COMPUTER ORGANIZATION AND DESIGN

THE HARDWARE/SOFTWARE INTERFACE

RISC-V EDITION



DAVID A. PATTERSON
JOHN L. HENNESSY

MORGAN KAUFMANN

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

"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. This new 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."

—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/are quickly becoming 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*

"*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 post-PC 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

RISC-V EDITION

Computer Organization and Design

THE HARDWARE/SOFTWARE INTERFACE

David A. Patterson is the Pardee Professor of Computer Science, Emeritus at the University of California at Berkeley, which he joined after graduating from UCLA in 1977. 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 coauthor, 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 US 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, CRA, and SIGARCH.

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 four 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 a Professor of Electrical Engineering and Computer Science at Stanford University, where he has been a member of the faculty since 1977 and was, from 2000 to 2016, its tenth President. 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.

R I S C - V 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

RISC-V updates and contributions by
Andrew S. Waterman
SiFive, Inc.

Yunsup Lee
SiFive, Inc.

Additional contributions by
Perry Alexander
The University of Kansas

Peter J. Ashenden
Ashenden Designs Pty Ltd

Jason D. Bakos
University of South Carolina

Javier Diaz Bruguera
Universidade de Santiago de Compostela

Jichuan Chang
Google

Matthew Farrens
University of California, Davis

David Kaeli
Northeastern University

Nicole Kaiyan
University of Adelaide

David Kirk
NVIDIA

Zachary Kurmas
Grand Valley State University

James R. Larus
School of Computer and Communications Science at EPFL

Jacob Leverich
Stanford University

Kevin Lim
Hewlett-Packard

Eric Love
University of California, Berkeley

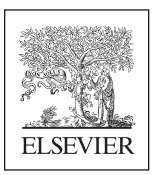
John Nickolls
NVIDIA

John Y. Oliver
Cal Poly, San Luis Obispo

Milos Prvulovic
Georgia Tech

Partha Ranganathan
Google

Mark Smotherman
Clemson University



MORGAN KAUFMANN PUBLISHERS
AN IMPRINT OF ELSEVIER
elsevier.com

Morgan Kaufmann is an imprint of Elsevier
50 Hampshire Street, 5th Floor, Cambridge, MA 02139, United States

Copyright © 2018 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, professional practices, or medical treatment may become necessary.

Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information, methods, compounds, or experiments 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.

RISC-V and the RISC-V logo are registered trademarks managed by the RISC-V Foundation, used under permission of the RISC-V Foundation. All rights reserved.

This publication is independent of the RISC-V Foundation, which is not affiliated with the publisher and the RISC-V Foundation does not authorize, sponsor, endorse or otherwise approve this publication.

All material relating to ARM® technology has been reproduced with permission from ARM Limited, and should only be used for education purposes. All ARM-based models shown or referred to in the text must not be used, reproduced or distributed for commercial purposes, and in no event shall purchasing this textbook be construed as granting you or any third party, expressly or by implication, estoppel or otherwise, a license to use any other ARM technology or know how. Materials provided by ARM are copyright © ARM Limited (or its affi liates).

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

Library of Congress Cataloging-in-Publication Data
A catalog record for this book is available from the Library of Congress

ISBN: 978-0-12-812275-4

For Information on all Morgan Kaufmann publications
visit our website at https://www.elsevier.com/books-and-journals



Publisher: Katey Birtcher
Acquisition Editor: Steve Merken
Development Editor: Nate McFadden
Production Project Manager: Lisa Jones
Designer: Victoria Pearson Esser

Typeset by MPS Limited, Chennai, India

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

ACKNOWLEDGMENTS

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

Figure 1.9 Courtesy of Chipworks (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 67
2.4 Signed and Unsigned Numbers 74
2.5 Representing Instructions in the Computer 81
2.6 Logical Operations 89
2.7 Instructions for Making Decisions 92
2.8 Supporting Procedures in Computer Hardware 98
2.9 Communicating with People 108
2.10 RISC-V Addressing for Wide Immediates and Addresses 113
2.11 Parallelism and Instructions: Synchronization 121
2.12 Translating and Starting a Program 124
2.13 A C Sort Example to Put it All Together 133
2.14 Arrays versus Pointers 141
2.15 Advanced Material: Compiling C and Interpreting Java 144

2.16 Real Stuff: MIPS Instructions 145
2.17 Real Stuff: x86 Instructions 146
2.18 Real Stuff: The Rest of the RISC-V Instruction Set 155
2.19 Fallacies and Pitfalls 157
2.20 Concluding Remarks 159

2.21 Historical Perspective and Further Reading 162
2.22 Exercises 162

3 Arithmetic for Computers 172

3.1 Introduction 174
3.2 Addition and Subtraction 174
3.3 Multiplication 177
3.4 Division 183
3.5 Floating Point 191
3.6 Parallelism and Computer Arithmetic: Subword Parallelism 216
3.7 Real Stuff: Streaming SIMD Extensions and Advanced Vector Extensions in x86 217
3.8 Going Faster: Subword Parallelism and Matrix Multiply 218
3.9 Fallacies and Pitfalls 222
3.10 Concluding Remarks 225

3.11 Historical Perspective and Further Reading 227
3.12 Exercises 227

4 The Processor 234

4.1 Introduction 236
4.2 Logic Design Conventions 240
4.3 Building a Datapath 243
4.4 A Simple Implementation Scheme 251
4.5 An Overview of Pipelining 262
4.6 Pipelined Datapath and Control 276
4.7 Data Hazards: Forwarding versus Stalling 294
4.8 Control Hazards 307
4.9 Exceptions 315
4.10 Parallelism via Instructions 321
4.11 Real Stuff: The ARM Cortex-A53 and Intel Core i7 Pipelines 334
4.12 Going Faster: Instruction-Level Parallelism and Matrix Multiply 342

4.13 Advanced Topic: An Introduction to Digital Design Using a Hardware Design Language to Describe and Model a Pipeline and More Pipelining Illustrations 345
4.14 Fallacies and Pitfalls 345
4.15 Concluding Remarks 346

4.16 Historical Perspective and Further Reading 347
4.17 Exercises 347

5 Large and Fast: Exploiting Memory Hierarchy 364

5.1 Introduction 366
5.2 Memory Technologies 370
5.3 The Basics of Caches 375
5.4 Measuring and Improving Cache Performance 390
5.5 Dependable Memory Hierarchy 410
5.6 Virtual Machines 416
5.7 Virtual Memory 419
5.8 A Common Framework for Memory Hierarchy 443
5.9 Using a Finite-State Machine to Control a Simple Cache 449
5.10 Parallelism and Memory Hierarchy: Cache Coherence 454
5.11 Parallelism and Memory Hierarchy: Redundant Arrays of Inexpensive Disks 458
5.12 Advanced Material: Implementing Cache Controllers 459
5.13 Real Stuff: The ARM Cortex-A53 and Intel Core i7 Memory Hierarchies 459
5.14 Real Stuff: The Rest of the RISC-V System and Special Instructions 464
5.15 Going Faster: Cache Blocking and Matrix Multiply 465
5.16 Fallacies and Pitfalls 468
5.17 Concluding Remarks 472
5.18 Historical Perspective and Further Reading 473
5.19 Exercises 473

6 Parallel Processors from Client to Cloud 490

6.1 Introduction 492
6.2 The Difficulty of Creating Parallel Processing Programs 494
6.3 SISD, MIMD, SIMD, SPMD, and Vector 499
6.4 Hardware Multithreading 506
6.5 Multicore and Other Shared Memory Multiprocessors 509
6.6 Introduction to Graphics Processing Units 514
6.7 Clusters, Warehouse Scale Computers, and Other Message-Passing Multiprocessors 521
6.8 Introduction to Multiprocessor Network Topologies 526
6.9 Communicating to the Outside World: Cluster Networking 529
6.10 Multiprocessor Benchmarks and Performance Models 530
6.11 Real Stuff: Benchmarking and Rooflines of the Intel Core i7 960 and the NVIDIA Tesla GPU 540
6.12 Going Faster: Multiple Processors and Matrix Multiply 545
6.13 Fallacies and Pitfalls 548
6.14 Concluding Remarks 550
6.15 Historical Perspective and Further Reading 553
6.16 Exercises 553

APPENDIX



The Basics of Logic Design A-2

A.1 Introduction A-3
A.2 Gates, Truth Tables, and Logic Equations A-4
A.3 Combinational Logic A-9
A.4 Using a Hardware Description Language A-20
A.5 Constructing a Basic Arithmetic Logic Unit A-26
A.6 Faster Addition: Carry Lookahead A-37
A.7 Clocks A-47
A.8 Memory Elements: Flip-Flops, Latches, and Registers A-49
A.9 Memory Elements: SRAMs and DRAMs A-57
A.10 Finite-State Machines A-66
A.11 Timing Methodologies A-71
A.12 Field Programmable Devices A-77
A.13 Concluding Remarks A-78
A.14 Exercises A-79

Index I-1

ONLINE CONTENT



Graphics and Computing GPUs B-2

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



Mapping Control to Hardware C-2

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



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

D.1 Introduction D-3
D.2 Addressing Modes and Instruction Formats D-5
D.3 Instructions: the MIPS Core Subset D-9
D.4 Instructions: Multimedia Extensions of the Desktop/Server RISCs D-16
D.5 Instructions: Digital Signal-Processing Extensions of the Embedded RISCs D-19
D.6 Instructions: Common Extensions to MIPS Core D-20
D.7 Instructions Unique to MIPS-64 D-25
D.8 Instructions Unique to Alpha D-27
D.9 Instructions Unique to SPARC v9 D-29
D.10 Instructions Unique to PowerPC D-32
D.11 Instructions Unique to PA-RISC 2.0 D-34
D.12 Instructions Unique to ARM D-36
D.13 Instructions Unique to Thumb D-38
D.14 Instructions Unique to SuperH D-39
D.15 Instructions Unique to M32R D-40
D.16 Instructions Unique to MIPS-16 D-40
D.17 Concluding Remarks D-43


Glossary G-1


Further Reading FR-1

This page intentionally left blank

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.

Why RISC-V for This Edition?

The choice of instruction set architecture is clearly critical to the pedagogy of a computer architecture textbook. We didn't want an instruction set that required describing unnecessary baroque features for someone's first instruction set, no matter how popular it is. Ideally, your initial instruction set should be an exemplar, just like your first love. Surprisingly, you remember both fondly.

Since there were so many choices at the time, for the first edition of *Computer Architecture: A Quantitative Approach* we invented our own RISC-style instruction set. Given the growing popularity and the simple elegance of the MIPS instruction set, we switched to it for the first edition of this book and to later editions of the other book. MIPS has served us and our readers well.

It's been 20 years since we made that switch, and while billions of chips that use MIPS continue to be shipped, they are typically in found embedded devices where the instruction set is nearly invisible. Thus, for a while now it's been hard to find a real computer on which readers can download and run MIPS programs.

The good news is that an open instruction set that adheres closely to the RISC principles has recently debuted, and it is rapidly gaining a following. RISC-V, which was developed originally at UC Berkeley, not only cleans up the quirks of the MIPS

instruction set, but it offers a simple, elegant, modern take on what instruction sets should look like in 2017.

Moreover, because it is not proprietary, there are open-source RISC-V simulators, compilers, debuggers, and so on easily available and even open-source RISC-V implementations available written in hardware description languages. In addition, there will soon be low-cost hardware platforms on which to run RISC-V programs. Readers will not only benefit from studying these RISC-V designs, they will be able to modify them and go through the implementation process in order to understand the impact of their hypothetical changes on performance, die size, and energy.

This is an exciting opportunity for the computing industry as well as for education, and thus at the time of this writing more than 40 companies have joined the RISC-V foundation. This sponsor list includes virtually all the major players except for ARM and Intel, including AMD, Google, Hewlett Packard Enterprise, IBM, Microsoft, NVIDIA, Oracle, and Qualcomm.

It is for these reasons that we wrote a RISC-V edition of this book, and we are switching *Computer Architecture: A Quantitative Approach* to RISC-V as well.

Given that RISC-V offers both 32-bit address instructions and 64-bit address instructions with essentially the same instruction set, we could have switched instruction sets but kept the address size at 32 bits. Our publisher polled the faculty who used the book and found that 75% either preferred larger addresses or were neutral, so we increased the address space to 64 bits, which may make more sense today than 32 bits.

The only changes for the RISC-V edition from the MIPS edition are those associated with the change in instruction sets, which primarily affects Chapter 2, Chapter 3, the virtual memory section in Chapter 5, and the short VMIPS example in Chapter 6. In Chapter 4, we switched to RISC-V instructions, changed several figures, and added a few "Elaboration" sections, but the changes were simpler than we had feared. Chapter 1 and the rest of the appendices are virtually unchanged. The extensive online documentation and combined with the magnitude of RISC-V make it difficult to come up with a replacement for the MIPS version of Appendix A ("Assemblers, Linkers, and the SPIM Simulator" in the MIPS Fifth Edition). Instead, Chapters 2, 3, and 5 include quick overviews of the hundreds of RISC-V instructions outside of the core RISC-V instructions that we cover in detail in the rest of the book.

Note that we are not (yet) saying that we are permanently switching to RISC-V. For example, in addition to this new RISC-V edition, there are ARMv8 and MIPS versions available for sale now. One possibility is that there will be a demand for all versions for future editions of the book, or for just one. We'll cross that bridge when we come to it. For now, we look forward to your reaction to and feedback on this effort.

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 main themes across the topics using margin icons that are introduced

early; update examples to reflect changeover from PC era to post-PC 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 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 parallels 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 A 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.

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)		
D. RISC Instruction-Set Architectures	D.1 to D.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)		
A. The Basics of Logic Design	A.1 to A.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)		
C. Mapping Control to Hardware	C.1 to C.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 5.17		
	5.18 (History)		
6. Parallel Process from Client to Cloud	6.1 to 6.8		
	6.9 (Networks)		
	6.10 to 6.14		
	6.15 (History)		
B. Graphics Processor Units	B.1 to B.13		

Read carefully Read if have time Reference

Review or read Read for culture

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. Chapter 4, The Processor, 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 post-PC 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 post-PC 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 A53 and the Intel Core i7, reflecting our post-PC era. Other highlights include 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 RISC-V 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 RISC-V assembly language programs.

Instructor Support

We have collected a great deal of material to help instructors teach courses using this book. Solutions to exercises, figures from the book, lecture slides, and other materials are available to instructors who register with the publisher. In addition, the companion Web site provides links to a free RISC-V software. Check the publisher's Web site for more information:

textbooks.elsevier.com/9780128122754

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

This edition is the third 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

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.

We are grateful for the assistance of **Khaled Benkrid** and his colleagues at ARM Ltd., who carefully reviewed the ARM-related material and provided helpful feedback.

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 Avizenis** 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 **Zachary Kurmas** of Grand Valley State University, who updated and created new exercises, based on originals created by **Perry Alexander** (The University of Kansas); **Jason Bakos** (University of South Carolina); **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); **Jichuan Chang** (Google); **Jacob Leverich** (Stanford); **Kevin Lim** (Hewlett-Packard); and **Partha Ranganathan** (Google).

Additional thanks goes to **Peter Ashenden** for updating the lecture slides.

I am grateful to the many instructors who have answered the publisher's surveys, reviewed our proposals, and attended focus groups. They include the following individuals: Focus Groups: 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), 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), Behnam Arad (Sacramento State University), 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), Mike Bright (Grove City College), 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), Gene Cooperman (Northeastern University), Robert D. Cupper (Allegheny College), Amy Csizmar Dalal (Carleton College), Daniel Dalle (Université de Sherbrooke), Edward W. Davis (North Carolina State University), Nathaniel J. Davis (Air Force Institute of Technology), Molisa Derk (Oklahoma City University), Andrea Di Blas (Stanford University), Derek Eager (University of Saskatchewan), Ata Elahi (Souther Connecticut State University), Ernest Ferguson (Northwest Missouri State University), Rhonda Kay Gaede (The University of Alabama), Etienne M. Gagnon (L'Université du Québec à Montréal), Costa Gerousis (Christopher Newport University), Paul Gillard (Memorial University of Newfoundland), Michael Goldweber (Xavier University), Georgia Grant (College of San Mateo), Paul V. Gratz (Texas A&M University), 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), Jayantha Herath (St. Cloud State University), Martin Herbordt (Boston University), Steve J. Hodges (Cabrillo College), Kenneth Hopkinson (Cornell University), Bill Hsu (San Francisco State 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), Adrian Lauf (University of Louisville), Robert N. Lea (University of Houston), Alvin Lebeck (Duke University), 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), Stephen Mann (University of Waterloo), Bill Mark (University of Texas at Austin), Ananda Mondal (Claflin University), Alvin Moser (Seattle University),

Walid Najjar (University of California, Riverside), Vijaykrishnan Narayanan (Penn State University), Danial J. Neebel (Loras College), Victor Nelson (Auburn University), John Nestor (Lafayette College), Jae C. Oh (Syracuse University), Joe Oldham (Centre College), Timour Paltashev, James Parkerson (University of Arkansas), Shaunak Pawagi (SUNY at Stony Brook), Steve Pearce, Ted Pedersen

(University of Minnesota), Lu Peng (Louisiana State University), Gregory D. Peterson (The University of Tennessee), William Pierce (Hood College), 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), Shai Simonson (Stonehill College), 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), Steve VanderLeest (Calvin College), Christopher Vickery (Queens College of CUNY), 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), Amr Zaky (Santa Clara University), Gerald D. Zarnett (Ryerson University), Nian Zhang (South Dakota School of Mines & Technology), Jiling Zhong (Troy University), Huiyang Zhou (North Carolina State University), 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 **Katey Birtcher**, **Steve Merken**, 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 **Victoria Pearson Esser**, who did the cover design. The cover cleverly connects the post-PC era content of this edition to the cover of the first edition.

Finally, I owe a huge debt to **Yunsup Lee** and **Andrew Waterman** for taking on this conversion to RISC-V in their spare time while founding a startup company. Kudos to **Eric Love** as well, who made RISC-V versions of the exercises in this edition while finishing his Ph.D. We're all excited to see what will happen with RISC-V in academia and beyond.

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

David A. Patterson

This page intentionally left blank

1

Computer Abstractions and Technology

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

Alfred North Whitehead,
An Introduction to Mathematics, 1911

1.1 **Introduction** 3
1.2 **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

Computer Organization and Design. DOI: http://dx.doi.org/10.1016/B978-0-12-812275-4.00001-4
© 2018 Elsevier Inc. All rights reserved.

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 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.

Traditional 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 distinct design requirements and employ the core hardware technologies in different ways. Broadly speaking, computers are used in three dissimilar 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 merely 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 sizable 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 higher 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. 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 thus a proportionally tiny 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}	exbibyte	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 to be as fast as necessary to handle its limited function, and beyond that, minimizing cost and power is the most important objective. 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 Post-PC 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

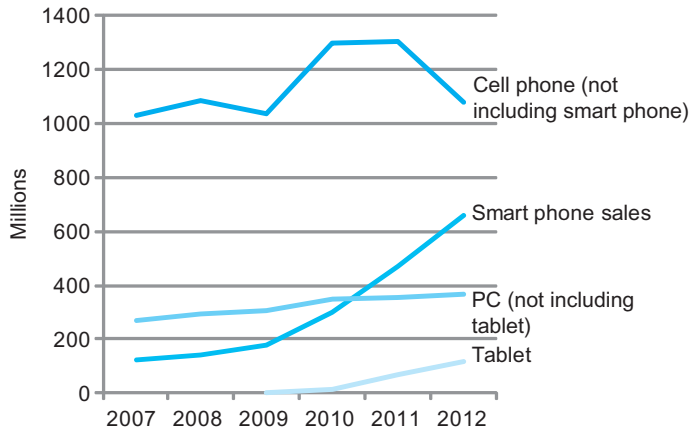


FIGURE 1.2 The number manufactured per year of tablets and smart phones, which reflect the post-PC era, versus personal computers and traditional cell phones. Smart phones represent the recent growth in the cell phone industry, and they passed PCs in 2011. Tablets are the fastest growing category, nearly 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 over time of tablets and smart phones versus that of PCs and traditional cell phones.

Taking over from the conventional 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.

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.

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 popular software. In the 1960s and 1970s, a primary constraint on computer performance was the size of the computer's memory. Thus, programmers often followed a simple credo: minimize memory space to make programs fast. In the

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

Programmers interested in performance now need to understand the issues that have replaced the simple memory model of the 1960s: the parallel nature of processors and the hierarchical nature of memories. We demonstrate the importance of this understanding in Chapters 3 to 6 by showing how to improve performance of a C program by a factor of 200. 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.

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

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

Understanding Program Performance

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

Hardware or software component	How this component affects performance	Where is this topic covered?
Algorithm	Determines both the number of source-level statements and the number of I/O operations executed	Other books!
Programming language, compiler, and architecture	Determines the number of computer instructions for each source-level statement	Chapters 2 and 3
Processor and memory system	Determines how fast instructions can be executed	Chapters 4, 5, and 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, as mentioned above, 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 2.0 to 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 4 to 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.

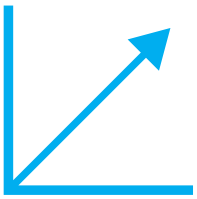
1. The number of embedded processors sold every year greatly outnumbers the number of PC and even post-PC processors. Can you confirm or deny this insight based on your own experience? Try to count the number of embedded processors in your home. How does it compare with the number of conventional computers 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 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.

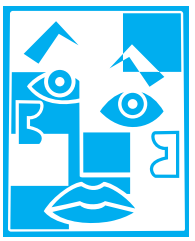
Design for Moore's Law

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.


MOORE'S LAW

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 characterize 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 usually 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!

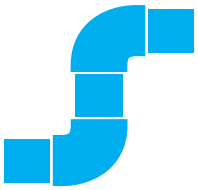

COMMON CASE FAST



PARALLELISM

Performance via Parallelism

Since the dawn of computing, computer architects have offered designs that get more performance by computing 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**.



PIPELINING

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.



PREDICTION

Performance via Prediction

Following the saying that it can be better to ask for forgiveness than to ask for permission, the next 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

Hierarchy of Memories

Programmers want the 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 the 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 almost 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

Dependability via Redundancy

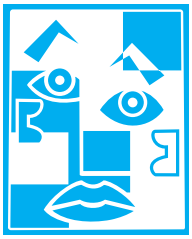
Computers not only need to be fast; they need to be dependable. Since any physical device can fail, we make systems **dependable** by including redundant components that can take over when a failure occurs *and* to help detect failures. We use the tractor-trailer as our icon, since the dual tires on each side of its rear axles 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

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

Mark Twain, *The Innocents Abroad*, 1869

A typical application, such as a word processor or a large database system, may consist of 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 primitive 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**.



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 the application 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.

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.

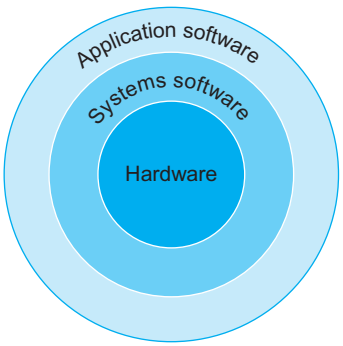


FIGURE 1.3 A simplified view of hardware and software as hierarchical layers, shown as concentric circles with hardware in the center and application software outermost. In complex applications, there are often multiple layers of application software as well. For example, a database system may run on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

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.

From a High-Level Language to the Language of Hardware

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

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

instruction A command that computer hardware understands and obeys.

```
1001010100101110
```

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 software to translate from symbolic notation to binary. The first of these programs was named an **assembler**. This program translates a symbolic version of an instruction into the binary version. For example, the programmer would write

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

```
add A, B
```

and the assembler would translate this notation into

```
1001010100101110
```

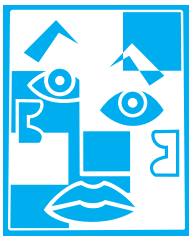
This instruction tells the computer to add the two numbers `A` and `B`. The name coined for this symbolic language, still used today, is **assembly language**. In contrast, the binary language that the machine understands is the **machine language**.

assembly language A symbolic representation of machine instructions.

machine language A binary representation of machine instructions.

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

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

High-level language program (in C)

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

Compiler

Assembly language program (for RISC-V)

```
swap:
      slli x6, x11, 3
      add  x6, x10, x6
      ld   x5, 0(x6)
      ld   x7, 8(x6)
      sd   x7, 0(x6)
      sd   x5, 8(x6)
      jalr x0, 0(x1)
```

Assembler

Binary machine language program (for RISC-V)

```
00000000000110101100100110010011
00000000011001010000001100110011
00000000000000011001100101000011
00000000100000110011001110000011
00000000011100110011000000100011
00000000010100110011010000100011
00000000000000000100000001100111
```

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.

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.

When we come to an important point in this book, a point so significant 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

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 BIG Picture

The five classic components of a computer are input, output, memory, datapath, and control, with the last two sometimes combined and called the processor. Figure 1.5 shows the standard organization of a computer. This organization is independent of hardware technology: you can place every piece of every computer, past and present, into one of these five categories. To help you keep all this in perspective, the five components of a computer are shown on the front page of each of the following chapters, with the portion of interest to that chapter highlighted.

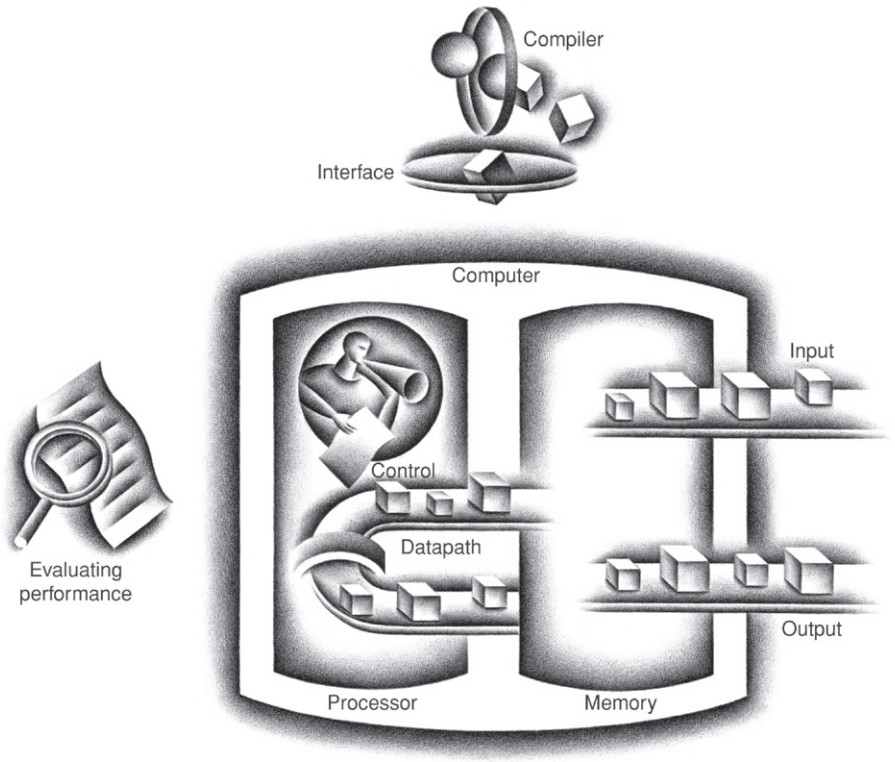


FIGURE 1.5 The organization of a computer, showing the five classic components. The processor gets instructions and data from memory. Input writes data to memory, and output reads data from memory. Control sends the signals that determine the operations of the datapath, memory, input, and output.

Through the Looking Glass

liquid crystal display (LCD) 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 LCDs use an **active matrix** that has a tiny transistor switch at each pixel to control current precisely 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 represent faithfully 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.

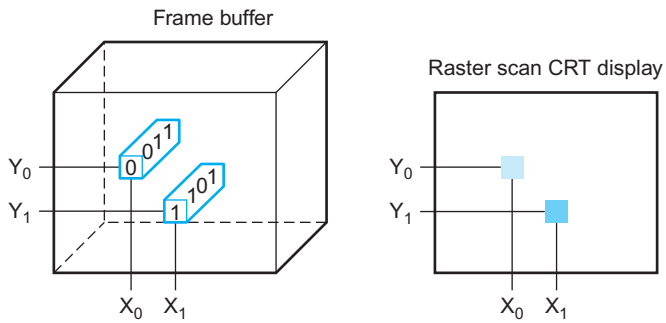


FIGURE 1.6 Each coordinate in the frame buffer on the left determines the shade of the corresponding coordinate for the raster scan CRT display on the right. Pixel (X_0, Y_0) contains the bit pattern 0011, which is a lighter shade on the screen than the bit pattern 1101 in pixel (X_1, Y_1).

Touchscreen

While PCs also use LCDs, the tablets and smartphones of the post-PC era have replaced the keyboard and mouse with touch-sensitive displays, which has the wonderful user interface advantage of users pointing directly at 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 recognizes 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, 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 Figure 1.8 contains two ARM processors that operate at 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. 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 six-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)

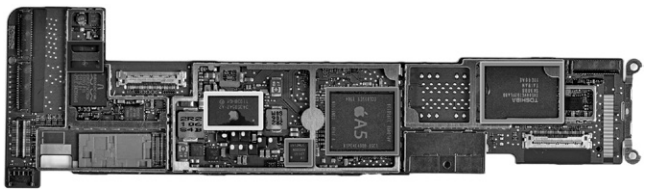


FIGURE 1.8 The logic board of Apple iPad 2 in Figure 1.7. The photo highlights five integrated circuits. The large integrated circuit in the middle is the Apple A5 chip, which contains dual ARM processor cores that run at 1 GHz as well as 512 MB of main memory inside the package. Figure 1.9 shows a photograph 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)

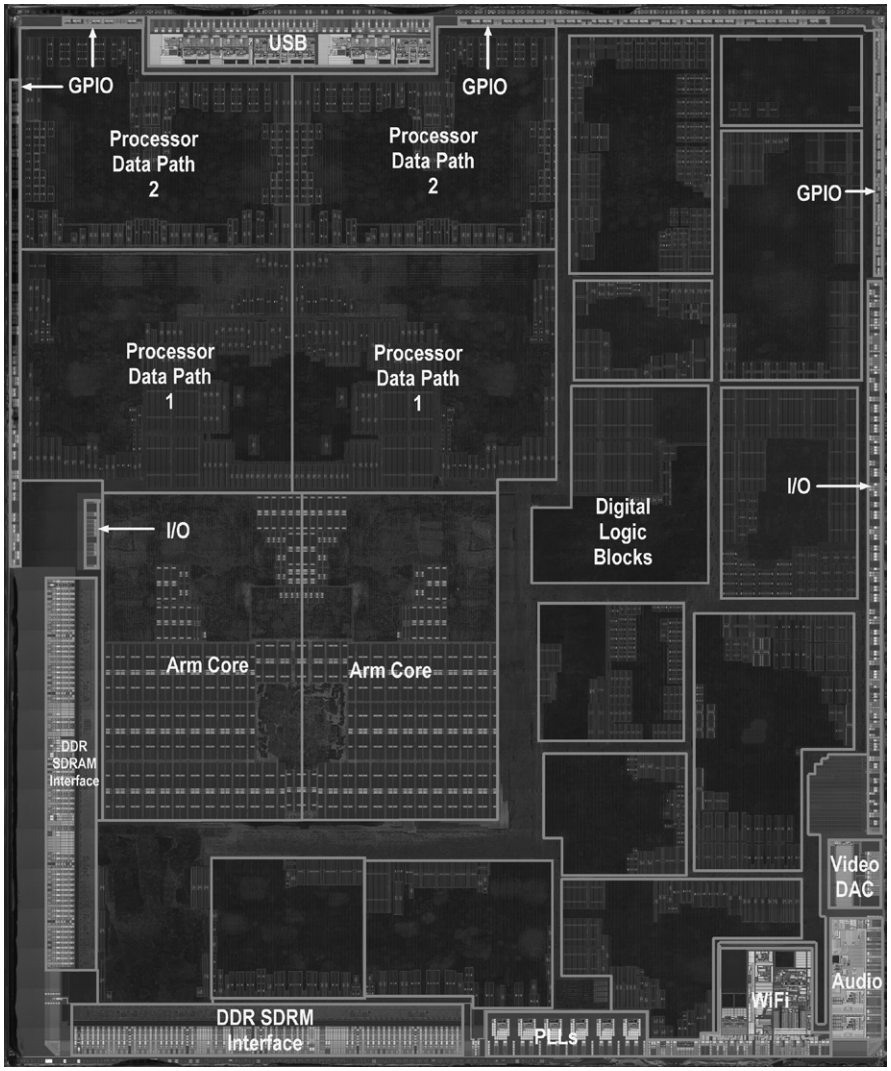


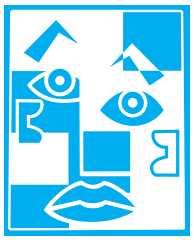
FIGURE 1.9 The processor integrated circuit inside the A5 package. The size of chip is 12.1 by 10.1 mm, and it was manufactured originally in a 45-nm process (see Section 1.5). It has two identical ARM processors or cores in the middle left of the chip and a PowerVR *graphics processing unit* (GPU) with 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 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.



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**.



ABSTRACTION

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)**.

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.

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) separately 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.

The 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.

implementation Hardware that obeys the architecture abstraction.

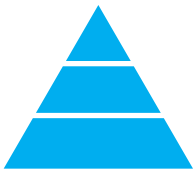
A Safe Place for Data

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 therefore a **nonvolatile memory** technology.

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.

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.



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.

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;

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 considerably cheaper and had a significantly 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 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 post-PC 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 forecasting 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**.

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.

This rate of increasing integration has been remarkably stable. Figure 1.11 shows the growth in DRAM capacity since 1977. For 35 years, the industry has consistently quadrupled capacity every 3 years, resulting in an increase in excess of 16,000 times!

To understand how to 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:

silicon A natural element that is a semiconductor.

semiconductor A substance that does not conduct electricity well.

- Excellent conductors of electricity (using either microscopic copper or aluminum wire)

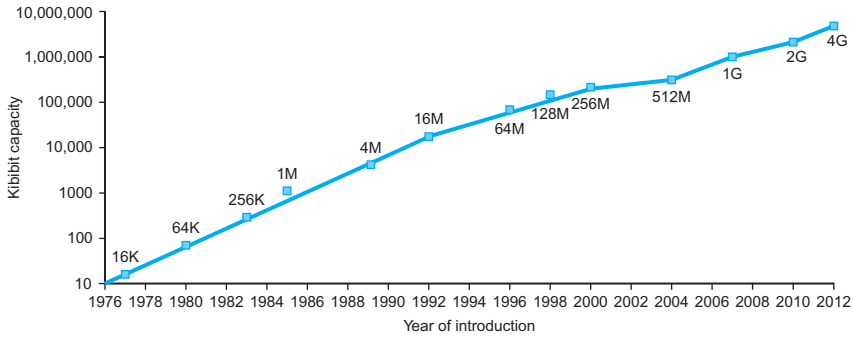


FIGURE 1.11 Growth of capacity per DRAM chip over time. The *y*-axis is measured in kibibits (2^{10} bits). The DRAM industry quadrupled capacity almost every three years, a 60% increase per year, for 20 years. In recent years, the rate has slowed down and is somewhat closer to doubling every two to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

- Excellent insulators from electricity (like plastic sheathing or glass)
- Areas that can conduct or insulate under specific conditions (as a switch)

Transistors fall into 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.

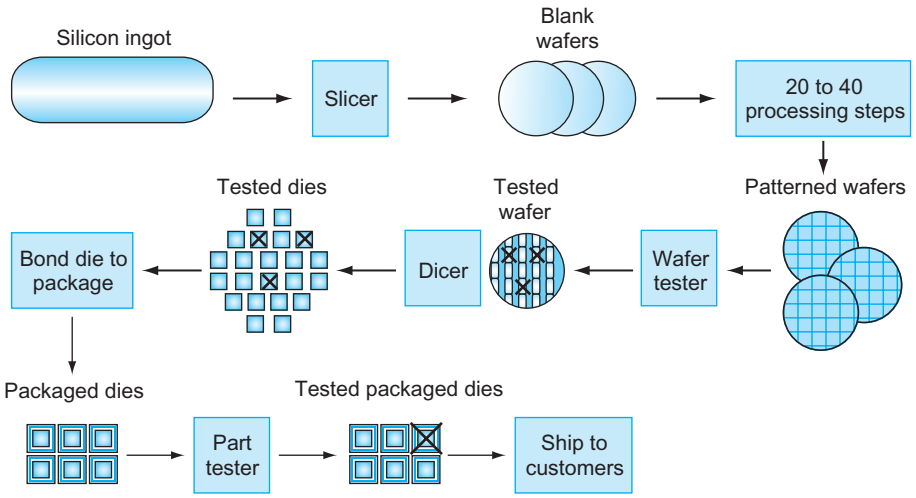


FIGURE 1.12 The chip manufacturing process. After being sliced from the silicon ingot, blank wafers are put through 20 to 40 steps to create patterned wafers (see Figure 1.13). These patterned wafers are then tested with a wafer tester, and a map of the good parts is made. Next, the wafers are diced 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,

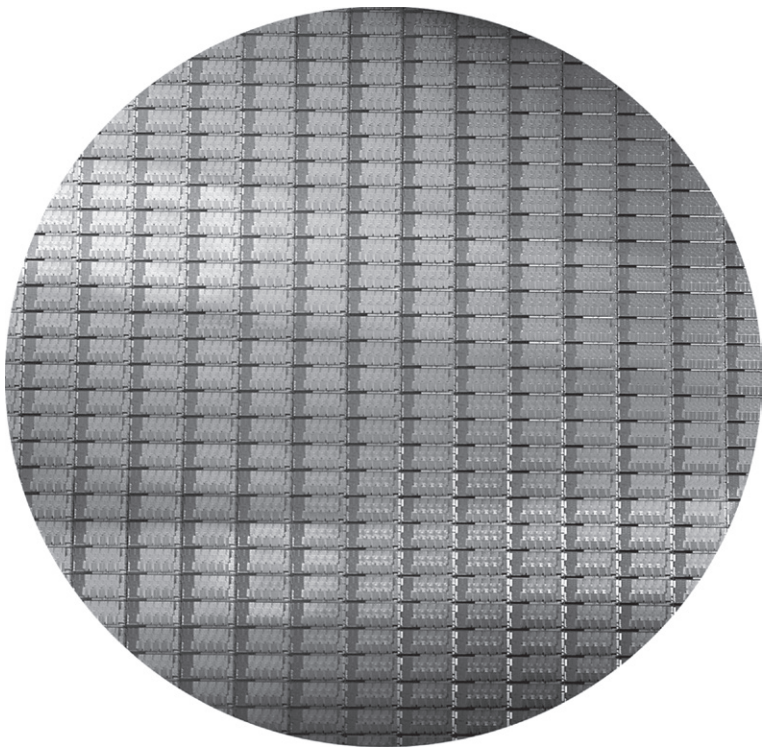


FIGURE 1.13 A 12-inch (300 mm) wafer of Intel Core i7 (Courtesy Intel). The number of dies on this 300 mm (12 inch) wafer at 100% yield is 280, each 20.7 by 10.5 mm. The several dozen partially rounded chips at the boundaries of the wafer are useless; they are included because it's easier to create the 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 to the fewer 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 those 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 distinct ways.

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

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.

execution time. Datacenter managers often care about 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

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

ANSWER

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

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

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

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

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

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

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

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

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

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

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

If X is *n* times 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

EXAMPLE

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

ANSWER

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

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

Thus the performance ratio is

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

and A is therefore 1.5 times 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 between operating systems.

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.

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

Understanding Program Performance

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

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

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

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

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 four 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

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

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

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.

Improving Performance

EXAMPLE

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 identical 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

EXAMPLE

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

ANSWER

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

$$\text{CPU clock cycles}_A = I \times 2.0$$
$$\text{CPU clock cycles}_B = I \times 1.2$$

Now we can compute the CPU time for each computer:

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

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

EXAMPLE

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

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

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

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

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

ANSWER

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

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

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

This yields

$$\text{CPU clock cycles}_1 = (2 \times 1) + (1 \times 2) + (2 \times 3) = 2 + 2 + 6 = 10 \text{ cycles}$$
$$\text{CPU clock cycles}_2 = (4 \times 1) + (1 \times 2) + (1 \times 3) = 4 + 2 + 3 = 9 \text{ cycles}$$

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

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

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

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

The BIG Picture

Figure 1.15 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:

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

Always bear in mind that the only complete and reliable measure of computer performance is time. For example, changing the instruction set to lower the instruction count may lead to an organization with a slower clock cycle time or higher CPI that offsets the improvement in instruction count. Similarly, because CPI depends on the 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 equal. 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.

Understanding Program Performance

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

Hardware or software component	Affects what?	How?
Algorithm	Instruction count, 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 varied 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 two 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:

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

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

c. $\dfrac{1.5 \times 1.1}{0.6} = 27.5\,\text{sec}$

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.

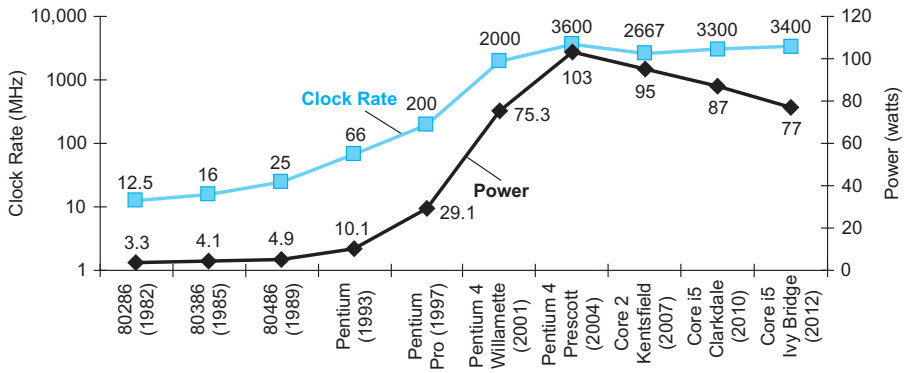


FIGURE 1.16 Clock rate and power for Intel x86 microprocessors over eight generations and 30 years. The Pentium 4 made a dramatic jump in clock rate and power but less so in performance. The Prescott thermal problems led to the abandonment of the Pentium 4 line. The Core 2 line reverts to a simpler pipeline 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 post-PC era the really valuable 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 evaluation 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:

$$Energy \propto Capacitive\ load \times Voltage^2$$

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

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

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

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

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 increased 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.

EXAMPLE

Relative Power

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

ANSWER

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

Thus the power ratio is

$$0.85^4 = 0.52$$

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

The modern problem 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 costly 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 one 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 the performance of their code as the number of cores increases.

To reinforce how the software and hardware systems work together, 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.

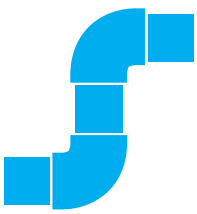


PARALLELISM

Hardware/ Software Interface

Parallelism has always been crucial 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 optimization 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 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.



PIPELINING

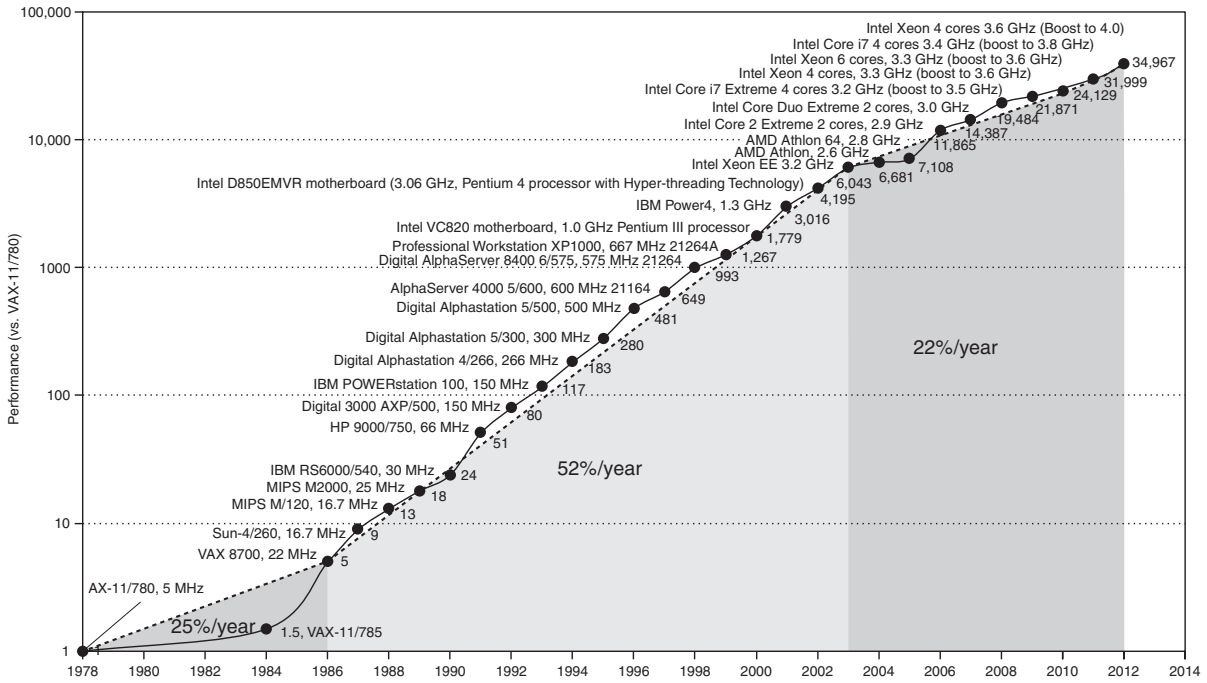


FIGURE 1.17 Growth in processor performance since the mid-1980s. This chart plots performance relative to the VAX 11/780 as measured by the SPECint benchmarks (see Section 1.10). Prior to the mid-1980s, processor performance growth was largely technology-driven and averaged about 25% per year. The increase 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 larger 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.

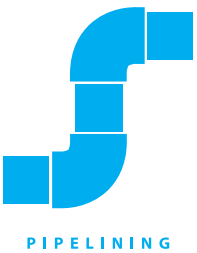
As an analogy, suppose the task was to write a newspaper story. Eight reporters working on the same story could potentially write a story eight times faster. To achieve this increased speed, one would need to break up the task so that each reporter had something to do at the same time. Thus, we must *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*

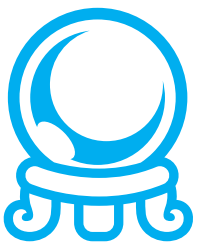
load evenly to get the desired speedup. Another danger would be if reporters had to spend a lot of time talking to each other to write their sections. You would also fall short if one part of the story, such as the conclusion, couldn't be written until all the other parts were completed. Thus, care must 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 has 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 provide 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 concurrently 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 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 various RAID levels.











PARALLELISM

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.

Appendix B 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**.

Appendix B describes the NVIDIA GPU and highlights parts of its parallel programming environment.

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

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

1.9 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

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.



COMMON CASE FAST

benchmark A program selected for use in comparing computer performance.

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 2.66 GHz Intel Core i7 920. As the equation on page 36 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 summarizing all 12 integer benchmarks. Dividing the execution time of a reference processor by the execution time of the evaluated 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, apply 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 Execution time ratio$_i$ is the execution time, normalized to the reference computer, for the ith program of a total of n in the workload, and

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

SPEC Power Benchmark

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
∑ssj_ops / ∑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 one number, called "overall ssj_ops per watt." The formula for this single summarizing metric is

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

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

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

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

The purpose of a section on fallacies and pitfalls, which will be found in every chapter, is to explain some commonly held misconceptions that you might encounter. We call 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 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.



COMMON CASE FAST

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**:

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

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

For this problem:

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

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

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

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

That is, there is *no amount* by which we can enhance-multiply to achieve a fivefold increase in performance, if multiply accounts for only 80% of the workload. The performance enhancement possible with a given improvement is limited by the amount that the improved feature is used. 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 possible 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 5 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 CO_2 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 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 the 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

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.

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

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 quicker computers mean bigger MIPS, which matches intuition.

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

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

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!

Check Yourself

Consider the following performance measurements for a program:

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

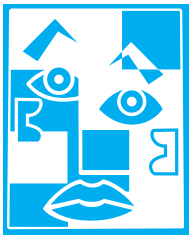
a. Which computer has the higher MIPS rating?

b. Which computer is faster?

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

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.

The BIG Picture

Execution time is the only valid and unimpeachable measure of performance. Many other metrics have been proposed and found wanting. Sometimes these metrics are flawed from the start by not reflecting execution time; other times a metric that is sound 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, normally 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 requiring 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 is 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 B**
- *Control*: Chapter 4, Chapter 6, and **Appendix B**
- *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 B** 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. Chapter 3 describes how computers handle arithmetic data. Appendix A introduces logic design.



1.12 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 studying 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



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. We may trace the development of an idea through a series of machines or describe some important projects, and we provide references in case you are interested in probing further.

The historical perspective for this chapter provides a background for some of the key ideas presented therein. Its purpose is to give you the human story behind the technological advances and to place achievements in their historical context. By learning the past, you may be better able to understand the forces that will shape computing in the future. Each historical perspective section ends with suggestions for additional reading, which are also collected separately in the online section "Further Reading."

The First Electronic Computers

J. Presper Eckert and John Mauchly at the Moore School of the University of Pennsylvania built what is widely accepted to be the world's first operational electronic, general-purpose computer. This machine, called ENIAC (*Electronic Numerical Integrator and Calculator*), was funded by the United States Army and started working during World War II but was not publicly disclosed until 1946. ENIAC was a general-purpose machine used for computing artillery-firing tables. Figure e1.12.1 shows the U-shaped computer, which was 80 feet long by 8.5 feet

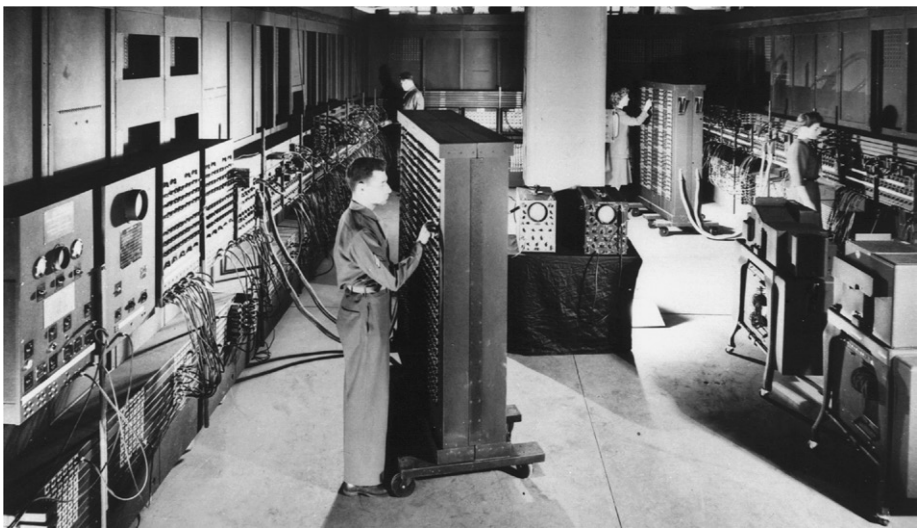


FIGURE e1.12.1 ENIAC, the world's first general-purpose electronic computer.

high and several feet wide. Each of the 20 10-digit registers was 2 feet long. In total, ENIAC used 18,000 vacuum tubes.

In size, ENIAC was two orders of magnitude bigger than machines built today, yet it was more than eight orders of magnitude slower, performing 1900 additions per second. ENIAC provided conditional jumps and was programmable, clearly distinguishing it from earlier calculators. Programming was done manually by plugging cables and setting switches, and data were entered on punched cards. Programming for typical calculations required from half an hour to a whole day. ENIAC was a general-purpose machine, limited primarily by a small amount of storage and tedious programming.

In 1944, John von Neumann was attracted to the ENIAC project. The group wanted to improve the way programs were entered and discussed storing programs as numbers; von Neumann helped crystallize the ideas and wrote a memo proposing a stored-program computer called EDVAC (*Electronic Discrete Variable Automatic Computer*). Herman Goldstine distributed the memo and put von Neumann's name on it, much to the dismay of Eckert and Mauchly, whose names were omitted. This memo has served as the basis for the commonly used term *von Neumann computer*. Several early pioneers in the computer field believe that this term gives too much credit to von Neumann, who wrote up the ideas, and too little to the engineers, Eckert and Mauchly, who worked on the machines. For this reason, the term does not appear elsewhere in this book or in the online sections.

In 1946, Maurice Wilkes of Cambridge University visited the Moore School to attend the latter part of a series of lectures on developments in electronic computers. When he returned to Cambridge, Wilkes decided to embark on a project to build a stored-program computer named EDSAC (*Electronic Delay Storage Automatic Calculator*). EDSAC started working in 1949 and was the world's first full-scale, operational, stored-program computer [Wilkes, 1985]. (A small prototype called the Mark-I, built at the University of Manchester in 1948, might be called the first operational stored-program machine.) Section 2.5 explains the stored-program concept.

In 1947, Eckert and Mauchly applied for a patent on electronic computers. The dean of the Moore School demanded that the patent be turned over to the university, which may have helped Eckert and Mauchly conclude that they should leave. Their departure crippled the EDVAC project, delaying completion until 1952.

Goldstine left to join von Neumann at the Institute for Advanced Study (IAS) at Princeton in 1946. Together with Arthur Burks, they issued a report based on the memo written earlier [Burks et al., 1946]. The paper was incredible for the period; reading it today, you would never guess this landmark paper was written more than 50 years ago, because it discusses most of the architectural concepts seen in modern computers. This paper led to the IAS machine built by Julian Bigelow. It had a total of 1024 40-bit words and was roughly 10 times faster than ENIAC. The group thought about uses for the machine, published a set of reports,

and encouraged visitors. These reports and visitors inspired the development of a number of new computers.

Recently, there has been some controversy about the work of John Atanasoff, who built a small-scale electronic computer in the early 1940s. His machine, designed at Iowa State University, was a special-purpose computer that was never completely operational. Mauchly briefly visited Atanasoff before he built ENIAC. The presence of the Atanasoff machine, together with delays in filing the ENIAC patents (the work was classified and patents could not be filed until after the war) and the distribution of von Neumann's EDVAC paper, was used to break the Eckert-Mauchly patent. Though controversy still rages over Atanasoff's role, Eckert and Mauchly are usually given credit for building the first working, general-purpose, electronic computer [Stern, 1980].

Another pioneering computer that deserves credit was a special-purpose machine built by Konrad Zuse in Germany in the late 1930s and early 1940s. Although Zuse had the design for a programmable computer ready, the German government decided not to fund scientific investigations taking more than 2 years because the bureaucrats expected the war would be won by that deadline.

Across the English Channel, during World War II special-purpose electronic computers were built to decrypt intercepted German messages. A team at Bletchley Park, including Alan Turing, built the Colossus in 1943. The machines were kept secret until 1970; after the war, the group had little impact on commercial British computers.

While work on ENIAC went forward, Howard Aiken was building an electro-mechanical computer called the Mark-I at Harvard (a name that Manchester later adopted for its machine). He followed the Mark-I with a relay machine, the Mark-II, and a pair of vacuum tube machines, the Mark-III and Mark-IV. In contrast to earlier machines like EDSAC, which used a single memory for instructions and data, the Mark-III and Mark-IV had separate memories for instructions and data. The machines were regarded as reactionary by the advocates of stored-program computers; the term *Harvard architecture* was coined to describe machines with distinct memories. Paying respect to history, this term is used today in a different sense to describe machines with a single main memory but with separate caches for instructions and data.

The Whirlwind project was begun at MIT in 1947 and was aimed at applications in real-time radar signal processing. Although it led to several inventions, its most important innovation was magnetic core memory. Whirlwind had 2048 16-bit words of magnetic core. Magnetic cores served as the main memory technology for nearly 30 years.

Commercial Developments

In December 1947, Eckert and Mauchly formed Eckert-Mauchly Computer Corporation. Their first machine, the BINAC, was built for Northrop and was shown in August 1949. After some financial difficulties, their firm was acquired by Remington-Rand, where they built the UNIVAC I (Universal Automatic

Computer), designed to be sold as a general-purpose computer (Figure e1.12.2). Originally delivered in June 1951, UNIVAC I sold for about $1 million and was the first successful commercial computer—48 systems were built! This early machine, along with many other fascinating pieces of computer lore, may be seen at the Computer History Museum in Mountain View, California.



FIGURE e1.12.2 UNIVAC I, the first commercial computer in the United States. It correctly predicted the outcome of the 1952 presidential election, but its initial forecast was withheld from broadcast because experts doubted the use of such early results.

IBM had been in the punched card and office automation business but didn't start building computers until 1950. The first IBM computer, the IBM 701, shipped in 1952, and eventually 19 units were sold. In the early 1950s, many people were pessimistic about the future of computers, believing that the market and opportunities for these "highly specialized" machines were quite limited.

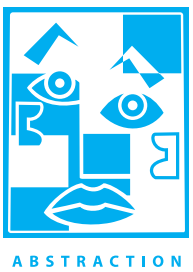
In 1964, after investing $5 billion, IBM made a bold move with the announcement of the System/360. An IBM spokesman said the following at the time:

We are not at all humble in this announcement. This is the most important product announcement that this corporation has ever made in its history. It's not a computer in any previous sense. It's not a product, but a line of products … that spans in performance from the very low part of the computer line to the very high.



FIGURE e1.12.3 IBM System/360 computers: models 40, 50, 65, and 75 were all introduced in 1964. These four models varied in cost and performance by a factor of almost 10; it grows to 25 if we include models 20 and 30 (not shown). The clock rate, range of memory sizes, and approximate price for only the processor and memory of average size: (a) model 40, 1.6 MHz, 32 KB–256 KB, $225,000; (b) model 50, 2.0 MHz, 128 KB–256 KB, $550,000; (c) model 65, 5.0 MHz, 256 KB–1 MB, $1,200,000; and (d) model 75, 5.1 MHz, 256 KB–1 MB, $1,900,000. Adding I/O devices typically increased the price by factors of 1.8 to 3.5, with higher factors for cheaper models.

Moving the idea of the architecture **abstraction** into commercial reality, IBM announced six implementations of the System/360 architecture that varied in price and performance by a factor of 25. Figure e1.12.3 shows four of these models. IBM bet its company on the success of a *computer family*, and IBM won. The System/360 and its successors dominated the large computer market.



About a year later, *Digital Equipment Corporation* (DEC) unveiled the PDP-8, the first commercial *minicomputer*. This small machine was a breakthrough in low-cost design, allowing DEC to offer a computer for under $20,000. Minicomputers were the forerunners of microprocessors, with Intel inventing the first microprocessor in 1971—the Intel 4004.

In 1963 came the announcement of the first *supercomputer*. This announcement came neither from the large companies nor even from the high-tech centers. Seymour Cray led the design of the Control Data Corporation CDC 6600 in Minnesota. This machine included many ideas that are beginning to be found in the latest microprocessors. Cray later left CDC to form Cray Research, Inc., in Wisconsin. In 1976, he announced the Cray-1 (Figure e1.12.4). This machine was simultaneously the fastest in the world, the most expensive, and the computer with the best cost/performance for scientific programs.



FIGURE e1.12.4 Cray-1, the first commercial vector supercomputer, announced in 1976. This machine had the unusual distinction of being both the fastest computer for scientific applications and the computer with the best price/performance for those applications. Viewed from the top, the computer looks like the letter *C*. Seymour Cray passed away in 1996 because of injuries sustained in an automobile accident. At the time of his death, this 70-year-old computer pioneer was working on his vision of the next generation of supercomputers. *(See www.cray.com for more details.)*

While Seymour Cray was creating the world's most expensive computer, other designers around the world were looking at using the microprocessor to create a computer so cheap that you could have it at home. There is no single fountainhead for the *personal computer*, but in 1977, the Apple IIe (Figure e1.12.5) from Steve Jobs and Steve Wozniak set standards for low cost, high volume, and high reliability that defined the personal computer industry.



FIGURE e1.12.5 The Apple IIc Plus. Designed by Steve Wozniak, the Apple IIc set standards of cost and reliability for the industry.

However, even with a 4-year head start, Apple's personal computers finished second in popularity. The IBM Personal Computer, announced in 1981, became the best-selling computer of any kind; its success gave Intel the most popular microprocessor and Microsoft the most popular operating system. Today, the most popular CD is the Microsoft operating system, even though it costs many times more than a music CD! Of course, over the more than 30 years that the IBM-compatible personal computer has existed, it has evolved greatly. In fact, the first personal computers had 16-bit processors and 64 kilobytes of **memory**, and a low-density, slow floppy disk was the only nonvolatile storage! Floppy disks were originally developed by IBM for loading diagnostic programs in mainframes, but were a major I/O device in personal computers for almost 20 years before the advent of CDs and networking made them obsolete as a method for exchanging data.



Of course, Intel microprocessors have also evolved since the first PC, which used a 16-bit processor with an 8-bit external interface! In Chapter 2, we write about the evolution of the Intel architecture.

The first personal computers were quite simple, with little or no graphics capability, no pointing devices, and primitive operating systems compared to those of today. The computer that inspired many of the architectural and software concepts that characterize the modern desktop machines was the Xerox Alto, shown in Figure e1.12.6. The Alto was created as an experimental prototype of a future computer; there were several hundred Altos built, including a significant



FIGURE e1.12.6 The Xerox Alto was the primary inspiration for the modern desktop computer. It included a mouse, a bit-mapped scheme, a Windows-based user interface, and a local network connection.

number that were donated to universities. Among the technologies incorporated in the Alto were:

- a bit-mapped graphics display integrated with a computer (earlier graphics displays acted as terminals, usually connected to larger computers)
- a mouse, which was invented earlier, but included on every Alto and used extensively in the user interface
- a local area network (LAN), which became the precursor to the Ethernet
- a user interface based on Windows and featuring a WYSIWYG (what you see is what you get) editor and interactive drawing programs

In addition, both file servers and print servers were developed and interfaced via the local area network, and connections between the local area network and the wide area ARPAnet produced the first versions of Internet-style networking. The Xerox Alto was incredibly influential and clearly affected the design of a wide variety of computers and software systems, including the Apple Macintosh, the IBM-compatible PC, MacOS and Windows, and Sun and other early workstations.

Measuring Performance

From the earliest days of computing, designers have specified performance goals—ENIAC was to be 1000 times faster than the Harvard Mark-I, and the IBM Stretch (7030) was to be 100 times faster than the fastest computer then in existence. What wasn't clear, though, was how this performance was to be measured.

The original measure of performance was the time required to perform an individual operation, such as addition. Since most instructions took the same execution time, the timing of one was the same as the others. As the execution times of instructions in a computer became more diverse, however, the time required for one operation was no longer useful for comparisons.

To consider these differences, an *instruction mix* was calculated by measuring the relative frequency of instructions in a computer across many programs. Multiplying the time for each instruction by its weight in the mix gave the user the *average instruction execution time*. (If measured in clock cycles, average instruction execution time is the same as average CPI.) Since instruction sets were similar, this was a more precise comparison than add times. From average instruction execution time, then, it was only a small step to MIPS. MIPS had the virtue of being easy to understand; hence, it grew in popularity.

The Quest for an Average Program

As processors were becoming more sophisticated and relied on memory hierarchies (the topic of Chapter 5) and pipelining (the topic of Chapter 4), a single execution time for each instruction no longer existed; neither execution time nor MIPS, therefore, could be calculated from the instruction mix and the manual.

Although it might seem obvious today that the right thing to do would have been to develop a set of real applications that could be used as standard benchmarks, this was a difficult task until relatively recent times. Variations in operating systems and language standards made it hard to create large programs that could be moved from computer to computer simply by recompiling.

Instead, the next step was benchmarking using synthetic programs. The Whetstone synthetic program was created by measuring scientific programs written in Algol-60 (see Curnow and Wichmann's [1976] description). This

program was converted to Fortran and was widely used to characterize scientific program performance. Whetstone performance is typically quoted in Whetstones per second—the number of executions of a single iteration of the Whetstone benchmark! Dhrystone is another synthetic benchmark that is still used in some embedded computing circles (see Weicker's [1984] description and methodology).

About the same time Whetstone was developed, the concept of *kernel benchmarks* gained popularity. Kernels are small, time-intensive pieces from real programs that are extracted and then used as benchmarks. This approach was developed primarily for benchmarking high-end computers, especially supercomputers. Livermore Loops and Linpack are the best-known examples. The Livermore Loops consist of a series of 21 small loop fragments. Linpack consists of a portion of a linear algebra subroutine package. Kernels are best used to isolate the performance of individual features of a computer and to explain the reasons for differences in the performance of real programs. Because scientific applications often use small pieces of code that execute for a long time, characterizing performance with kernels is most popular in this application class. Although kernels help illuminate performance, they frequently overstate the performance on real applications.

SPECulating about Performance

An important advance in performance evaluation was the formation of the System Performance Evaluation Cooperative (SPEC) group in 1988. SPEC comprises representatives of many computer companies—the founders being Apollo/Hewlett-Packard, DEC, MIPS, and Sun—who have agreed on a set of real programs and inputs that all will run. It is worth noting that SPEC couldn't have come into being before portable operating systems and the popularity of high-level languages. Now compilers, too, are accepted as a proper part of the performance of computer systems and must be measured in any evaluation.

History teaches us that while the SPEC effort may be useful with current computers, it will not meet the needs of the next generation without changing. In 1991, a throughput measure was added, based on running multiple versions of the benchmark. It is most useful for evaluating timeshared usage of a uniprocessor or a multiprocessor. Other system benchmarks that include OS-intensive and I/O-intensive activities have also been added. Another change was the decision to drop some benchmarks and add others. One result of the difficulty in finding benchmarks was that the initial version of the SPEC benchmarks (called SPEC89) contained six floating-point benchmarks but only four integer benchmarks. Calculating a single summary measurement using the geometric mean of execution times normalized to a VAX-11/780 meant that this measure favored computers with strong floating-point performance.

In 1992, a new benchmark set (called SPEC92) was introduced. It incorporated additional benchmarks, dropped matrix300, and provided separate means (SPEC INT and SPECFP) for integer and floating-point programs. In addition, the SPECbase measure, which disallows program-specific optimization flags, was added to provide users with a performance measurement that would more closely match what they might experience on their own programs. The SPECFP numbers show the largest increase versus the base SPECFP measurement, typically ranging from 15% to 30% higher.

In 1995, the benchmark set was once again updated, adding some new integer and floating-point benchmarks, as well as removing some benchmarks that suffered from flaws or had running times that had become too small given the factor of 20 or more performance improvement since the first SPEC release. SPEC95 also changed the base computer for normalization to a Sun SPARC Station 10/40, since operating versions of the original base computer were becoming difficult to find!

The most recent version of SPEC is SPEC2006. What is perhaps most surprising is that all floating-point programs in SPEC2006 are new, and for integer programs just two are from SPEC2000, one from SPEC95, none from SPEC92, and one from SPEC89. The sole survivor from SPEC89 is the gcc compiler.

SPEC has also added benchmark suites beyond the original suites targeted at CPU performance. In 2008, SPEC provided benchmark sets for graphics, high-performance scientific computing, object-oriented computing, file systems, Web servers and clients, Java, engineering CAD applications, and power.

The Growth of Embedded Computing

Embedded processors have been around for a very long time; in fact, the first minicomputers and the first microprocessors were originally developed for controlling functions in a laboratory or industrial application. For many years, the dominant use of embedded processors was for industrial control applications, and although this use continued to grow, the processors tended to be very cheap and the performance relatively low. For example, the best-selling processor in the world remains an 8-bit micro controller used in cars, some home appliances, and other simple applications.

The late 1980s and early 1990s saw the emergence of new opportunities for embedded processors, ranging from more advanced video games and set-top boxes to cell phones and personal digital assistants. The rapidly increasing number of information appliances and the growth of networking have driven dramatic surges in the number of embedded processors, as well as the performance requirements. To evaluate performance, the embedded community was inspired by SPEC to create the *Embedded Microprocessor Benchmark Consortium* (*EEMBC*). Started in 1997, it consists of a collection of kernels organized into suites that address different portions of the embedded industry. They announced the second generation of these benchmarks in 2007.

A Half-Century of Progress



Since 1951, there have been thousands of new computers using a wide range of technologies and having widely varying capabilities. Figure e1.12.7 summarizes the key characteristics of some machines mentioned in this section and shows the dramatic changes that have occurred in just over **50 years**. After adjusting for inflation, price/performance has improved by almost 100 billion in 55 years, or about 58% per year. Another way to say it is we've seen a factor of 10,000 improvement in cost and a factor of 10,000,000 improvement in performance.

Year	Name	Size (cu. ft.)	Power (watts)	Performance (adds/sec)	Memory (KB)	Price	Price/ performance vs. UNIVAC	Adjusted price (2007 $)	Adjusted price/ performance vs. UNIVAC
1951	UNIVAC I	1000	125,000	2000	48	$1,000,000	1	$7,670,724	1
1964	IBM S/360 model 50	60	10,000	500,000	64	$1,000,000	263	$6,018,798	319
1965	PDP-8	8	500	330,000	4	$16,000	10,855	$94,685	13,367
1976	Cray-1	58	60,000	166,000,000	32,000	$4,000,000	21,842	$13,509,798	47,127
1981	IBM PC	1	150	240,000	256	$3000	42,105	$6859	134,208
1991	HP 9000/ model 750	2	500	50,000,000	16,384	$7400	3,556,188	$11,807	16,241,889
1996	Intel PPro PC (200 MHz)	2	500	400,000,000	16,384	$4400	47,846,890	$6211	247,021,234
2003	Intel Pentium 4 PC (3.0 GHz)	2	500	6,000,000,000	262,144	$1600	1,875,000,000	$2009	11,451,750,000
2007	AMD Barcelona PC (2.5 GHz)	2	250	20,000,000,000	2,097,152	$800	12,500,000,000	$800	95,884,051,042

FIGURE e1.12.7 Characteristics of key commercial computers since 1950, in actual dollars and in 2007 dollars adjusted for inflation. The last row assumes we can fully utilize the potential performance of the four cores in Barcelona. In contrast to Figure e1.12.3, here the price of the IBM S/360 model 50 includes I/O devices. *(Source: The Computer History Museum and Producer Price Index for Industrial Commodities.)*

Readers interested in computer history should consult *Annals of the History of Computing*, a journal devoted to the history of computing. Several books describing the early days of computing have also appeared, many written by the pioneers including Goldstine [1972], Metropolis et al. [1980], and Wilkes [1985].

Further Reading

Barroso, L. and U. Hölzle [2007]. "The case for energy-proportional computing", *IEEE Computer* December.

A plea to change the nature of computer components so that they use much less power when lightly utilized.

Bell, C. G. [1996]. *Computer Pioneers and Pioneer Computers*, ACM and the Computer Museum, videotapes.

Two videotapes on the history of computing, produced by Gordon and Gwen Bell, including the following machines and their inventors: Harvard Mark-I, ENIAC, EDSAC, IAS machine, and many others.

Burks, A. W., H. H. Goldstine, and J. von Neumann [1946]. "Preliminary discussion of the logical design of an electronic computing instrument," Report to the U.S. Army Ordnance Department, p. 1; also appears in *Papers of John von Neumann*, W. Aspray and A. Burks (Eds.), MIT Press, Cambridge, MA, and Tomash Publishers, Los Angeles, 1987, 97–146.

A classic paper explaining computer hardware and software before the first stored-program computer was built. We quote extensively from it in Chapter 3. It simultaneously explained computers to the world and was a source of controversy because the first draft did not give credit to Eckert and Mauchly.

Campbell-Kelly, M. and W. Aspray [1996]. *Computer: A History of the Information Machine*, Basic Books, New York.

Two historians chronicle the dramatic story. The New York Times calls it well written and authoritative.

Ceruzzi, P. F. [1998]. *A History of Modern Computing*, MIT Press, Cambridge, MA.

Contains a good description of the later history of computing: the integrated circuit and its impact, personal computers, UNIX, and the Internet.

Curnow, H. J. and B. A. Wichmann [1976]. "A synthetic benchmark", *The Computer J.* 19(1):80.

Describes the first major synthetic benchmark, Whetstone, and how it was created.

Flemming, P. J. and J. J. Wallace [1986]. "How not to lie with statistics: The correct way to summarize benchmark results", *Comm. ACM* 29:3 (March), 218–21.

Describes some of the underlying principles in using different means to summarize performance results.

Goldstine, H. H. [1972]. *The Computer: From Pascal to von Neumann*, Princeton University Press, Princeton, NJ.

A personal view of computing by one of the pioneers who worked with von Neumann.

Hayes, B. [2007]. "Computing in a parallel universe", *American Scientist* Vol. 95(November–December):476–480.

An overview of the parallel computing challenge written for the layman.

Hennessy, J. L. and D. A. Patterson [2007]. *Chapter 1 of Computer Architecture: A Quantitative Approach*, fourth edition, Morgan Kaufmann Publishers, San Francisco.

Section 1.5 goes into more detail on power, Section 1.6 contains much more detail on the cost of integrated circuits and explains the reasons for the difference between price and cost, and Section 1.8 gives more details on evaluating performance.

Lampson, B. W. [1986]. "Personal distributed computing; The Alto and Ethernet software." In ACM Conference on the History of Personal Workstations (January).

Thacker, C. R. [1986]. "Personal distributed computing: The Alto and Ethernet hardware," In ACM Conference on the History of Personal Workstations (January).

These two papers describe the software and hardware of the landmark Alto.

Metropolis, N., J. Howlett, and G.-C. Rota (Eds.) [1980]. *A History of Computing in the Twentieth Century*, Academic Press, New York.

A collection of essays that describe the people, software, computers, and laboratories involved in the first experimental and commercial computers. Most of the authors were personally involved in the projects. An excellent bibliography of early reports concludes this interesting book.

Public Broadcasting System [1992]. *The Machine That Changed the World*, videotapes.

These five 1-hour programs include rare footage and interviews with pioneers of the computer industry.

Slater, R. [1987]. *Portraits in Silicon*, MIT Press, Cambridge, MA.

Short biographies of 31 computer pioneers.

Stern, N. [1980]. "Who invented the first electronic digital computer?" *Annals of the History of Computing* 2:4 (October), 375–76.

A historian's perspective on Atanasoff versus Eckert and Mauchly.

Weicker, R. P. [1984]. "Dhrystone: a synthetic systems programming benchmark", *Communications of the ACM* 27(10):1013–1030.

Description of a synthetic benchmarking program for systems code.

Wilkes, M. V. [1985]. *Memoirs of a Computer Pioneer*, MIT Press, Cambridge, MA.

A personal view of computing by one of the pioneers.

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 (classes 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 is faster: P1 or P2?

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, has 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 processors 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/cm^2. Assume a 20 cm diameter wafer has a cost of 15, contains 100 dies, and has 0.031 defects/cm^2.

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 mm^2.

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 highest 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 processors.

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 50×10^6 FP instructions, 110×10^6 INT instructions, 80×10^6 L/S instructions, and 16×10^6 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).

Answers to Check Yourself

§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.

2

Instructions: Language of the Computer

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

Charles V, Holy Roman Emperor
(1500–1558)

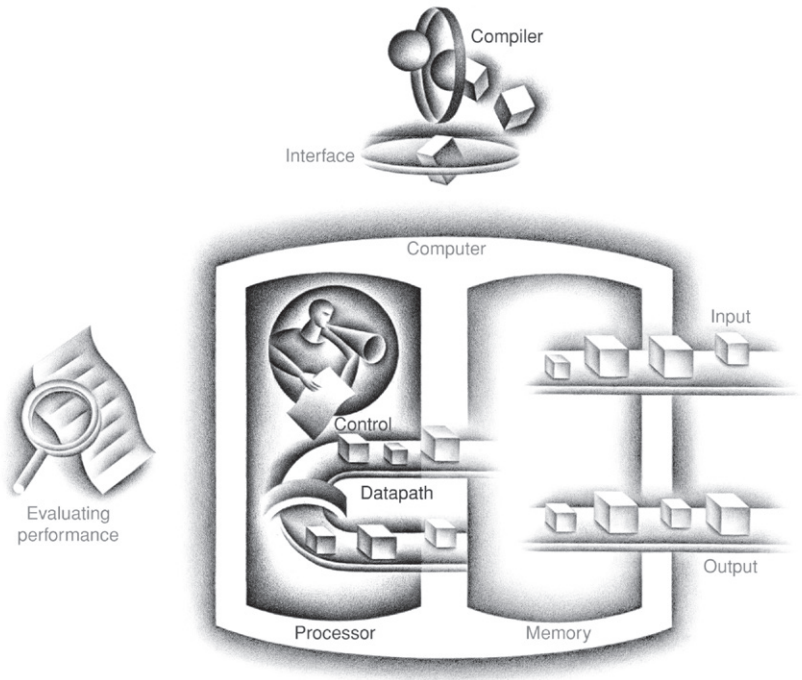
2.1 **Introduction** 62
2.2 **Operations of the Computer Hardware** 63
2.3 **Operands of the Computer Hardware** 67
2.4 **Signed and Unsigned Numbers** 74
2.5 **Representing Instructions in the Computer** 81
2.6 **Logical Operations** 89
2.7 **Instructions for Making Decisions** 92

Computer Organization and Design. DOI: http://dx.doi.org/10.1016/B978-0-12-812275-4.00002-6

© 2018 Elsevier Inc. All rights reserved.

2.8 **Supporting Procedures in Computer Hardware** 98
2.9 **Communicating with People** 108
2.10 **RISC-V Addressing for Wide Immediates and Addresses** 113
2.11 **Parallelism and Instructions: Synchronization** 121
2.12 **Translating and Starting a Program** 124
2.13 **A C Sort Example to Put it All Together** 133
2.14 **Arrays versus Pointers** 141
2.15 **Advanced Material: Compiling C and Interpreting Java** 144
2.16 **Real Stuff: MIPS Instructions** 145
2.17 **Real Stuff: x86 Instructions** 146
2.18 **Real Stuff: The Rest of the RISC-V Instruction Set** 155
2.19 **Fallacies and Pitfalls** 157
2.20 **Concluding Remarks** 159
2.21 **Historical Perspective and Further Reading** 162
2.22 **Exercises** 162

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 actual 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 independent languages. Hence, once you learn one, it is easy to pick up others.

The chosen instruction set is RISC-V, which was originally developed at UC Berkeley starting in 2010.

To demonstrate how easy it is to pick up other instruction sets, we will also take a quick look at two other popular instruction sets.

1. MIPS is an elegant example of the instruction sets designed since the 1980s. In several respects, RISC-V follows a similar design.
2. The Intel x86 originated in the 1970s, but still today powers both the PC and the Cloud of the post-PC era.

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.*
>
> Burks, Goldstine, and von Neumann, 1947

The "simplicity of the equipment" is as valuable a consideration for today's computers as it was for those of the 1950s. The goal of this chapter is to teach an instruction set that follows this advice, showing both how it is represented in

hardware and the relationship between high-level programming languages and this more primitive one. Our examples are in the C programming language; **Section 2.15** shows how these would change for an object-oriented language like Java.

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

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

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.

2.2 Operations of the Computer Hardware

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

Burks, Goldstine, and von Neumann, 1947

Every computer must be able to perform arithmetic. The RISC-V 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 RISC-V 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 double slashes (`//`) 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 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

RISC-V operands

Name	Example	Comments
32 registers	x0-x31	Fast locations for data. In RISC-V, data must be in registers to perform arithmetic. Register x0 always equals 0.
2^{61} memory words	Memory[0], Memory[8], ..., Memory[18,446,744,073,709,551,608]	Accessed only by data transfer instructions. RISC-V uses byte addresses, so sequential doubleword accesses differ by 8. Memory holds data structures, arrays, and spilled registers.

RISC-V assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	add x5, x6, x7	x5 = x6 + x7	Three register operands; add
	Subtract	sub x5, x6, x7	x5 = x6 - x7	Three register operands; subtract
	Add immediate	addi x5, x6, 20	x5 = x6 + 20	Used to add constants
Data transfer	Load doubleword	ld x5, 40(x6)	x5 = Memory[x6 + 40]	Doubleword from memory to register
	Store doubleword	sd x5, 40(x6)	Memory[x6 + 40] = x5	Doubleword from register to memory
	Load word	lw x5, 40(x6)	x5 = Memory[x6 + 40]	Word from memory to register
	Load word, unsigned	lwu x5, 40(x6)	x5 = Memory[x6 + 40]	Unsigned word from memory to register
	Store word	sw x5, 40(x6)	Memory[x6 + 40] = x5	Word from register to memory
	Load halfword	lh x5, 40(x6)	x5 = Memory[x6 + 40]	Halfword from memory to register
	Load halfword, unsigned	lhu x5, 40(x6)	x5 = Memory[x6 + 40]	Unsigned halfword from memory to register
	Store halfword	sh x5, 40(x6)	Memory[x6 + 40] = x5	Halfword from register to memory
	Load byte	lb x5, 40(x6)	x5 = Memory[x6 + 40]	Byte from memory to register
	Load byte, unsigned	lbu x5, 40(x6)	x5 = Memory[x6 + 40]	Byte unsigned from memory to register
	Store byte	sb x5, 40(x6)	Memory[x6 + 40] = x5	Byte from register to memory
	Load reserved	lr.d x5, (x6)	x5 = Memory[x6]	Load; 1st half of atomic swap
	Store conditional	sc.d x7, x5, (x6)	Memory[x6] = x5; x7 = 0/1	Store; 2nd half of atomic swap
	Load upper immediate	lui x5, 0x12345	x5 = 0x12345000	Loads 20-bit constant shifted left 12 bits
Logical	And	and x5, x6, x7	x5 = x6 & x7	Three reg. operands; bit-by-bit AND
	Inclusive or	or x5, x6, x8	x5 = x6 | x8	Three reg. operands; bit-by-bit OR
	Exclusive or	xor x5, x6, x9	x5 = x6 ^ x9	Three reg. operands; bit-by-bit XOR
	And immediate	andi x5, x6, 20	x5 = x6 & 20	Bit-by-bit AND reg. with constant
	Inclusive or immediate	ori x5, x6, 20	x5 = x6 | 20	Bit-by-bit OR reg. with constant
	Exclusive or immediate	xori x5, x6, 20	x5 = x6 ^ 20	Bit-by-bit XOR reg. with constant
Shift	Shift left logical	sll x5, x6, x7	x5 = x6 << x7	Shift left by register
	Shift right logical	srl x5, x6, x7	x5 = x6 >> x7	Shift right by register
	Shift right arithmetic	sra x5, x6, x7	x5 = x6 >> x7	Arithmetic shift right by register
	Shift left logical immediate	slli x5, x6, 3	x5 = x6 << 3	Shift left by immediate
	Shift right logical immediate	srli x5, x6, 3	x5 = x6 >> 3	Shift right by immediate
	Shift right arithmetic immediate	srai x5, x6, 3	x5 = x6 >> 3	Arithmetic shift right by immediate

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

Conditional branch	Branch if equal	`beq x5, x6, 100`	`if (x5 == x6) go to PC+100`	PC-relative branch if registers equal
	Branch if not equal	`bne x5, x6, 100`	`if (x5 != x6) go to PC+100`	PC-relative branch if registers not equal
	Branch if less than	`blt x5, x6, 100`	`if (x5 < x6) go to PC+100`	PC-relative branch if registers less
	Branch if greater or equal	`bge x5, x6, 100`	`if (x5 >= x6) go to PC+100`	PC-relative branch if registers greater or equal
	Branch if less, unsigned	`bltu x5, x6, 100`	`if (x5 < x6) go to PC+100`	PC-relative branch if registers less, unsigned
	Branch if greater or equal, unsigned	`bgeu x5, x6, 100`	`if (x5 >= x6) go to PC+100`	PC-relative branch if registers greater or equal, unsigned
Unconditional branch	Jump and link	`jal x1, 100`	`x1 = PC+4; go to PC+100`	PC-relative procedure call
	Jump and link register	`jalr x1, 100(x5)`	`x1 = PC+4; go to x5+100`	Procedure return; indirect call

FIGURE 2.1 (Continued).

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 RISC-V

EXAMPLE

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

```
a = b + c;
d = a – e;
```

The *compiler* translates from C to RISC-V assembly language instructions. Show the RISC-V code produced by a compiler.

ANSWER

A RISC-V 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 RISC-V assembly language instructions:

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

EXAMPLE

Compiling a Complex C Assignment into RISC-V

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

```
f = (g + h) – (i + j);
```

What might a C compiler produce?

ANSWER

The compiler must break this statement into several assembly instructions, since only one operation is performed per RISC-V instruction. The first RISC-V 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. RISC-V 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 RISC-V instruction set. To get performance close to the equivalent C program, Java systems today typically compile Java bytecodes into the native instruction sets like RISC-V. 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 RISC-V architecture is 64 bits; groups of 64 bits occur so frequently that they are given the name **doubleword** in the RISC-V architecture. (Another popular size is a group of 32 bits, called a **word** in the RISC-V architecture.)

doubleword Another natural unit of access in a computer, usually a group of 64 bits; corresponds to the size of a register in the RISC-V architecture.

word A natural unit of access in a computer, usually a group of 32 bits.

One major difference between the variables of a programming language and registers is the limited number of registers, typically 32 on current computers, like RISC-V. (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 RISC-V language, in this section we have added the restriction that the three operands of RISC-V arithmetic instructions must each be chosen from one of the 32 64-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. Even so, 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 that 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 RISC-V convention is `x` followed by the number of the register, except for a few register names that we will cover later.

EXAMPLE

Compiling a C Assignment Using Registers

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

```
f = (g + h) – (i + j);
```

The variables `f`, `g`, `h`, `i`, and `j` are assigned to the registers `x19`, `x20`, `x21`, `x22`, and `x23`, respectively. What is the compiled RISC-V 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, `x5` and `x6`, which correspond to the temporary variables above:

```
add x5, x20, x21  // register x5 contains g + h
add x6, x22, x23  // register x6 contains i + j
sub x19, x5, x6   // f gets x5 – x6, 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 composite 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.

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

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

As explained above, arithmetic operations occur only on registers in RISC-V instructions; thus, RISC-V must include instructions that transfer data between memory and registers. Such instructions are called **data transfer instructions**. To access a word or doubleword 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.

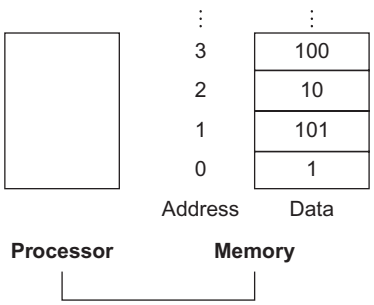


FIGURE 2.2 Memory addresses and contents of memory at those locations. If these elements were doublewords, these addresses would be incorrect, since RISC-V actually uses byte addressing, with each doubleword representing 8 bytes. Figure 2.3 shows the correct memory addressing for sequential doubleword 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 register and a constant 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 real RISC-V name for this instruction is `ld`, standing for *load doubleword*.

Compiling an Assignment When an Operand Is in Memory

EXAMPLE

Let's assume that `A` is an array of 100 doublewords and that the compiler has associated the variables `g` and `h` with the registers `x20` and `x21` as before. Let's also assume that the starting address, or *base address*, of the array is in `x22`. Compile this C assignment statement:

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

ANSWER

Although there is a single operation in this assignment statement, one of the operands is in memory, so we must first transfer `A[8]` to a register. The address of this array element is the sum of the base of the array `A`, found in register `x22`, 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

```
ld   x9, 8(x22) // Temporary reg x9 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 `x9` (which equals `A[8]`) since it is in a register. The instruction must add `h` (contained in `x21`) to `A[8]` (contained in `x9`) and put the sum in the register corresponding to `g` (associated with `x20`):

```
add  x20, x21, x9 // g = h + A[8]
```

The register added to form the address (`x22`) is called the *base register*, and the constant in a data transfer instruction (8) is called the *offset*.

Hardware/ Software Interface

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

Since 8-bit *bytes* are useful in many programs, virtually all architectures today address individual bytes. Therefore, the address of a doubleword matches the address of one of the 8 bytes within the doubleword, and addresses of sequential

doublewords differ by 8. For example, Figure 2.3 shows the actual RISC-V addresses for the doublewords in Figure 2.2; the byte address of the third doubleword is 16.

Computers divide into those that use the address of the leftmost or "big end" byte as the doubleword address versus those that use the rightmost or "little end" byte. RISC-V belongs to the latter camp, referred to as *little-endian*. Since the order matters only if you access the identical data both as a doubleword and as eight individual bytes, few need to be aware of the "endianness."

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* `x22` *must be 8 × 8, or 64*, so that the load address will select `A[8]` and not `A[8/8]`. (See the related *Pitfall* on page 159 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 the base register, and finally the offset to select the array element. Once again, the RISC-V address is specified in part by a constant and in part by the contents of a register. The actual RISC-V name is `sd`, standing for *store doubleword*.

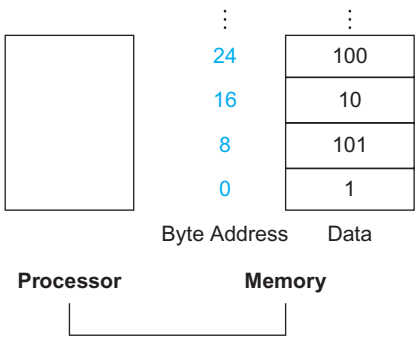


FIGURE 2.3 Actual RISC-V memory addresses and contents of memory for those doublewords. The changed addresses are highlighted to contrast with Figure 2.2. Since RISC-V addresses each byte, doubleword addresses are multiples of 8: there are 8 bytes in a doubleword.

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

Elaboration: In many architectures, words must start at addresses that are multiples of 4 and doublewords must start at addresses that are multiples of 8. This requirement is called an **alignment restriction**. (Chapter 4 suggests why alignment leads to faster data transfers.) RISC-V and Intel x86 do *not* have alignment restrictions, but MIPS does.

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

EXAMPLE

Assume variable `h` is associated with register `x21` and the base address of the array `A` is in `x22`. What is the RISC-V assembly code for the C assignment statement below?

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

ANSWER

Although there is a single operation in the C statement, now two of the operands are in memory, so we need even more RISC-V 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 register instruction to select `A[8]`, and the `add` instruction places the sum in `x9`:

```
ld    x9, 64(x22)    // Temporary reg x9 gets A[8]
add   x9, x21, x9    // Temporary reg x9 gets h + A[8]
```

The final instruction stores the sum into `A[12]`, using 96 (8 × 12) as the offset and register `x22` as the base register.

```
sd    x9, 96(x22)    // Stores h + A[8] back into A[12]
```

Load doubleword and store doubleword are the instructions that copy doublewords between memory and registers in the RISC-V architecture. Some brands of computers use other instructions along with load and store to transfer data. An architecture with such alternatives is the Intel x86, described in Section 2.17.

Hardware/ Software Interface

Many programs have more variables than computers have registers. Consequently, the compiler tries to keep the most frequently used variables in registers and places the rest in memory, using loads and stores to move variables between registers and memory. The process of putting less frequently 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 suggestion is indeed the case; data accesses are faster if data are in registers instead of memory.

Moreover, data are more useful when in a register. A RISC-V arithmetic instruction can read two registers, operate on them, and write the result. A RISC-V 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 considerably faster to access and simpler to use. Accessing registers also uses much less energy than accessing memory. To achieve the highest performance and conserve energy, an instruction set architecture must have enough registers, and compilers must use registers efficiently.

Elaboration: Let's put the energy and performance of registers versus memory into perspective. Assuming 64-bit data, registers are roughly 200 times faster (0.25 vs. 50 nanoseconds) and are 10,000 times more energy efficient (0.1 vs. 1000 picoJoules) than DRAM in 2015. These large differences led to caches, which reduce the performance and energy penalties of going to memory (see Chapter 5).

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 RISC-V 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 `x22`, we could use the code

```
ld  x9, AddrConstant4(x3)       // x9 = constant 4
add x22, x22, x9                //  x22 = x22 + x9 (where x9 == 4)
```

assuming that `x3 + 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 `x22`, we just write

```
addi     x22, x22, 4      // x22 = x22 + 4
```

Constant operands occur frequently; indeed, `addi` is the most popular instruction in most RISC-V programs. By including constants inside arithmetic instructions, operations are much faster and use less energy than if constants were loaded from memory.



COMMON CASE FAST

The constant zero has another role, which is to simplify the instruction set by offering useful variations. For example, you can negate the value in a register by using the `sub` instruction with zero for the first operand. Hence, RISC-V dedicates register `x0` to be hard-wired to the value zero. Using frequency to justify the inclusions of constants is another example of the great idea from Chapter 1 of making the **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 RISC-V registers in this book are 64 bits wide, the RISC-V architects conceived multiple variants of the ISA. In addition to this variant, known as RV64, a variant named RV32 has 32-bit registers, whose reduced cost make RV32 better suited to very low-cost processors.

Elaboration: The RISC-V 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: The migration from 32-bit address computers to 64-bit address computers left compiler writers a choice of the size of data types in C. Clearly, pointers should be 64 bits, but what about integers? Moreover, C has the data types `int`, `long int`, and `long long int`. The problems come from converting from one data type to another and having an unexpected overflow in C code that is not fully standard compliant, which unfortunately is not rare code. The table below shows the two popular options:

Operating System	pointers	int	long int	long long int
Microsoft Windows	64 bits	32 bits	32 bits	64 bits
Linux, Most Unix	64 bits	32 bits	64 bits	64 bits

While each compiler could have different choices, generally the compilers associated with each operating system make the same decision. To keep the examples simple, in this book we'll assume pointers are all 64 bits and declare all C integers as `long long int` to keep them the same size. We also follow C99 standard and declare variables used as indexes to arrays to be `size_t`, which guarantees they are the right size no matter how big the array. They are typically declared the same as `long int`.

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.)

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

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 doubleword: simply use the power of the base for that bit. We subscript decimal numbers with *ten* and binary numbers with *two*. For example,

$$1011_{two}$$

represents

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

We number the bits 0, 1, 2, 3, … from *right to left* in a doubleword. The drawing below shows the numbering of bits within a RISC-V doubleword and the placement of the number 1011_{two}, (which we must unfortunately split in half to fit on the page of the book):

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
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

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

(64 bits wide, split into two 32-bit rows)

least significant bit The rightmost bit in an RISC-V doubleword.

most significant bit The leftmost bit in an RISC-V doubleword.

Since doublewords 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 63).

The RISC-V doubleword is 64 bits long, so we can represent 2^{64} different 64-bit patterns. It is natural to let these combinations represent the numbers from 0 to $2^{64}-1$ ($18{,}446{,}774{,}073{,}709{,}551{,}615_{ten}$):

$$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000_{two} = 0_{ten}$$
$$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001_{two} = 1_{ten}$$
$$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000010_{two} = 2_{ten}$$
$$\ldots \qquad\qquad \ldots$$
$$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111101_{two} = 18{,}446{,}774{,}073{,}709{,}551{,}613_{ten}$$
$$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_{two} = 18{,}446{,}744{,}073{,}709{,}551{,}614_{ten}$$
$$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111_{two} = 18{,}446{,}744{,}073{,}709{,}551{,}615_{ten}$$

That is, 64-bit binary numbers can be represented in terms of the bit value times a power of 2 (here x_i means the ith bit of x):

$$(x_{63} \times 2^{63}) + (x_{62} \times 2^{62}) + (x_{61} \times 2^{61}) + \cdots + (x_1 \times 2^1) + (x_0 \times 2^0)$$

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. Because 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:

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000_{two} = 0_{ten}$

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001_{two} = 1_{ten}$

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000010_{two} = 2_{ten}$

. . . . . .

$01111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111101_{two} = 9{,}223{,}372{,}036{,}854{,}775{,}805_{ten}$

$01111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_{two} = 9{,}223{,}372{,}036{,}854{,}775{,}806_{ten}$

$01111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111_{two} = 9{,}223{,}372{,}036{,}854{,}775{,}807_{ten}$

$10000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000_{two} = -9{,}223{,}372{,}036{,}854{,}775{,}808_{ten}$

$10000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001_{two} = -9{,}223{,}372{,}036{,}854{,}775{,}807_{ten}$

$10000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000010_{two} = -9{,}223{,}372{,}036{,}854{,}775{,}806_{ten}$

… . . .

$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111101_{two} = -3_{ten}$

$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_{two} = -2_{ten}$

$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111_{two} = -1_{ten}$

The positive half of the numbers, from 0 to $9{,}223{,}372{,}036{,}854{,}775{,}807_{ten}$ ($2^{63}-1$), use the same representation as before. The following bit pattern ($1000\ \ldots\ 0000_{two}$) represents the most negative number $-9{,}223{,}372{,}036{,}854{,}775{,}808_{ten}$ (-2^{63}). It is followed by a declining set of negative numbers: $-9{,}223{,}372{,}036{,}854{,}775{,}807_{ten}$ ($1000\ \ldots\ 0001_{two}$) down to -1_{ten} ($1111\ \ldots\ 1111_{two}$).

Two's complement does have one negative number that has no corresponding positive number: $-9{,}223{,}372{,}036{,}854{,}775{,}808_{ten}$. 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. Thus, hardware needs to test only this bit to see if a number is positive or negative (with the number 0 is considered positive). This bit is often called the *sign bit*. By recognizing the role of the sign bit, we can represent positive and negative 64-bit numbers in terms of the bit value times a power of 2:

$$(x_{63} \times -2^{63}) + (x_{62} \times 2^{62}) + (x_{61} \times 2^{61}) + \cdots + (x_1 \times 2^1) + (x_0 \times 2^0)$$

The sign bit is multiplied by -2^{63}, 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 64-bit two's complement number?

$$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111100_{two}$$

ANSWER

Substituting the number's bit values into the formula above:

$$\begin{aligned}
&(1 \times -2^{63}) + (1 \times 2^{62}) + (1 \times 2^{61}) + \cdots + (1 \times 2^1) + (0 \times 2^1) + (0 \times 2^0) \\
&= -2^{63} + 2^{62} + 2^{61} + \cdots + 2^2 + 0 + 0 \\
&= -9,223,372,036,854,775,808_{ten} + 9,223,372,036,854,775,804_{ten} \\
&= -4_{ten}
\end{aligned}$$

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

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

Hardware/ Software Interface

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 64-bit doubleword into a 64-bit register, the point is moot; signed and unsigned loads are identical. RISC-V does offer two flavors of byte loads: *load byte unsigned* (`lbu`) treats the byte as an unsigned number and thus zero-extends to fill the leftmost bits of the register, while *load byte* (`lb`) works with signed integers. Since C programs almost always use bytes to represent characters rather than consider bytes as very short signed integers, `lbu` is used practically exclusively for byte loads.

Hardware/ Software Interface

Unlike the signed 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 `long long int` in the program) and the latter *unsigned integers* (`unsigned long long int`). Some C style guides even recommend declaring the former as `signed long long int` to keep the distinction clear.

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

EXAMPLE

Negation Shortcut

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

ANSWER

2_{ten} = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000010$_{two}$

Negating this number by inverting the bits and adding one,

$$
\begin{array}{l}
\quad 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111101_{two} \\
+ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 1_{two} \\
\hline
= 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_{two} \\
= -2_{ten}
\end{array}
$$

Going the other direction,

$$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_{two}$$

is first inverted and then incremented:

$$
\begin{array}{l}
\quad 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001_{two} \\
+ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 1_{two} \\
\hline
= 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000010_{two} \\
= 2_{ten}
\end{array}
$$

Our next shortcut tells us how to convert a binary number represented in n bits to a number represented with more than n bits. 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 doubleword. This shortcut is commonly called *sign extension*.

EXAMPLE

Sign Extension Shortcut

Convert 16-bit binary versions of 2_{ten} and -2_{ten} to 64-bit binary numbers.

ANSWER

The 16-bit binary version of the number 2 is

$$00000000\ 00000010_{two} = 2_{ten}$$

It is converted to a 64-bit number by making 48 copies of the value in the most significant bit (0) and placing that in the left of the doubleword. The right part gets the old value:

$$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000010_{two} = 2_{ten}$$

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

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

becomes

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

Creating a 64-bit version of the negative number means copying the sign bit 48 times and placing it on the left:

$$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_{two} = -2_{ten}$$

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

Summary

The main point of this section is that we need to represent both positive and negative integers within a computer, 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 data size, binary and hexadecimal (see Figure 2.4) bit strings can encode the sign; therefore, we do not normally use "+" or "−" with binary or hexadecimal notation.

Check Yourself

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

$$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111000_{two}$$

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

Elaboration: Two's complement gets its name from the rule that the unsigned sum of an n-bit number and its 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 $\overline{x}$. 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 … 000_{two} and the most positive value by 11 … 11_{two}, with 0 typically having the value 10 … 00_{two}. This representation 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 … 000_{two} and the most positive value by 01 … 11_{two}, leaving an equal number of negatives and positives but ending up with two zeros, one positive (00 … 00_{two}) and one negative (11 … 11_{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 … 000_{two} and the most positive value by 11 … 11_{two}, with 0 typically having the value 10 … 00_{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. The 32 registers of RISC-V are just referred to by their number, from 0 to 31.

EXAMPLE

Translating a RISC-V Assembly Instruction into a Machine Instruction

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

```
add x9, x20, x21
```

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

ANSWER

The decimal representation is

0	21	20	0	9	51

Each of these segments of an instruction is called a *field*. The first, fourth, and sixth fields (containing 0, 0, and 51 in this case) collectively tell the RISC-V computer that this instruction performs addition. The second field gives the number of the register that is the second source operand of the addition operation (21 for `x21`), and the third field gives the other source operand for the addition (20 for `x20`). The fifth field contains the number of the register that is to receive the sum (9 for `x9`). Thus, this instruction adds register `x20` to register `x21` and places the sum in register `x9`.

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

0000000	10101	10100	000	01001	0110011
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

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

This layout of the instruction is called the **instruction format**. As you can see from counting the number of bits, this RISC-V instruction takes exactly 32 bits—a word, or one half of a doubleword. In keeping with our design principle that simplicity favors regularity, RISC-V instructions are all 32 bits long.

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

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

hexadecimal Numbers in base 16.

It would appear that you would now be reading and writing long, tiresome 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. As base 16 is a power of 2, we can trivially convert by replacing each group of four binary digits by a single hexadecimal digit, and vice versa. Figure 2.4 converts between hexadecimal and binary.

Hexadecimal	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

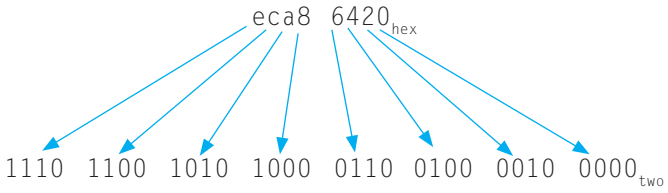
EXAMPLE

Convert the following 8-digit hexadecimal and 32-bit binary numbers into the other base:

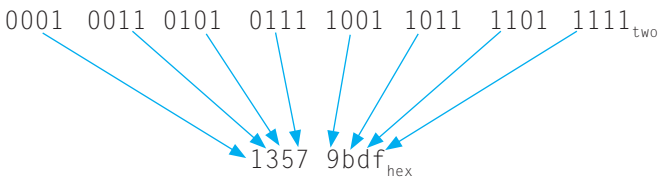
$$\text{eca8 6420}_{\text{hex}}$$
$$0001\ 0011\ 0101\ 0111\ 1001\ 1011\ 1101\ 1111_{\text{two}}$$

ANSWER

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



And then the other direction:



RISC-V Fields

RISC-V fields are given names to make them easier to discuss:

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Here is the meaning of each name of the fields in RISC-V instructions:

- ***opcode***: Basic operation of the instruction, and this abbreviation is its traditional name.
- *rd:* The register destination operand. It gets the result of the operation.
- *funct3:* An additional opcode field.
- *rs1:* The first register source operand.
- *rs2:* The second register source operand.
- *funct7:* An additional opcode field.

opcode The field that denotes the operation and format of an instruction.

A problem occurs when an instruction needs longer fields than those shown above. For example, the load register 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 largest constant within the load register instruction would be limited to only 2^5-1 or 31. This constant is used to select elements from arrays or data structures, and it often needs to be much larger than 31. 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 conflict leads us to the final hardware design principle:

Design Principle 3: Good design demands good compromises.

The compromise chosen by the RISC-V designers is to keep all instructions the same length, thereby requiring distinct instruction formats for different kinds of instructions. For example, the format above is called *R-type* (for register). A second type of instruction format is *I-type* and is used by arithmetic operands with one constant operand, including `addi`, and by load instructions. The fields of the I-type format are

immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

The 12-bit immediate is interpreted as a two's complement value, so it can represent integers from -2^{11} to $2^{11}-1$. When the I-type format is used for load instructions, the immediate represents a byte offset, so the load doubleword instruction can refer to any doubleword within a region of $\pm 2^{11}$ or 2048 bytes ($\pm 2^8$ or 256 doublewords) of the base address in the base register rd. We see that more than 32 registers would be difficult in this format, as the rd and rs1 fields would each need another bit, making it harder to fit everything in one word.

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

```
ld x9, 64(x22) // Temporary reg x9 gets A[8]
```

Here, 22 (for `x22`) is placed in the rs1 field, 64 is placed in the immediate field, and 9 (for `x9`) is placed in the rd field. We also need a format for the store doubleword instruction, `sd`, which needs two source registers (for the base address and the store data) and an immediate for the address offset. The fields of the S-type format are

immediate[11:5]	rs2	rs1	funct3	immediate[4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

The 12-bit immediate in the S-type format is split into two fields, which supply the lower 5 bits and upper 7 bits. The RISC-V architects chose this design because it keeps the rs1 and rs2 fields in the same place in all instruction formats. Keeping the instruction formats as similar as possible reduces hardware complexity. Similarly,

the opcode and funct3 fields are the same size in all locations, and they are always in the same place.

In case you were wondering, the formats are distinguished by the values in the opcode field: each format is assigned a distinct set of opcode values in the first field (opcode) so that the hardware knows how to treat the rest of the instruction. Figure 2.5 shows the numbers used in each field for the RISC-V instructions covered so far.

Instruction	Format	funct7	rs2	rs1	funct3	rd	opcode
`add` (add)	R	0000000	reg	reg	000	reg	0110011
`sub` (sub)	R	0100000	reg	reg	000	reg	0110011
Instruction	**Format**	**immediate**		**rs1**	**funct3**	**rd**	**opcode**
`addi` (add immediate)	I	constant		reg	000	reg	0010011
`ld` (load doubleword)	I	address		reg	011	reg	0000011
Instruction	**Format**	**immed -iate**	**rs2**	**rs1**	**funct3**	**immed -iate**	**opcode**
`sd` (store doubleword)	S	address	reg	reg	011	address	0100011

FIGURE 2.5 RISC-V instruction encoding. In the table above, "reg" means a register number between 0 and 31 and "address" means a 12-bit address or constant. The funct3 and funct7 fields act as additional opcode fields.

EXAMPLE

Translating RISC-V 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 `x10` has the base of the array `A` and `x21` corresponds to `h`, the assignment statement

```
A[30] = h + A[30] + 1;
```

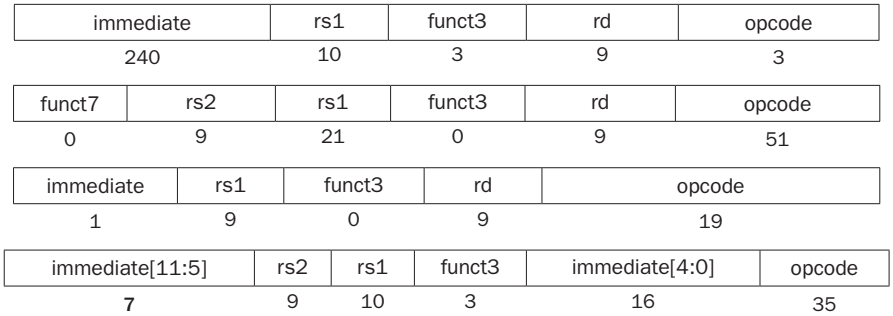
is compiled into

```
ld   x9, 240(x10)    // Temporary reg x9 gets A[30]
add  x9, x21, x9     // Temporary reg x9 gets h+A[30]
addi x9, x9, 1       // Temporary reg x9 gets h+A[30]+1
sd   x9, 240(x10)    // Stores h+A[30]+1 back into A[30]
```

ANSWER

What is the RISC-V machine language code for these three instructions?

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:



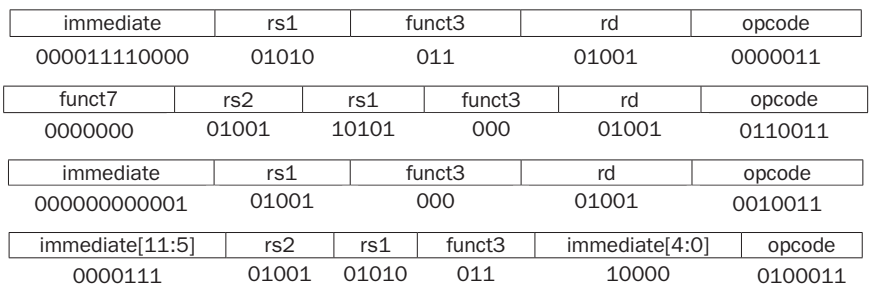
The `ld` instruction is identified by 3 (see Figure 2.5) in the opcode field and 3 in the funct3 field. The base register 10 is specified in the rs1 field, and the destination register 9 is specified in the rd field. The offset to select `A[30]` ($240 = 30 \times 8$) is found in the immediate field.

The `add` instruction that follows is specified with 51 in the opcode field, 0 in the funct3 field, and 0 in the funct7 field. The three register operands (9, 21, and 9) are found in the rd, rs1, and rs2 fields.

The subsequent `addi` instruction is specified with 19 in the opcode field and 0 in the funct3 field. The register operands (9 and 9) are found in the rd and rs1 fields, and the constant addend 1 is found in the immediate field.

The `sd` instruction is identified with 35 in the opcode field and 3 in the funct3 field. The register operands (9 and 10) are found in the rs2 and rs1 fields, respectively. The address offset 240 is split across the two immediate fields. Since the upper part of the immediate holds bits 5 and above, we can decompose the offset 240 by dividing by 2^5. The upper part of the immediate holds the quotient, 7, and the lower part holds the remainder, 16.

Since $240_{ten} = 0000\ 1111\ 0000_{two}$, the binary equivalent to the decimal form is:



Elaboration: RISC-V assembly language programmers aren't forced to use `addi` when working with constants. The programmer simply writes `add`, and the assembler generates the proper opcode and the proper instruction format depending on whether the operands are all registers (R-type) or if one is a constant (I-type). We use the explicit names in RISC-V for the different opcodes and formats as we think it is less confusing when introducing assembly language versus machine language.

Elaboration: Although RISC-V has both `add` and `sub` instructions, it does not have a `subi` counterpart to `addi`. This is because the immediate field represents a two's complement integer, so `addi` can be used to subtract constants.

Hardware/Software Interface

The desire to keep all instructions the same size conflicts 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.

Figure 2.6 summarizes the portions of RISC-V 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 RISC-V architecture.

R-type Instructions	funct7	rs2	rs1	funct3	rd	opcode	Example
`add` (add)	0000000	00011	00010	000	00001	0110011	add x1, x2, x3
`sub` (sub)	0100000	00011	00010	000	00001	0110011	sub x1, x2, x3
I-type Instructions	**immediate**		**rs1**	**funct3**	**rd**	**opcode**	**Example**
`addi` (add immediate)	001111101000		00010	000	00001	0010011	addi x1, x2, 1000
`ld` (load doubleword)	001111101000		00010	011	00001	0000011	ld x1, 1000 (x2)
S-type Instructions	**immed -iate**	**rs2**	**rs1**	**funct3**	**immed -iate**	**opcode**	**Example**
`sd` (store doubleword)	0011111	00001	00010	011	01000	0100011	sd x1, 1000(x2)

FIGURE 2.6 RISC-V architecture revealed through Section 2.5. The three RISC-V instruction formats so far are R, I, and S. The R-type format has two source register operand and one destination register operand. The I-type format replaces one source register operand with a 12-bit *immediate* field. The S-type format has two source operands and a 12-bit immediate field, but no destination register operand. The S-type immediate field is split into two parts, with bits 11—5 in the leftmost field and bits 4—0 in the second-rightmost 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.

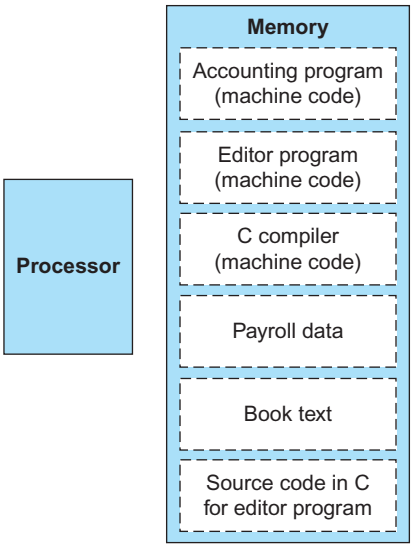


FIGURE 2.7 The stored-program concept. Stored programs allow a computer that performs accounting to become, in the blink of an eye, a computer that helps an author write a book. The switch happens simply by loading memory with programs and data and then telling the computer to begin executing at a given 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.

Check Yourself

What RISC-V instruction does this represent? Choose from one of the four options below.

funct7	rs2	rs1	funct3	rd	opcode
32	9	10	000	11	51

1. `sub x9, x10, x11`
2. `add x11, x9, x10`
3. `sub x11, x10, x9`
4. `sub x11, x9, x10`

2.6 Logical Operations

"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

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 RISC-V.

Logical operations	C operators	Java operators	RISC-V instructions
Shift left	<<	<<	`sll, slli`
Shift right	>>	>>>	`srl, srli`
Shift right arithmetic	>>	>>	`sra, srai`
Bit-by-bit AND	&	&	`and, andi`
Bit-by-bit OR	|	|	`or, ori`
Bit-by-bit XOR	^	^	`xor, xori`
Bit-by-bit NOT	~	~	`xori`

FIGURE 2.8 C and Java logical operators and their corresponding RISC-V instructions. One way to implement NOT is to use XOR with one operand being all ones (FFFF FFFF FFFF FFFF$_{hex}$).

The first class of such operations is called *shifts*. They move all the bits in a doubleword to the left or right, filling the emptied bits with 0s. For example, if register `x19` contained

$$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00001001_{two} = 9_{ten}$$

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

$$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 10010000_{two} = 144_{ten}$$

The dual of a shift left is a shift right. The actual names of the two RISC-V shift instructions are *shift left logical immediate* (`slli`) and *shift right logical immediate* (`srli`). The following instruction performs the operation above, if the original value was in register `x19` and the result should go in register `x11`:

```
slli x11, x19, 4 // reg x11 = reg x19 << 4 bits
```

These shift instructions use the I-type format. Since it isn't useful to shift a 64-bit register by more than 63 bits, only the lower 6 bits of the I-type format's 12-bit immediate are actually used. The remaining 6 bits are repurposed as an additional opcode field, funct6.

funct6	immediate	rs1	funct3	rd	opcode
0	4	19	1	11	19

The encoding of `slli` is 19 in the opcode field, rd contains 11, funct3 contains 1, rs1 contains 19, immediate contains 4, and funct6 contains 0.

Shift left logical provides a bonus benefit. Shifting left by i bits gives the identical 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 `slli` 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. RISC-V provides a third type of shift, *shift right arithmetic* (`srai`). This variant is similar to `srli`, except rather than filling the vacated bits on the left with zeros, it fills them with copies of the old sign bit. It also provides variants of all three shifts that take the shift amount from a register, rather than from an immediate: `sll`, `srl`, and `sra`.

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

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 `x11` contains

00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000$_{two}$

and register x10 contains

00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000$_{two}$

then, after executing the RISC-V instruction

```
and x9, x10, x11      // reg x9 = reg x10 & reg x11
```

the value of register x9 would be

00000000 00000000 00000000 00000000 00000000 00000000 00001100 00000000$_{two}$

As you can see, AND can apply a bit pattern to a set of bits to force 0s where there is a 0 in the bit pattern. Such a bit pattern in conjunction with AND is traditionally called a *mask*, since the mask "conceals" some bits.

To place a value into one of these seas of 0s, there is the dual to AND, called **OR**. It is a bit-by-bit operation that places a 1 in the result if *either* operand bit is a 1. To elaborate, if the registers x10 and x11 are unchanged from the preceding example, the result of the RISC-V instruction

```
or x9, x10, x11 // reg x9 = reg x10 | reg x11
```

is this value in register x9:

00000000 00000000 00000000 00000000 00000000 00000000 00111101 11000000$_{two}$

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 $\overline{x}$.

In keeping with the three-operand format, the designers of RISC-V decided to include the instruction **XOR** (exclusive OR) instead of NOT. Since exclusive OR creates a 0 when bits are the same and a 1 if they are different, the equivalent to NOT is an xor 111…111.

If the register x10 is unchanged from the preceding example and register x12 has the value 0, the result of the RISC-V instruction

```
xor x9, x10, x12 // reg x9 = reg x10 ^ reg x12
```

is this value in register x9:

00000000 00000000 00000000 00000000 00000000 00000000 00110001 11000000$_{two}$

Figure 2.8 above shows the relationship between the C and Java operators and the RISC-V instructions. Constants are useful in logical operations as well as in arithmetic operations, so RISC-V also provides the instructions *and immediate* (andi), *or immediate* (ori), and *exclusive or immediate* (xori).

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.

XOR A logical bit-by-bit operation with two operands that calculates the exclusive OR of the two operands. That is, it calculates a 1 only if the values are different in the two operands.

Elaboration: C allows *bit fields* or *fields* to be defined within doublewords, both allowing objects to be packed within a doubleword and to match an externally enforced interface such as an I/O device. All fields must fit within a single doubleword. Fields are unsigned integers that can be as short as 1 bit. C compilers insert and extract fields using logical instructions in RISC-V: `andi`, `ori`, `slli`, and `srli`.

Check Yourself

Which operations can isolate a field in a doubleword?

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

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

Burks, Goldstine, and von Neumann, 1947

2.7 Instructions for Making Decisions

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

```
beq rs1, rs2, L1
```

This instruction means go to the statement labeled `L1` if the value in register `rs1` equals the value in register rs2. The mnemonic `beq` stands for *branch if equal*. The second instruction is

```
bne rs1, rs2, L1
```

It means go to the statement labeled `L1` if the value in register rs1 does *not* equal the value in register rs2. The mnemonic `bne` stands for *branch if not equal*. These two instructions are traditionally called **conditional branches**.

conditional branch An instruction that tests a value and that allows for a subsequent transfer of control to a new address in the program based on the outcome of the test.

Compiling *if-then-else* into Conditional Branches

EXAMPLE

In the following code segment, `f`, `g`, `h`, `i`, and `j` are variables. If the five variables `f` through `j` correspond to the five registers `x19` through `x23`, what is the compiled RISC-V code for this C *if* statement?

```
if (i == j) f = g + h; else f = g – h;
```

ANSWER

Figure 2.9 shows a flowchart of what the RISC-V code should do. The first expression compares for equality between two variables in registers. It would seem that we would want to branch if I and j are equal (`beq`). In general, the code will be more efficient if we test for the opposite condition to branch over the code that branches if the values are *not* equal (`bne`). Here is the code:

```
bne x22, x23, 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 x19, x20, x21       // 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. One way to express an unconditional branch in RISC-V is to use a conditional branch whose condition is always true:

```
beq x0, x0, Exit        // if 0 == 0, 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 x19, x20, x21    // 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).

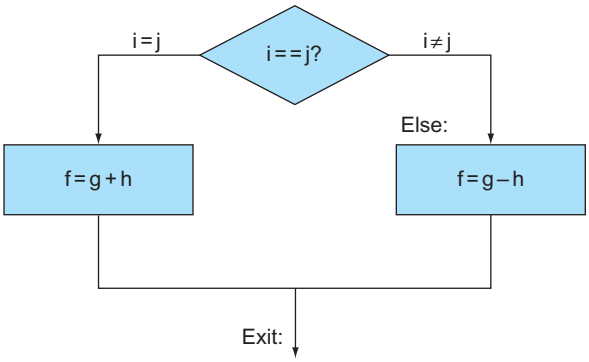


FIGURE 2.9 Illustration of the options in the *if* statement above. The left box corresponds to the *then* part of the *if* statement, and the right box corresponds to the *else* part.

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 `x22` and `x24` and the base of the array `save` is in `x25`. What is the RISC-V assembly code corresponding to this C code?

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 8 due to the byte addressing issue. Fortunately, we can use shift left, since shifting left by 3 bits multiplies by 2^3 or 8 (see page 90 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: slli x10, x22, 3      // Temp reg x10 = i * 8
```

To get the address of `save[i]`, we need to add `x10` and the base of save in `x25`:

```
add x10, x10, x25        // x10 = address of save[i]
```

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

```
ld x9, 0(x10)        // Temp reg x9 = save[i]
```

The next instruction performs the loop test, exiting if `save[i]` ≠ `k`:

```
bne x9, x24, Exit      // go to Exit if save[i] ≠ k
```

The following instruction adds 1 to `i`:

```
addi x22, x22, 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:

```
beq x0, x0, Loop        // go to Loop

Exit:
```

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

Hardware/ Software Interface

Such sequences of instructions that end in a branch are so fundamental to compiling that they are given their own buzzword: a **basic block** is a sequence of instructions without branches, except possibly at the end, and without branch targets or branch labels, except possibly at the beginning. One of the first early phases of compilation is breaking the program into basic blocks.

basic block A sequence of instructions without branches (except possibly at the end) and without branch targets or branch labels (except possibly at the beginning).

The test for equality or inequality is probably the most popular test, but there are many other relationships between two numbers. For example, a *for* loop may want to test to see if the index variable is less than 0. The full set of comparisons is less than (<), less than or equal (≤), greater than (>), greater than or equal (≥), equal (=), and not equal (≠).

Comparison of bit patterns must also 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.) RISC-V provides instructions that handle both cases. These instructions have the same form as `beq` and `bne`, but perform different comparisons. The branch if less than (`blt`) instruction compares the values in registers rs1 and rs2 and takes the branch if the value in rs1 is smaller, when they are treated as two's complement numbers. Branch if greater than or equal (`bge`) takes the branch in the opposite case, that is, if the value in rs1 is at least the value in rs2. Branch if less than, unsigned (`bltu`) takes the branch if the value in rs1 is smaller than the value in rs2 when the values are treated as unsigned numbers. Finally, branch if greater than or equal, unsigned (`bgeu`) takes the branch in the opposite case.

An alternative to providing these additional branch instructions is to set a register based upon the result of the comparison, then branch on the value in that temporary register with the `beq` or `bne` instructions. This approach, used by the MIPS instruction set, can make the processor datapath slightly simpler, but it takes more instructions to express a program.

Yet another alternative, used by ARM's instruction sets, is to keep extra bits that record what occurred during an instruction. These additional bits, called *condition codes* or *flags*, indicate, for example, if the result of an arithmetic operation was negative, or zero, or resulted in overflow.

Conditional branches then use combinations of these condition codes to perform the desired test.

One downside to condition codes is that if many instructions always set them, it will create dependencies that will make it difficult for pipelined execution (see Chapter 4).

Bounds Check Shortcut

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 $x < y$ checks if x is negative as well as if x is less than y.

EXAMPLE

Use this shortcut to reduce an index-out-of-bounds check: branch to `IndexOutOfBounds` if `x20` ≥ `x11` or if `x20` is negative.

ANSWER

The checking code just uses unsigned greater than or equal to do both checks:

```
bgeu x20, x11, IndexOutOfBounds // if x20 >= x11 or
   x20 < 0, goto IndexOutOfBounds
```

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 **branch address table** or **branch table**, and the program needs only to index into the table and then branch to the appropriate sequence. The branch table is therefore just an array of double-words containing addresses that correspond to labels in the code. The program loads the appropriate entry from the branch table into a register. It then needs to branch using the address in the register. To support such situations, computers like RISC-V include an *indirect jump* instruction, which performs an unconditional branch to the address specified in a register. In RISC-V, the jump-and-link register instruction (`jalr`) serves this purpose. We'll see an even more popular use of this versatile instruction in the next section.

branch address table Also called **branch 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

I. C has many statements for decisions and loops, while RISC-V has few. Which of the following does or does not explain this imbalance? Why?

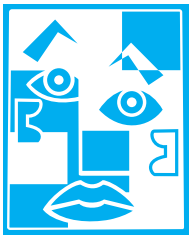
1. More decision statements make code easier to read and understand.
2. Fewer decision statements simplify the task of the underlying layer that is responsible for execution.
3. More decision statements mean fewer lines of code, which generally reduces coding time.
4. More decision statements mean fewer 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 RISC-V doesn't?

1. Logical operations AND and ORR implement & and |, while conditional branches implement && and ||.
2. The previous statement has it backwards: && and || correspond to logical operations, while & and | map to conditional branches.
3. They are redundant and mean the same thing: && and || are simply inherited from the programming language B, the predecessor of C.

2.8 Supporting Procedures in Computer Hardware

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



ABSTRACTION

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 the spymaster.

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. RISC-V software follows the following convention for procedure calling in allocating its 32 registers:

- `x10-x17`: eight parameter registers in which to pass parameters or return values.
- `x1`: one return address register to return to the point of origin.

jump-and-link instruction An instruction that branches to an address and simultaneously saves the address of the following instruction in a register (usually `x1` in RISC-V).

In addition to allocating these registers, RISC-V assembly language includes an instruction just for the procedures: it branches to an address and simultaneously saves the address of the following instruction to the destination register rd. The **jump-and-link instruction** (`jal`) is written

```
jal x1, ProcedureAddress       // jump to
   ProcedureAddress and write return address to x1
```

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 `x1`, is called the **return address**. The return address is needed because the same procedure could be called from several parts of the program.

return address A link to the calling site that allows a procedure to return to the proper address; in RISC-V it is stored in register `x1`.

To support the return from a procedure, computers like RISC-V use an indirect jump, like the jump-and-link instruction (`jalr`) introduced above to help with case statements:

```
jalr x0,  0(x1)
```

The jump-and-link register instruction branches to the address stored in register `x1`—which is just what we want. Thus, the calling program, or **caller**, puts the parameter values in `x10-x17` and uses `jal x1, X` to branch to procedure `X` (sometimes named the **callee**). The callee then performs the calculations, places the results in the same parameter registers, and returns control to the caller using `jalr x0, 0(x1)`.

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.

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 RISC-V architecture, although a more sensible name would have been *instruction address register*. The `jal` instruction actually saves PC + 4 in its designation register (usually `x1`) to link to the byte address of the following instruction to set up the procedure return.

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

Elaboration: The jump-and-link instruction can also be used to perform an unconditional branch within a procedure by using x0 as the destination register. Since x0 is hard-wired to zero, the effect is to discard the return address:

```
jal x0, Label  // unconditionally branch to Label
```

Using More Registers

Suppose a compiler needs more registers for a procedure than the eight argument registers. Since we must cover our tracks after our mission is complete, any registers needed by the caller must be restored to the values that they contained *before* the procedure was invoked. This situation is an example in which we need to spill registers to memory, as mentioned in the *Hardware/Software Interface* section on page 69.

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. In RISC-V, the **stack pointer** is register `x2`, also known by the name `sp`. The stack pointer is adjusted by one doubleword for each register that is saved or restored. Stacks are so popular that they have their own buzzwords for transferring data to and from the stack: placing data onto the stack is called a **push**, and removing data from the stack is called a **pop**.

stack 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 RISC-V, it is register `sp`, or `x2`.

push Add element to stack.

pop Remove element from stack.

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.

EXAMPLE

Compiling a C Procedure That Doesn't Call Another Procedure

Let's turn the example on page 66 from Section 2.2 into a C procedure:

```
long long int leaf_example (long long int g, long long
int h, long long int i, long long int j)
{
       long long int f;

        f = (g + h) – (i + j);
        return f;
}
```

What is the compiled RISC-V assembly code?

ANSWER

The parameter variables `g`, `h`, `i`, and `j` correspond to the argument registers `x10`, `x11`, `x12`, and `x13`, and `f` corresponds to `x20`. The compiled program starts with the label of the procedure:

```
leaf_example:
```

The next step is to save the registers used by the procedure. The C assignment statement in the procedure body is identical to the example on page 67, which uses two temporary registers (`x5` and `x6`). Thus, we need to save three registers: `x5`, `x6`, and `x20`. We "push" the old values onto the stack by creating space for three doublewords (24 bytes) on the stack and then store them:

```
addi sp, sp, -24          // adjust stack to make room for 3 items
sd   x5, 16(sp)           // save register x5 for use afterwards
sd   x6, 8(sp)            // save register x6 for use afterwards
sd   x20, 0(sp)           // save register x20 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 67:

```
add x5, x10, x11   // register x5 contains g + h
add x6, x12, x13   // register x6 contains i + j
sub x20, x5, x6    // f = x5 – x6, which is (g + h) – (i + j)
```

To return the value of f, we copy it into a parameter register:

```
addi x10, x20, 0 // returns f (x10 = x20 + 0)
```

Before returning, we restore the three old values of the registers we saved by "popping" them from the stack:

```
ld x20, 0(sp)    // restore register x20 for caller
ld x6, 8(sp)     // restore register x6 for caller
ld x5, 16(sp)    // restore register x5 for caller
addi sp, sp, 24  // adjust stack to delete 3 items
```

The procedure ends with a branch register using the return address:

```
jalr x0, 0(x1)       // branch 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, RISC-V software separates 19 of the registers into two groups:

- x5–x7 and x28–x31: temporary registers that are *not* preserved by the callee (called procedure) on a procedure call
- x8–x9 and x18–x27: saved registers that must be preserved on a procedure call (if used, the callee saves and restores them)

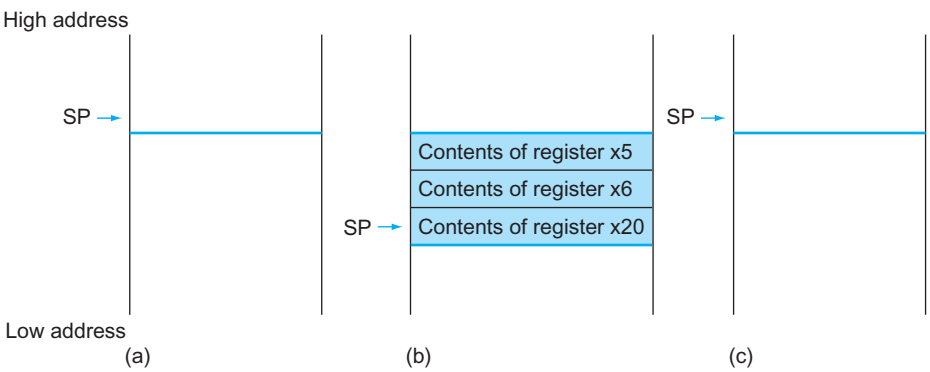


FIGURE 2.10 The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call. The stack pointer always points to the "top" of the stack, or the last doubleword in the stack in this drawing.

This simple convention reduces register spilling. In the example above, since the caller does not expect registers x5 and x6 to be preserved across a procedure call, we can drop two stores and two loads from the code. We still must save and restore x20, 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, attention must be paid 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 x10 and then using jal x1, A. Then suppose that procedure A calls procedure B via jal x1, B with an argument of 7, also placed in x10. Since A hasn't finished its task yet, there is a conflict over the use of register x10. Similarly, there is a conflict over the return address in register x1, 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 on the stack, just as we did with the saved registers. The caller pushes any argument registers (x10-x17) or temporary registers (x5-x7 and x28-x31) that are needed after the call. The callee pushes the return address register x1 and any saved registers (x8-x9 and x18-x27) 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.

EXAMPLE

Compiling a Recursive C Procedure, Showing Nested Procedure Linking

Let's tackle a recursive procedure that calculates factorial:

```
long long int fact (long long int n)
{
    if (n < 1) return (1);
        else return (n * fact(n - 1));
}
```

What is the RISC-V assembly code?

ANSWER

The parameter variable n corresponds to the argument register x10. The compiled program starts with the label of the procedure and then saves two registers on the stack, the return address and x10:

```
fact:
    addi sp, sp, -16   // adjust stack for 2 items
      sd x1, 8(sp)     // save the return address
      sd x10, 0(sp)    // save the argument n
```

The first time `fact` is called, `sd` 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`.

```
addi    x5, x10, -1        // x5 = n - 1
bge     x5, x0, L1         // if (n - 1) >= 0, 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 `x10`. It then pops the two saved values off the stack and branches to the return address:

```
addi    x10, x0, 1   // return 1
addi    sp, sp, 16   // pop 2 items off stack
jalr    x0, 0(x1)    // return to caller
```

Before popping two items off the stack, we could have loaded `x1` and `x10`. Since `x1` and `x10` 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 x10, x10, -1  // n >= 1: argument gets (n – 1)
    jal  x1, fact      // call fact with (n – 1)
```

The next instruction is where `fact` returns; its result is in `x10` . Now the old return address and old argument are restored, along with the stack pointer:

```
addi  x6, x10, 0   // return from jal: move result of fact
                      (n - 1) to x6:
ld    x10, 0(sp)   // restore argument n
ld    x1, 8(sp)    // restore the return address
addi  sp, sp, 16   // adjust stack pointer to pop 2 items
```

Next, argument register `x10` gets the product of the old argument and the result of fact(n - 1), now in `x6`. We assume a multiply instruction is available, even though it is not covered until Chapter 3:

```
mul     x10, x10, x6     // return n * fact (n – 1)
```

Finally, `fact` branches again to the return address:

```
jalr     x0, 0(x1)     // return to the caller
```

Hardware/ Software Interface

A C variable is generally a location in storage, and its interpretation depends both on its *type* and *storage class*. Example types 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, some RISC-V compilers reserve a register `x3` for use as the **global pointer**, or `gp`.

global pointer The register that is reserved to point to the static area.

Figure 2.11 summarizes what is preserved across a procedure call. Note that several schemes preserve the stack, guaranteeing that the caller will get the same data back on a load from the stack as it stored onto the stack. The stack above `sp` is preserved simply by making sure the callee does not write above `sp`; `sp` is itself preserved by the callee adding exactly the same amount that was subtracted from it; and the other registers are preserved by saving them on the stack (if they are used) and restoring them from there.

Preserved	Not preserved
Saved registers: `x8-x9, x18-x27`	Temporary registers: `x5-x7, x28-x31`
Stack pointer register: `x2(sp)`	Argument/result registers: `x10-x17`
Frame pointer: `x8(fp)`	
Return address: `x1(ra)`	
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 global pointer register, discussed in the following subsections, it is also preserved.

procedure frame Also called **activation record**. The segment of the stack containing a procedure's saved registers and local variables.

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.

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

Some RISC-V compilers use a **frame pointer** `fp`, or register `x8` to point to the first doubleword 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

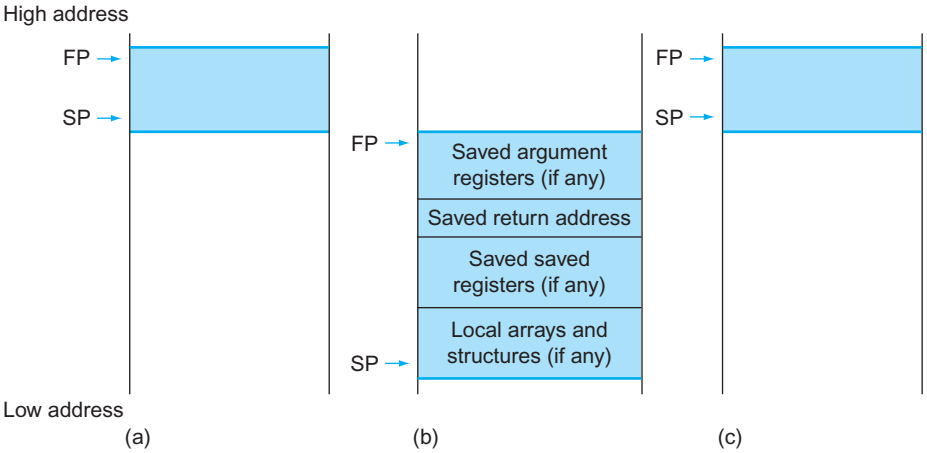


FIGURE 2.12 Illustration of the stack allocation (a) before, (b) during, and (c) after the procedure call. The frame pointer (`fp` or `x8`) points to the first doubleword of the frame, often a saved argument register, and the stack pointer (`sp`) points to the top of the stack. The stack is adjusted 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 RISC-V Reference Data Card at the front of this book.

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 to and exit from the procedure.

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 RISC-V convention for allocation of memory when running the Linux operating system. The stack starts in the high end of the user addresses space (see Chapter 5) and grows down. The first part of the low end of memory is reserved, followed by the home of the RISC-V 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.

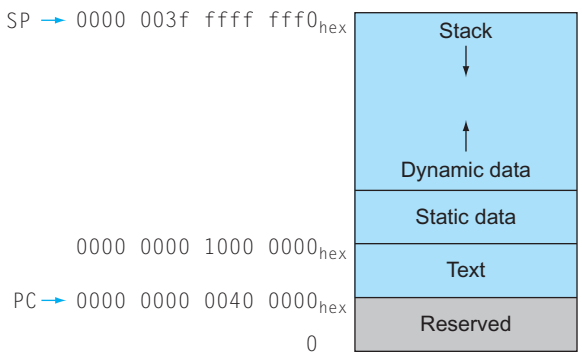


FIGURE 2.13 The RISC-V memory allocation for program and data. These addresses are only a software convention, and not part of the RISC-V architecture. The user address space is set to 2^{38} of the potential 2^{64} total address space given a 64-bit architecture (see Chapter 5). The stack pointer is initialized to 0000 003f ffff fff0$_{hex}$ and grows down toward the data segment. At the other end, the program code ("text") starts at 0000 0000 0040 0000$_{hex}$. The static data starts immediately after the end of the text segment; in this example, we assume that address is 0000 0000 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*. This information is also found in Column 4 of the RISC-V 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. C programs control memory allocation, which is the source of many common and difficult bugs. Forgetting to free space leads to a "memory leak," which ultimately 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.



COMMON CASE FAST

Figure 2.14 summarizes the register conventions for the RISC-V assembly language. This convention is another example of making the **common case fast**: most procedures can be satisfied with up to eight argument registers, twelve saved registers, and seven temporary registers without ever going to memory.

Elaboration: What if there are more than eight parameters? The RISC-V convention is to place the extra parameters on the stack just above the frame pointer. The procedure then expects the first eight parameters to be in registers `x10` through `x17` 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 RISC-V C compiler only uses a frame pointer in procedures that change the stack pointer in the body of the procedure.

Name	Register number	Usage	Preserved on call?
x0	0	The constant value 0	n.a.
x1 (ra)	1	Return address (link register)	yes
x2 (sp)	2	Stack pointer	yes
x3 (gp)	3	Global pointer	yes
x4 (tp)	4	Thread pointer	yes
x5-x7	5–7	Temporaries	no
x8-x9	8–9	Saved	yes
x10-x17	10–17	Arguments/results	no
x18-x27	18–27	Saved	yes
x28-x31	28–31	Temporaries	no

FIGURE 2.14 RISC-V register conventions. This information is also found in Column 2 of the RISC-V Reference Data Card at the front of this book.

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:

```
long long int sum (long long int n, long long 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 `x10 = n, x11 = acc`, and the result goes into `x12`):

```
sum: ble x10, x0, sum_exit   // go to sum_exit if n <= 0
     add x11, x11, x10       // add n to acc
     addi x10, x10, -1       // subtract 1 from n
     jal x0, sum             // jump to sum
sum_exit:
     addi x12, x11, 0        // return value acc
     jalr x0, 0(x1)          // return to caller
```

Check Yourself

Which of the following statements about C and Java is generally true?

1. C programmers manage data explicitly, while it's automatic in Java.
2. C leads to more pointer bugs and memory leak bugs than does Java.

!(@ | = > (wow open tab at bar is great)

Fourth line of the keyboard poem "Hatless Atlas," 1991 (some give names to ASCII characters: "!" is "wow," "(" is open, "|" is bar, and so on).

2.9 Communicating with People

Computers were invented to crunch numbers, but as soon as they became commercially viable they were used to process text. Most computers today offer 8-bit bytes to represent characters, with the *American Standard Code for Information Interchange* (ASCII) being the representation that nearly everyone follows. Figure 2.15 summarizes ASCII.

ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

FIGURE 2.15 ASCII representation of characters. Note that upper- and lowercase letters differ by exactly 32; this observation can lead to shortcuts in checking or changing upper- and lowercase. Values not shown include formatting characters. For example, 8 represents a backspace, 9 represents a tab character, and 13 a carriage return. Another useful value is 0 for null, the value the programming language C uses to mark the end of a string.

ASCII versus Binary Numbers

EXAMPLE

We could represent numbers as strings of ASCII digits instead of as integers. How much does storage increase if the number 1 billion is represented in ASCII versus a 32-bit integer?

ANSWER

One billion is 1,000,000,000, so it would take 10 ASCII digits, each 8 bits long. Thus the storage expansion would be (10 × 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.

A series of instructions can extract a byte from a doubleword, so load register and store register are sufficient for transferring bytes as well as words. Because of the popularity of text in some programs, however, RISC-V provides instructions to move bytes. *Load byte* unsigned (`lbu`) loads a byte from memory, placing it in the rightmost 8 bits of a register. *Store byte* (`sb`) takes a byte from the rightmost 8 bits of a register and writes it to memory. Thus, we copy a byte with the sequence

```
lbu x12, 0(x10)    // Read byte from source
sb  x12, 0(x11)    // 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, and 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[])
{
    size_t i;
    i = 0;
    while ((x[i] = y[i]) != '\0') /* copy & test byte */
        i += 1;
}
```

What is the RISC-V assembly code?

ANSWER

Below is the basic RISC-V assembly code segment. Assume that base addresses for arrays `x` and `y` are found in `x10` and `x11`, while `i` is in `x19`. `strcpy` adjusts the stack pointer and then saves the saved register `x19` on the stack:

```
strcpy:
    addi  sp, sp, -8      // adjust stack for 1 more item
    sd    x19, 0(sp)      // save x19
```

To initialize `i` to 0, the next instruction sets `x19` to 0 by adding 0 to 0 and placing that sum in `x19`:

```
add  x19, x0, x0     // 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  x5, x19, x11  // address of y[i] in x5
```

Note that we don't have to multiply `i` by 8 since `y` is an array of *bytes* and not of doublewords, as in prior examples.

To load the character in `y[i]`, we use load byte unsigned, which puts the character into `x6`:

```
lbu  x6, 0(x5)  // x6 = y[i]
```

A similar address calculation puts the address of `x[i]` in `x7`, and then the character in `x6` is stored at that address.

```
add    x7, x19, x10     // address of x[i] in x7
sb     x6, 0(x7)        // 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 x6, x0, L2
```

If not, we increment `i` and loop back:

```
addi  x19, x19, 1   // i = i + 1
jal   x0, L1        // go to L1
```

If we don't loop back, it was the last character of the string; we restore `x19` and the stack pointer, and then return.

```
L2: ld      x19, 0(sp)     // restore old x19
    addi    sp, sp, 8      // pop 1 doubleword off stack
    jalr    x0, 0(x1)     // 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 `x19`. Hence, instead of thinking of these 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 RISC-V instruction set has explicit instructions to load and store such 16-bit quantities, called *halfwords*. *Load half unsigned* loads a halfword from memory, placing it in the rightmost 16 bits of a register, filling the leftmost 48 bits with zeros. Like load byte, *load half* (`lh`) treats the halfword as a signed number and thus sign-extends to fill the 48 leftmost bits of the register. *Store half* (`sh`) takes a halfword from the rightmost 16 bits of a register and writes it to memory. We copy a halfword with the sequence

```
lhu x19, 0(x10) // Read halfword (16 bits) from source
sh  x19, 0(x11) // Write halfword (16 bits) to dest
```

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: RISC-V software is required to keep the stack aligned to "quadword" (16 byte) addresses to get better performance. This convention means that a `char` variable allocated on the stack may occupy as much as 16 bytes, even though it needs less. However, a C string variable or an array of bytes *will* pack 16 bytes per quadword, and a Java string variable or array of shorts packs 8 halfwords per quadword.

Elaboration: Reflecting the international nature of the web, most web pages today use Unicode instead of ASCII. Hence, Unicode may be even more popular than ASCII today.

Elaboration: RISC-V also includes instructions to move 32-bit values to and from memory. *Load word unsigned* (`lwu`) loads a 32-bit word from memory into the rightmost 32 bits of a register, filling the leftmost 32 bits with zeros. *Load word* (`lw`) instead fills the leftmost 32 bits with copies of bit 31. *Store word* (`sw`) takes a word from the rightmost 32 bits of a register and stores it to memory.

Check Yourself

I. Which of the following statements about characters and strings in C and Java is true?

1. A string in C takes about half the memory as the same string in Java.
2. Strings are just an informal name for single-dimension arrays of characters in C and Java.
3. Strings in C and Java use null (0) to mark the end of a string.
4. Operations on strings, like length, are faster in C than in Java.

II. Which type of variable that can contain $1{,}000{,}000{,}000_{ten}$ takes the most memory space?

1. `long long int` in C
2. `string` in C
3. `string` in Java

2.10 RISC-V Addressing for Wide Immediates and Addresses

Although keeping all RISC-V instructions 32 bits long simplifies the hardware, there are times where it would be convenient to have 32-bit or larger constants or addresses. This section starts with the general solution for large constants, and then shows the optimizations for instruction addresses used in branches.

Wide Immediate Operands

Although constants are frequently short and fit into the 12-bit fields, sometimes they are bigger.

The RISC-V instruction set includes the instruction *Load upper immediate* (`lui`) to load a 20-bit constant into bits 12 through 31 of a register. The leftmost 32 bits are filled with copies of bit 31, and the rightmost 12 bits are filled with zeros. This instruction allows, for example, a 32-bit constant to be created with

two instructions. `lui` uses a new instruction format, U-type, as the other formats cannot accommodate such a large constant.

EXAMPLE

Loading a 32-Bit Constant

What is the RISC-V assembly code to load this 64-bit constant into register `x19`?

```
00000000 00000000 00000000 00000000 00000000 00111101 00000101 00000000
```

ANSWER

First, we would load bits 12 through 31 with that bit pattern, which is 976 in decimal, using `lui`:

```
lui    x19, 976 // 976decimal = 0000 0000 0011 1101 0000
```

The value of register `x19` afterward is:

```
0000000 00000000 00000000 00000000 00000000 00111101 00000000 00000000
```

The next step is to add in the lowest 12 bits, whose decimal value is 1280:

```
addi    x19, x19, 1280     // 1280decimal = 00000101 00000000
```

The final value in register `x19` is the desired value:

```
00000000 00000000 00000000 00000000 00000000 00111101 00000101 00000000
```

Elaboration: In the previous example, bit 11 of the constant was 0. If bit 11 had been set, there would have been an additional complication: the 12-bit immediate is sign-extended, so the addend would have been negative. This means that in addition to adding in the rightmost 11 bits of the constant, we would have also subtracted 2^{12}. To compensate for this error, it suffices to add 1 to the constant loaded with lui, since the lui constant is scaled by 2^{12}.

Hardware/ Software Interface

Either the compiler or the assembler must break large constants into pieces and then reassemble them into a register. As you might expect, the immediate field's size restriction may be a problem for memory addresses in loads and stores as well as for constants in immediate instructions.

Hence, the symbolic representation of the RISC-V 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.

Addressing in Branches

The RISC-V branch instructions use the RISC-V instruction format called SB-type. This format can represent branch addresses from −4096 to 4094, in multiples of 2. For reasons revealed shortly, it is only possible to branch to even addresses. The SB-type format consists of a 7-bit opcode, a 3-bit function code, two 5-bit register operands (rs1 and rs2), and a 12-bit address immediate. The address uses an unusual encoding, which simplifies datapath design but complicates assembly. The instruction

```
bne  x10, x11, 2000  // if x10 != x11, go to location 2000ten = 0111 1101 0000
```

could be assembled into this format (it's actually a bit more complicated, as we will see):

0	111110	01011	01010	001	1000	0	1100111
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode

where the opcode for conditional branches is 1100111_{two} and `bne`'s funct3 code is 001_{two}.

The unconditional jump-and-link instruction (`jal`) is the only instruction that uses the *UJ-type* format. This instruction consists of a 7-bit opcode, a 5-bit destination register operand (rd), and a 20-bit address immediate. The link address, which is the address of the instruction following the `jal`, is written to rd.

Like the SB-type format, the UJ-type format's address operand uses an unusual immediate encoding, and it cannot encode odd addresses. So,

```
jal  x0, 2000  // go to location 2000ten = 0111 1101 0000
```

is assembled into this format:

0	1111101000	0	00000000	00000	1101111
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode

If addresses of the program had to fit in this 20-bit field, it would mean that no program could be bigger than 2^{20}, 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 offset, so that a branch instruction would calculate the following:

$$\text{Program counter} = \text{Register} + \text{Branch offset}$$

This sum allows the program to be as large as 2^{64} 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 $\pm 2^{10}$ words of the current instruction, or jump within $\pm 2^{18}$ 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 smaller than 2^{10} words, so the PC is the ideal choice. This form of branch addressing is called **PC-relative addressing**.

PC-relative addressing An addressing regime in which the address is the sum of the *program counter* (PC) and a constant in the instruction.

Like most recent computers, RISC-V uses PC-relative addressing for both conditional branches and unconditional jumps, because the destination of these instructions is likely to be close to the branch. On the other hand, procedure calls may require jumping more than 2^{18} words away, since there is no guarantee that the callee is close to the caller. Hence, RISC-V allows very long jumps to any 32-bit address with a two-instruction sequence: `lui` writes bits 12 through 31 of the address to a temporary register, and `jalr` adds the lower 12 bits of the address to the temporary register and jumps to the sum.

Since RISC-V instructions are 4 bytes long, the RISC-V branch instructions could have been designed to stretch their reach by having the PC-relative address refer to the number of *words* between the branch and the target instruction, rather than the number of bytes. However, the RISC-V architects wanted to support the possibility of instructions that are only 2 bytes long, so the branch instructions represent the number of *halfwords* between the branch and the branch target. Thus, the 20-bit address field in the `jal` instruction can encode a distance of $\pm 2^{19}$ halfwords, or ± 1 MiB from the current PC. Similarly, the 12-bit field in the conditional branch instructions is also a halfword address, meaning that it represents a 13-bit byte address.

EXAMPLE

Showing Branch Offset in Machine Language

The *while* loop on page 94 was compiled into this RISC-V assembler code:

```
Loop:slli x10, x22, 3     // Temp reg x10 = i * 8
     add  x10, x10, x25   // x10 = address of save[i]
     ld   x9, 0(x10)      // Temp reg x9 = save[i]
     bne  x9, x24, Exit   // go to Exit if save[i] != k
     addi x22, x22, 1     // i = i + 1
     beq  x0, x0, Loop    // go to Loop
Exit:
```

ANSWER

If we assume we place the loop starting at location 80000 in memory, what is the RISC-V machine code for this loop?

The assembled instructions and their addresses are:

Address	Instruction					
80000	0000000	00011	10110	001	01010	0010011
80004	0000000	11001	01010	000	01010	0110011
80008	0000000	00000	01010	011	01001	0000011
80012	0000000	11000	01001	001	01100	1100011
80016	0000000	00001	10110	000	10110	0010011
80020	1111111	00000	00000	000	01101	1100011

Remember that RISC-V instructions have byte addresses, so addresses of sequential words differ by 4. The `bne` instruction on the fourth line adds 3 words or 12 bytes to the address of the instruction, specifying the branch destination relative to the branch instruction (12 + 80012) and not using the full destination address (80024). The branch instruction on the last line does a similar calculation for a backwards branch (−20 + 80020), 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 12-bit address in the conditional branch instruction. The assembler comes to the rescue just as it did with large addresses or constants: it inserts an unconditional branch to the branch target, and inverts the condition so that the conditional branch decides whether to skip the unconditional branch.

EXAMPLE

Branching Far Away

Given a branch on register `x10` being equal to zero,

```
beq   x10, x0, L1
```

ANSWER

replace it by a pair of instructions that offers a much greater branching distance. These instructions replace the short-address conditional branch:

```
      bne     x10, x0, L2
      jal     x0, L1
L2:
```

RISC-V Addressing Mode Summary

Multiple forms of addressing are generically called **addressing modes**. Figure 2.17 shows how operands are identified for each addressing mode. The addressing modes of the RISC-V instructions are the following:

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

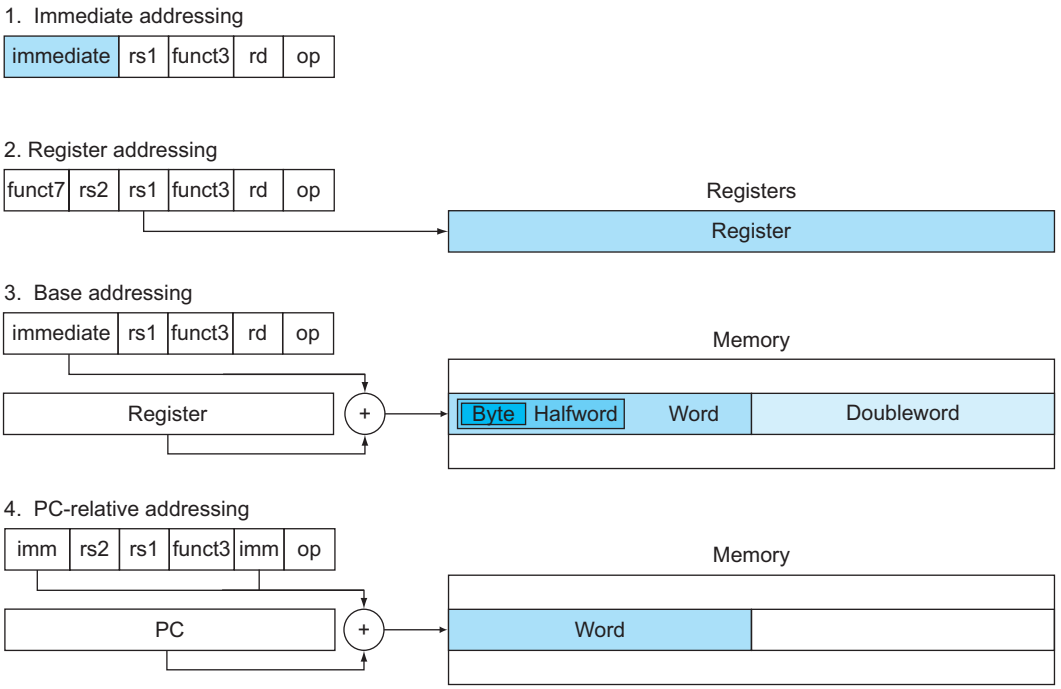


FIGURE 2.17 Illustration of four RISC-V addressing modes. The operands are shaded in color. The operand of mode 3 is in memory, whereas the operand for mode 2 is a register. Note that versions of load and store access bytes, halfwords, words, or doublewords. For mode 1, the operand is part of the instruction itself. Mode 4 addresses instructions in memory, with mode 4 adding a long address to 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.

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.

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 2.18 shows the RISC-V encoding of the opcodes for the RISC-V machine language. This figure helps when translating by hand between assembly language and machine language.

Format	Instruction	Opcode	Funct3	Funct6/7
R-type	add	0110011	000	0000000
	sub	0110011	000	0100000
	sll	0110011	001	0000000
	xor	0110011	100	0000000
	srl	0110011	101	0000000
	sra	0110011	101	0000000
	or	0110011	110	0000000
	and	0110011	111	0000000
	lr.d	0110011	011	0001000
	sc.d	0110011	011	0001100
I-type	lb	0000011	000	n.a.
	lh	0000011	001	n.a.
	lw	0000011	010	n.a.
	ld	0000011	011	n.a.
	lbu	0000011	100	n.a.
	lhu	0000011	101	n.a.
	lwu	0000011	110	n.a.
	addi	0010011	000	n.a.
	slli	0010011	001	000000
	xori	0010011	100	n.a.
	srli	0010011	101	000000
	srai	0010011	101	010000
	ori	0010011	110	n.a.
	andi	0010011	111	n.a.
	jalr	1100111	000	n.a.
S-type	sb	0100011	000	n.a.
	sh	0100011	001	n.a.
	sw	0100011	010	n.a.
	sd	0100011	111	n.a.
SB-type	beq	1100111	000	n.a.
	bne	1100111	001	n.a.
	blt	1100111	100	n.a.
	bge	1100111	101	n.a.
	bltu	1100111	110	n.a.
	bgeu	1100111	111	n.a.
U-type	lui	0110111	n.a.	n.a.
UJ-type	jal	1101111	n.a.	n.a.

FIGURE 2.18 RISC-V instruction encoding. All instructions have an opcode field, and all formats except U-type and UJ-type use the funct3 field. R-type instructions use the funct7 field, and immediate shifts (`slli`, `srli`, `srai`) use the funct6 field.

EXAMPLE

Decoding Machine Code

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

00578833_{hex}

ANSWER

The first step is converting hexadecimal to binary:

```
0000 0000 0101 0111 1000 1000 0011 0011
```

To know how to interpret the bits, we need to determine the instruction format, and to do that we first need to determine the `opcode`. The opcode is the rightmost 7 bits, or 0110011. Searching Figure 2.20 for this value, we see that the opcode corresponds to the R-type arithmetic instructions. Thus, we can parse the binary format into fields listed in Figure 2.21:

funct7	rs2	rs1	funct3	rd	opcode
0000000	00101	01111	000	10000	0110011

We decode the rest of the instruction by looking at the field values. The funct7 and funct3 fields are both zero, indicating the instruction is `add`. The decimal values for the register operands are 5 for the rs2 field, 15 for rs1, and 16 for rd. These numbers represent registers `x5`, `x15`, and `x16`. Now we can reveal the assembly instruction:

```
add x16, x15, x5
```

Figure 2.19 shows all the RISC-V instruction formats. Figure 2.1 on pages 64–65 shows the RISC-V assembly language revealed in this chapter. The next chapter covers RISC-V instructions for multiply, divide, and arithmetic for real numbers.

Name	Field						Comments
(Field size)	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

FIGURE 2.19 RISC-V instruction formats.

Check Yourself

I. What is the range of byte addresses for conditional branches in RISC-V (K = 1024)?

1. Addresses between 0 and 4K − 1
2. Addresses between 0 and 8K − 1
3. Addresses up to about 2K before the branch to about 2K after
4. Addresses up to about 4K before the branch to about 4K after

II. What is the range of byte addresses for the jump-and-link instruction in RISC-V (M = 1024K)?

1. Addresses between 0 and 512K − 1
2. Addresses between 0 and 1M − 1
3. Addresses up to about 512K before the branch to about 512K after
4. Addresses up to about 1M before the branch to about 1M after

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.

data race Two memory accesses form a data race if they are from different threads to the same location, at least one is a write, and they occur one after another.

For example, recall the analogy of the eight reporters writing a story on pages 44–45 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 system programmers will use the primitives 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 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 prevented, 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 was 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 pair of instructions.

In RISC-V this pair of instructions includes a special load called a *load-reserved doubleword* (`lr.d`) and a special store called a *store-conditional doubleword* (`sc.d`). These instructions are used in sequence: if the contents of the memory location specified by the load-reserved are changed before the store-conditional to the same address occurs, then the store-conditional fails and does not write the value to memory. The store-conditional is defined to both store the value of a (presumably different) register in memory *and* to change the value of another register to a 0 if it succeeds and to a nonzero value if it fails. Thus, `sc.d` specifies three registers: one to hold the address, one to indicate whether the atomic operation failed or succeeded, and one to hold the value to be stored in memory if it succeeded. Since the load-reserved returns the initial value, and the store-conditional returns 0 only if it succeeds, the following

sequence implements an atomic exchange on the memory location specified by the contents of x20:

```
again:lr.d x10, (x20)          // load-reserved
      sc.d x11, x23, (x20)     // store-conditional
      bne  x11, x0, again      // branch if store fails
      addi x23, x10, 0         // put loaded value in x23
```

Any time a processor intervenes and modifies the value in memory between the lr.d and sc.d instructions, the sc.d writes a nonzero value into x11, causing the code sequence to try again. At the end of this sequence, the contents of x23 and the memory location specified by x20 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).

Elaboration: An advantage of the load-reserved/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 lr.d and the sc.d, but not too many.

Since the store-conditional will fail after either another attempted store to the load reservation address or any exception, care must be taken in choosing which instructions are inserted between the two instructions. In particular, only integer arithmetic, forward branches, and backward branches out of the load-reserved/store-conditional block can safely be permitted; otherwise, it is possible to create deadlock situations where the processor can never complete the sc.d because of repeated page faults. In addition, the number of instructions between the load-reserved 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.

Elaboration: While the code above implemented an atomic exchange, the following code would more efficiently acquire a lock at the location in register x20, where the value of 0 means the lock was free and 1 to mean lock was acquired:

```
        addi x12, x0, 1       // copy locked value
again:  lr.d x10, (x20)       // load-reserved to read lock
        bne  x10, x0, again   // check if it is 0 yet
        sc.d x11, x12, (x20)  // attempt to store new value
        bne  x11, x0, again   // branch if store fails
```

We release the lock just using a regular store to write 0 into the location:

```
        sd x0, 0(x20)     // free lock by writing 0
```

Check Yourself

When do you use primitives like load-reserved 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.

2.12 Translating and Starting a Program

This section describes the four steps in transforming a C program in a file from storage (disk or flash memory) into a program running on a computer. Figure 2.20 shows the translation hierarchy. Some systems combine these steps to reduce translation time, but programs go through these four logical phases. This section follows this translation hierarchy.

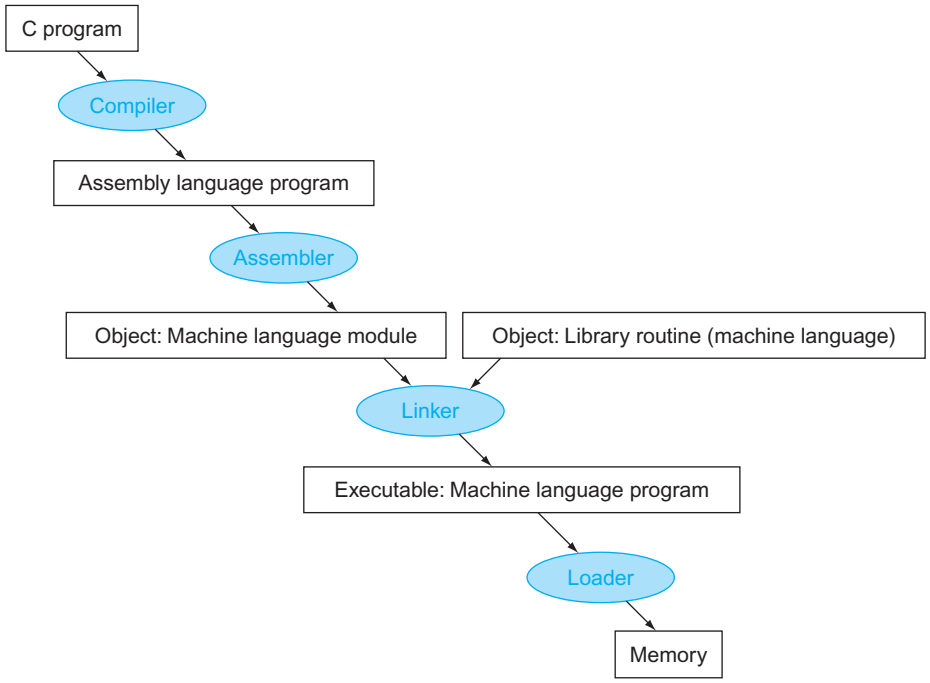


FIGURE 2.20 A translation hierarchy for C. A high-level language program is first compiled into an assembly language program and then assembled into an object module in machine language. The linker combines multiple modules with library routines to resolve all references. The loader then places the machine 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.

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.

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**.

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

As mentioned above, the RISC-V hardware makes sure that register `x0` always has the value 0. That is, whenever register `x0` is used, it supplies a 0, and if the programmer attempts to change the value in `x0`, the new value is simply discarded. Register `x0` is used to create the assembly language instruction that copies the contents of one register to another. Thus, the RISC-V assembler accepts the following instruction even though it is not found in the RISC-V machine language:

```
li x9, 123     // load immediate value 123 into register x9
```

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

```
addi x9, x0, 123 // register x9 gets register x0 + 123
```

The RISC-V assembler also converts `mv` (move) into an `addi` instruction. Thus

```
mv x10, x11  // register x10 gets register x11
```

becomes

```
addi x10, x11, 0  // register x10 gets register x11 + 0
```

The assembler also accepts `j` Label to unconditionally branch to a label, as a stand-in for `jal x0, Label`. It also converts branches to faraway locations into a branch and a jump. As mentioned above, the RISC-V assembler allows large constants to be loaded into a register despite the limited size of the immediate instructions. Thus, the *load immediate* (`li`) pseudoinstruction introduced above can

create constants larger than `addi`'s immediate field can contain; the *load address* (`la`) macro works similarly for symbolic addresses. Finally, it can simplify the instruction set by determining which variation of an instruction the programmer wants. For example, the RISC-V assembler does not require the programmer to specify the immediate version of the instruction when using a constant for arithmetic and logical instructions; it just generates the proper opcode. Thus

```
and x9, x10, 15 // register x9 gets x10 AND 15
```

becomes

```
andi x9, x10, 15 // register x9 gets x10 AND 15
```

We include the "`i`" on the instructions to remind the reader that `andi` produces a different opcode in a different instruction format than the `and` instruction with no immediate operands.

In summary, pseudoinstructions give RISC-V a richer set of assembly language instructions than those implemented by the hardware. If you are going to write assembly programs, use pseudoinstructions to simplify your task. To understand the RISC-V architecture and be sure to get best performance, however, study the real RISC-V instructions found in Figures 2.1 and 2.18.

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. RISC-V assemblers use hexadecimal and octal.

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.

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

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

- The *object file header* describes the size and position of the other pieces of the object file.
- The *text segment* contains the machine language code.
- The *static data segment* contains data allocated for the life of the program. (UNIX allows programs to use 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.
- 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. The reason a linker is useful is that it is much faster to patch code than it is to recompile and reassemble.

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

There are three steps for the linker:

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

The linker uses the relocation information and symbol table in each object module to resolve all undefined labels. Such references occur in branch instructions 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.

If all external references are resolved, the linker next determines the memory locations each module will occupy. Recall that Figure 2.13 on page 106 shows the RISC-V 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.

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.

EXAMPLE

Linking Object Files

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

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

Object file header			
	Name	Procedure A	
	Text size	100_{hex}	
	Data size	20_{hex}	
Text segment	Address	Instruction	
	0	`ld x10, 0(x3)`	
	4	`jal x1, 0`	
	. . .	. . .	
Data segment	0	(X)	
	. . .	. . .	
Relocation information	Address	Instruction type	Dependency
	0	`ld`	X
	4	`jal`	B
Symbol table	Label	Address	
	X	–	
	B	–	
	Name	Procedure B	
	Text size	200_{hex}	
	Data size	30_{hex}	
Text segment	Address	Instruction	
	0	`sd x11, 0(x3)`	
	4	`jal x1, 0`	
	. . .	. . .	
Data segment	0	(Y)	
	. . .	. . .	
Relocation information	Address	Instruction type	Dependency
	0	`sd`	Y
	4	`jal`	A
Symbol table	Label	Address	
	Y	–	
	A	–	

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`

ANSWER

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 2.14 on page 107, we know that the text segment starts at address $0000\ 0000\ 0040\ 0000_{hex}$ and the data segment at $0000\ 0000\ 1000\ 0000_{hex}$. 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
	$0000\ 0000\ 0040\ 0000_{hex}$	ld x10, 0(x3)
	$0000\ 0000\ 0040\ 0004_{hex}$	jal x1, 252_{ten}
	. . .	. . .
	$0000\ 0000\ 0040\ 0100_{hex}$	sd x11, 32(x3)
	$0000\ 0000\ 0040\ 0104_{hex}$	jal x1, -260_{ten}
	. . .	. . .
Data segment	Address	
	$0000\ 0000\ 1000\ 0000_{hex}$	(X)
	. . .	. . .
	$0000\ 0000\ 1000\ 0020_{hex}$	(Y)
	. . .	. . .

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 three types here:

1. The jump and link instructions use PC-relative addressing. Thus, for the jal at address $40\ 0004_{hex}$ to go to $40\ 0100_{hex}$ (the address of procedure B), it must put ($40\ 0100_{hex}$ – $40\ 0004_{hex}$) or 252_{ten} in its address field. Similarly, since $40\ 0000_{hex}$ is the address of procedure A, the jal at $40\ 0104_{hex}$ gets the negative number -260_{ten} ($40\ 0000_{hex}$ – $40\ 0104_{hex}$) in its address field.

2. The load addresses are harder because they are relative to a base register. This example uses x3 as the base register, assuming it is initialized to $0000\ 0000\ 1000\ 0000_{hex}$. To get the address $0000\ 0000\ 1000\ 0000_{hex}$ (the address of doubleword X), we place 0_{ten} in the address field of ld at address $40\ 0000_{hex}$. Similarly, we place 20_{hex} in the address field of sd at address $40\ 0100_{hex}$ to get the address $0000\ 0000\ 1000\ 0020_{hex}$ (the address of doubleword Y).

3. Store addresses are handled just like load addresses, except that their S-type instruction format represents immediates differently than loads' I-type format. We place 32_{ten} in the address field of sd at address $40\ 0100_{hex}$ to get the address $0000\ 0000\ 1000\ 0020_{hex}$ (the address of doubleword Y).

Loader

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

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 processor registers and sets the stack pointer to the first free location.
6. Branches 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.

Dynamically Linked Libraries

Virtually every problem in computer science can be solved by another level of indirection.

David Wheeler

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 on a RISC-V system running the Linux operating system is 1.5 MiB.

dynamically linked libraries (DLLs) Library routines that are linked to a program during execution.

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 original version of DLLs, the loader ran a dynamic linker, using the extra information in the file to find the appropriate libraries and to update all external references.

The downside of the initial version of DLLs was that it still linked all routines of the library that might be called, versus just 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 2.21 shows the technique. It starts with the nonlocal routines calling a set of

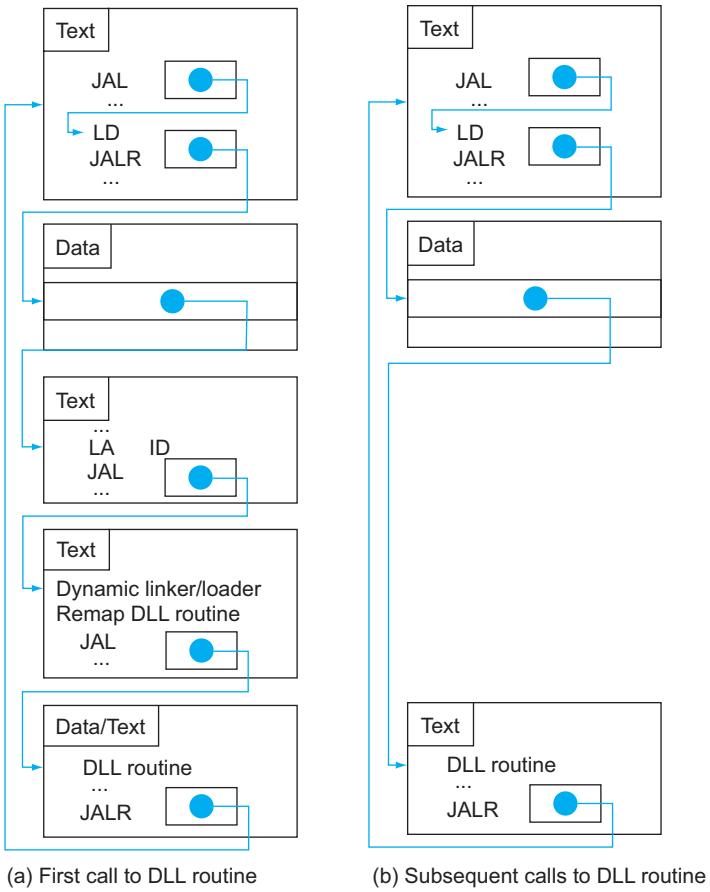


FIGURE 2.21 Dynamically linked library via lazy procedure linkage. (a) Steps for the first time a call is made to the DLL routine. (b) The steps to find the routine, remap it, and link it are skipped on subsequent calls. As we will see in Chapter 5, the operating system may avoid copying the desired routine by remapping it using virtual memory management.

dummy routines at the end of the program, with one entry per nonlocal routine. These dummy entries each contain an indirect branch.

The first time the library routine is called, the program calls the dummy entry and follows the indirect branch. It points to code that puts a number in a register to identify the desired library routine and then branches to the dynamic linker/loader. The linker/loader finds the wanted routine, remaps it, and changes the address in the indirect branch location to point to that routine. It then branches to it. When the routine completes, it returns to the original calling site. Thereafter, the call to the library routine branches indirectly to the routine without the extra hops.

In summary, DLLs require additional 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 branch 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 particular 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.

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

Figure 2.22 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.

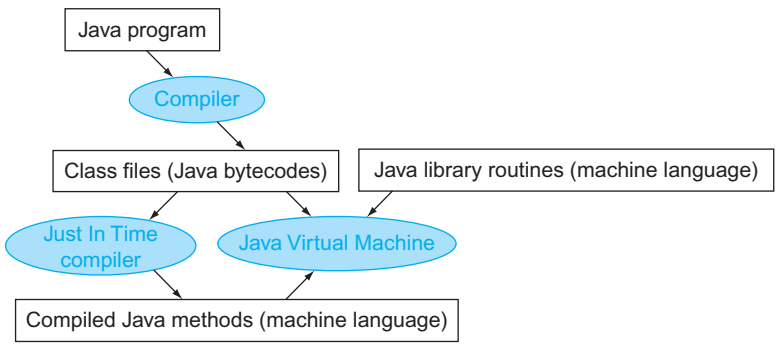


FIGURE 2.22 A translation hierarchy for Java. A Java program is first compiled into a binary version of Java bytecodes, with all addresses defined by the compiler. The Java program is now ready to run on the interpreter, called the *Java Virtual Machine* (JVM). The JVM links to desired methods in the 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.

A software interpreter, called a **Java Virtual Machine (JVM)**, can execute Java bytecodes. An interpreter is a program that simulates an instruction set architecture. For example, the RISC-V 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 billions of devices, in everything from cell phones to Internet browsers.

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

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

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

As computers get faster so that compilers can do more, and as researchers invent betters ways to compile Java on the fly, the performance gap between Java and C or C++ is closing. 🌐 **Section 2.15** goes into much greater depth on the implementation of Java, Java bytecodes, JVM, and JIT compilers.

Check Yourself

Which of the advantages of an interpreter over a translator was the 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 RISC-V code from two procedures written in C: one to swap array elements and one to sort them.

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

FIGURE 2.23 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 2.23. This procedure simply swaps two locations in memory. When translating from C to assembly language by hand, we follow these general steps:

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

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

Register Allocation for `swap`

As mentioned on page 98, the RISC-V convention on parameter passing is to use registers `x10` to `x17`. Since swap has just two parameters, `v` and `k`, they will be found in registers `x10` and `x11`. The only other variable is `temp`, which we associate with register `x5` since swap is a leaf procedure (see page 102). This register allocation corresponds to the variable declarations in the first part of the swap procedure in Figure 2.23.

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 RISC-V refers to the *byte* address, and so doublewords are really 8 bytes apart. Hence, we need to multiply the index `k` by 8 before adding it to the address. *Forgetting that sequential doubleword addresses differ by 8 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 8 via a shift left by 3:

```
slli    x6, x11, 3       // reg x6 = k * 8
add     x6, x10, x6      // reg x6 = v + (k * 8)
```

Now we load `v[k]` using `x6`, and then `v[k+1]` by adding 8 to `x6`:

```
ld    x5, 0(x6)     // reg x5 (temp) = v[k]
ld    x7, 8(x6)     // reg x7 = v[k + 1]
                    // refers to next element of v
```

Next we store `x9` and `x11` to the swapped addresses:

```
sd    x7, 0(x6)     // v[k] = reg x7
sd    x5, 8(x6)     // v[k+1] = reg x5 (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. All that remains is to add the procedure label and the return branch.

```
swap:
  slli   x6, x11, 3   // reg x6 = k * 8
  add    x6, x10, x6  // reg x6 = v + (k * 8)
  ld     x5, 0(x6)    // reg x5 (temp) = v[k]
  ld     x7, 8(x6)    // reg x7 = v[k + 1]
  sd     x7, 0(x6)    // v[k] = reg x7
  sd     x5, 8(x6)    // v[k+1] = reg x5 (temp)
  jalr   x0, 0(x1)    // return to calling routine
```

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 2.24 shows the C version of the program. Once again, we present this procedure in several steps, concluding with the full procedure.

```
void sort (long long int v[], size_t int n)
{
   size_t 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 2.24 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 `x10` and `x11`, and we assign register `x19` to `i` and register `x20` 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:

```
li  x19, 0
```

(Remember that `li` is a pseudoinstruction provided by the assembler for the convenience of the assembly language programmer; see page 125.) It also takes just one instruction to increment `i`, the last part of the *for* statement:

```
addi  x19, x19, 1  // i += 1
```

The loop should be exited if `i < n` is *not* true or, said another way, should be exited if `i ≥ n`. This test takes just one instruction:

```
for1tst: bge x19, x11, exit1  // go to exit1 if x19 ≥ x1 (i≥n)
```

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

```
        j  for1tst    // branch to test of outer loop
exit1:
```

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

```
    li x19, 0                // i = 0
for1tst:
    bge x19, x11, exit1      // go to exit1 if x19 ≥ x1 (i≥n)
        …
```

```
        (body of first for loop)
        ...
        addi x19, x19, 1    // i += 1
        j  for1tst          // branch 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   x20, x19, -1    // j = i - 1
```

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

```
addi   x20, x20, -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:
     blt x20, x0, exit2   // go to exit2 if x20 < 0 (j < 0)
```

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

The second test exits if `v[j] > v[j + 1]` is *not* true, or exits if `v[j]` ≤ `v[j + 1]`. First we create the address by multiplying `j` by 8 (since we need a byte address) and add it to the base address of `v`:

```
slli      x5, x20, 3        // reg x5 = j * 8
add       x5, x10, x5       // reg x5 = v + (j * 8)
```

Now we load `v[j]`:

```
ld        x6, 0(x5)         // reg x6 = v[j]
```

Since we know that the second element is just the following doubleword, we add 8 to the address in register `x5` to get `v[j + 1]`:

```
ld        x7, 8(x5)         // reg x7 = v[j + 1]
```

We test `v[j]` ≤ `v[j + 1]` to exit the loop:

```
ble       x6, x7, exit2     // go to exit2 if x6 ≤ x7
```

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

```
j         for2tst           // branch to test of inner loop
```

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

```
         addi x20, x19, -1  // j = i - 1
for2tst: blt x20, x0, exit2 // go to exit2 if x20 < 0 (j < 0)
```

```
        slli  x5, x20, 3     // reg x5 = j * 8
        add   x5, x10, x5    // reg x5 = v + (j * 8)
        ld    x6, 0(x5)      // reg x6 = v[j]
        ld    x7, 8(x5)      // reg x7 = v[j + 1]
        ble   x6, x7, exit2  // go to exit2 if x6 ≤ x7
             . . .
             (body of second for loop)
             . . .
        addi x20, x20, -1    // j -= 1
        j    for2tst         // branch 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 x1, swap
```

Passing Parameters in `sort`

The problem comes when we want to pass parameters because the `sort` procedure needs the values in registers `x10` and `x11`, 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 `x10` and `x11` available for the call of `swap`. (This copy is faster than saving and restoring on the stack.) We first copy `x10` and `x11` into `x21` and `x22` during the procedure:

```
mv x21, x10          // copy parameter x10 into x21
mv x22, x11          // copy parameter x11 into x22
```

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

```
mv x10, x21          // first swap parameter is v
mv x11, x20          // 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 `x1`, since `sort` is a procedure and is itself called. The `sort` procedure also uses the callee-saved registers `x19`, `x20`, `x21`, and `x22`, so they must be saved. The prologue of the `sort` procedure is then

```
addi    sp, sp, -40      // make room on stack for 5 regs
sd      x1, 32(sp)       // save x1 on stack
sd      x22, 24(sp)      // save x22 on stack
sd      x21, 16(sp)      // save x21 on stack
```

```
sd      x20, 8(sp)      // save x20 on stack
sd      x19, 0(sp)      // save x19 on stack
```

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

The Full Procedure `sort`

Now we put all the pieces together in Figure 2.25, being careful to replace references to registers `x10` and `x11` in the *for* loops with references to registers `x21` and `x22`. Once again, to make the code easier to follow, we identify each block of code with

Saving registers			
	sort:	addi sp, sp, -40	# make room on stack for 5 registers
		sd x1, 32(sp)	# save return address on stack
		sd x22, 24(sp)	# save x22 on stack
		sd x21, 16(sp)	# save x21 on stack
		sd x20, 8(sp)	# save x20 on stack
		sd x19, 0(sp)	# save x19 on stack

Procedure body			
Move parameters		mv x21, x10	# copy parameter x10 into x21
		mv x22, x11	# copy parameter x11 into x22
Outer loop		li x19, 0	# i = 0
	for1tst:	bge x19, x22, exit1	# go to exit1 if i >= n
Inner loop		addi x20, x19, -1	# j = i - 1
	for2tst:	blt x20, x0, exit2	# go to exit2 if j < 0
		slli x5, x20, 3	# x5 = j * 8
		add x5, x21, x5	# x5 = v + (j * 8)
		ld x6, 0(x5)	# x6 = v[j]
		ld x7, 8(x5)	# x7 = v[j + 1]
		ble x6, x7, exit2	# go to exit2 if x6 < x7
Pass parameters and call		mv x10, x21	# first swap parameter is v
		mv x11, x20	# second swap parameter is j
		jal x1, swap	# call swap
Inner loop		addi x20, x20, -1	j for2tst
		j for2tst	# go to for2tst
Outer loop	exit2:	addi x19, x19, 1	# i += 1
		j for1tst	# go to for1tst

Restoring registers			
	exit1:	ld x19, 0(sp)	# restore x19 from stack
		ld x20, 8(sp)	# restore x20 from stack
		ld x21, 16(sp)	# restore x21 from stack
		ld x22, 24(sp)	# restore x22 from stack
		ld x1, 32(sp)	# restore return address from stack
		addi sp, sp, 40	# restore stack pointer

Procedure return			
		jalr x0, 0(x1)	# return to calling routine

FIGURE 2.25 RISC-V assembly version of procedure `sort` in Figure 2.27.

its purpose in the procedure. In this example, nine lines of the `sort` procedure in C became 34 lines in the RISC-V 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.

Understanding Program Performance

Figure 2.26 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 2.27 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 jalr 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 increase 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).

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 2.26 Comparing performance, instruction count, and CPI using compiler optimization for Bubble Sort. The programs sorted 100,000 32-bit 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	O1	2.37	1.50	1562
	Compiler	O2	2.38	1.50	1555
	Compiler	O3	2.41	1.91	1955
Java	Interpreter	–	0.12	0.05	1050
	JIT compiler	–	2.13	0.29	338

FIGURE 2.27 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 2.29. 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 RISC-V assembly versions of two procedures to clear a sequence of doublewords in memory: one using array indices and one with pointers. Figure 2.28 shows the two C procedures.

The purpose of this section is to show how pointers map into RISC-V 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.

```
clear1(long long int array[], size_t int size)
{
   size_t i;
   for (i = 0; i < size; i += 1)
        array[i] = 0;
}
clear2(long long int *array, size_t int size)
{
   long long int *p;
   for (p = &array[0]; p < &array[size]; p = p + 1)
        *p = 0;
}
```

FIGURE 2.28 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 bottom 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 RISC-V instructions to increment `p` by eight, the number of bytes in an RISC-V integer. The assignment in the loop places 0 in the object pointed to by `p`.

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 `x10` and `x11`, and that `i` is allocated to register `x5`.

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

```
li    x5, 0    // i = 0 (register x5 = 0)
```

To set `array[i]` to 0 we must first get its address. Start by multiplying `i` by 8 to get the byte address:

```
loop1: slli    x6, x5, 3    // x6 = i * 8
```

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  x7, x10, x6    // x7 = address of array[i]
```

Finally, we can store 0 in that address:

```
sd  x0, 0(x7)    // array[i] = 0
```

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

```
addi x5, x5, 1    // i = i + 1
```

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

```
blt  x5, x11, loop1 // if (i < size) go to loop1
```

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

```
       li     x5, 0           // i = 0
loop1: slli   x6, x5, 3       // x6 = i * 8
       add    x7, x10, x6     // x7 = address of array[i]
       sd     x0, 0(x7)       // array[i] = 0
       addi   x5, x5, 1       // i = i + 1
       blt    x5, x11, 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 `x10` and `x11` and allocates `p` to register `x5`. The code for the

second procedure starts with assigning the pointer `p` to the address of the first element of the array:

```
mv    x5, x10    // p = address of array[0]
```

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

```
loop2: sd    x0, 0(x5)    // 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 doubleword:

```
addi    x5, x5, 8    // p = p + 8
```

Incrementing a pointer by 1 means moving the pointer to the next sequential object in C. Since `p` is a pointer to integers declared as `long long int`, each of which uses 8 bytes, the compiler increments `p` by 8.

The loop test is next. The first step is calculating the address of the last element of `array`. Start with multiplying `size` by 8 to get its byte address:

```
slli    x6, x11, 3    // x6 = size * 8
```

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

```
add    x7, x10, x6      // x7 = address of array[size]
```

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

```
bltu x5, x7, 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:

```
            mv   x5, x10          // p = address of array[0]
loop2:      sd   x0, 0(x5)        // Memory[p] = 0
            addi x5, x5, 8        // p = p + 8
            slli x6, x11, 3       // x6 = size * 8
            add  x7, x10, x6      // x7 = address of array[size]
            bltu x5, x7, 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:

```
            mv    x5, x10         // p = address of array[0]
            slli  x6, x11, 3      // x6 = size * 8
            add   x7, x10, x6     // x7 = address of array[size]
loop2:      sd    x0, 0(x5)       // Memory[p] = 0
            addi  x5, x5, 8       // p = p + 8
            bltu  x5, x7, 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):

```
      li   x5, 0           // i = 0
loop1: slli x6, x5, 3      // x6 = i * 8
      add  x7, x10, x6     // x7 = address of array[i]
      sd   x0, 0(x7)       // array[i] = 0
      addi x5, x5, 1       // i = i + 1
      blt  x5, x11, loop1  // if (i < size) go to loop1
```

```
      mv   x5, x10         // p = address of array[0]
      slli x6, x11, 3      // x6 = size * 8
      add  x7, x10, x6     // x7 = address of array[size]
loop2: sd   x0, 0(x5)      // Memory[p] = 0
      addi x5, x5, 8       // p = p + 8
      bltu x5, x7, loop2   // if (p < &array[size]) go to loop2
```

The version on the left must have the "multiply" and add inside the loop because `i` is incremented and each address must be recalculated from the new index. The memory pointer version on the right increments the pointer `p` directly. The pointer version moves the scaling shift and the array bound addition outside the loop, thereby reducing the instructions executed per iteration from five to three. 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 branch to the instruction after the loop with `blt x0, x11, afterLoop`.

Understanding Program Performance

People were once 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.



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



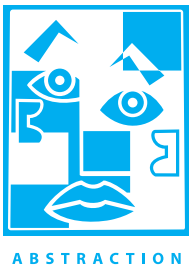
Advanced Material: Compiling C and Interpreting Java

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, starting on page 150.e15, is for readers interested in seeing how an objected-oriented language like Java executes on the RISC-V architecture. It shows the Java bytecodes used for interpretation and the RISC-V 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 just-in-time (JIT) compilers.

Compiling C

This first part of the section introduces the internal **anatomy** of a compiler. To start, Figure e2.15.1 shows the structure of recent compilers, and we describe the optimizations in the order of the passes of that structure.



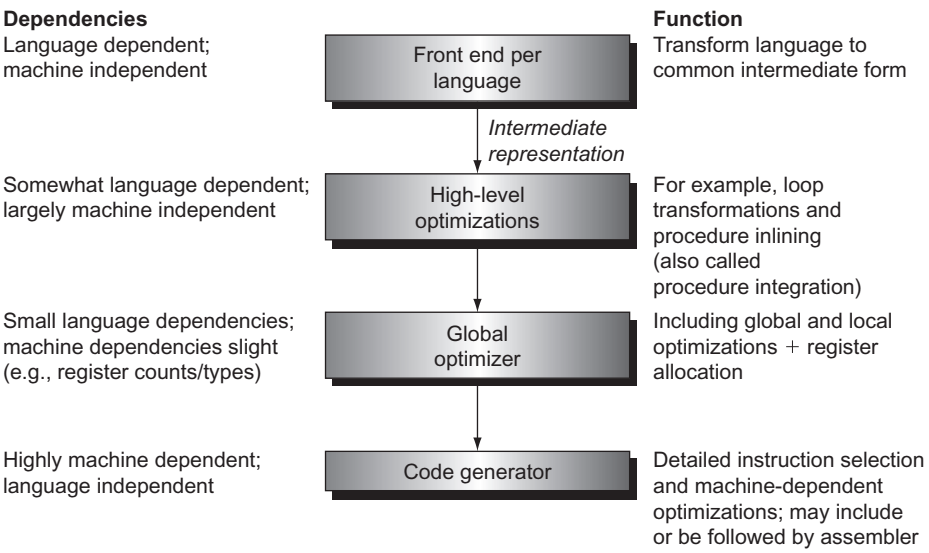


FIGURE e2.15.1 The structure of a modern optimizing compiler consists of a number of passes or phases. Logically, each pass can be thought of as running to completion before the next occurs. In practice, some passes may handle one procedure at a time, essentially interleaving with another pass.

To illustrate the concepts in this part of this section, we will use the C version of a *while* loop from page 95:

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

The Front End

The function of the front end is to read in a source program; check the syntax and semantics; and translate the source program to an intermediate form that interprets most of the language-specific operation of the program. As we will see, intermediate forms are usually simple, and some are, in fact, similar to the Java bytecodes (see Figure e2.15.8).

The front end is typically broken into four separate functions:

1. *Scanning* reads in individual characters and creates a string of tokens. Examples of *tokens* are reserved words, names, operators, and punctuation symbols. In the above example, the token sequence is `while`, `(`, `save`, `[`, `i`, `]`, `==`, `k`, `)`, `i`, `+=`, `1`. A word like `while` is recognized as a reserved word in C, but `save`, `i`, and `j` are recognized as names, and `1` is recognized as a number.

2. *Parsing* takes the token stream, ensures the syntax is correct, and produces an *abstract syntax tree*, which is a representation of the syntactic structure of the program. Figure e2.15.2 shows what the abstract syntax tree might look like for this program fragment.

3. *Semantic analysis* takes the abstract syntax tree and checks the program for semantic correctness. Semantic checks normally ensure that variables and types are properly declared and that the types of operators and objects match, a step called *type checking*. During this process, a symbol table representing all the named objects—classes, variables, and functions—is usually created and used to type-check the program.

4. *Generation of the intermediate representation* (IR) takes the symbol table and the abstract syntax tree and generates the intermediate representation that is the output of the front end. Intermediate representations usually use simple operations on a small set of primitive types, such as integers, characters, and reals. Java bytecodes represent one type of intermediate form. In modern compilers, the most common intermediate form looks much like the RISC-V instruction set but with an infinite number of virtual registers; later, we describe how to map these virtual registers to a finite set of real registers. Figure e2.15.3 shows how our example might be represented in such an intermediate form.

The intermediate form specifies the functionality of the program in a manner independent of the original source. After this front end has created the intermediate form, the remaining passes are largely language independent.

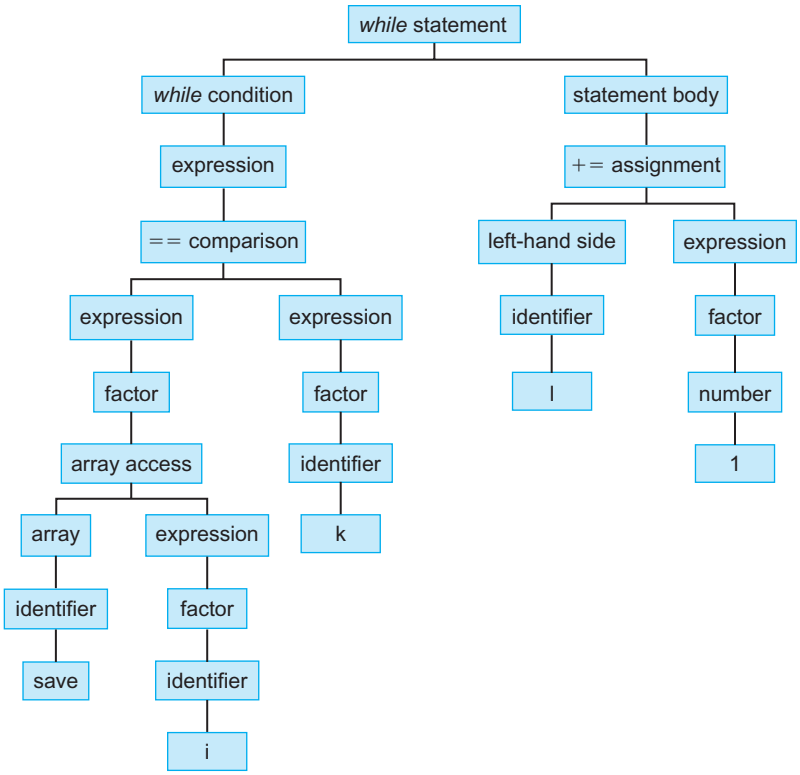


FIGURE e2.15.2 An abstract syntax tree for the *while* example. The roots of the tree consist of the informational tokens such as numbers and names. Long chains of straight-line descendents are often omitted in constructing the tree.

High-Level Optimizations

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

The most common high-level transformation is probably *procedure inlining*, which replaces a call to a function by the body of the function, substituting the caller's arguments for the procedure's parameters. Other high-level optimizations involve loop transformations that can reduce loop overhead, improve memory access, and exploit the hardware more effectively. For example, in loops that execute many iterations, such as those traditionally controlled by a *for* statement, the optimization of **loop-unrolling** is often useful. Loop-unrolling involves taking a loop, replicating the body multiple times, and executing the transformed loop fewer times. Loop-unrolling reduces the loop overhead and provides opportunities for many other optimizations. Other types of high-level transformations include

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

```
loop:
  # comments are written like this--source code often included
  # while (save[i] == k)
  la   r100, save          # r100 = &save[0]
  ld   r101, i
  li   r102, 8
  mul  r103, r101, r102
  add  r104, r103, r100
  ld   r105, 0(r104)       # r105 = save[i]
  ld   r106, k
  bne  r105, r106, exit
  # i += 1
  ld   r106, i
  addi r107, r106, i       # increment
  sd   r107, i
  j    loop                # next iteration
exit:
```

FIGURE e2.15.3 The *while* loop example is shown using a typical intermediate representation. In practice, the names `save`, `i`, and `k` would be replaced by some sort of address, such as a reference to either the local stack pointer or a global pointer, and an offset, similar to the way `save[i]` is accessed. Note that the format of the RISC-V instructions is different from the rest of the chapter, because they represent intermediate representations here using `rXX` notation for virtual registers.

sophisticated loop transformations such as interchanging nested loops and blocking loops to obtain better memory behavior; see Chapter 5 for examples.

Local and Global Optimizations

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

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

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

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

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

The address calculation for `x[i]` occurs twice and is identical since neither the starting address of `x` nor the value of `i` changes. Thus, the calculation can be reused. Let's look at the intermediate code for this fragment, since it allows several other optimizations to be performed. The unoptimized intermediate code is on the left. On the right is the optimized code, using common subexpression elimination to replace the second address calculation with the first. Note that the register allocation has not yet occurred, so the compiler is using virtual register numbers like `r100` here.

```
// x[i] + 4
la r100,x
ld r101,i
mul r102,r101,8
add r103,r100,r102
ld r104, 0(r103)
//
addi r105, r104,4
la r106,x
ld r107,i
mul r108,r107,8
add r109,r106,r107
sd r105,0(r109)
```

```
// x[i] + 4
la r100,x
ld r101,i
slli r102,r101,3
add r103,r100,r102
ld r104, 0(r103)
// value of x[i] is in r104
addi r105, r104,4
sd r105, 0(r103)
```

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

Let's consider some of the other optimizations:

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

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

Understanding Program Performance

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

Global Code Optimizations

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

There are two other important global optimizations: code motion and induction variable elimination. Both are loop optimizations; that is, they are aimed at code in loops. *Code motion* finds code that is loop invariant: a particular piece of code computes the same value on every iteration of the loop and, hence, may be computed once outside the loop. *Induction variable elimination* is a combination of transformations that reduce overhead on indexing arrays, essentially replacing array indexing with pointer accesses. Rather than examine induction variable elimination in depth, we point the reader to Section 2.14, which compares the use of array indexing and pointers; for most loops, a modern optimizing compiler can perform the transformation from the more obvious array code to the faster pointer code.

Implementing Local Optimizations

Local optimizations are implemented on basic blocks by scanning the basic block in instruction execution order, looking for optimization opportunities. In the assignment statement example on page 150.e6, the duplication of the entire address calculation is recognized by a series of sequential passes over the code. Here is how the process might proceed, including a description of the checks that are needed:

1. Determine that the two `LDA` operations return the same result by observing that the operand `x` is the same and that the value of its address has not been changed between the two `LDA` operations.
2. Replace all uses of `R106` in the basic block by `R101`.
3. Observe that `i` cannot change between the two `LDURs` that reference it. So replace all uses of `R107` with `R101`.
4. Observe that the `MUL` instructions now have the same input operands, so that `R108` may be replaced by `R102`.
5. Observe that now the two `ADD` instructions have identical input operands (`R100` and `R102`), so replace the `R109` with `R103`.
6. Use dead store code elimination to delete the second set of `LDA`, `LDUR`, `MUL`, and `ADD` instructions since their results are unused.

Throughout this process, we need to know when two instances of an operand have the same value. This is easy to determine when they refer to virtual registers, since our intermediate representation uses such registers only once, but the problem can be trickier when the operands are variables in memory, even though we are only considering references within a basic block.

It is reasonably easy for the compiler to make the common subexpression elimination determination in a conservative fashion in this case; as we will see in the next subsection, this is more difficult when branches intervene.

Implementing Global Optimizations

To understand the challenge of implementing global optimizations, let's consider a few examples:

- Consider the case of an opportunity for common subexpression elimination, say, of an IR statement like `ADD Rx, R20, R50`. To determine whether two such statements compute the same value, we must determine whether the values of `R20` and `R50` are identical in the two statements. In practice, this means that the values of `R20` and `R50` have not changed between the first statement and the second. For a single basic block, this is easy to decide; it is more difficult for a more complex program structure involving multiple basic blocks and branches.
- Consider the second `LDUR` of `i` into `R107` within the earlier example: how do we know whether its value is used again? If we consider only a single basic

block, and we know that all uses of R107 are within that block, it is easy to see. As optimization proceeds, however, common subexpression elimination and copy propagation may create other uses of a value. Determining that a value is unused and the code is dead is more difficult in the case of multiple basic blocks.

- Finally, consider the load of k in our loop, which is a candidate for code motion. In this simple example, we might argue that it is easy to see that k is not changed in the loop and is, hence, loop invariant. Imagine, however, a more complex loop with multiple nestings and *if* statements within the body. Determining that the load of k is loop invariant is harder in such a case.

The information we need to perform these global optimizations is similar: we need to know where each operand in an IR statement could have been changed or *defined* (use-definition information). The dual of this information is also needed: that is, finding all the uses of that changed operand (definition-use information). *Data flow analysis* obtains both types of information.

Global optimizations and data flow analysis operate on a *control flow graph*, where the nodes represent basic blocks and the arcs represent control flow between basic blocks. Figure e2.15.4 shows the control flow graph for our simple loop example, with one important transformation introduced. We describe the transformation in the caption, but see if you can discover it, and why it was done, on your own!

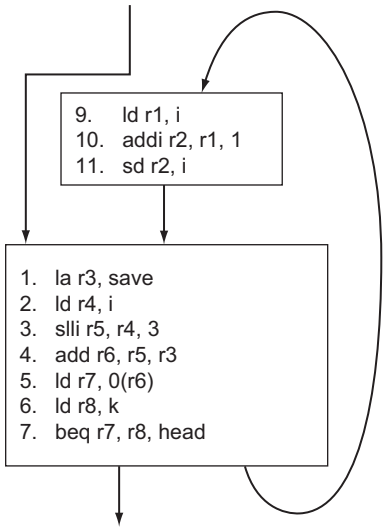


FIGURE e2.15.4 A control flow graph for the *while* loop example. Each node represents a basic block, which terminates with a branch or by sequential fall-through into another basic block that is also the target of a branch. The IR statements have been numbered for ease in referring to them. The important transformation performed was to move the *while* test and conditional branch to the end. This eliminates the unconditional branch that was formerly inside the loop and places it before the loop. This transformation is so important that many compilers do it during the generation of the IR. The mul was also replaced with ("strength-reduced to") an slli.

Suppose we have computed the use-definition information for the control flow graph in Figure e2.15.4. How does this information allow us to perform code motion? Consider IR statements number 1 and 6: in both cases, the use-definition information tells us that there are no definitions (changes) of the operands of these statements within the loop. Thus, these IR statements can be moved outside the loop. Notice that if the `LDA` of `save` and the `LDUR` of `k` are executed once, just prior to the loop entrance, the computational effect is the same, but the program now runs faster since these two statements are outside the loop. In contrast, consider IR statement 2, which loads the value of `i`. The definitions of `i` that affect this statement are both outside the loop, where `i` is initially defined, and inside the loop in statement 10 where it is stored. Hence, this statement is not loop invariant.

Figure e2.15.5 shows the code after performing both code motion and induction variable elimination, which simplifies the address calculation. The variable `i` can still be register allocated, eliminating the need to load and store it every time, and we will see how this is done in the next subsection.

Before we turn to register allocation, we need to mention a caveat that also illustrates the complexity and difficulty of optimizers. Remember that the compiler must be cautious. To be conservative, a compiler must consider the following question: Is there *any way* that the variable `k` could possibly ever change in this loop? Unfortunately, there is one way. Suppose that the variable `k` and the variable `i` actually refer to the same memory location, which could happen if they were accessed by pointers or reference parameters.

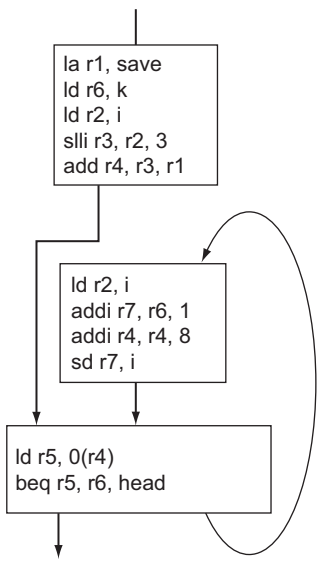


FIGURE e2.15.5 The control flow graph showing the representation of the *while* loop example after code motion and induction variable elimination. The number of instructions in the inner loop has been reduced from 10 to 6.

I am sure that many readers are saying, "Well, that would certainly be a stupid piece of code!" Alas, this response is not open to the compiler, which must translate the code as it is written. Recall too that the aliasing information must also be conservative; thus, compilers often find themselves negating optimization opportunities because of a possible alias that exists in one place in the code or because of incomplete information about aliasing.

Register Allocation

Register allocation is perhaps the most important optimization for modern load-store architectures. Eliminating a load or a store gets rid of an instruction. Furthermore, register allocation enhances the value of other optimizations, such as common subexpression elimination. Fortunately, the trend toward larger register counts in modern architectures has made register allocation simpler and more effective. Register allocation is done on both a local basis and a global basis, that is, across multiple basic blocks but within a single function. Local register allocation is usually done late in compilation, as the final code is generated. Our focus here is on the more challenging and more opportunistic global register allocation.

Modern global register allocation uses a region-based approach, where a region (sometimes called a *live range*) represents a section of code during which a particular variable could be allocated to a particular register. How is a region selected? The process is iterative:

1. Choose a definition (change) of a variable in a given basic block; add that block to the region.
2. Find any uses of that definition, which is a data flow analysis problem; add any basic blocks that contain such uses, as well as any basic block that the value passes through to reach a use, to the region.
3. Find any other definitions that also can affect a use found in the previous step and add the basic blocks containing those definitions, as well as the blocks the definitions pass through to reach a use, to the region.
4. Repeat steps 2 and 3 using the definitions discovered in step 3 until convergence.

The set of basic blocks found by this technique has a special property: if the designated variable is allocated to a register in all these basic blocks, then there is no need for loading and storing the variable.

Modern global register allocators start by constructing the regions for every virtual register in a function. Once the regions are constructed, the key question is how to allocate a register to each region: the challenge is that certain regions overlap and may not use the same register. Regions that do not overlap (i.e., share no common basic blocks) can share the same register. One way to record the interference among regions is with an *interference graph*, where each node represents a region, and the arcs between nodes represent that the regions have some basic blocks in common.

Once an interference graph has been constructed, the problem of allocating registers is equivalent to a famous problem called *graph coloring:* find a color for each node in a graph such that no two adjacent nodes have the same color. If the number of colors equals the number of registers, then coloring an interference graph is equivalent to allocating a register for each region! This insight was the initial motivation for the allocation method now known as region-based allocation, but originally called the graph-coloring approach. Figure e2.15.6 shows the flow graph representation of the *while* loop example after register allocation.

What happens if the graph cannot be colored using the number of registers available? The allocator must spill registers until it can complete the coloring. By doing the coloring based on a priority function that takes into account the number of memory references saved and the cost of tying up the register, the allocator attempts to avoid spilling for the most important candidates.

Spilling is equivalent to splitting up a region (or live range); if the region is split, fewer other regions will interfere with the two separate nodes representing the original region. A process of splitting regions and successive coloring is used to allow the allocation process to complete, at which point all candidates will have been allocated a register. Of course, whenever a region is split, loads and stores must be introduced to get the value from memory or to store it there. The location chosen to split a region must balance the cost of the loads and stores that must be introduced against the advantage of freeing up a register and reducing the number of interferences.

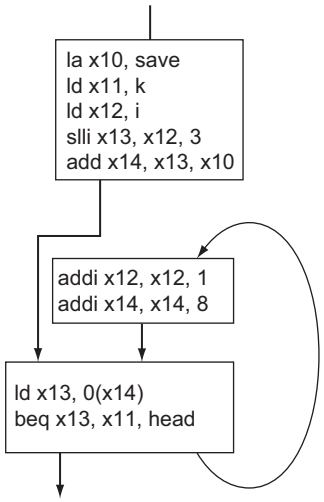


FIGURE e2.15.6 The control flow graph showing the representation of the *while* loop example after code motion and induction variable elimination and register allocation, using the RISC-V register names. The number of IR statements in the inner loop has now dropped to only four from six before register allocation and 10 before any global optimizations. The value of `i` resides in `x12` at the end of the loop and may need to be stored eventually to maintain the program semantics. If `i` were unused after the loop, not only could the store be avoided, but also the increment inside the loop could be eliminated!

Modern register allocators are incredibly effective in using the large register counts available in modern processors. In many programs, the effectiveness of register allocation is limited not by the availability of registers but by the possibilities of aliasing that cause the compiler to be conservative in its choice of candidates.

Code Generation

The final steps of the compiler are code generation and assembly. Most compilers do not use a stand-alone assembler that accepts assembly language source code; to save time, they instead perform most of the same functions: filling in symbolic values and generating the binary code as the last stage of code generation.

In modern processors, code generation is reasonably straightforward, since the simple architectures make the choice of instruction relatively obvious. Code generation is more complex for the more complicated architectures, such as the x86, since multiple IR instructions may collapse into a single machine instruction. In modern compilers, this compilation process uses pattern matching with either a tree-based pattern matcher or a pattern matcher driven by a parser.

During code generation, the final stages of machine-dependent optimization are also performed. These include some constant folding optimizations, as well as localized instruction scheduling (see Chapter 4).

Optimization Summary

Figure e2.15.7 gives examples of typical optimizations, and the last column indicates where the optimization is performed in the gcc compiler. It is sometimes difficult to separate some of the simpler optimizations—local and processor-dependent optimizations—from transformations done in the code generator, and some optimizations are done multiple times, especially local optimizations, which may be performed before and after global optimization as well as during code generation.

Hardware/ Software Interface



Today, essentially all programming for desktop and server applications is done in high-level languages, as is most programming for embedded applications. This development means that since most instructions executed are the output of a compiler, an instruction set architecture is mainly a compiler target. With **Moore's Law** comes the temptation of adding sophisticated operations in an instruction set. The challenge is that they may not exactly match what the compiler needs to produce or may be so general that they aren't fast. For example, consider special loop instructions found in some computers. Suppose that instead of decrementing by one, the compiler wanted to increment by four, or instead of branching on not equal zero, the compiler wanted to branch if the index was less than or equal to the limit. The loop instruction may be a mismatch. When faced with such objections,

the instruction set designer might next generalize the operation, adding another operand to specify the increment and perhaps an option on which branch condition to use. Then the danger is that a common case, say, incrementing by one, will be slower than a sequence of simple operations.

Elaboration Some more sophisticated compilers, and many research compilers, use an analysis technique called *interprocedural analysis* to obtain more information about functions and how they are called. Interprocedural analysis attempts to discover what properties remain true across a function call. For example, we might discover that a function call can never change any global variables, which might be useful in optimizing a loop that calls such a function. Such information is called *may-information* or *flow-insensitive information* and can be obtained reasonably efficiently, although analyzing a call to a function F requires analyzing all the functions that F calls, which makes the process somewhat time consuming for large programs. A more costly property to discover is that a function *must* always change some variable; such information is called *must-information* or *flow-sensitive information*. Recall the dictate to be conservative: may-information can never be used as must-information—just because a function *may* change a variable does not mean that it *must* change it. It is conservative, however, to use the negation of may-information, so the compiler can rely on the fact that a function *will* never change a variable in optimizations around the call site of that function.

Optimization name	Explanation	gcc level
High level	*At or near the source level; processor independent*	
Procedure integration	Replace procedure call by procedure body	O3
Local	*Within straight-line code*	
Common subexpression elimination	Replace two instances of the same computation by single copy	O1
Constant propagation	Replace all instances of a variable that is assigned a constant with the constant	O1
Stack height reduction	Rearrange expression tree to minimize resources needed for expression evaluation	O1
Global	*Across a branch*	
Global common subexpression elimination	Same as local, but this version crosses branches	O2
Copy propagation	Replace all instances of a variable *A* that has been assigned *X* (i.e., *A* = *X*) with *X*	O2
Code motion	Remove code from a loop that computes the same value each iteration of the loop	O2
Induction variable elimination	Simplify/eliminate array addressing calculations within loops	O2
Processor dependent	*Depends on processor knowledge*	
Strength reduction	Many examples; replace multiply by a constant with shifts	O1
Pipeline scheduling	Reorder instructions to improve pipeline performance	O1
Branch offset optimization	Choose the shortest branch displacement that reaches target	O1

FIGURE e2.15.7 Major types of optimizations and explanation of each class. The third column shows when these occur at different levels of optimization in gcc. The GNU organization calls the three optimization levels medium (O1), full (O2), and full with integration of small procedures (O3).

One of the most important uses of interprocedural analysis is to obtain so-called alias information. An *alias* occurs when two names may designate the same variable. For example, it is quite helpful to know that two pointers passed to a function may never designate the same variable. Alias information is usually flow-insensitive and must be used conservatively.

Interpreting Java

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

This second part of the section is for readers interested in seeing how an **object-oriented language** like Java executes on an RISC-V architecture. It shows the Java bytecodes used for interpretation and the RISC-V code for the Java version of some of the C segments in prior sections, including Bubble Sort.

Let's quickly review the Java lingo to make sure we are all on the same page. The big idea of object-oriented programming is for programmers to think in terms of abstract objects, and operations are associated with each *type* of object. New types can often be thought of as refinements to existing types, and so the new types use some operations for the existing types without change. The hope is that the programmer thinks at a higher level, and that code can be reused more readily if the programmer implements the common operations on many different types.

This different perspective led to a different set of terms. The type of an object is a *class*, which is the definition of a new data type together with the operations that are defined to work on that data type. A particular object is then an *instance* of a class, and creating an object from a class is called *instantiation*. The operations in a class are called *methods*, which are similar to C procedures. Rather than call a procedure as in C, you *invoke* a method in Java. The other members of a class are *fields*, which correspond to variables in C. Variables inside objects are called *instance fields*. Rather than access a structure with a pointer, Java uses an *object reference* to access an object. The syntax for method invocation is `x.y`, where `x` is an object reference and `y` is the method name.

The parent–child relationship between older and newer classes is captured by the verb "extends": a child class *extends* (or subclasses) a parent class. The child class typically will redefine some of the methods found in the parent to match the new data type. Some methods work fine, and the child class *inherits* those methods.

To reduce the number of errors associated with pointers and explicit memory deallocation, Java automatically frees unused storage, using a separate garbage collector that frees memory when it is full. Hence, `new` creates a new instance of a dynamic object on the heap, but there is no `free` in Java. Java also requires array bounds to be checked at runtime to catch another class of errors that can occur in C programs.

Interpretation

As mentioned before, Java programs are distributed as Java bytecodes, and the Java Virtual Machine (JVM) executes Java byte codes. The JVM understands a binary format called the *class file* format. A class file is a stream of bytes for a single class, containing a table of valid methods with their bytecodes, a pool of constants that acts in part as a symbol table, and other information such as the parent class of this class.

When the JVM is first started, it looks for the class method `main`. To start any Java class, the JVM dynamically loads, links, and initializes a class. The JVM loads a class by first finding the binary representation of the proper class (class file) and then creating a class from that binary representation. Linking combines the class into the runtime state of the JVM so that it can be executed. Finally, it executes the class initialization method that is included in every class.

Figure e2.15.8 shows Java bytecodes and their corresponding RISC-V instructions, illustrating five major differences between the two:

1. To simplify compilation, Java uses a stack instead of registers for operands. Operands are pushed on the stack, operated on, and then popped off the stack.
2. The designers of the JVM were concerned about code size, so bytecodes vary in length between one and five bytes, versus the four-byte, fixed-size RISC-V instructions. To save space, the JVM even has redundant instructions of varying lengths whose only difference is size of the immediate. This decision illustrates a code size variation of our third design principle: make the common case *small*.
3. The JVM has safety features embedded in the architecture. For example, array data transfer instructions check to be sure that the first operand is a reference and that the second index operand is within bounds.
4. To allow garbage collectors to find all live pointers, the JVM uses different instructions to operate on addresses versus integers so that the JVM can know what operands contain addresses. RISC-V generally lumps integers and addresses together.
5. Finally, unlike RISC-V, Java bytecodes include Java-specific instructions that perform complex operations, like allocating an array on the heap or invoking a method.

Category	Operation	Java bytecode	Size (bits)	RISC-V instr.	Meaning
Arithmetic	add	iadd	8	add	NOS=TOS+NOS; pop
	subtract	isub	8	sub	NOS=TOS–NOS; pop
	increment	iinc I8a I8b	8	addi	Frame[I8a]= Frame[I8a] + I8b
Data transfer	load local integer/address	iload I8/aload I 8	16	ld	TOS=Frame[I8]
	load local integer/addres s	iload_/aload_{0,1,2,3 }	8	ld	TOS= Frame[{0,1,2,3}]
	store local integer/addres s	istore I8/astore I 8	16	sd	Frame[I8]=TOS; pop
	load integer/address from a rray	iaload/aaloa d	8	ld	NOS=*NOS[TOS]; pop
	store integer/address into a rray	iastore/aastor e	8	sd	*NNOS[NOS]=TOS; pop2
	load half from array	saload	8	lh	NOS=*NOS[TOS]; pop
	store half into array	sastore	8	sh	*NNOS[NOS]=TOS; pop2
	load byte from array	baload	8	lb	NOS=*NOS[TOS]; pop
	store byte into array	bastore	8	sb	*NNOS[NOS]=TOS; pop2
	load immediat e	bipush I8 , sipush I1 6	16, 24	addi	push; TOS=I8 or I16
	load immediat e	iconst_{–1,0,1,2,3,4,5 }	8	addi	push; TOS={–1,0,1,2,3,4,5}
Logical	and	iand	8	and	NOS=TOS&NOS; pop
	or	ior	8	or	NOS=TOS |NOS; pop
	shift left	ishl	8	sll	NOS=NOS << TOS; pop
	shift right	iushr	8	srl	NOS=NOS >> TOS; pop
Conditional branch	branch on equa l	if_icompeq I1 6	24	beq	if TOS == NOS , go to I16; pop2
	branch on not equa l	if_icompne I1 6	24	bne	if TOS != NOS , go to I16; pop2
	compare	if_icomp{lt,le,gt,ge} I16	24	blt/bge	if TOS {<,<=,>,>=} NOS, go to I16; pop2
Unconditional jump	jump	goto I16	24	jal	go to I16
	return	ret, ireturn	8	jalr	
	jump to subroutine	jsr I16	24	jal	go to I16; push; TOS=PC+3
Stack management	remove from stack	pop, pop2	8		pop, pop2
	duplicate on stack	dup	8		push; TOS=NOS
	swap top 2 positions on stack	swap	8		T=NOS; NOS=TOS; TOS=T
Safety check	check for null reference	ifnull I16, ifnotnull I16	24		if TOS {==,!=} null, go to I16
	get length of array	arraylength	8		push; TOS = length of array
	check if object a type	instanceof I16	24		TOS = 1 if TOS matches type of Const[I16]; TOS = 0 otherwise
Invocation	invoke metho d	invokevirtual I1 6	24		Invoke method in Const[I16] , dispatching on type
Allocation	create new class instanc e	new I16	24		Allocate object type Const[I16] on heap
	create new array	newarray I16	24		Allocate array type Const[I16] on heap

FIGURE e2.15.8 Java bytecode architecture versus RISC-V. Although many bytecodes are simple, those in the last half-dozen rows above are complex and specific to Java. Bytecodes are one to five bytes in length, hence their name. The Java mnemonics uses the prefix `i` for 32-bit integer, `a` for reference (address), `s` for 16-bit integers (short), and `b` for 8-bit bytes. We use `I8` for an 8-bit constant and `I16` for a 16-bit constant. RISC-V uses registers for operands, but the JVM uses a stack. The compiler knows the maximum size of the operand stack for each method and simply allocates space for it in the current frame. Here is the notation in the Meaning column: `TOS`: top of stack; `NOS`: next position below `TOS`; `NNOS`: next position below `NOS`; `pop`: remove `TOS`; `pop2`: remove `TOS` and `NOS`; and `push`: add a position to the stack. `*NOS` and `*NNOS` mean access the memory location pointed to by the address in the stack at those positions. `Const[]` refers to the runtime constant pool of a class created by the JVM, and `Frame[]` refers to the variables of the local method frame. The missing Java bytecodes from Figure e2.1 are a few arithmetic and logical operators, some tricky stack management, compares to 0 and branch, support for branch tables, type conversions, more variations of the complex, Java-specific instructions plus operations on floating-point data, 64-bit integers (longs), and 16-bit characters.

Compiling a *while* Loop in Java Using Bytecodes

EXAMPLE

Compile the *while* loop from page 95, this time using Java bytecodes:

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

Assume that `i`, `k`, and `save` are the first three local variables. Show the addresses of the bytecodes. The RISC-V version of the C loop in Figure e2.15.3 took six instructions and 24 bytes. How big is the bytecode version?

ANSWER

The first step is to put the array reference in `save` on the stack:

```
0 aload_3 // Push local variable 3 (save[]) onto stack
```

This 1-byte instruction informs the JVM that an address in local variable 3 is being put on the stack. The 0 on the left of this instruction is the byte address of this first instruction; bytecodes for each method start at 0. The next step is to put the index on the stack:

```
1 iload_1 // Push local variable 1 (i) onto stack
```

Like the prior instruction, this 1-byte instruction is a short version of a more general instruction that takes 2 bytes to load a local variable onto the stack. The next instruction is to get the value from the array element:

```
2 iaload // Put array element (save[i]) onto stack
```

This 1-byte instruction checks the prior two operands, pops them off the stack, and then puts the value of the desired array element onto the new top of the stack. Next, we place `k` on the stack:

```
3 iload_2 // Push local variable 2 (k) onto stack
```

We are now ready for the *while* test:

```
4 if_icompne, Exit // Compare and exit if not equal
```

This 3-byte instruction compares the top two elements of the stack, pops them off the stack, and branches if they are not equal. We are finally prepared for the body of the loop:

```
7 iinc, 1, 1 // Increment local variable 1 by 1 (i+=1)
```

This unusual 3-byte instruction increments a local variable by 1 without using the operand stack, an optimization that again saves space. Finally, we return to the top of the loop with a 3-byte branch:

```
10 go to 0 // Go to top of Loop (byte address 0)
```

Thus, the bytecode version takes seven instructions and 13 bytes, just over half the size of the RISC-V C code. (As before, we can optimize this code to branch less.)

Compiling for Java

Since Java is derived from C and Java has the same built-in types as C, the assignment statement examples in Sections 2.2 to 2.6 are the same in Java as they are in C. The same is true for the *if* statement example in Section 2.7.

The Java version of the *while* loop is different, however. The designers of C leave it up to the programmers to be sure that their code does not exceed the array bounds. The designers of Java wanted to catch array bound bugs, and thus require the compiler to check for such violations. To check bounds, the compiler needs to know what they are. Java includes an extra doubleword in every array that holds the upper bound. The lower bound is defined as 0.

EXAMPLE

Compiling a *while* Loop in Java

Modify the RISC-V code for the *while* loop on page 95 to include the array bounds checks that are required by Java. Assume that the length of the array is located just before the first element of the array.

ANSWER

Let's assume that Java arrays reserved the first two doublewords of arrays before the data start. We'll see the use of the first doubleword soon, but the second doubleword has the array length. Before we enter the loop, let's load the length of the array into a temporary register:

```
ld  x5, 8(x25)    // Temp reg x5 = length of array save
```

Before we multiply `i` by 8, we must test to see if it's less than 0 or greater than the last element of the array. The first step is to check if `i` is less than 0:

```
Loop: blt  x22, x0, IndexOutOfBounds  // if i<0, goto Error
```

Since the array starts at 0, the index of the last array element is one less than the length of the array. Thus, the test of the upper array bound is to be sure that `i` is

less than the length of the array. Thus, the second step is to branch to an error if it's greater than or equal to `length`.

```
bge  x22, x5, IndexOutOfBounds  //if i>=length, goto Error
```

The next two lines of the RISC-V *while* loop are unchanged from the C version:

```
slli  x10, x22, 3    // Temp reg x10 = i * 8
add   x10, x10, x25  // x10 = address of save[i]
```

We need to account for the first 16 bytes of an array that are reserved in Java. We do that by changing the address field of the load from 0 to 16:

```
ld x9, 16(x10)      // Temp reg x9 = save[i]
```

The rest of the RISC-V code from the C *while* loop is fine as is:

```
bne   x9, x24, Exit  // go to Exit if save[i] ≠ k
addi  x22, x22, 1    // i = i + 1
beq   x0, x0, Loop   // go to Loop
Exit:
```

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

Invoking Methods in Java

The compiler picks the appropriate method depending on the type of object. In a few cases, it is unambiguous, and the method can be invoked with no more overhead than a C procedure. In general, however, the compiler knows only that a given variable contains a pointer to an object that belongs to some subtype of a general class. Since it doesn't know at compile time which subclass the object is, and thus which method should be invoked, the compiler will generate code that first tests to be sure the pointer isn't null and then uses the code to load a pointer to a table with all the legal methods for that type. The first doubleword of the object has the method table address, which is why Java arrays reserve two doublewords. Let's say it's using the fifth method that was declared for that class. (The method order is the same for all subclasses.) The compiler then takes the fifth address from that table and invokes the method at that address.

The cost of object orientation in general is that method invocation takes five steps:

1. A conditional branch to be sure that the pointer to the object is valid;
2. A load to get the address of the table of available methods;
3. Another load to get the address of the proper method;

4. Placing a return address into the return register; and finally
5. A branch register to invoke the method.

A Sort Example in Java

public A Java keyword that allows a method to be invoked by any other method.

protected A Java key word that restricts invocation of a method to other methods in that package.

package Basically a directory that contains a group of related classes.

static method A method that applies to the whole class rather than to an individual object. It is unrelated to static in C.

Figure e2.15.9 shows the Java version of exchange sort. A simple difference is that there is no need to pass the length of the array as a separate parameter, since Java arrays include their length: `v.length` denotes the length of `v`.

A more significant difference is that Java methods are prepended with keywords not found in the C procedures. The `sort` method is declared `public static` while `swap` is declared `protected static`. **Public** means that `sort` can be invoked from any other method, while **protected** means `swap` can only be called by other methods within the same **package** and from methods within derived classes. A **static method** is another name for a class method—methods that perform class-wide operations and do not apply to an individual object. Static methods are essentially the same as C procedures.

This straightforward translation from C into static methods means there is no ambiguity on method invocation, and so it can be just as efficient as C. It also is limited to sorting integers, which means a different sort has to be written for each data type.

To demonstrate the object orientation of Java, Figure e2.15.10 shows the new version with the changes highlighted. First, we declare `v` to be of the type `Comparable` and replace `v[j] > v[j + 1]` with an invocation of `compareTo`. By changing `v` to this new class, we can use this code to `sort` many data types.

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

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

FIGURE e2.15.9 An initial Java procedure that performs a sort on the array v. Changes from Figures e2.24 and e2.26 are highlighted.

```
public class sort {
   public static void sort (Comparable[] v) {
      for (int i = 0; i < v.length; i += 1) {
        for (int j = i - 1; j >= 0 && v[j].compareTo(v[j + 1]);j -= 1) {

                    swap(v, j);
          }
   }

   protected static void swap(Comparable[] v, int k) {
      Comparable temp = v[k];
      v[k] = v[k+1];
      v[k+1] = temp;
   }}
public class Comparable {
      public int(compareTo (int x)
      { return value - x; }
      public int value;
}
```

FIGURE e2.15.10 A revised Java procedure that sorts on the array v that can take on more types. Changes from Figure e2.15.9 are highlighted.

The method `compareTo` compares two elements and returns a value greater than 0 if the parameter is larger than the object, 0 if it is equal, and a negative number if it is smaller than the object. These two changes generalize the code so it can sort integers, characters, strings, and so on, if there are subclasses of `Comparable` with each of these types and if there is a version of `compareTo` for each type. For pedagogic purposes, we redefine the class `Comparable` and the method `compareTo` here to compare integers. The actual definition of `Comparable` in the Java library is considerably different.

Starting from the RISC-V code that we generated for C, we show what changes we made to create the RISC-V code for Java.

For `swap`, the only significant differences are that we must check to be sure the object reference is not null and that each array reference is within bounds. The first test checks that the address in the first parameter is not zero:

```
swap: beq x10, x0, Error  x10, NullPointer  // if X0==0,goto Error
```

Next, we load the length of v into a register and check that index k is OK.

```
ld x5, 8(x10)                    // Temp reg x5 = length of array v
blt x11, x0, IndexOutOfBounds    // if k < 0, goto Error
bge x11, x5, IndexOutOfBounds    // if k >= length, goto Error
```

This check is followed by a check that k+1 is within bounds.

```
addi x6,x11,1                    // Temp reg x6 = k+1
blt x6, x0, IndexOutOfBounds     // if k+1 < 0, goto Error
bge x6, x5, IndexOutOfBounds     // if k+1 >= length, goto Error
```

Figure e2.15.11 highlights the extra RISC-V instructions in swap that a Java compiler might produce. We again must adjust the offset in the load and store to account for two doublewords reserved for the method table and length.

Figure e2.15.12 shows the method body for those new instructions for sort. (We can take the saving, restoring, and return from Figure e2.28.)

The first test is again to make sure the pointer to v is not null:

```
beq  x10, x0, Error    // if x10==0,goto Error
```

Bounds check

```
swap: beq x10, x0, NullPointer          # If x10==0, goto Error
      ld x5, 8(x10)                     # Temp reg x5 = length of array v
      blt x11, x0, IndexOutOfBounds     # If k < 0, goto Error
      bge x11, x5, IndexOutOfBounds     # If k >= length, goto Error
      addi x6, x11, 1                   # Temp reg x6 = k+1
      blt x6, x0, IndexOutOfBounds      # If k+1 < 0, goto Error
      bge x6, x5, IndexOutOfBOunds      # If k+1 >= length, goto Error
```

Method body

```
      slli x11, x11, 3                  # reg X11 = k * 8
      add x11, x11, x10                 # reg X11 = v + (k * 8)

      ld x12, 0(x11)                    # reg x12 = v[k]
      ld x13, 8(x11)                    # reg x13 = v[k+1]

      ld x13, 0(x11)                    # v[k] = reg x13
      ld x12, 8(x11)                    # v[k+1] = reg x12
```

Method return

```
      jalr x0,0(x1)                     # return to calling routine
```

FIGURE e2.15.11 RISC-V assembly code of the procedure swap in Figure e2.24.

Method body			
Move parameters		mv x21, x10	# Copy parameter x10 into x21
Test ptr null		beq x10, x0, NullPointer	# If x10==0, goto Error
Get array length		ld x22, 8(x10)	# x22 = length of array v
Outer loop head		li x19, 0	# i = 0
	for1tst:	bge x19, x22, exit1	# If i >= length, go to exit1
Inner loop head		addi x20, x19, -1	# j = i - 1
	for2tst:	blt x20, x0, exit1	# If j < 0, goto exit2
Test if j too big		bge x20, x22, IndexOutOfBounds	# If j >= length, goto error
Get v[j]		slli x5, x20, 3	# x5 = j * 8
		add x5, x21, x5	# x5 = v + (j * 8)
		ld x6, 0(x5)	# x6 = v[j]
Test if j+1 < 0 or too big		addi x7, x20, 1	# x7 = j + 1
		blt x7, x0, IndexOutOfBounds	# If j + 1 < 0, goto Error
		bge x7, x22, IndexOutOfBounds	# If j + 1 >= length, goto Error
Get v[j+1]		ld x7, 8(x5)	# x7 = v[j+1]
Load method table		ld x28, 0(x10)	# x28 = address of method table
Get method address		ld x28, 16(x28)	# x28 = address of third method
Pass parameters		mv x10, x6	# 1st parameter is v[j]
	for2tst:	mv x11, x7	# 2nd parameter is v[j+1]
Call method indirectly		jalr x1, 0(x28)	# Call compareTo
Test if should skip swap		ble x10, x0, exit2	# If result <= 0, skip swap
Pass parameters and call swap		mv x10, x21	# 1st parameter is v
		mv x11, x20	# 2nd parameter is j
		jal x1, swap	# Invoke swap routine (Figure 2.34)
Inner loop end		addi x20,x20,-1	# j -= 1
		j for2tst	# Go to for2tst
Outer loop end	exit2:	addi x19,x19,1	# i += 1
		j for1tst	# Go to for1tst

FIGURE e2.15.12 RISC-V assembly version of the method body of the Java version of `sort`. The new code is highlighted in this figure. We must still add the code to save and restore registers and the return from the RISC-V code found in Figure e2.27. To keep the code similar to that figure, we load `v.length` into `x22` instead of into a temporary register. To reduce the number of lines of code, we make the simplifying assumption that `compareTo` is a leaf procedure and we do not need to push registers to be saved on the stack.

Next, we load the length of the array (we use register `x22` to keep it similar to the code for the C version of sort):

```
ld   x22, 8(x10)    // x22 = length of array v
```

Now we must ensure that the index is within bounds. Since the first test of the inner loop is to test if `j` is negative, we can skip that initial bound test. That leaves the test for too big:

```
bge x20, x22, IndexOutOfBounds  // if j > = length, goto Error
```

The code for testing `j + 1` is quite similar to the code for checking `k + 1` in `swap`, so we skip it here.

The key difference is the invocation of `compareTo`. We first load the address of the table of legal methods, which we assume is two doublewords before the beginning of the array:

```
ld  x28, 0(x10)      // x28 = address of method table
```

Given the address of the method table for this object, we then get the desired method. Let's assume `compareTo` is the third method in the `Comparable` class. To pick the address of the third method, we load that address into a temporary register:

```
ld  x28, 16(x28)    // x28 = address of third method
```

We are now ready to call `compareTo`. The next step is to save the necessary registers on the stack. Fortunately, we don't need the temporary registers or argument registers after the method invocation, so there is nothing to save. Thus, we simply pass the parameters for `compareTo`:

```
mv x10, x6     // 1st parameter of compareTo is v[j]
mv x11, x7     // 2nd parameter of compareTo is v[j+1]
```

Then, we use the jump-and-link register to invoke compareTo:

```
jalr  x1, 0(x28)  // invoke compareTo, and save return address in x1
```

The method returns, with `x10` determining which of the two elements is larger. If `x10 > 0`, then `v[j] >v[j+1]`, and we need to `swap`. Thus, to skip the `swap`, we need to test if `x10` ≤ `0`:

```
ble x10, x0, exit2    // go to exit2 if v[j] ≤ v[j+1]
```

The RISC-V code for `compareTo` is left as an exercise.

Hardware/ Software Interface

The main changes for the Java versions of `sort` and `swap` are testing for null object references and index out-of-bounds errors, and the extra method invocation to give a more general compare. This method invocation is more expensive than a C procedure call, since it requires, a conditional branch, a pair of chained loads, and an indirect branch. As we see in Chapter 4, dependent loads and indirect branches can be relatively slow on modern processors. The increasing popularity of Java suggests that many programmers today are willing to leverage the high performance of modern processors to pay for error checking and code reuse.

Elaboration Although we test each reference to j and $j + 1$ to be sure that these indices are within bounds, an assembly language programmer might look at the code and reason as follows:

1. The inner *for* loop is only executed if $j \leq 0$ and since $j + 1 > j$, there is no need to test $j + 1$ to see if it is less than 0.
2. Since i takes on the values, 0, 1, 2, …, (data.length − 1) and since j takes on the values i − 1, i − 2, …, 2, 1, 0, there is no need to test if $j \leq$ data.length since the largest value j can be is data.length − 2.
3. Following the same reasoning, there is no need to test whether $j + 1 \leq$ data.length since the largest value of j+1 is data.length − 1.

There are coding tricks in Chapter 2 and superscalar execution in Chapter 4 that lower the effective cost of such bounds checking, but only high optimizing compilers can reason this way. Note that if the compiler inlined the swap method into sort, many checks would be unnecessary.

Elaboration Look carefully at the code for swap in Figure e2.15.11. See anything wrong in the code, or at least in the explanation of how the code works? It implicitly assumes that each `Comparable` element in `v` is 8 bytes long. Surely, you need much more than 8 bytes for a complex subclass of `Comparable`, which could contain any number of fields. Surprisingly, this code does work, because an important property of Java's semantics forces the use of the same, small representation for all variables, fields, and array elements that belong to `Comparable` or its subclasses.

Java types are divided into *primitive types*—the predefined types for numbers, characters, and Booleans—and *reference types*—the built-in classes like String, user-defined classes, and arrays. Values of reference types are pointers (also called *references*) to anonymous objects that are themselves allocated in the heap. For the programmer, this means that assigning one variable to another does not create a new object, but instead makes both variables refer to the same object. Because these objects are anonymous, and programs therefore have no way to refer to them directly, a program must use indirection through a variable to read or write any objects' fields (variables). Thus, because the data structure allocated for the array `v` consists entirely of pointers, it is safe to assume they are all the same size, and the same swapping code works for all of `Comparable`'s subtypes.

To write sorting and swapping functions for arrays of primitive types requires that we write new versions of the functions, one for each type. This replication is for two reasons. First, primitive type values do not include the references to dispatching tables that we used on `Comparables` to determine at runtime how to compare values. Second, primitive values come in different sizes: 1, 2, 4, or 8 bytes.

The pervasive use of pointers in Java is elegant in its consistency, with the penalty being a level of indirection and a requirement that objects be allocated on the heap. Furthermore, in any language where the lifetimes of the heap-allocated anonymous objects are independent of the lifetimes of the named variables, fields, and array elements that reference them, programmers must deal with the problem of deciding when it is safe to deallocate heap-allocated storage. Java's designers chose to use

garbage collection. Of course, use of garbage collection rather than explicit user memory management also improves program safety.

C++ provides an interesting contrast. Although programmers can write essentially the same pointer-manipulating solution in C++, there is another option. In C++, programmers can elect to forgo the level of indirection and directly manipulate an array of objects, rather than an array of pointers to those objects. To do so, C++ programmers would typically use the template capability, which allows a class or function to be parameterized by the *type* of data on which it acts. Templates, however, are compiled using the equivalent of macro expansion. That is, if we declared an instance of sort capable of sorting types X and Y, C++ would create two copies of the code for the class: one for sort<X> and one for sort<Y>, each specialized accordingly. This solution increases code size in exchange for making comparison faster (since the function calls would not be indirect, and might even be subject to inline expansion). Of course, the speed advantage would be canceled if swapping the objects required moving large amounts of data instead of just single pointers. As always, the best design depends on the details of the problem.

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 an RISC-V architecture. It shows the Java byte-codes used for interpretation and the RISC-V 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.

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

The rest of **Section 2.15** can be found online.

2.16 Real Stuff: MIPS Instructions

The instruction set most similar to RISC-V, MIPS, also originated in academia, but is now owned by Imagination Technologies. MIPS and RISC-V share the same design philosophy, despite MIPS being 25 years more senior than RISC-V. The good news is that if you know RISC-V, it will be very easy to pick up MIPS. To show their similarity, Figure 2.29 compares instruction formats for RISC-V and MIPS.

The MIPS ISA has both 32-bit address and 64-bit address versions, sensibly called MIPS-32 and MIPS-64. These instruction sets are virtually identical except for the larger address size needing 64-bit registers instead of 32-bit registers. Here are the common features between RISC-V and MIPS:

- All instructions are 32 bits wide for both architectures.
- Both have 32 general-purpose registers, with one register being hardwired to 0.
- The only way to access memory is via load and store instructions on both architectures.
- Unlike some architectures, there are no instructions that can load or store many registers in MIPS or RISC-V.
- Both have instructions that branch if a register is equal to zero and branch if a register is not equal to zero.
- Both sets of addressing modes work for all word sizes.

One of the main differences between RISC-V and MIPS is for conditional branches other than equal or not equal. Whereas RISC-V simply provides branch instructions to compare two registers, MIPS relies on a comparison instruction that sets a register to 0 or 1 depending on whether the comparison is true. Programmers then follow that comparison instruction with a branch on equal to or not equal to zero depending on the desired outcome of the comparison. Keeping with its minimalist philosophy, MIPS only performs less than comparisons, leaving it up to the programmer to switch order of operands or to switch the condition being tested

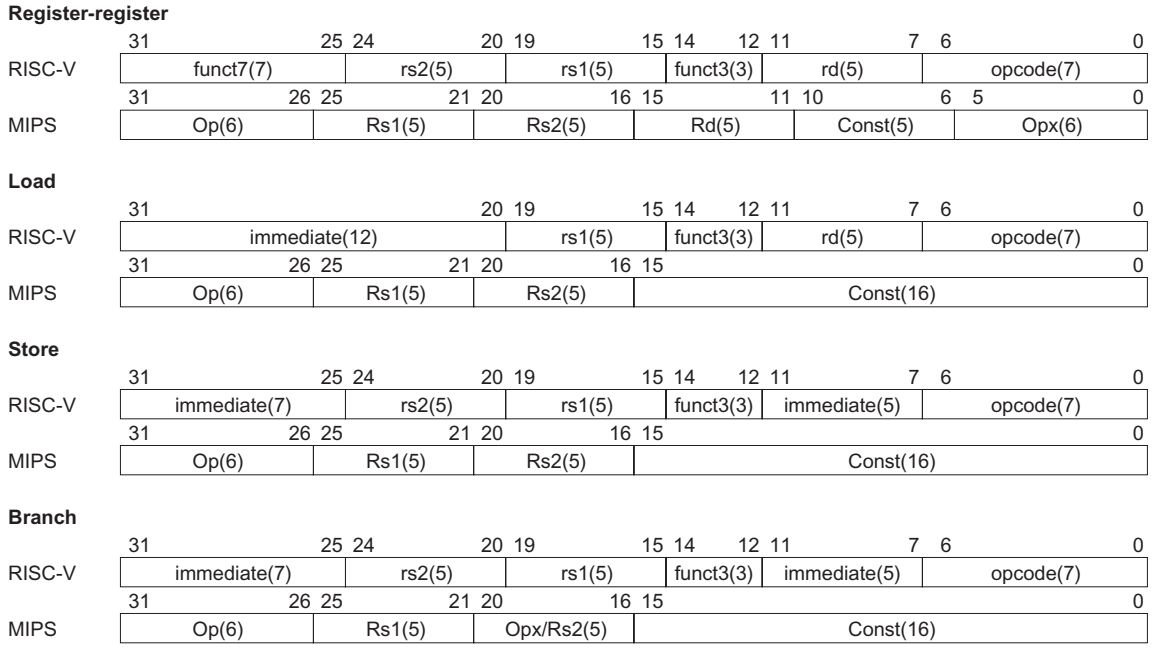


FIGURE 2.29 Instruction formats of RISC-V and MIPS. The similarities result in part from both instruction sets having 32 registers.

by the branch to get all the desired outcomes. MIPS has both signed and unsigned versions of the set on less than instructions: `slt` and `sltu`.

When we look beyond the core instructions that are most commonly used, the other main difference is that the full MIPS is a much larger instruction set than RISC-V, as we shall see in Section 2.18.

2.17 Real Stuff: x86 Instructions

Beauty is altogether in the eye of the beholder.

Margaret Wolfe Hungerford, *Molly Bawn*, 1877

Designers of instruction sets sometimes provide more powerful operations than those found in RISC-V 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

RISC-V and MIPS were the vision of single groups working at the same time; the pieces of these architectures fit nicely together. Such is not the case for the x86; it is the product of several independent groups who evolved the architecture over almost 40 years, adding new features to the original instruction set as someone might add clothing to a packed bag. Here are important x86 milestones.

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

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

- **1980**: The Intel 8087 floating-point coprocessor is announced. This architecture extends the 8086 with about 60 floating-point instructions. Instead of using registers, it relies on a stack (see **Section 2.21** 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 expanded 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 (see 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 10 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 RISC-V.
- **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 post-PC era. Manufacturing 350M x86 chips per year may seem small compared to 14 billion ARM 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 2.30). 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 RISC-V and MIPS programs can use four times as many.

Figure 2.31 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; RISC-V 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 RISC-V 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.

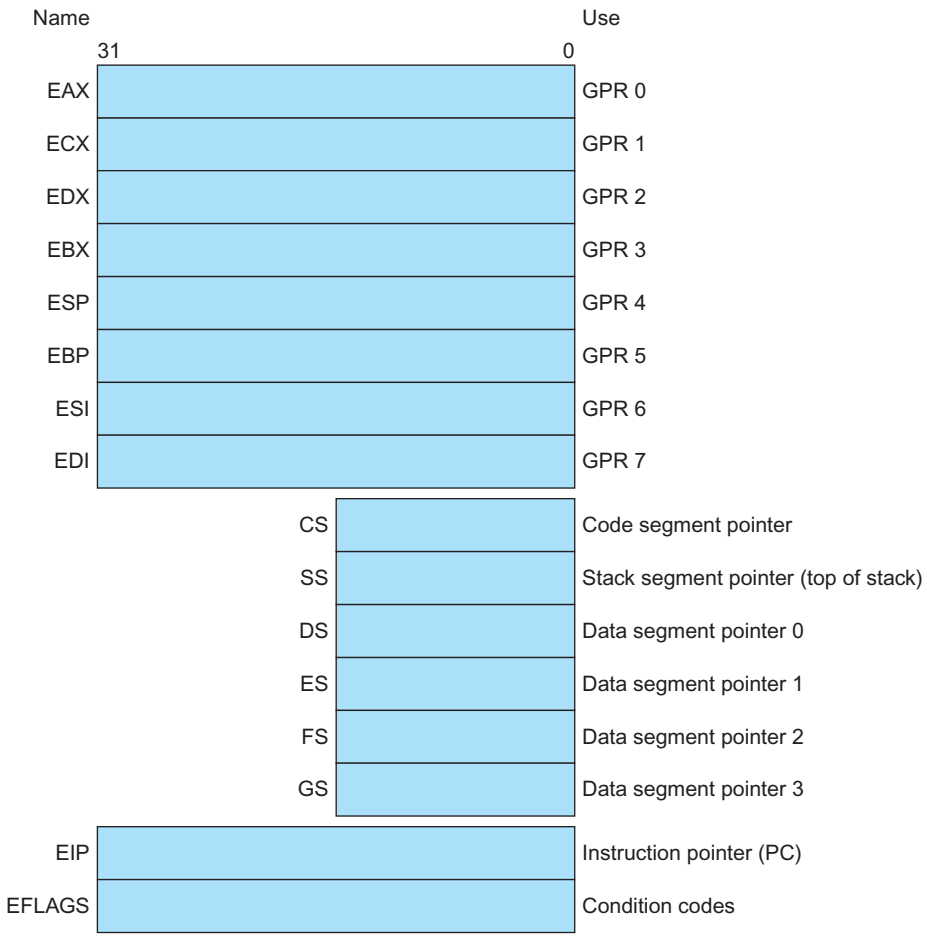


FIGURE 2.30 The 80386 register set. Starting with the 80386, the top eight registers were extended to 32 bits and could also be used as general-purpose registers.

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

FIGURE 2.31 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. Immediates may be 8, 16, or 32 bits in length; a register is any one of the 14 major registers in Figure 2.33 (not EIP or EFLAGS).

Mode	Description	Register restrictions	RISC-V equivalent
Register indirect	Address is in a register.	Not ESP or EBP	`ld x10, 0(x11)`
Based mode with 8- or 32-bit displacement	Address is contents of base register plus displacement.	Not ESP	`ld x10, 40(x11)`

Base plus scaled index	The address is Base + (2^{Scale} × Index) where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	`slli x12, x12, 3`
			`add x11, x11, x12`
			`ld x10, 0(x11)`

Base plus scaled index with 8- or 32-bit displacement	The address is Base + (2^{Scale} × Index) + Displacement where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	`slli x12, x12, 3`
			`add x11, x11, x12`
			`ld x10, 40(x11)`

FIGURE 2.32 x86 32-bit addressing modes with register restrictions and the equivalent RISC-V code. The Base plus Scaled Index addressing mode, not found in RISC-V or MIPS, is included to avoid the multiplies by 8 (scale factor of 3) to turn an index in a register into a byte address (see Figures 2.26 and 2.28). A scale factor of 1 is used for 16-bit data, and a scale factor of 2 for 32-bit data. A scale factor of 0 means the address is not scaled. If the displacement is longer than 12 bits in the second or fourth modes, then the RISC-V equivalent mode would need more instructions, usually a `lui` to load bits 12 through 31 of the displacement, followed by an `add` to sum these bits with the base register. (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.)

Although a memory operand can use any addressing mode, there are restrictions on which *registers* can be used in a mode. Figure 2.32 shows the x86 addressing modes and which GPRs cannot be used with each mode, as well as how to get the same effect using RISC-V 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 (*doublewords*) in the x86. (AMD64 adds 64-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 dominate 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 branches, 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 2.33 shows some typical x86 instructions and their functions.

Instruction	Function
`je` name	`if equal(condition code){EIP=name};` `EIP-128 <= name < EIP+128`
`jmp` name	`EIP=name`
`call` name	`SP=SP-4; M[SP]=EIP+5; EIP=name;`
`movw EBX,[EDI+45]`	`EBX=M[EDI+45]`
`push ESI`	`SP=SP-4; M[SP]=ESI`
`pop EDI`	`EDI=M[SP]; SP=SP+4`
`add EAX,#6765`	`EAX= EAX+6765`
`test EDX,#42`	Set condition code (flags) with EDX and 42
`movsl`	`M[EDI]=M[ESI];` `EDI=EDI+4; ESI=ESI+4`

FIGURE 2.33 Some typical x86 instructions and their functions. A list of frequent operations appears in Figure 2.37. The CALL saves the EIP of the next instruction on the stack. (EIP is the Intel PC.)

Conditional branches on the x86 are based on *condition codes* or *flags*. Condition codes are set as a side effect of an operation; most are used to compare the value of a result to 0. Branches then test the condition codes. PC-relative branch addresses must be specified in the number of bytes, since unlike RISC-V and MIPS, 80386 instructions have no alignment restriction.

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 157).

Figure 2.34 lists some of the integer x86 instructions. Many of the instructions are available in both byte and word formats.

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
`lods`	Loads a byte, word, or doubleword of a string into the EAX register

FIGURE 2.34 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.

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 is only one operand, up to 15 bytes.

Figure 2.35 shows the instruction format for several of the example instructions in Figure 2.33. 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 of the instructions that address memory. The base plus scaled index mode uses a second postbyte, labeled "sc, index, base."

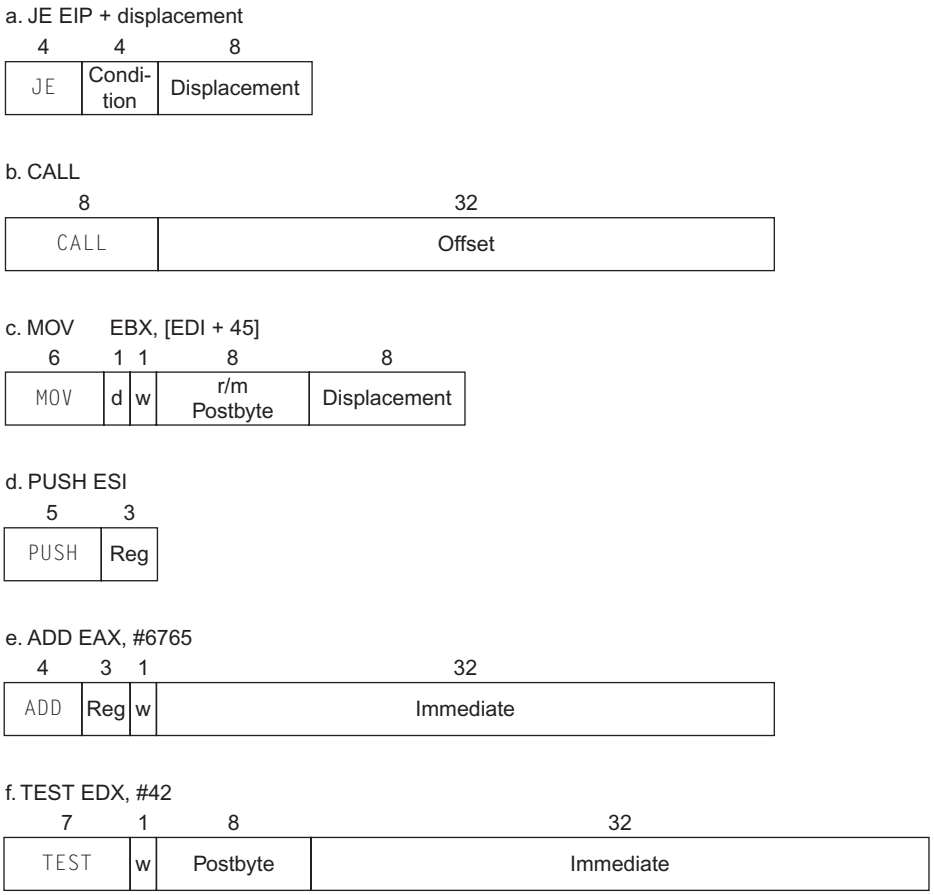


FIGURE 2.35 Typical x86 instruction formats. Figure 2.39 shows the encoding of the postbyte. Many instructions contain the 1-bit field *w*, which says whether the operation is a byte or a doubleword. The *d* field in MOV is used in instructions that may move to or from memory and shows the direction 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.

Figure 2.36 shows the encoding of the two postbyte address specifiers for both 16-bit and 32-bit modes. 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

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

FIGURE 2.36 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 2.35. When *mod* = 3, the *r/m* field indicates a register, using the same encoding as the reg field combined with the *w* bit.

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 RISC-V and MIPS, but the large market meant in the PC era that AMD and Intel could afford more resources to help overcome the added complexity. What the x86 lacks in style, it rectifies with 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, compilers must avoid the portions of the architecture that are hard to implement fast.

In the post-PC era, however, despite considerable architectural and manufacturing expertise, x86 has not yet been competitive in the personal mobile device.

2.18 Real Stuff: The Rest of the RISC-V Instruction Set

With the goal of making an instruction set architecture suitable for a wide variety of computers, the RISC-V architects partitioned the instruction set into a *base architecture* and several *extensions*. Each is named with a letter of the alphabet, and the base architecture is named I for *integer*. The base architecture has few instructions relative to other popular instruction sets today; indeed, this chapter has already covered nearly all of them. This section rounds out the base architecture, then describes the five standard extensions.

Additional Instructions in RISC-V Base Architecture

Instruction	Name	Format	Description
Add upper immediate to PC	`auipc`	U	Add 20-bit upper immediate to PC; write sum to register
Set if less than	`slt`	R	Compare registers; write Boolean result to register
Set if less than, unsigned	`sltu`	R	Compare registers; write Boolean result to register
Set if less than, immediate	`slti`	I	Compare registers; write Boolean result to register
Set if less than immediate, unsigned	`sltiu`	I	Compare registers; write Boolean result to register
Add word	`addw`	R	Add 32-bit numbers
Subtract word	`subw`	R	Subtract 32-bit numbers
Add word immediate	`addiw`	I	Add constant to 32-bit number
Shift left logical word	`sllw`	R	Shift 32-bit number left by register
Shift right logical word	`srlw`	R	Shift 32-bit number right by register
Shift right arithmetic word	`sraw`	R	Shift 32-bit number right arithmetically by register
Shift left logical word immedate	`slliw`	I	Shift 32-bit number left by immediate
Shift right logical word immediate	`srliw`	I	Shift 32-bit number right by immediate
Shift right arithmetic word immediate	`sraiw`	I	Shift 32-bit number right arithmetically by immediate

FIGURE 2.37 The remaining 14 instructions in the base RISC-V instruction set architecture.

Figure 2.37 lists the remaining instructions in the base RISC-V architecture. The first instruction, `auipc`, is used for PC-relative memory addressing. Like the `lui` instruction, it holds a 20-bit constant that corresponds to bits 12 through 31 of an integer. `auipc`'s effect is to add this number to the PC and write the sum to a register. Combined with an instruction like `addi`, it is possible to address any byte of memory within 4 GiB of the PC. This feature is useful for *position-independent code*, which can execute correctly no matter where in memory it is loaded. It is most frequently used in dynamically linked libraries.

The next four instructions compare two integers, then write the Boolean result of the comparison to a register. `slt` and `sltu` compare two registers as signed and unsigned numbers, respectively, then write 1 to a register if the first value is less than the second value, or 0 otherwise. `slti` and `sltiu` perform the same comparisons, but with an immediate for the second operand.

The remaining instructions should all look familiar, as their names are the same as other instructions discussed in this chapter, but with the letter `w`, short for *word*, appended. These instructions perform the same operation as the similarly named ones we've discussed, except these only operate on the lower 32 bits of their operands, ignoring bits 32 through 63. Additionally, they produce sign-extended 32-bit results: that is, bits 32 through 63 are all the same as bit 31. The RISC-V architects included these `w` instructions because operations on 32-bit numbers remain very common on computers with 64-bit addresses. The main reason is that the popular data type `int` remains 32 bits in Java and in most implementations of the C language.

RISC-V Base and Extensions

Mnemonic	Description	Insn. Count
I	Base architecture	51
M	Integer multiply/divide	13
A	Atomic operations	22
F	Single-precision floating point	30
D	Double-precision floating point	32
C	Compressed instructions	36

FIGURE 2.38 The RISC-V instruction set architecture is divided into the base ISA, named I, and five standard extensions, M, A, F, D, and C.

That's it for the base architecture! Figure 2.38 lists the five standard extensions. The first, M, adds instructions to multiply and divide integers. Chapter 3 will introduce several instructions in the M extension.

The second extension, A, supports atomic memory operations for multiprocessor synchronization. The load-reserved (`lr.d`) and store-conditional (`sc.d`) instructions introduced in Section 2.11 are members of the A extension. Also included are versions that operate on 32-bit words (`lr.w` and `sc.w`). The remaining 18 instructions are optimizations of common synchronization patterns, like atomic exchange and atomic addition, but do not add any additional functionality over load-reserved and store-conditional.

The third and fourth extensions, F and D, provide operations on floating-point numbers, which are described in Chapter 3.

The last extension, C, provides no new functionality at all. Rather, it takes the most popular RISC-V instructions, like `addi`, and provides equivalent instructions that are only 16 bits in length, rather than 32. It thereby allows programs to be expressed in fewer bytes, which can reduce cost and, as we will see in Chapter 5, can improve performance. To fit in 16 bits, the new instructions have restrictions on their operands: for example, some instructions can only access some of the 32 registers, and the immediate fields are narrower.

Taken together, the RISC-V base and extensions have 184 instructions, plus 13 system instructions that will be introduced at the end of Chapter 5.

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 subsequent instruction until

a counter steps 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 as 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 these 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 protracted 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 recent computers. Writing in higher-level language instead of assembly language not only allows future compilers to tailor the code to forthcoming 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 2.39 shows that the x86 architecture has grown dramatically. The average is more than one instruction per month over its 35-year lifetime!

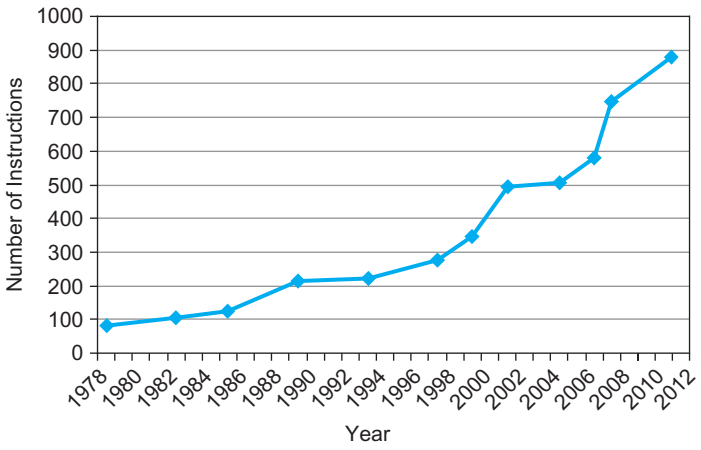


FIGURE 2.39 Growth of x86 instruction set over time. While there is clear technical value to some of these extensions, this rapid change also increases the difficulty for other companies to try to build compatible processors.

Pitfall: Forgetting that sequential word or doubleword 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 or doubleword can be found by incrementing the address in a register by one instead of by the word or doubleword 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.

2.20 Concluding Remarks

Less is more.

Robert Browning, *Andrea del Sarto*, 1855

The two principles of the *stored-program* computer are the use of instructions that are indistinguishable from numbers and the use of alterable memory for programs. These principles allow a single machine to aid cancer researchers, 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 tricky tradeoff:

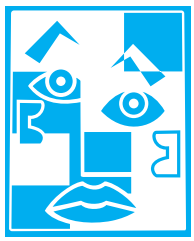
1. *Simplicity favors regularity.* Regularity motivates many features of the RISC-V instruction set: keeping all instructions a single size, always requiring register operands in arithmetic instructions, and keeping the register fields in the same place in all instruction formats.
2. *Smaller is faster.* The desire for speed is the reason that RISC-V has 32 registers rather than many more.
3. *Good design demands good compromises.* One RISC-V example is the compromise between providing for larger addresses and constants in instructions and keeping all instructions the same length.



COMMON CASE FAST

We also saw the great idea from Chapter 1 of making the **common cast fast** applied to instruction sets as well as computer architecture. Examples of making the common RISC-V 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 2.40 lists the RISC-V instructions we have covered so far, both real and pseudoinstructions. Hiding details from the higher level is another example of the great idea of **abstraction**.



ABSTRACTION

Each category of RISC-V 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 branches 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 2.41 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.

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

RISC-V Instructions	Name	Format	Pseudo RISC-V	Name	Real Instruction
Add	add	R	Move	mv	addi
Subtract	sub	R	Load immediate	li	addi
Add immediate	addi	I	Jump	j	jal
Load doubleword	ld	I	Load address	la	lui+addi
Store doubleword	sd	S			
Load word	lw	I			
Load word, unsigned	lwu	I			
Store word	sw	S			
Load halfword	lh	I			
Load halfword, unsigned	lhu	I			
Store halfword	sh	S			
Load byte	lb	I			
Load byte, unsigned	lbu	I			
Store byte	sb	S			
Load reserved	lr.d	R			
Store conditional	sc.d	R			
Load upper immediate	lui	U			
And	and	R			
Inclusive or	or	R			
Exclusive or	xor	R			
And immediate	andi	I			
Inclusive or immediate	ori	I			
Exclusive or immediate	xori	I			
Shift left logical	sll	R			
Shift right logical	srl	R			
Shift right arithmetic	sra	R			
Shift left logical immediate	slli	I			
Shift right logical immediate	srli	I			
Shift right arithmetic immediate	srai	I			
Branch if equal	beq	SB			
Branch if not equal	bne	SB			
Branch if less than	blt	SB			
Branch if greater or equal	bge	SB			
Branch if less, unsigned	bltu	SB			
Branch if greatr/eq, unsigned	bgeu	SB			
Jump and link	jal	UJ			
Jump and link register	jalr	I			

FIGURE 2.40 The RISC-V instruction set covered so far, with the real RISC-V instructions on the left and the pseudoinstructions on the right. Figure 2.1 shows more details of the RISC-V architecture revealed in this chapter. The information given here is also found in Columns 1 and 2 of the RISC-V Reference Data Card at the front of the book.

Instruction class	RISC-V examples	HLL correspondence	Frequency	
			Integer	Fl. Pt.
Arithmetic	`add, sub, addi`	Operations in assignment statements	16%	48%
Data transfer	`ld, sd, lw, sw, lh, sh, lb, sb, lui`	References to data structures in memory	35%	36%
Logical	`and, or, xor, sll, srl, sra`	Operations in assignment statements	12%	4%
Branch	`beq, bne, blt, bge, bltu, bgeu`	*If* statements; loops	34%	8%
Jump	`jal, jalr`	Procedure calls & returns; *switch* statements	2%	0%

FIGURE 2.41 RISC-V instruction classes, examples, correspondence to high-level program language constructs, and percentage of RISC-V instructions executed by category for the average integer and floating point SPEC CPU2006 benchmarks. Figure 3.24 in Chapter 3 shows average percentage of the individual RISC-V instructions executed.



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 the x86 and ARM's 32-bit architecture, ARMv7. 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

2.1 [5] <§2.2> For the following C statement, write the corresponding RISC-V assembly code. Assume that the C variables `f`, `g`, and `h`, have already been placed in registers `x5`, `x6`, and `x7` respectively. Use a minimal number of RISC-V assembly instructions.

```
f = g + (h − 5);
```



Historical Perspective and Further Reading

This section surveys the history of instruction set architectures 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 ARMv7 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.

Accumulator Architectures

Hardware was precious in the earliest stored-program computers. Consequently, computer pioneers could not afford the number of registers found in today's architectures. In fact, these architectures had a single register for arithmetic instructions. Since all operations would accumulate in one register, it was called the **accumulator**, and this style of instruction set is given the same name. For example, EDSAC in 1949 had a single accumulator.

accumulator *Archaic term for register. On-line use of it as a synonym for "register" is a fairly reliable indication that the user has been around quite a while.*

Eric Raymond, *The New Hacker's Dictionary*, 1991

The three-operand format of RISC-V suggests that a single register is at least two registers shy of our needs. Having the accumulator as both a source operand *and* the destination of the operation fills part of the shortfall, but it still leaves us one operand short. That final operand is found in memory. Accumulator architectures have the memory-based operand-addressing mode suggested earlier. It follows that the add instruction of an accumulator instruction set would look like this:

```
ADD     200
```

This instruction means add the accumulator to the word in memory at address 200 and place the sum back into the accumulator. No registers are specified because the accumulator is known to be both a source and a destination of the operation.

The next step in the evolution of instruction sets was the addition of registers dedicated to specific operations. Hence, registers might be included to act as indices for array references in data transfer instructions, to act as separate accumulators for multiply or divide instructions, and to serve as the top-of-stack pointer. Perhaps the best-known example of this style of instruction set is found in the Intel 80x86. This style of instruction set is labeled *extended accumulator*, *dedicated register*, or *special-purpose register*. Like the single-register accumulator architectures, one operand may be in memory for arithmetic instructions. Like the RISC-V architecture, however, there are also instructions where all the operands are registers.

General-Purpose Register Architectures

The generalization of the dedicated-register architecture allows all the registers to be used for any purpose, hence the name *general-purpose register*. RISC-V is an example of a general-purpose register architecture. This style of instruction set may be further divided into those that allow one operand to be in memory (as found in accumulator architectures), called a *register-memory* architecture, and those that demand that operands always be in registers, called either a **load-store** or a **register-register** architecture. Figure e2.22.1 shows a history of the number of registers in some popular computers.

load-store architecture Also called **register-register** architecture. An instruction set architecture in which all operations are between registers and data memory may only be accessed via loads or stores.

The first load-store architecture was the CDC 6600 in 1963, considered by many to be the first supercomputer. RISC-V, ARMv7, ARMv8, and MIPS are more recent examples of a load-store architecture.

Machine	Number of general-purpose registers	Architectural style	Year
EDSAC	1	Accumulator	1949
IBM 701	1	Accumulator	1953
CDC 6600	8	Load-store	1963
IBM 360	16	Register-memory	1964
DEC PDP-8	1	Accumulator	1965
DEC PDP-11	8	Register-memory	1970
Intel 8008	1	Accumulator	1972
Motorola 6800	2	Accumulator	1974
DEC VAX	16	Register-memory, memory-memory	1977
Intel 8086	1	Extended accumulator	1978
Motorola 68000	16	Register-memory	1980
Intel 80386	8	Register-memory	1985
ARM	16	Load-store	1985
MIPS	32	Load-store	1985
HP PA-RISC	32	Load-store	1986
SPARC	32	Load-store	1987
PowerPC	32	Load-store	1992
DEC Alpha	32	Load-store	1992
HP/Intel IA-64	128	Load-store	2001
AMD64 (EMT64)	16	Register-memory	2003
RISC-V	32	Load-store	2010

FIGURE e2.22.1 The number of general-purpose registers in popular architectures over the years.

The 80386 was Intel's attempt to transform the 8086 into a general-purpose register-memory instruction set. Perhaps the best-known register-memory instruction set is the IBM 360 architecture, first announced in 1964. This instruction set is still at the core of IBM's mainframe computers—responsible for a large part

of the business of the largest computer company in the world. Register-memory architectures were the most popular in the 1960s and the first half of the 1970s.

Digital Equipment Corporation's VAX architecture took memory operands one step further in 1977. It allowed an instruction to use any combination of registers and memory operands. A style of architecture in which all operands can be in memory is called *memory-memory*. (In truth the VAX instruction set, like almost all other instruction sets since the IBM 360, is a hybrid, since it also has general-purpose registers.)

The Intel x86 has many versions of a 64-bit add to specify whether an operand is in memory or is in a register. In addition, the memory operand can be accessed with more than seven addressing modes. This combination of address modes and register-memory operands means that there are dozens of variants of an x86 add instruction. Clearly, this variability makes x86 implementations more challenging.

Compact Code and Stack Architectures

When memory is scarce, it is also important to keep programs small, so architectures like the Intel x86, IBM 360, and VAX had variable-length instructions, both to match the varying operand specifications and to minimize code size. Intel x86 instructions are from 1 to 15 bytes long; IBM 360 instructions are 2, 4, or 6 bytes long; and VAX instruction lengths are anywhere from 1 to 54 bytes.

One place where code size is still important is embedded applications. In recognition of this need, ARM, MIPS, and RISC-V all made versions of their instructions sets that offer both 16-bit instruction formats and 32-bit instruction formats: Thumb and Thumb-2 for ARM, MIPS-16, and RISC-V Compressed. Despite being limited to just two sizes, Thumb, Thumb-2, MIPS-16, and RISC-V Compressed programs are about 25% to 30% smaller, which makes their code sizes smaller than those of the 80x86. Smaller code sizes have the added benefit of improving instruction cache hit rates (see Chapter 5).

In the 1960s, a few companies followed a radical approach to instruction sets. In the belief that it was too hard for compilers to utilize registers effectively, these companies abandoned registers altogether! Instruction sets were based on a *stack model* of execution, like that found in the older Hewlett-Packard handheld calculators. Operands are pushed on the stack from memory or popped off the stack into memory. Operations take their operands from the stack and then place the result back onto the stack. In addition to simplifying compilers by eliminating register allocation, stack architectures lent themselves to compact instruction encoding, thereby removing memory size as an excuse not to program in high-level languages.

Memory space was perceived to be precious again for Java, both because memory space is limited to keep costs low in embedded applications and because programs may be downloaded over the Internet or phone lines as Java applets, and smaller programs take less time to transmit. Hence, compact instruction encoding was desirable for Java bytecodes.

High-Level-Language Computer Architectures

In the 1960s, systems software was rarely written in high-level languages. For example, virtually every commercial operating system before UNIX was programmed in assembly language, and more recently even OS/2 was originally programmed at that same low level. Some people blamed the code density of the instruction sets, rather than the programming languages and the compiler technology.

Hence, an architecture design philosophy called *high-level-language computer architecture* was advocated, with the goal of making the hardware more like the programming languages. More efficient programming languages and compilers, plus expanding memory, doomed this movement to a historical footnote. The Burroughs B5000 was the commercial fountainhead of this philosophy, but today there is no significant commercial descendant of this 1960s radical.

Reduced Instruction Set Computer Architectures

This language-oriented design philosophy was replaced in the 1980s by *RISC (reduced instruction set computer)*. Improvements in programming languages, compiler technology, and memory cost meant that less programming was being done at the assembly level, so instruction sets could be measured by how well compilers used them, in contrast to how skillfully assembly language programmers used them.

Virtually all new instruction sets since 1982 have followed this RISC philosophy of fixed instruction lengths, load-store instruction sets, limited addressing modes, and limited operations. ARMv7, ARMv8 Hitachi SH, IBM PowerPC, MIPS, Sun SPARC, and, of course, RISC-V, are all examples of RISC architectures.

A Brief History of the ARMv7

ARM started as the processor for the Acorn computer, hence its original name of Acorn RISC Machine. The Berkeley RISC papers influenced its architecture.

One of the most important early applications was emulation of the AM 6502, a 16-bit microprocessor. This emulation was to provide most of the software for the Acorn computer. As the 6502 had a variable-length instruction set that was a multiple of bytes, 6502 emulation helps explain the emphasis on shifting and masking in the ARMv7 instruction set.

Its popularity as a low-power embedded computer began with its selection as the processor for the ill-fated Apple Newton personal digital assistant. Although the Newton was not as popular as Apple hoped, Apple's blessing gave visibility to the earlier ARM instruction sets, and they subsequently caught on in several markets, including cell phones. Unlike the Newton experience, the extraordinary success of cell phones explains why 12 billion ARM processors were shipped in 2014.

One of the major events in ARM's history is the 64-bit address extension called version 8. ARM took the opportunity to redesign the instruction set to make it look much more like MIPS than like earlier ARM versions.

A Brief History of the x86

The ancestors of the x86 were the first microprocessors, produced starting in 1972. The Intel 4004 and 8008 were extremely simple 4-bit and 8-bit accumulator-style architectures. Morse et al. [1980] describe the evolution of the 8086 from the 8080 in the late 1970s as an attempt to provide a 16-bit architecture with better throughput. At that time, almost all programming for microprocessors was done in assembly language—both memory and compilers were in short supply. Intel wanted to keep its base of 8080 users, so the 8086 was designed to be "compatible" with the 8080. The 8086 was *never* object-code compatible with the 8080, but the architectures were close enough that translation of assembly language programs could be done automatically.

In early 1980, IBM selected a version of the 8086 with an 8-bit external bus, called the 8088, for use in the IBM PC. They chose the 8-bit version to reduce the cost of the architecture. This choice, together with the tremendous success of the IBM PC, has made the 8086 architecture ubiquitous. The success of the IBM PC was due in part because IBM opened the architecture of the PC and enabled the PC-clone industry to flourish. As discussed in Section 2.18, the 80286, 80386, 80486, Pentium, Pentium Pro, Pentium II, Pentium III, Pentium 4, and AMD64 have extended the architecture and provided a series of performance enhancements.

Although the 68000 was chosen for the Macintosh, the Mac was never as pervasive as the PC, partly because Apple did not allow Mac clones based on the 68000, and the 68000 did not acquire the same software following that which the 8086 enjoys. The Motorola 68000 may have been more significant *technically* than the 8086, but the impact of IBM's selection and open architecture strategy dominated the technical advantages of the 68000 in the market.

Some argue that the inelegance of the x86 instruction set is unavoidable, the price that must be paid for rampant success by any architecture. We reject that notion. Obviously, no successful architecture can jettison features that were added in previous implementations, and over time, some features may be seen as undesirable. The awkwardness of the x86 begins at its core with the 8086 instruction set and was exacerbated by the architecturally inconsistent expansions found in the 8087, 80286, 80386, MMX, SSE, SSE2, SSE3, SSE4, AMD64 (EM64T), and AVX.

A counterexample is the IBM 360/370 architecture, which is much older than the x86. It dominated the mainframe market just as the x86 dominated the PC market. Due undoubtedly to a better base and more compatible enhancements, this instruction set makes much more sense than the x86 50 years after its first implementation.

Extending the x86 to 64-bit addressing means the architecture may last for several more decades. Instruction set anthropologists of the future will peel off layer after layer from such architectures until they uncover artifacts from the first microprocessor. Given such a find, how will they judge today's computer architecture?

A Brief History of Programming Languages

In 1954, John Backus led a team at IBM to create a more natural notation for scientific programming. The goal of Fortran, for "FORmula TRANslator," was to reduce the time to develop programs. Fortran included many ideas found in programming languages today, including assignment statements, expressions, typed variables, loops, and arrays. The development of the language and the compiler went hand in hand. This language became a standard that has evolved over time to improve programmer productivity and program portability. The evolutionary steps are Fortran I, II, IV, 77, and 90.

Fortran was developed for IBM's second commercial computer, the 704, which was also the cradle of another important programming language: Lisp. John McCarthy invented the "LISt Processing" language in 1958. Its mantra is that programming can be considered as manipulating lists, so the language contains operations to follow links and to compose new lists from old ones. This list notation is used for the code as well as the data, so modifying or composing Lisp programs is common. The big contribution was dynamic data structures and, hence, pointers. Given that its inventor was a pioneer in artificial intelligence, Lisp became popular in the AI community. Lisp has no type declarations, and Lisp traditionally reclaims storage automatically via built-in garbage collection. Lisp was originally interpreted, although compilers were later developed for it.

Fortran inspired the international community to invent a programming language that was more natural to express algorithms than Fortran, with less emphasis on coding. This language became Algol, for "ALGOrithmic Language." Like Fortran, it included type declarations, but it added recursive procedure calls, nested *if-then-else* statements, *while* loops, *begin-end* statements to structure code, and call-by-name. Algol-60 became the classic language for academics to teach programming in the 1960s.

Although engineers, AI researchers, and computer scientists had their own programming languages, the same could not be said for business data processing. Cobol, for "COmmon Business-Oriented Language," was developed as a standard for this purpose contemporary with Algol-60. Cobol was created to be easy to read, so it follows English vocabulary and punctuation. It added records to programming languages, and separated description of data from description of code.

Niklaus Wirth was a member of the Algol-68 committee, which was supposed to update Algol-60. He was bothered by the complexity of the result, and so he wrote a minority report to show that a programming language could combine the algorithmic power of Algol-60 with the record structure from Cobol and be simple to understand, easy to implement, yet still powerful. This minority report became Pascal. It was first implemented with an interpreter and a set of Pascal bytecodes. The ease of implementation led to its being widely deployed, much more than Algol-68, and it soon replaced Algol-60 as the most popular language for academics to teach programming.

In the same period, Dennis Ritchie invented the C programming language to use in building UNIX. Its inventors say it is not a "very high level" programming language or a big one, and it is not aimed at a particular application. Given its birthplace, it was very good at systems programming, and the UNIX operating system and C compiler were written in C. UNIX's popularity helped spur C's popularity.

The concept of object orientation is first captured in Simula-67, a simulation language successor to Algol-60. Invented by Ole-Johan Dahl and Kristen Nygaard at the University of Oslo in 1967, it introduced objects, classes, and inheritance.

Object orientation proved to be a powerful idea. It led Alan Kay and others at Xerox Palo Alto Research Center to invent Smalltalk in the 1970s. Smalltalk-80 married the typeless variables and garbage collection from Lisp and the object orientation of Simula-67. It relied on interpretation that was defined by a Smalltalk virtual machine with a Smalltalk bytecode instruction set. Kay and his colleagues argued that processors were getting faster, and that we must eventually be willing to sacrifice some performance to improve program development. Another example was CLU, which demonstrated that an object-oriented language could be defined that allowed compile-time type checking. Simula-67 also inspired Bjarne Stroustrup of Bell Labs to develop an object-oriented version of C called C++ in the 1980s. C++ became widely used in industry.

Dissatisfied with C++, a group at Sun led by James Gosling invented Oak in the early 1990s. It was invented as an object-oriented C dialect for embedded devices as part of a major Sun project. To make it portable, it was interpreted and had its own virtual machine and bytecode instruction set. Since it was a new language, it had a more elegant object-oriented design than C++ and was much easier to learn and compile than Smalltalk-80. Since Sun's embedded project failed, we might never have heard of it had someone not made the connection between Oak and programmable browsers for the World Wide Web. It was rechristened Java, and in 1995, Netscape announced that it would be shipping with its browser. It soon became extraordinarily popular. Java had the rare distinction of becoming the standard language for new business data processing applications *and* the favored language for academics to teach programming. Java and languages like it encourage reuse of code, and hence programmers make heavy use of libraries, whereas in the past they were more likely to write everything from scratch.

A Brief History of Compilers

Backus and his group were very concerned that Fortran would be unsuccessful if skeptics found examples where the Fortran version ran at half the speed of the equivalent assembly language program. Their success with one of the first compilers created a beachhead that many others followed.

Early compilers were ad hoc programs that performed the steps described in Section 2.15 online. These ad hoc approaches were replaced with a solid theoretical foundation for each of these steps. Each time the theory was established, a tool was built based on that theory that automated the creation of that step.

The theoretical roots underlying scanning and parsing derive from automata theory, and the relationship between languages and automata was known early. The scanning task corresponds to recognition of a language accepted by a finite-state automata, and parsing corresponds to recognition of a language by a push-down automata (basically an automata with a stack). Languages are described by grammars, which are a set of rules that tell how any legal program can be generated.

The scanning pass of a compiler was well understood early, but parsing is harder. The earliest parsers use precedence techniques, which derived from the structure of arithmetic statements, and were then generalized. The great breakthrough in modern parsing was made by Donald Knuth in the invention of LR-parsing, which codified the two key steps in the parsing technique, pushing a token on the stack or reducing a set of tokens on the stack using a grammar rule. The strong theory formulation for scanning and parsing led to the development of automated tools for compiler constructions, such as `lex` and `yacc`, the tools developed as part of UNIX.

Optimizations occurred in many compilers, and it is harder to determine the first examples in most cases. However, Victor Vyssotsky did the first papers on data flow analysis in 1963, and William McKeeman is generally credited with the first peephole optimizer in 1965. The group at IBM, including John Cocke and Fran Allan, developed many of the early optimization concepts, as well as defining and extending the concepts of flow analysis. Important contributions were also made by Al Aho and Jeff Ullman.

One of the biggest challenges for optimization was register allocation. It was so difficult that some architects used stack architectures just to avoid the problem. The breakthrough came when researchers working on compilers for the 801, an early RISC architecture, recognized that coloring a graph with a minimum number of colors was equivalent to allocating a fixed number of registers to the unlimited number of virtual registers used in intermediate forms.

Compilers also played an important role in the open-source movement. Richard Stallman's self-appointed mission was to make a public domain version of UNIX. He built the GNU C Compiler (gcc) as an open-source compiler in 1987. It soon was ported to many architectures, and is used in many systems today.

Further Reading

Bayko, J. [1996]. "Great microprocessors of the past and present," search for it on the *http://www.cpushack.com/CPU/cpu.html*.

A personal view of the history of both representative and unusual microprocessors, from the Intel 4004 to the Patriot Scientific ShBoom!

Kane, G. and J. Heinrich [1992]. *MIPS RISC Architecture*, Prentice Hall, Englewood Cliffs, NJ.

This book describes the MIPS architecture in greater detail than Appendix A.

Levy, H. and R. Eckhouse [1989]. *Computer Programming and Architecture*, The VAX, Digital Press, Boston.

This book concentrates on the VAX, but also includes descriptions of the Intel 8086, IBM 360, and CDC 6600.

Morse, S., B. Ravenal, S. Mazor, and W. Pohlman [1980]. "Intel microprocessors—8080 to 8086", *Computer* 13 10 (October).

The architecture history of the Intel from the 4004 to the 8086, according to the people who participated in the designs.

Wakerly, J. [1989]. *Microcomputer Architecture and Programming*, Wiley, New York.

The Motorola 6800 is the main focus of the book, but it covers the Intel 8086, Motorola 6809, TI 9900, and Zilog Z8000.

2.2 [5] <§2.2> Write a single C statement that corresponds to the two RISC-V assembly instructions below.

```
add f, g, h
add f, i, f
```

2.3 [5] <§§2.2, 2.3> For the following C statement, write the corresponding RISC-V assembly code. Assume that the variables `f`, `g`, `h`, `i`, and `j` are assigned to registers `x5`, `x6`, `x7`, `x28`, and `x29`, respectively. Assume that the base address of the arrays `A` and `B` are in registers `x10` and `x11`, respectively.

```
B[8] = A[i-j];
```

2.4 [10] <§§2.2, 2.3> For the RISC-V assembly instructions below, what is the corresponding C statement? Assume that the variables `f`, `g`, `h`, `i`, and `j` are assigned to registers `x5`, `x6`, `x7`, `x28`, and `x29`, respectively. Assume that the base address of the arrays A and B are in registers `x10` and `x11`, respectively.

```
slli x30, x5, 3     // x30 = f*8
add  x30, x10, x30  // x30 = &A[f]
slli x31, x6, 3     // x31 = g*8
add  x31, x11, x31  // x31 = &B[g]
ld   x5, 0(x30)     // f = A[f]

addi x12, x30, 8
ld   x30, 0(x12)
add  x30, x30, x5
sd   x30, 0(x31)
```

2.5 [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 are stored starting at address 0 and that the word size is 4 bytes.

2.6 [5] <§2.4> Translate `0xabcdef12` into decimal.

2.7 [5] <§§2.2, 2.3> Translate the following C code to RISC-V. Assume that the variables `f`, `g`, `h`, `i`, and `j` are assigned to registers `x5`, `x6`, `x7`, `x28`, and `x29`, respectively. Assume that the base address of the arrays `A` and `B` are in registers `x10` and `x11`, respectively. Assume that the elements of the arrays `A` and `B` are 8-byte words:

```
B[8] = A[i] + A[j];
```

2.8 [10] <§§2.2, 2.3> Translate the following RISC-V code to C. Assume that the variables `f`, `g`, `h`, `i`, and `j` are assigned to registers `x5`, `x6`, `x7`, `x28`, and `x29`,

respectively. Assume that the base address of the arrays `A` and `B` are in registers `x10` and `x11`, respectively.

```
addi  x30, x10, 8
addi  x31, x10, 0
sd    x31, 0(x30)
ld    x30, 0(x30)
add   x5,  x30, x31
```

2.9 [20] <§§2.2, 2.5> For each RISC-V instruction in Exercise 2.8, show the value of the opcode (op), source register (rs1), and destination register (rd) fields. For the I-type instructions, show the value of the immediate field, and for the R-type instructions, show the value of the second source register (rs2). For non U- and UJ-type instructions, show the funct3 field, and for R-type and S-type instructions, also show the funct7 field.

2.10 Assume that registers `x5` and `x6` hold the values `0x8000000000000000` and `0xD000000000000000`, respectively.

2.10.1 [5] <§2.4> What is the value of `x30` for the following assembly code?

```
add x30, x5, x6
```

2.10.2 [5] <§2.4> Is the result in `x30` the desired result, or has there been overflow?

2.10.3 [5] <§2.4> For the contents of registers `x5` and `x6` as specified above, what is the value of `x30` for the following assembly code?

```
sub x30, x5, x6
```

2.10.4 [5] <§2.4> Is the result in `x30` the desired result, or has there been overflow?

2.10.5 [5] <§2.4> For the contents of registers `x5` and `x6` as specified above, what is the value of `x30` for the following assembly code?

```
add x30, x5, x6
add x30, x30, x5
```

2.10.6 [5] <§2.4> Is the result in `x30` the desired result, or has there been overflow?

2.11 Assume that `x5` holds the value 128_{ten}.

2.11.1 [5] <§2.4> For the instruction `add x30, x5, x6`, what is the range(s) of values for `x6` that would result in overflow?

2.11.2 [5] <§2.4> For the instruction `sub x30, x5, x6`, what is the range(s) of values for `x6` that would result in overflow?

2.11.3 [5] <§2.4> For the instruction `sub x30, x6, x5`, what is the range(s) of values for `x6` that would result in overflow?

2.12 [5] <§§2.2, 2.5> Provide the instruction type and assembly language instruction for the following binary value:

$0000\ 0000\ 0001\ 0000\ 1000\ 0000\ 1011\ 0011_{two}$

Hint: Figure 2.20 may be helpful.

2.13 [5] <§§2.2, 2.5> Provide the instruction type and hexadecimal representation of the following instruction:

```
sd x5, 32(x30)
```

2.14 [5] <§2.5> Provide the instruction type, assembly language instruction, and binary representation of instruction described by the following RISC-V fields:

```
opcode=0x33, funct3=0x0, funct7=0x20, rs2=5, rs1=7, rd=6
```

2.15 [5] <§2.5> Provide the instruction type, assembly language instruction, and binary representation of instruction described by the following RISC-V fields:

```
opcode=0x3, funct3=0x3, rs1=27, rd=3, imm=0x4
```

2.16 Assume that we would like to expand the RISC-V register file to 128 registers and expand the instruction set to contain four times as many instructions.

2.16.1 [5] <§2.5> How would this affect the size of each of the bit fields in the R-type instructions?

2.16.2 [5] <§2.5> How would this affect the size of each of the bit fields in the I-type instructions?

2.16.3 [5] <§§2.5, 2.8, 2.10> How could each of the two proposed changes decrease the size of a RISC-V assembly program? On the other hand, how could the proposed change increase the size of an RISC-V assembly program?

2.17 Assume the following register contents:

```
x5 = 0x00000000AAAAAAAA, x6 = 0x1234567812345678
```

2.17.1 [5] <§2.6> For the register values shown above, what is the value of `x7` for the following sequence of instructions?

```
slli x7, x5, 4
or   x7, x7, x6
```

2.17.2 [5] <§2.6> For the register values shown above, what is the value of x7 for the following sequence of instructions?

```
slli x7, x6, 4
```

2.17.3 [5] <§2.6> For the register values shown above, what is the value of x7 for the following sequence of instructions?

```
srli  x7, x5, 3
andi  x7, x7, 0xFEF
```

2.18 [10] <§2.6> Find the shortest sequence of RISC-V instructions that extracts bits 16 down to 11 from register x5 and uses the value of this field to replace bits 31 down to 26 in register x6 without changing the other bits of registers x5 or x6. (Be sure to test your code using x5 = 0 and x6 = 0xffffffffffffffff. Doing so may reveal a common oversight.)

2.19 [5] <§2.6> Provide a minimal set of RISC-V instructions that may be used to implement the following pseudoinstruction:

```
not x5, x6     // bit-wise invert
```

2.20 [5] <§2.6> For the following C statement, write a minimal sequence of RISC-V assembly instructions that performs the identical operation. Assume x6 = A, and x17 is the base address of C.

```
A = C[0] << 4;
```

2.21 [5] <§2.7> Assume x5 holds the value 0x0000000001010000. What is the value of x6 after the following instructions?

```
      bge x5, x0, ELSE
      jal x0, DONE
ELSE: ori x6, x0, 2
DONE:
```

2.22 Suppose the *program counter* (PC) is set to 0x20000000.

2.22.1 [5] <§2.10> What range of addresses can be reached using the RISC-V *jump-and-link* (jal) instruction? (In other words, what is the set of possible values for the PC after the jump instruction executes?)

2.22.2 [5] <§2.10> What range of addresses can be reached using the RISC-V *branch if equal* (beq) instruction? (In other words, what is the set of possible values for the PC after the branch instruction executes?)

2.23 Consider a proposed new instruction named `rpt`. This instruction combines a loop's condition check and counter decrement into a single instruction. For example `rpt x29, loop` would do the following:

```
if (x29 > 0) {
        x29 = x29 −1;
        goto loop
    }
```

2.23.1 [5] <§2.7, 2.10> If this instruction were to be added to the RISC-V instruction set, what is the most appropriate instruction format?

2.23.2 [5] <§2.7> What is the shortest sequence of RISC-V instructions that performs the same operation?

2.24 Consider the following RISC-V loop:

```
LOOP:  beq  x6, x0, DONE
       addi x6, x6, -1
       addi x5, x5, 2
       jal  x0, LOOP
DONE:
```

2.24.1 [5] <§2.7> Assume that the register `x6` is initialized to the value 10. What is the final value in register `x5` assuming the `x5` is initially zero?

2.24.2 [5] <§2.7> For the loop above, write the equivalent C code. Assume that the registers `x5` and `x6` are integers `acc` and `i`, respectively.

2.24.3 [5] <§2.7> For the loop written in RISC-V assembly above, assume that the register `x6` is initialized to the value N. How many RISC-V instructions are executed?

2.24.4 [5] <§2.7> For the loop written in RISC-V assembly above, replace the instruction "`beq x6, x0, DONE`" with the instruction "`blt x6, x0, DONE`" and write the equivalent C code.

2.25 [10] <§2.7> Translate the following C code to RISC-V assembly code. Use a minimum number of instructions. Assume that the values of `a`, `b`, `i`, and `j` are in registers `x5`, `x6`, `x7`, and `x29`, respectively. Also, assume that register `x10` 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;
```

2.26 [5] <§2.7> How many RISC-V instructions does it take to implement the C code from Exercise 2.25? 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 RISC-V instructions executed to complete the loop?

2.27 [5] <§2.7> Translate the following loop into C. Assume that the C-level integer `i` is held in register `x5`, `x6` holds the C-level integer called `result`, and `x10` holds the base address of the integer `MemArray`.

```
        addi x6, x0, 0
        addi x29, x0, 100
LOOP:   ld   x7, 0(x10)
        add  x5, x5, x7
        addi x10, x10, 8
        addi x6, x6, 1
        blt  x6, x29, LOOP
```

2.28 [10] <§2.7> Rewrite the loop from Exercise 2.27 to reduce the number of RISC-V instructions executed. Hint: Notice that variable `i` is used only for loop control.

2.29 [30] <§2.8> Implement the following C code in RISC-V assembly. Hint: Remember that the stack pointer must remain aligned on a multiple of 16.

```
int fib(int n){
    if (n==0)
       return 0;
    else if (n == 1)
       return 1;
    else
       return fib(n-1) + fib(n-2);
}
```

2.30 [20] <§2.8> For each function call in Exercise 2.29, show the contents of the stack after the function call is made. Assume the stack pointer is originally at address `0x7ffffffc`, and follow the register conventions as specified in Figure 2.11.

2.31 [20] <§2.8> Translate function `f` into RISC-V assembly language. Assume the function declaration for `g` is `int g(int a, int b)`. The code for function `f` is as follows:

```
int f(int a, int b, int c, int d){

  return g(g(a,b), c+d);
}
```

2.32 [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?

2.33 [5] <§2.8> Right before your function `f` from Exercise 2.31 returns, what do we know about contents of registers `x10-x14`, `x8`, `x1`, and `sp`? Keep in mind that we know what the entire function `f` looks like, but for function `g` we only know its declaration.

2.34 [30] <§2.9> Write a program in RISC-V assembly to convert an ASCII string containing a positive or negative integer decimal string to an integer. Your program should expect register `x10` to hold the address of a null-terminated string containing an optional "+" or "−" followed by 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 `x10`. If a non-digit character appears anywhere in the string, your program should stop with the value −1 in register `x10`. For example, if register `x10` points to a sequence of three bytes 50_{ten}, 52_{ten}, 0_{ten} (the null-terminated string "24"), then when the program stops, register `x10` should contain the value 24_{ten}. The RISC-V `mul` instruction takes two registers as input. There is no "`muli`" instruction. Thus, just store the constant 10 in a register.

2.35 Consider the following code:

```
lb x6, 0(x7)
sd x6, 8(x7)
```

Assume that the register `x7` contains the address `0×10000000` and the data at address is `0×1122334455667788`.

2.35.1 [5] <§2.3, 2.9> What value is stored in `0×10000008` on a big-endian machine?

2.35.2 [5] <§2.3, 2.9> What value is stored in `0×10000008` on a little-endian machine?

2.36 [5] <§2.10> Write the RISC-V assembly code that creates the 64-bit constant $\texttt{0x1122334455667788}_{two}$ and stores that value to register `x10`.

2.37 [10] <§2.11> Write the RISC-V assembly code to implement the following C code as an atomic "set max" operation using the `lr.d/sc.d` instructions. Here, the argument `shvar` contains the address of a shared variable which should be replaced by `x` if `x` is greater than the value it points to:

```
void setmax(int* shvar, int x) {
  // Begin critical section
  if (x > *shvar)
     *shvar = x;
  // End critical section}
}
```

2.38 [5] <§2.11> Using your code from Exercise 2.37 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.

2.39 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.

2.39.1 [5] <§§1.6, 2.13> 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%, while increasing the clock cycle time by only 10%. Is this a good design choice? Why?

2.39.2 [5] <§§1.6, 2.13> 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?

2.40 Assume that for a given program 70% of the executed instructions are arithmetic, 10% are load/store, and 20% are branch.

2.40.1 [5] <§§1.6, 2.13> Given this instruction mix and the assumption that an arithmetic instruction requires two cycles, a load/store instruction takes six cycles, and a branch instruction takes three cycles, find the average CPI.

2.40.2 [5] <§§1.6, 2.13> 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?

2.40.3 [5] <§§1.6, 2.13> 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?

2.41 [10] <§2.19> Suppose the RISC-V ISA included a scaled offset addressing mode similar to the x86 one described in Section 2.17 (Figure 2.35). Describe how you would use scaled offset loads to further reduce the number of assembly instructions needed to carry out the function given in Exercise 2.4.

2.42 [10] <§2.19> Suppose the RISC-V ISA included a scaled offset addressing mode similar to the x86 one described in Section 2.17 (Figure 2.35). Describe how you would use scaled offset loads to further reduce the number of assembly instructions needed to implement the C code given in Exercise 2.7.

Answers to Check Yourself

§2.2, page 66: RISC-V, C, Java.
§2.3, page 73: 2) Very slow.
§2.4, page 80: 2) -8_{ten}
§2.5, page 89: 3) `sub x11, x10, x9`
§2.6, page 92: Both. AND with a mask pattern of 1s will leaves 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 doubleword, with 0s in the rest of the doubleword. Note that AND leaves the field where it was originally, and the shift pair moves the field into the rightmost part of the doubleword.
§2.7, page 97: I. All are true. II. 1).
§2.8, page 108: Both are true.
§2.9, page 113: I. 1) and 2) II. 3).
§2.10, page 121: I. 4) ±4 K. II. 4) ± 1 M.
§2.11, page 124: Both are true.
§2.12, page 133: 4) Machine independence.

3

Arithmetic for Computers

Numerical precision is the very soul of science.

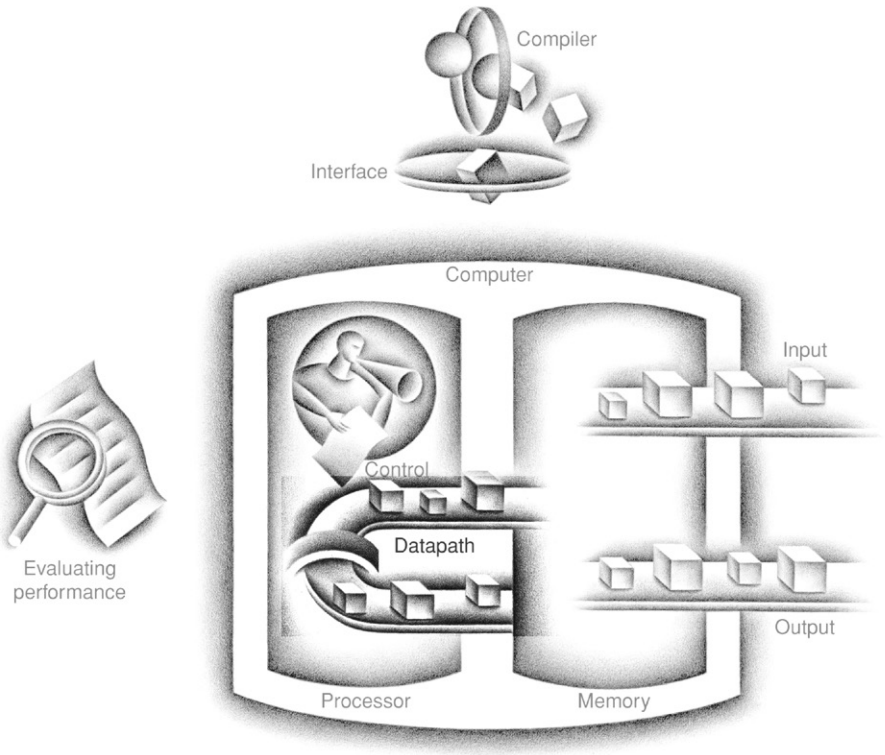
Sir D'arcy Wentworth Thompson,
On Growth and Form, 1917

3.1 **Introduction** 174
3.2 **Addition and Subtraction** 174
3.3 **Multiplication** 177
3.4 **Division** 183
3.5 **Floating Point** 191
3.6 **Parallelism and Computer Arithmetic: Subword Parallelism** 216
3.7 **Real Stuff: Streaming SIMD Extensions and Advanced Vector Extensions in x86** 217

Computer Organization and Design. DOI: http://dx.doi.org/10.1016/B978-0-12-812275-4.00003-8
© 2018 Elsevier Inc. All rights reserved.

3.8 **Going Faster: Subword Parallelism and Matrix Multiply** 218
3.9 **Fallacies and Pitfalls** 222
3.10 **Concluding Remarks** 225
3.11 **Historical Perspective and Further Reading** 227
3.12 **Exercises** 227

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.

$$\begin{array}{rl} & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000111_{two} = 7_{ten} \\ + & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000110_{two} = 6_{ten} \\ \hline = & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00001101_{two} = 13_{ten} \end{array}$$

The 4 bits to the right have all the action; Figure 3.1 shows the sums and carries. Parentheses identify the carries, with the arrows illustrating how they are passed.

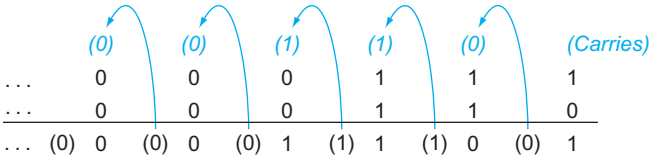


FIGURE 3.1 Binary addition, showing carries from right to left. The rightmost bit adds 1 to 0, resulting in the sum of this bit being 1 and the carry out from this bit being 0. Hence, the operation for the second digit to the right is 0 + 1 + 1. This generates a 0 for this sum bit and a carry out of 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.

ANSWER

Subtracting 6_{ten} from 7_{ten} can be done directly:

$$\begin{array}{rl} & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000111_{two} = 7_{ten} \\ - & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000110_{two} = 6_{ten} \\ \hline = & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001_{two} = 1_{ten} \end{array}$$

or via addition using the two's complement representation of −6:

$$\begin{array}{rl} & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000111_{two} = 7_{ten} \\ + & 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111010_{two} = -6_{ten} \\ \hline = & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001_{two} = 1_{ten} \end{array}$$

Recall that overflow occurs when the result from an operation cannot be represented with the available hardware, in this case a 64-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 64 bits and the sum is no larger than an operand, the sum must fit in 64 bits as well. Therefore, no overflow can occur when adding positive and negative operands.

There are similar restrictions to the occurrence of overflow during subtract, but it's just the opposite principle: when the signs of the operands are the *same*, overflow cannot occur. To see this, remember that $c - a = c + (-a)$ because we subtract by negating the second operand and then add. Therefore, when we subtract operands of the same sign we end up *adding* operands of *different* signs. From the prior paragraph, we know that overflow cannot occur in this case either.

Knowing when an 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 64-bit numbers can yield a result that needs 65 bits to be fully expressed.

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.

The lack of a 65th bit means that when an 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.

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.

Arithmetic Logic Unit (ALU) Hardware that performs addition, subtraction, and usually logical operations such as AND and OR.

Fortunately, the compiler can easily check for unsigned overflow using a branch instruction. Addition has overflowed if the sum is less than either of the addends, whereas subtraction has overflowed if the difference is greater than the minuend.

Appendix A describes the hardware that performs addition and subtraction, which is called an **Arithmetic Logic Unit** or **ALU**.

Hardware/ Software Interface

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 an overflow occurs.

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 recognizing overflow, so today all computers have a way to detect it.

Check Yourself

Some programming languages allow two's complement integer arithmetic on variables declared byte and half, whereas RISC-V only has integer arithmetic operations on full words. As we recall from Chapter 2, RISC-V does have data transfer operations for bytes and halfwords. What RISC-V instructions should be generated for byte and halfword arithmetic operations?

1. Load with `lb`, `lh`; arithmetic with `add`, `sub`, `mul`, `div`, using `and` to mask result to 8 or 16 bits after each operation; then store using `sb`, `sh`.
2. Load with `lb`, `lh`; arithmetic with `add`, `sub`, `mul`, `div`; 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 the 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: The speed of addition depends on how quickly the carry into the high-order bits is computed. 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 A.6 in Appendix A describes.

3.3 Multiplication

Multiplication is vexation, Division is as bad; The rule of three doth puzzle me, And practice drives me mad.

Anonymous, Elizabethan manuscript, 1570

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}:

```
Multiplicand        1000ten
Multiplier    X     1001ten
                    ------
                    1000
                   0000
                  0000
                 1000
                 -------
Product          1001000ten
```

The first operand is called the *multiplicand* and the second the *multiplier*. The final result is called the *product*. As you may recall, the algorithm learned in grammar school is to take the digits of the multiplier one at a time from right to left, multiplying the multiplicand by the single digit of the multiplier, and shifting the intermediate product one digit to the left of the earlier intermediate products.

The first observation is that the number of digits in the product is considerably larger than the number in either the multiplicand or the multiplier. In fact, if we ignore the sign bits, the length of the multiplication of an n-bit multiplicand and an m-bit multiplier is a product that is $n + m$ bits long. That is, $n + m$ bits are required to represent all possible products. Hence, like add, multiply must cope with overflow because we frequently want a 64-bit product as the result of multiplying two 64-bit numbers.

In this example, we restricted the decimal digits to 0 and 1. With only two choices, each step of the multiplication is simple:

1. Just place a copy of the multiplicand (1 × multiplicand) in the proper place if the multiplier digit is a 1, or
2. Place 0 (0 × multiplicand) in the proper place if the digit is 0.

Although the decimal example above 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.

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 flow from top to bottom to resemble more closely the paper-and-pencil method.

Let's assume that the multiplier is in the 64-bit Multiplier register and that the 128-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 64 steps, a 64-bit multiplicand would move 64 bits to the left. Hence, we need a 128-bit Multiplicand register, initialized with the 64-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 128-bit Product register.

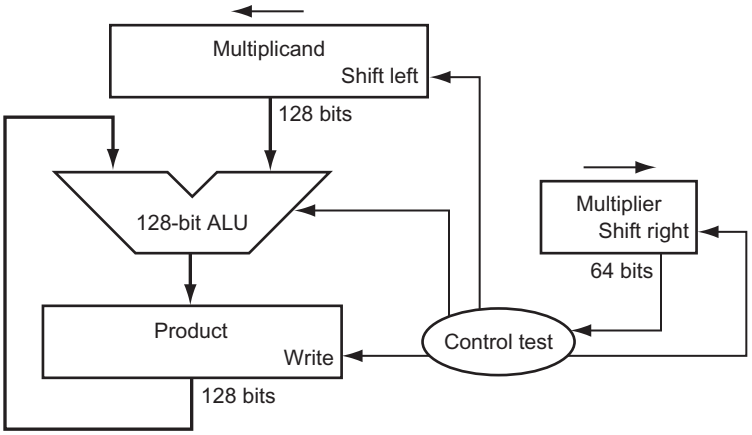


FIGURE 3.3 First version of the multiplication hardware. The Multiplicand register, ALU, and Product register are all 128 bits wide, with only the Multiplier register containing 64 bits. (Appendix A describes ALUs.) The 64-bit multiplicand starts in the right half of the Multiplicand register and is shifted left 1 bit on each step. The multiplier is shifted in the opposite direction at each step. The algorithm starts with the product initialized to 0. Control decides when to shift the Multiplicand and Multiplier registers and when to write new values into the Product register.

Figure 3.4 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 64 times to obtain the product. If each step took a clock cycle, this algorithm would require almost 200 clock cycles to multiply two 64-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 several clock cycles without significantly affecting performance. However, Amdahl's Law (see Section 1.10) reminds us that even a moderate frequency for a slow operation can limit performance.

This algorithm and hardware are easily refined to take one 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.

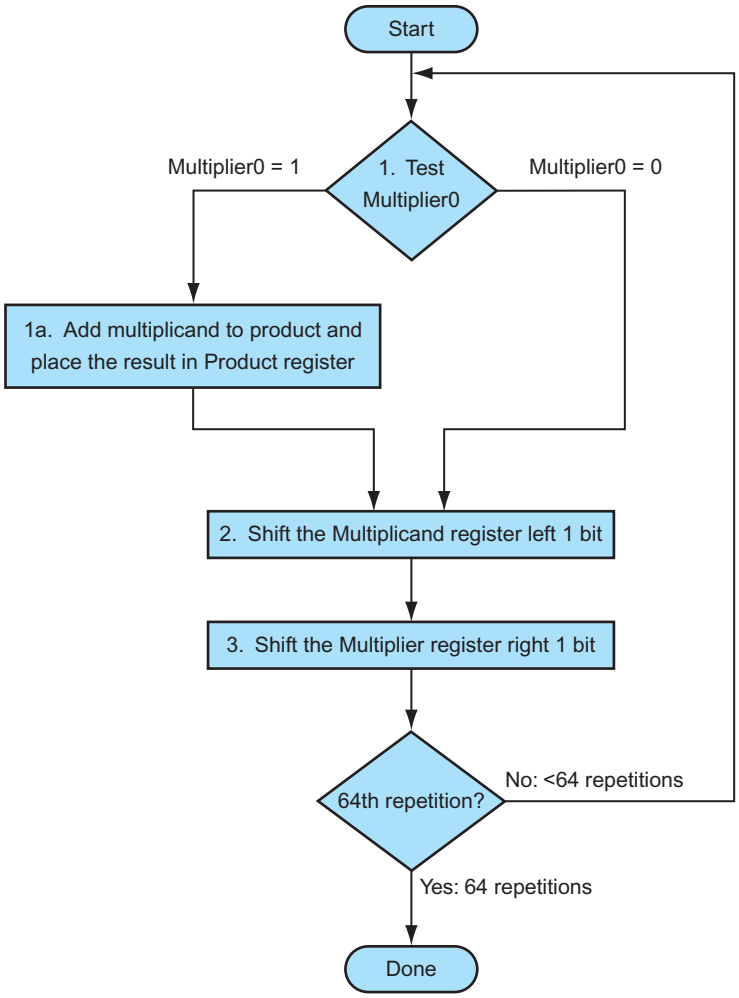


FIGURE 3.4 The first multiplication algorithm, using the hardware shown in Figure 3.3. If the least significant bit of the multiplier is 1, add the multiplicand to the product. If not, go to the next step. Shift the multiplicand left and the multiplier right in the next two steps. These three steps are repeated 64 times.

Hardware/ Software Interface

Replacing arithmetic by shifts can also occur when multiplying by constants. Some compilers replace multiplies by short constants with a series of shifts and adds. Because one bit to the left represents a number twice as large in base 2, shifting the bits left has the same effect as multiplying by a power of 2. As mentioned in Chapter 2, almost every compiler will perform the strength reduction optimization of substituting a left shift for a multiply by a power of 2.

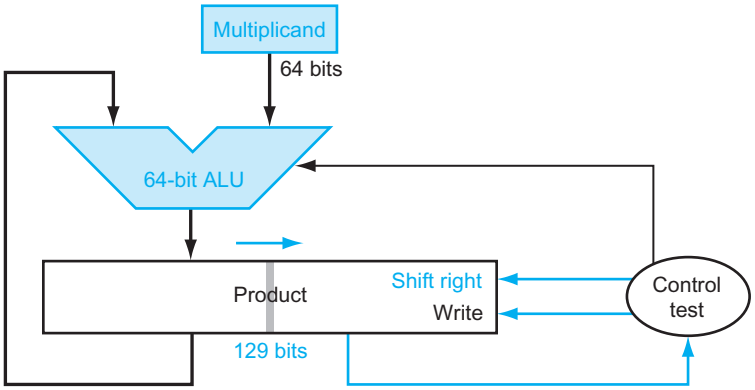


FIGURE 3.5 Refined version of the multiplication hardware. Compare with the first version in Figure 3.3. The Multiplicand register and ALU have been reduced to 64 bits. Now the product is shifted right. The separate Multiplier register also disappeared. The multiplier is placed instead in the right half of the Product register, which has grown by one bit to 129 bits to hold the carry-out of the adder. These changes are highlighted in color.

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	001(1)	0000 0010	0000 0000
1	1a: 1 ⇒ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	000(1)	0000 0100	0000 0010
2	1a: 1 ⇒ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	000(0)	0000 1000	0000 0110
3	1: 0 ⇒ No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	000(0)	0001 0000	0000 0110
4	1: 0 ⇒ No operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

FIGURE 3.6 Multiply example using algorithm in Figure 3.4. The bit examined to determine the next step is circled in color.

A Multiply Algorithm

EXAMPLE

Using 4-bit numbers to save space, multiply $2_{ten} \times 3_{ten}$, or $0010_{two} \times 0011_{two}$.

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 $0000\ 0110_{two}$ or 6_{ten}. Color is used to indicate the register values that change on that step, and the bit circled is the one examined to determine the operation of the next step.

Signed Multiplication

So far, we have dealt with positive numbers. The easiest way to understand how to deal with signed numbers is to first convert the multiplier and multiplicand to positive numbers and then remember their original signs. The algorithms should next 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, if we remember that we are dealing with numbers that have infinite digits, and we are only representing them with 64 bits. Hence, the shifting steps would need to extend the sign of the product for signed numbers. When the algorithm completes, the lower doubleword would have the 64-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 64 multiplier bits. Faster multiplications are possible by essentially providing one 64-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 64 high. An alternative way to organize these 64 additions is in a parallel tree, as Figure 3.7 shows. Instead of waiting for 64 add times, we wait just the $\log_2$ (64) or six 64-bit add times.

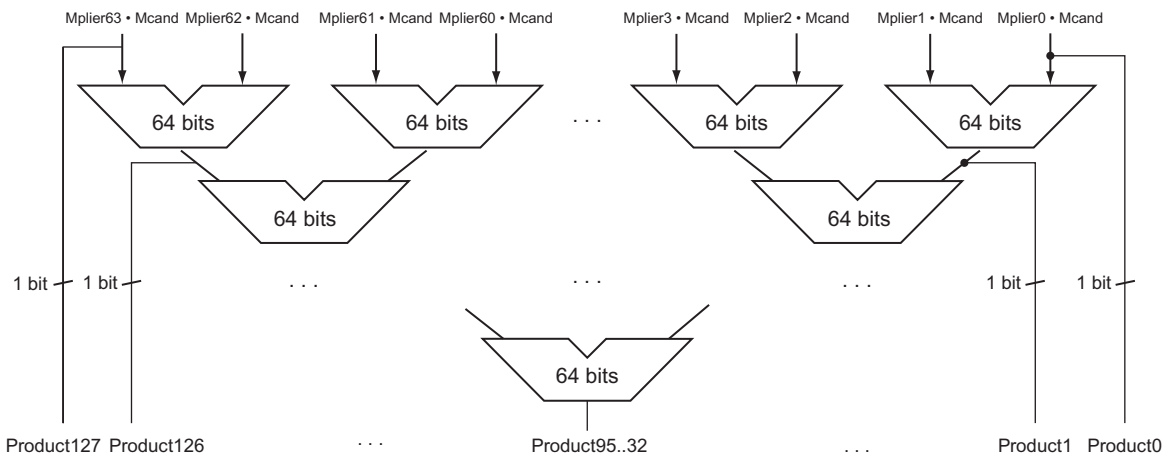
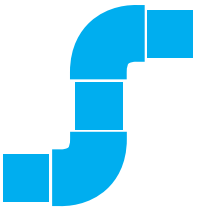


FIGURE 3.7 Fast multiplication hardware. Rather than use a single 64-bit adder 63 times, this hardware "unrolls the loop" to use 63 adders and then organizes them to minimize delay.

In fact, multiply can go even faster than six add times because of the use of *carry save adders* (see Section A.6 in Appendix A), and because it is easy to **pipeline** such a design to be able to support many multiplies simultaneously (see Chapter 4).



Multiply in RISC-V

To produce a properly signed or unsigned 128-bit product, RISC-V has four instructions: *multiply* (`mul`), *multiply high* (`mulh`), *multiply high unsigned* (`mulhu`), and *multiply high signed-unsigned* (`mulhsu`). To get the integer 64-bit product, the programmer uses `mul`. To get the upper 64 bits of the 128-bit product, the programmer uses (`mulh`) if both operands are signed, (`mulhu`) if both operands are unsigned, or (`mulhsu`) if one operand is signed and the other is unsigned.

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

Software can use the multiply-high instructions to check for overflow from 64-bit multiplication. There is no overflow for 64-bit unsigned multiplication if `mulhu`'s result is zero. There is no overflow for 64-bit signed multiplication if all of the bits in `mulh`'s result are copies of the sign bit of `mul`'s result.

3.4 Division

Divide et impera.

Latin for "Divide and rule," ancient political maxim cited by Machiavelli, 1532

The reciprocal operation of multiply is divide, an operation that is even less frequent and even quirkier. 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 division algorithm from grammar school. For reasons similar to those in the previous section, we limit the decimal digits to just 0 or 1. The example is dividing $1{,}001{,}010_{ten}$ by 1000_{ten}:

$$
\begin{array}{rll}
 & 1001_{ten} & \text{Quotient} \\
\text{Divisor } 1000_{ten} & \overline{)\,1001010_{ten}} & \text{Dividend} \\
 & -1000 & \\
 & \overline{10} & \\
 & 101 & \\
 & 1010 & \\
 & -1000 & \\
 & \overline{10_{ten}} & \text{Remainder}
\end{array}
$$

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.

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 division algorithm from grammar school 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 just 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 64-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 64-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 128-bit Divisor register and shift it right 1 bit each step to align it with the dividend. The Remainder register is initialized with the dividend.

Figure 3.9 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

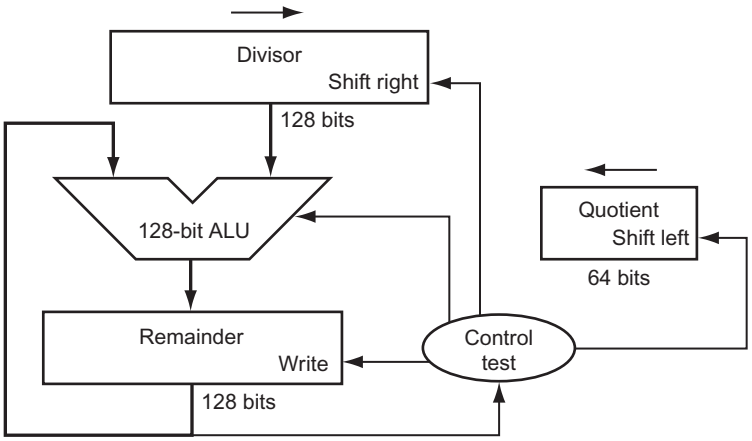


FIGURE 3.8 First version of the division hardware. The Divisor register, ALU, and Remainder register are all 128 bits wide, with only the Quotient register being 62 bits. The 64-bit divisor starts in the left half of the Divisor register and is shifted right 1 bit each iteration. The remainder is initialized with the dividend. Control decides when to shift the Divisor and Quotient registers and when to write the new value into the Remainder register.

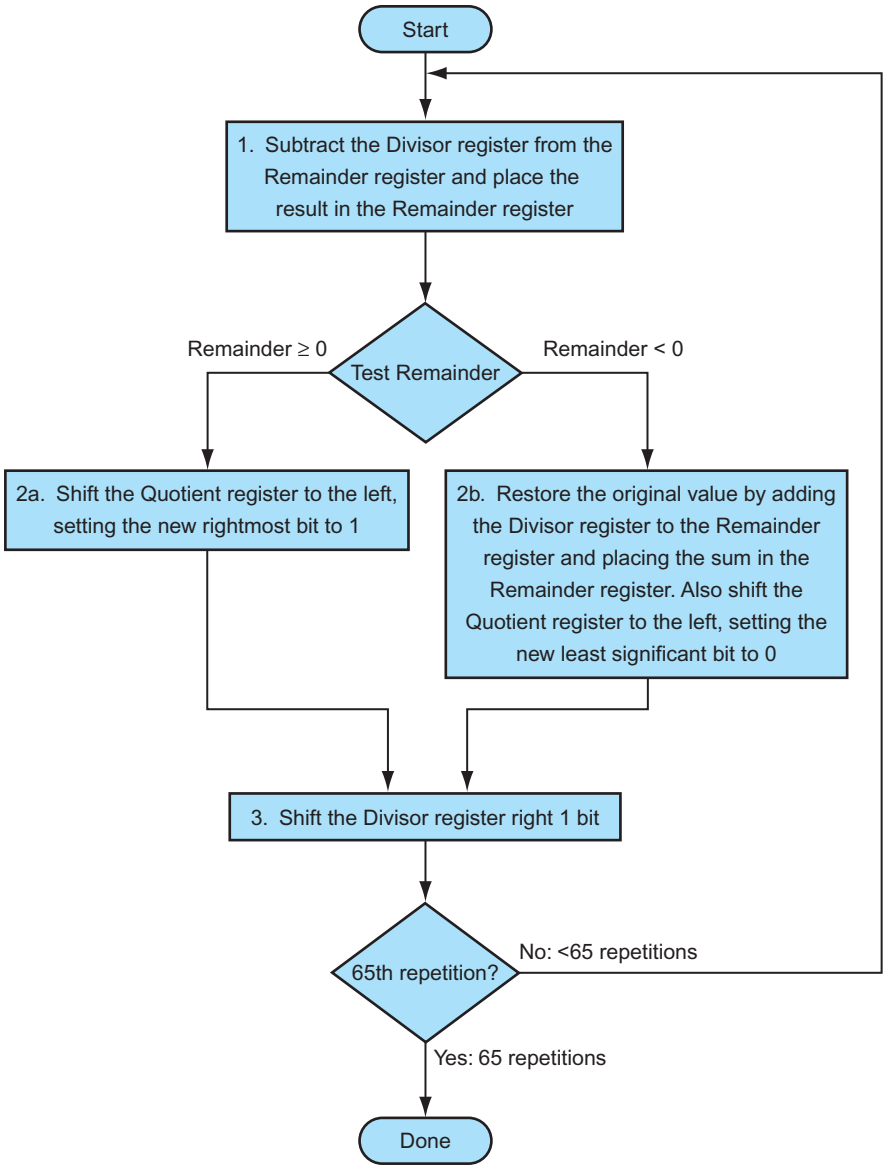


FIGURE 3.9 A division algorithm, using the hardware in Figure 3.8. If the remainder is positive, the divisor did go into the dividend, so step 2a generates a 1 in the quotient. A negative remainder after step 1 means that the divisor did not go into the dividend, so step 2b generates a 0 in the quotient 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 65 times.

than the dividend. It must first subtract the divisor in step 1; remember that this is how we performed comparison. 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 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 $0000\ 0111_{two}$ 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 checks whether the sign bit of the Remainder register is a 0 or 1. The surprising requirement of this algorithm is that it takes $n + 1$ steps to get the proper quotient and remainder.

This algorithm and hardware can be refined to be faster and cheaper. The speed-up comes from shifting the operands and the quotient simultaneously with the

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	①110 0111
	2b: Rem < 0 ⟹ +Div, SLL Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	①111 0111
	2b: Rem < 0 ⟹ +Div, SLL Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	①111 1111
	2b: Rem < 0 ⟹ +Div, SLL Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	⓪000 0011
	2a: Rem ≥ 0 ⟹ SLL Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	⓪000 0001
	2a: Rem ≥ 0 ⟹ SLL Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

FIGURE 3.10 Division example using the algorithm in Figure 3.9. The bit examined to determine the next step is circled in color.

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.

Signed Division

So far, we have ignored signed numbers in division. The simplest solution is to remember the signs of the divisor and dividend and then negate the quotient if the signs disagree.

Elaboration: The one complication of signed division is that we must also set the sign of the remainder. Remember that the following equation must always hold:

$$\text{Dividend} = \text{Quotient} \times \text{Divisor} + \text{Remainder}$$

To understand how to set the sign of the remainder, let's look at the example of dividing all the combinations of $\pm 7_{ten}$ by $\pm 2_{ten}$. The first case is easy:

$$+7 \div +2\text{: Quotient} = +3, + \text{Remainder} = +1$$

Checking the results:

$$+7 = 3 \times 2 + (+1) = 6 + 1$$

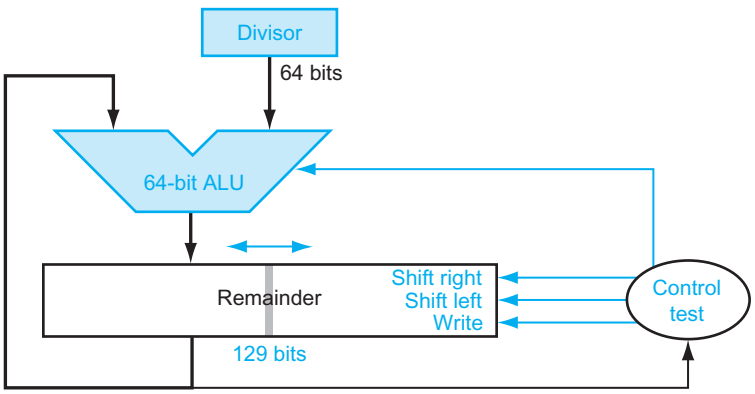


FIGURE 3.11 An improved version of the division hardware. The Divisor register, ALU, and Quotient register are all 64 bits wide. Compared to Figure 3.8, the ALU and Divisor registers are halved and the remainder is shifted left. This version also combines the Quotient register with the right half of the Remainder register. As in Figure 3.5, the Remainder register has grown to 129 bits to make sure the carry out of the adder is not lost.

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:

$$\begin{aligned}\text{Remainder} &= (\text{Dividend} - \text{Quotient} \times \text{Divisor}) = -7 - (-3x + 2)\\ &= -7 - (-6) = \mathbf{-1}\end{aligned}$$

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 identical 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 64 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 224 in Section 3.8 shows what can happen if the table is incorrect.

Divide in RISC-V

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 128-bit register that can shift left or right and a 64-bit ALU that adds or subtracts.

To handle both signed integers and unsigned integers, RISC-V has two instructions for division and two instructions for remainder: *divide* (`div`), *divide unsigned* (`divu`), *remainder* (`rem`), and *remainder unsigned* (`remu`).

Summary

The common hardware support for multiply and divide allows RISC-V to provide a single pair of 64-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 RISC-V architecture for the last two sections.

Hardware/ Software Interface

RISC-V 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. RISC-V 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 $(r + d) \times 2 - d = r - 2 + d \times 2 - d = r \times 2 + d$. This *nonrestoring* division algorithm, which takes one 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.

RISC-V assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	`add x5, x6, x7`	`x5 = x6 + x7`	Three register operands
	Subtract	`sub x5, x6, x7`	`x5 = x6 - x7`	Three register operands
	Add immediate	`addi x5, x6, 20`	`x5 = x6 + 20`	Used to add constants
	Set if less than	`slt x5, x6, x7`	`x5 = 1 if x5 < x6, else 0`	Three register operands
	Set if less than, unsigned	`sltu x5, x6, x7`	`x5 = 1 if x5 < x6, else 0`	Three register operands
	Set if less than, immediate	`slti x5, x6, x7`	`x5 = 1 if x5 < x6, else 0`	Comparison with immediate
	Set if less than immediate, uns.	`sltiu x5, x6, x7`	`x5 = 1 if x5 < x6, else 0`	Comparison with immediate
	Multiply	`mul x5, x6, x7`	`x5 = x6 × x7`	Lower 64 bits of 128-bit product
	Multiply high	`mulh x5, x6, x7`	`x5 = (x6 × x7) >> 64`	Upper 64 bits of 128-bit signed product
	Multiply high, unsigned	`mulhu x5, x6, x7`	`x5 = (x6 × x7) >> 64`	Upper 64 bits of 128-bit unsigned product
	Multiply high, signed-unsigned	`mulhsu x5, x6, x7`	`x5 = (x6 × x7) >> 64`	Upper 64 bits of 128-bit signed-unsigned product
	Divide	`div x5, x6, x7`	`x5 = x6 / x7`	Divide signed 64-bit numbers
	Divide unsigned	`divu x5, x6, x7`	`x5 = x6 / x7`	Divide unsigned 64-bit numbers
	Remainder	`rem x5, x6, x7`	`x5 = x6 % x7`	Remainder of signed 64-bit division
	Remainder unsigned	`remu x5, x6, x7`	`x5 = x6 % x7`	Remainder of unsigned 64-bit division
Data transfer	Load doubleword	`ld x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Doubleword from memory to register
	Store doubleword	`sd x5, 40(x6)`	`Memory[x6 + 40] = x5`	Doubleword from register to memory
	Load word	`lw x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Word from memory to register
	Load word, unsigned	`lwu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Unsigned word from memory to register
	Store word	`sw x5, 40(x6)`	`Memory[x6 + 40] = x5`	Word from register to memory
	Load halfword	`lh x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Halfword from memory to register
	Load halfword, unsigned	`lhu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Unsigned halfword from memory to register
	Store halfword	`sh x5, 40(x6)`	`Memory[x6 + 40] = x5`	Halfword from register to memory
	Load byte	`lb x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Byte from memory to register
	Load byte, unsigned	`lbu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Byte halfword from memory to register
	Store byte	`sb x5, 40(x6)`	`Memory[x6 + 40] = x5`	Byte from register to memory
	Load reserved	`lr.d x5, (x6)`	`x5 = Memory[x6]`	Load; 1st half of atomic swap
	Store conditional	`sc.d x7, x5, (x6)`	`Memory[x6] = x5; x7 = 0/1`	Store; 2nd half of atomic swap
	Load upper immediate	`lui x5, 0x12345`	`x5 = 0x12345000`	Loads 20-bit constant shifted left 12 bits
	Add upper immediate to PC	`auipc x5, 0x12345`	`x5 = PC + 0x12345000`	Used for PC-relative data addressing
Logical	And	`and x5, x6, x7`	`x5 = x6 & x7`	Three reg. operands; bit-by-bit AND
	Inclusive or	`or x5, x6, x8`	`x5 = x6 | x8`	Three reg. operands; bit-by-bit OR
	Exclusive or	`xor x5, x6, x9`	`x5 = x6 ^ x9`	Three reg. operands; bit-by-bit XOR
	And immediate	`andi x5, x6, 20`	`x5 = x6 & 20`	Bit-by-bit AND reg. with constant
	Inclusive or immediate	`ori x5, x6, 20`	`x5 = x6 | 20`	Bit-by-bit OR reg. with constant
	Exclusive or immediate	`xori x5, x6, 20`	`x5 = x6 ^ 20`	Bit-by-bit XOR reg. with constant
Shift	Shift left logical	`sll x5, x6, x7`	`x5 = x6 << x7`	Shift left by register
	Shift right logical	`srl x5, x6, x7`	`x5 = x6 >> x7`	Shift right by register
	Shift right arithmetic	`sra x5, x6, x7`	`x5 = x6 >> x7`	Arithmetic shift right by register
	Shift left logical immediate	`slli x5, x6, 3`	`x5 = x6 << 3`	Shift left by immediate
	Shift right logical immediate	`srli x5, x6, 3`	`x5 = x6 >> 3`	Shift right by immediate
	Shift right arithmetic immediate	`srai x5, x6, 3`	`x5 = x6 >> 3`	Arithmetic shift right by immediate
Conditional branch	Branch if equal	`beq x5, x6, 100`	`if (x5 == x6) go to PC+100`	PC-relative branch if registers equal
	Branch if not equal	`bne x5, x6, 100`	`if (x5 != x6) go to PC+100`	PC-relative branch if registers not equal
	Branch if less than	`blt x5, x6, 100`	`if (x5 < x6) go to PC+100`	PC-relative branch if registers less
	Branch if greater or equal	`bge x5, x6, 100`	`if (x5 >= x6) go to PC+100`	PC-relative branch if registers greater or equal
	Branch if less, unsigned	`bltu x5, x6, 100`	`if (x5 < x6) go to PC+100`	PC-relative branch if registers less
	Branch if greatr/eq, unsigned	`bgeu x5, x6, 100`	`if (x5 >= x6) go to PC+100`	PC-relative branch if registers greater or equal
Uncondit-ional branch	Jump and link	`jal x1, 100`	`x1 = PC+4; go to PC+100`	PC-relative procedure call
	Jump and link register	`jalr x1, 100(x5)`	`x1 = PC+4; go to x5+100`	Procedure return; indirect call

FIGURE 3.12 RISC-V core architecture. RISC-V machine language is listed in the RISC-V Reference Data Card at the front of this book.

3.5 Floating Point

Speed gets you nowhere if you're headed the wrong way.

American proverb

Going beyond signed and unsigned integers, programming languages support numbers with fractions, which are called *reals* in mathematics. Here are some examples of reals:

$3.14159265\ldots_{ten}$ (pi)

$2.71828\ldots_{ten}$ *(e)*

0.000000001_{ten} or $1.0_{ten} \times 10^{-9}$ (seconds in a nanosecond)

$3{,}155{,}760{,}000_{ten}$ or $3.15576_{ten} \times 10^{9}$ (seconds in a typical century)

Notice that in the last case, the number didn't represent a small fraction, but it was bigger than we could represent with a 32-bit signed integer. The alternative notation for the last two numbers is called **scientific notation**, which has a single digit to the left of the decimal point. A number in scientific notation that has no leading 0s is called a **normalized** number, which is the usual way to write it. For example, $1.0_{ten} \times 10^{-9}$ is in normalized scientific notation, but $0.1_{ten} \times 10^{-8}$ and $10.0_{ten} \times 10^{-10}$ are not.

scientific notation A notation that renders numbers with a single digit to the left of the decimal point.

normalized A number in floating-point notation that has no leading 0s.

Just as we can show decimal numbers in scientific notation, we can also show binary numbers in scientific notation:

$$1.0_{two} \times 2^{-1}$$

To keep a binary number in the normalized form, we need a base that we can increase or decrease by exactly the number of bits the number must be shifted to have one nonzero digit to the left of the decimal point. Only a base of 2 fulfills our need. Since the base is not 10, we also need a new name for decimal point; *binary point* will do fine.

Computer arithmetic that supports such numbers is called **floating point** because it represents numbers in which the binary point is not fixed, as it is for integers. The programming language C uses the name *float* for such numbers. Just as in scientific notation, numbers are represented as a single nonzero digit to the left of the binary point. In binary, the form is

floating point Computer arithmetic that represents numbers in which the binary point is not fixed.

$$1.xxxxxxxxx_{two} \times 2^{yyyy}$$

(Although the computer represents the exponent in base 2 as well as the rest of the number, to simplify the notation we show the exponent in decimal.)

A standard scientific notation for reals in the 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 real digits to the right of the binary point replace the unnecessary leading 0s.

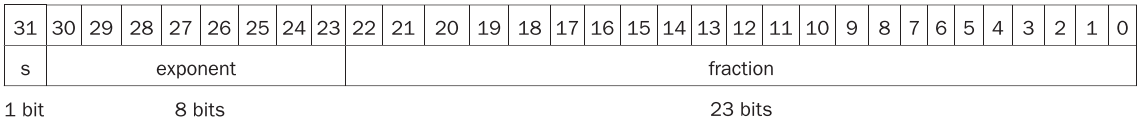
Floating-Point Representation

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.

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 RISC-V 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.



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 RISC-V does something slightly more sophisticated.)

overflow (floating-point) A situation in which a positive exponent becomes too large to fit in the exponent field.

These chosen sizes of exponent and fraction give RISC-V computer arithmetic an extraordinary range. Fractions almost as small as $2.0_{ten} \times 10^{-38}$ and numbers almost as large as $2.0_{ten} \times 10^{38}$ can be represented in a computer. Alas, extraordinary differs from infinite, so it is still possible for numbers to be too large. Thus, overflow interrupts can occur in floating-point arithmetic as well as in integer arithmetic. Notice that **overflow** here means that the exponent is too large to be represented in the exponent field.

underflow (floating-point) A situation in which a negative exponent becomes too large to fit in the exponent field.

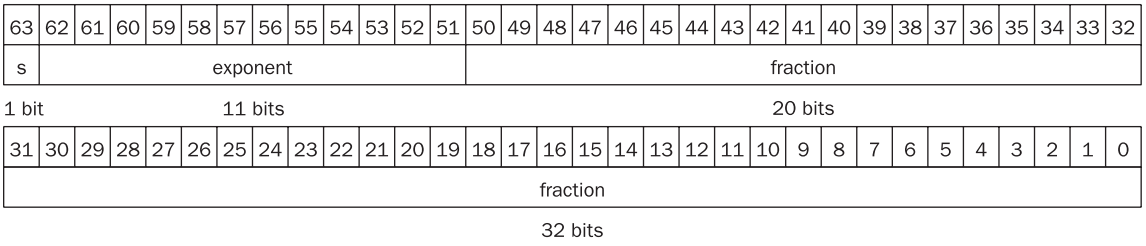
Floating point offers a new kind of exceptional event as well. Just as programmers will want to know when they have calculated a number that is too large to be represented, they will want to know if the nonzero fraction they are calculating has become so small that it cannot be represented; either event could result in a program giving incorrect answers. To distinguish it from overflow, 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 the chances of underflow or overflow is to offer another format that has a larger exponent. In C, this number is called *double*, and operations on doubles are called **double precision** floating-point arithmetic; **single precision** floating point is the name of the earlier format.

double precision A floating-point value represented in a 64-bit doubleword.

single precision A floating-point value represented in a 32-bit word.

The representation of a double precision floating-point number takes one RISC-V doubleword, 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.



RISC-V double precision allows numbers almost as small as $2.0_{ten} \times 10^{-308}$ and almost as large as $2.0_{ten} \times 10^{308}$. Although double precision does increase the exponent range, its primary advantage is its greater precision because of the much larger fraction.

Exceptions and Interrupts

What should happen on an overflow or underflow to let the user know that a problem occurred? Some computers signal these events by raising an **exception**, sometimes called an **interrupt**. 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.) RISC-V computers do *not* raise an exception on overflow or underflow; instead, software can read the *floating-point control and status register* (fcsr) to check whether overflow or underflow has occurred.

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.)

IEEE 754 Floating-Point Standard

These formats go beyond RISC-V. 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 number, IEEE 754 makes the leading 1 bit of normalized binary numbers implicit. Hence, the number is actually 24 bits long in single precision (implied 1 and a 23-bit fraction), and 53 bits long in double precision (1 + 52). To be precise, we use the term *significand* to represent the 24- or

53-bit number that is 1 plus the fraction, and *fraction* when we mean the 23- or 52-bit number. Since 0 has no leading 1, it is given the reserved exponent value 0 so that the hardware won't attach a leading 1 to it.

Thus 00 … 00_{two} represents 0; the representation of the rest of the numbers uses the form from before with the hidden 1 added:

$$(-1)^S \times (1 + \text{Fraction}) \times 2^E$$

where the bits of the fraction represent a number between 0 and 1 and E specifies the value in the exponent field, to be given in detail shortly. If we number the bits of the fraction from *left to right* s1, s2, s3, …, then the value is

$$(-1)^S \times (1 + (s1 \times 2^{-1}) + (s2 \times 2^{-2}) + (s3 \times 2^{-3}) + (s4 \times 2^{-4}) + \ldots) \times 2^E$$

Figure 3.13 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 output 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.)

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	± denormalized number
1–254	Anything	1–2046	Anything	± floating-point number
255	0	2047	0	± infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

FIGURE 3.13 IEEE 754 encoding of floating-point numbers. A separate sign bit determines the sign. Denormalized numbers are described in the *Elaboration* on page 216. This information is also found in Column 4 of the RISC-V Reference Data Card at the front of this book.

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, $1.0_{two} \times 2^{-1}$ would be represented in a single precision 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
●	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	0	0

(Remember that the leading 1 is implicit in the significand.) The value $1.0_{two} \times 2^{+1}$ would look like the smaller binary number

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

The desirable notation must therefore represent the most negative exponent as $00 \ldots 00_{two}$ and the most positive as $11 \ldots 11_{two}$. 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 $-1 + 127_{ten}$, or $126_{ten} = 0111\ 1110_{two}$, and $+1$ is represented by $1 + 127$, or $128_{ten} = 1000\ 0000_{two}$. 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

$$\pm 1.00000000000000000000000_{two} \times 2^{-126}$$

to as large as

$$\pm 1.11111111111111111111111_{two} \times 2^{+127}.$$

Let's demonstrate.

EXAMPLE

Show the IEEE 754 binary representation of the number -0.75_{ten} in single and double precision.

ANSWER

The number -0.75_{ten} is also

$$-3/4_{ten} \text{ or } -3/2^2_{ten}$$

It is also represented by the binary fraction

$$-11_{two}/2^2_{ten} \text{ or } -0.11_{two}$$

In scientific notation, the value is

$$-0.11_{two} \times 2^0$$

and in normalized scientific notation, it is

$$-1.1_{two} \times 2^{-1}$$

The general representation for a single precision number is

$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent}-127)}$$

Subtracting the bias 127 from the exponent of $-1.1_{two} \times 2^{-1}$ yields

$$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 000_{two}) \times 2^{(126-127)}$$

The single precision binary representation of -0.75_{ten} is then

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

1 bit 8 bits 23 bits

The double precision representation is

$$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two}) \times 2^{(1022-1023)}$$

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

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

32 bits

Now let's try going the other direction.

Converting Binary to Decimal Floating Point

EXAMPLE

What decimal number does this single precision float represent?

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	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"). The revised standard also adds decimal floating point arithmetic.

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 old hex format.

Floating-Point Addition

Let's add numbers in scientific notation by hand to illustrate the problems in floating-point addition: $9.999_{ten} \times 10^{1} + 1.610_{ten} \times 10^{-1}$. Assume that we can store only four decimal digits of the significand and two decimal digits of the exponent.

Step 1. To be able to add these numbers properly, we must align the decimal point of the number that has the smaller exponent. Hence, we need a form of the smaller number, $1.610_{ten} \times 10^{-1}$, that matches the larger exponent. We obtain this by observing that there are multiple representations of an unnormalized floating-point number in scientific notation:

$$1.610_{ten} \times 10^{-1} = 0.1610_{ten} \times 10^{0} = 0.01610_{ten} \times 10^{1}$$

The number on the right is the version we desire, since its exponent matches the exponent of the larger number, $9.999_{ten} \times 10^{1}$. Thus, the first step shifts the significand of the smaller number to the right until its corrected exponent matches that of the larger number. But we can represent only four decimal digits so, after shifting, the number is really

$$0.016 \times 10^{1}$$

Step 2. Next comes the addition of the significands:

$$\begin{array}{rr} & 9.999_{ten} \\ + & 0.016_{ten} \\ \hline & 10.015_{ten} \end{array}$$

The sum is $10.015_{ten} \times 10^{1}$.

Step 3. This sum is not in normalized scientific notation, so we need to adjust it:

$$10.015_{ten} \times 10^{1} = 1.0015_{ten} \times 10^{2}$$

Thus, after the addition we may have to shift the sum to put it into normalized form, adjusting the exponent appropriately. This example shows shifting to the right, but if one number were positive and the other were negative, it would be possible for the sum to have many leading 0s, requiring left shifts. Whenever the exponent is increased or decreased, we must check for overflow or underflow—that is, we must make sure that the exponent still fits in its field.

Step 4. Since we assumed that the significand could be only four digits long (excluding the sign), we must round the number. In our grammar school algorithm, the rules truncate the number if the digit to the right of the desired point is between 0 and 4 and add 1 to the digit if the number to the right is between 5 and 9. The number

$$1.0015_{ten} \times 10^{2}$$

is rounded to four digits in the significand to

$$1.002_{ten} \times 10^{2}$$

since the fourth digit to the right of the decimal point was between 5 and 9. Notice that if we have bad luck on rounding, such as adding 1 to a string of 9s, the sum may no longer be normalized and we would need to perform step 3 again.

Figure 3.14 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 216). For the example below, remember that for single precision, the maximum exponent is 127, and the minimum exponent is −126.

Binary Floating-Point Addition

EXAMPLE

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:

$$\begin{aligned}
0.5_{ten} &= 1/2_{ten} &&= 1/2^{1}_{ten} \\
&= 0.1_{two} &&= 0.1_{two} \times 2^{0} &&= 1.000_{two} \times 2^{-1} \\
-0.4375_{ten} &= -7/16_{ten} &&= -7/2^{4}_{ten} \\
&= -0.0111_{two} &&= -0.0111_{two} \times 2^{0} &&= -1.110_{two} \times 2^{-2}
\end{aligned}$$

Now we follow the algorithm:

Step 1. The significand of the number with the lesser exponent ($-1.11_{two} \times 2^{-2}$) is shifted right until its exponent matches the larger number:

$$-1.110_{two} \times 2^{-2} = -0.111_{two} \times 2^{-1}$$

Step 2. Add the significands:

$$1.000_{two} \times 2^{-1} + (-0.111_{two} \times 2^{-1}) = 0.001_{two} \times 2^{-1}$$

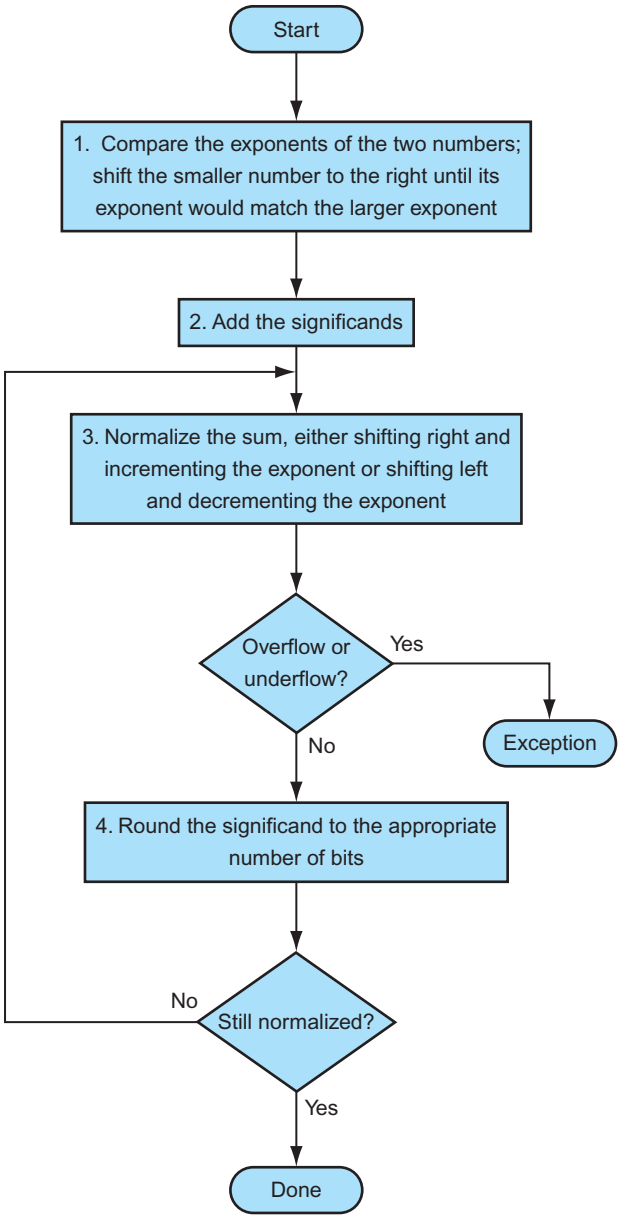


FIGURE 3.14 Floating-point addition. The normal path is to execute steps 3 and 4 once, but if rounding causes the sum to be unnormalized, we must repeat step 3.

Step 3. Normalize the sum, checking for overflow or underflow:

$$0.001_{two} \times 2^{-1} = 0.010_{two} \times 2^{-2} = 0.100_{two} \times 2^{-3} = 1.000_{two} \times 2^{-4}$$

Since $127 \geq -4 \geq -126$, there is no overflow or underflow. (The biased exponent would be $-4 + 127$, or 123, which is between 1 and 254, the smallest and largest unreserved biased exponents.)

Step 4. Round the sum:

$$1.000_{two} \times 2^{-4}$$

The sum already fits exactly in 4 bits, so there is no change to the bits due to rounding.

This sum is then

$$1.000_{two} \times 2^{-4} = 0.0001000_{two} = 0.0001_{two} = 1/2^{4}_{ten} = 1/16_{ten} = 0.0625_{ten}$$

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: $1.110_{ten} \times 10^{10} \times 9.200_{ten} \times 10^{-5}$. Assume that we can store only four digits of the significand and two digits of the exponent.

Step 1. Unlike addition, we calculate the exponent of the product by simply adding the exponents of the operands together:

$$\text{New exponent} = 10 + (-5) = 5$$

Let's do this with the biased exponents as well to make sure we obtain the same result: $10 + 127 = 137$, and $-5 + 127 = 122$, so

$$\text{New exponent} = 137 + 122 = 259$$

This result is too large for the 8-bit exponent field, so something is amiss! The problem is with the bias because we are adding the biases as well as the exponents:

$$\text{New exponent} = (10 + 127) + (-5 + 127) = (5 + 2 \times 127) = 259$$

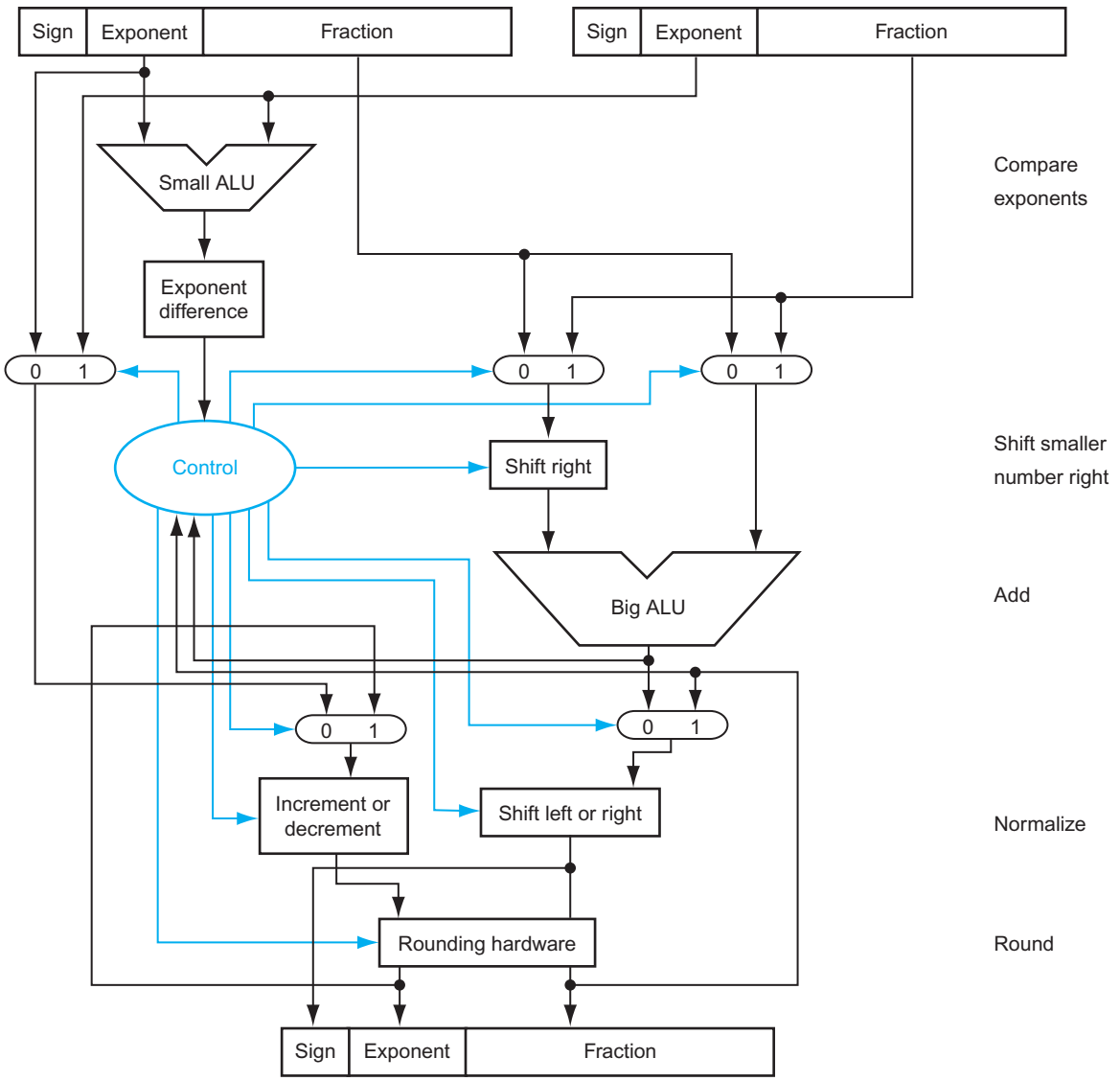


FIGURE 3.15 Block diagram of an arithmetic unit dedicated to floating-point addition. The steps of Figure 3.14 correspond to each block, from top to bottom. First, the exponent of one operand is subtracted from the other using the small ALU to determine which is larger and by how much. This difference 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.

Accordingly, to get the correct biased sum when we add biased numbers, we must subtract the bias from the sum:

$$\text{New exponent} = 137 + 122 - 127 = 259 - 127 = 132 = (5 + 127)$$

and 5 is indeed the exponent we calculated initially.

Step 2. Next comes the multiplication of the significands:

$$\begin{array}{r} 1.110_{\text{ten}} \\ \times \quad 9.200_{\text{ten}} \\ \hline 0000 \\ 0000 \\ 2220 \\ 9990 \\ \hline 1110000_{\text{ten}} \end{array}$$

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:

$$10.212000_{\text{ten}}$$

If 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 signs of the operands determine the sign of the product.

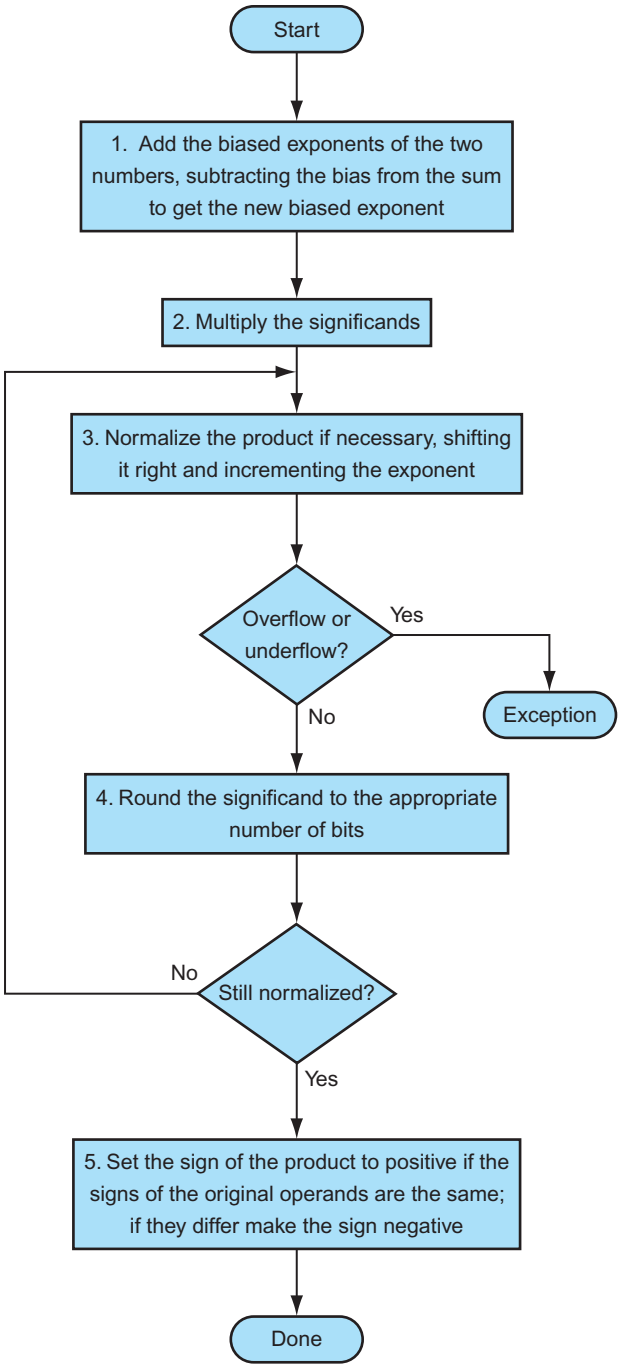


FIGURE 3.16 Floating-point multiplication. The normal path is to execute steps 3 and 4 once, but if rounding causes the sum to be unnormalized, we must repeat step 3.

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).

Binary Floating-Point Multiplication

EXAMPLE

Let's try multiplying the numbers 0.5_{ten} and -0.4375_{ten}, using the steps in Figure 3.16.

ANSWER

In binary, the task is multiplying $1.000_{two} \times 2^{-1}$ by $-1.110_{two} \times 2^{-2}$.

Step 1. Adding the exponents without bias:

$$-1 + (-2) = -3$$

or, using the biased representation:

$$(-1 + 127) + (-2 + 127) - 127 = (-1 - 2) + (127 + 127 - 127)$$
$$= -3 + 127 = 124$$

Step 2. Multiplying the significands:

$$\begin{array}{r} 1.000_{two} \\ \times\ 1.110_{two} \\ \hline 0000 \\ 1000 \\ 1000 \\ 1000 \\ \hline 1110000_{two} \end{array}$$

The product is $1.110000_{two} \times 2^{-3}$, but we need to keep it to 4 bits, so it is $1.110_{two} \times 2^{-3}$.

Step 3. Now we check the product to make sure it is normalized, and then check the exponent for overflow or underflow. The product is already normalized and, since $127 \geq -3 \geq -126$, there is no overflow or underflow. (Using the biased representation, $254 \geq 124 \geq 1$, so the exponent fits.)

Step 4. Rounding the product makes no change:

$$1.110_{two} \times 2^{-3}$$

Step 5. Since the signs of the original operands differ, make the sign of the product negative. Hence, the product is

$$-1.110_{\text{two}} \times 2^{-3}$$

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 RISC-V

RISC-V supports the IEEE 754 single-precision and double-precision formats with these instructions:

- Floating-point *addition, single* (`fadd.s`) and *addition, double* (`fadd.d`)
- Floating-point *subtraction, single* (`fsub.s`) and *subtraction, double* (`fsub.d`)
- Floating-point *multiplication, single* (`fmul.s`) and *multiplication, double* (`fmul.d`)
- Floating-point *division, single* (`fdiv.s`) and *division, double* (`fdiv.d`)
- Floating-point square root, single (`fsqrt.s`) and square root, double (`fsqrt.d`)
- Floating-point equals, single (`feq.s`) and equals, double (`feq.d`)
- Floating-point less-than, single (`flt.s`) and less-than, double (`flt.d`)
- Floating-point less-than-or-equals, single (`fle.s`) and less-than-or-equals, double (`fle.d`)

The comparison instructions, `feq`, `flt`, and `fle`, set an integer register to 0 if the comparison is false and 1 if it is true. Software can thus branch on the result of a floating-point comparison using the integer branch instructions `beq` and `bne`.

The RISC-V designers decided to add separate floating-point registers. They are called `f0`, `f1`, …, `f31`. Hence, they included separate loads and stores for floating-point registers: `fld` and `fsd` for double-precision and `flw` and `fsw` for single-precision. The base registers for floating-point data transfers which are used for addresses remain integer registers. The RISC-V code to load two single precision numbers from memory, add them, and then store the sum might look like this:

```
flw    f0, 0(x10) //  Load 32-bit F.P. number into f0
flw    f1, 4(x10) //  Load 32-bit F.P. number into f1
fadd.s f2, f0, f1 //  f2 = f0 + f1, single precision
fsw    f2, 8(x10) //  Store 32-bit F.P. number from f2
```

RISC-V floating-point operands

Name	Example	Comments
32 floating-point registers	f0-f31	An *f*-register can hold either a single-precision floating-point number or a double-precision floating-point number.
2^{61} memory double words	Memory[0], Memory[8], ..., Memory[18,446,744,073,709,551,608]	Accessed only by data transfer instructions. RISC-V uses byte addresses, so sequential doubleword accesses differ by 8. Memory holds data structures, arrays, and spilled registers.

RISC-V floating-point assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	FP add single	fadd.s f0, f1, f2	f0 = f1 + f2	FP add (single precision)
	FP subtract single	fsub.s f0, f1, f2	f0 = f1 - f2	FP subtract (single precision)
	FP multiply single	fmul.s f0, f1, f2	f0 = f1 * f2	FP multiply (single precision)
	FP divide single	fdiv.s f0, f1, f2	f0 = f1 / f2	FP divide (single precision)
	FP square root single	fsqrt.s f0, f1	f0 = √f1	FP square root (single precision)
	FP add double	fadd.d f0, f1, f2	f0 = f1 + f2	FP add (double precision)
	FP subtract double	fsub.d f0, f1, f2	f0 = f1 - f2	FP subtract (double precision)
	FP multiply double	fmul.d f0, f1, f2	f0 = f1 * f2	FP multiply (double precision)
	FP divide double	fdiv.d f0, f1, f2	f0 = f1 / f2	FP divide (double precision)
	FP square root double	fsqrt.d f0, f1	f0 = √f1	FP square root (double precision)
Comparison	FP equality single	feq.s x5, f0, f1	x5 = 1 if f0 == f1, else 0	FP comparison (single precision)
	FP less than single	flt.s x5, f0, f1	x5 = 1 if f0 < f1, else 0	FP comparison (single precision)
	FP less than or equals single	fle.s x5, f0, f1	x5 = 1 if f0 <= f1, else 0	FP comparison (single precision)
	FP equality double	feq.d x5, f0, f1	x5 = 1 if f0 == f1, else 0	FP comparison (double precision)
	FP less than double	flt.d x5, f0, f1	x5 = 1 if f0 < f1, else 0	FP comparison (double precision)
	FP less than or equals double	fle.d x5, f0, f1	x5 = 1 if f0 <= f1, else 0	FP comparison (double precision)
Data transfer	FP load word	flw f0, 4(x5)	f0 = Memory[x5 + 4]	Load single-precision from memory
	FP load doubleword	fld f0, 8(x5)	f0 = Memory[x5 + 8]	Load double-precision from memory
	FP store word	fsw f0, 4(x5)	Memory[x5 + 4] = f0	Store single-precision from memory
	FP store doubleword	fsd f0, 8(x5)	Memory[x5 + 8] = f0	Store double-precision from memory

FIGURE 3.17 RISC-V floating-point architecture revealed thus far. This information is also found in column 2 of the RISC-V Reference Data Card at the front of this book.

A single precision register is just the lower half of a double-precision register. Note that, unlike integer register x0, floating-point register f0 is *not* hard-wired to the constant 0.

Figure 3.17 summarizes the floating-point portion of the RISC-V architecture revealed in this chapter, with the new pieces to support floating point shown in color. The floating-point instructions use the same format as their integer counterparts: loads use the I-type format, stores use the S-type format, and arithmetic instructions use the R-type format.

Hardware/ Software Interface

One issue that architects face in supporting floating-point arithmetic is whether to select 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 distinct 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 RISC-V Assembly Code

Let's convert a temperature in Fahrenheit to Celsius:

```
float f2c (float fahr)
{
    return ((5.0f/9.0f) *(fahr - 32.0f));
}
```

Assume that the floating-point argument `fahr` is passed in `f10` and the result should also go in `f10`. What is the RISC-V assembly code?

ANSWER

We assume that the compiler places the three floating-point constants in memory within easy reach of register `x3`. The first two instructions load the constants 5.0 and 9.0 into floating-point registers:

```
f2c:
    flw f0, const5(x3) // f0 = 5.0f
    flw f1, const9(x3) // f1 = 9.0f
```

They are then divided to get the fraction 5.0/9.0:

```
    fdiv.s f0, f0, f1 // f0 = 5.0f / 9.0f
```

(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` (`f10`):

```
flw    f1, const32(x3) // f1 = 32.0f
fsub.s f10, f10, f1    // f10 = fahr - 32.0f
```

Finally, we multiply the two intermediate results, placing the product in `f10` as the return result, and then return

```
fmul.s f10, f0, f10 // f10 = (5.0f / 9.0f)*(fahr - 32.0f)
jalr   x0, 0(x1)   // 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 RISC-V

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[][])
{
          size_t 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 `x10`, `x11`, and `x12`. Assume that the integer variables are in `x5`, `x6`, and `x7`, respectively. What is the RISC-V assembly code for the body of the procedure?

ANSWER

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 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 pseudoinstruction li, which loads a constant into a register.

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 x28, 32    //  x28 = 32 (row size/loop end)
       li x5, 0      //  i = 0; initialize 1st for loop
L1:    li x6, 0      //  j = 0; initialize 2nd for loop
L2:    li x7, 0      //  k = 0; initialize 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-dimensional 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:

```
slli  x30, x5, 5    //  x30 = i * 2^5(size of row of c)
```

Now we add the second index to select the jth element of the desired row:

```
add  x30, x30, x6    //  x30 = 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 three:

```
slli  x30, x30, 3    //  x30 = 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 f0:

```
add  x30, x10, x30  //  x30 = byte address of c[i][j]
fld  f0, 0(x30)     //  f0 = 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: slli  x29, x7, 5     //  x29 = k * 2^5(size of row of b)
    add   x29, x29, x6   //  x29 = k * size(row) + j
    slli  x29, x29, 3    //  x29 = byte offset of [k][j]
    add   x29, x12, x29  //  x29 = byte address of b[k][j]
    fld   f1, 0(x29)     //  f1 = 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]`.

```
slli  x29, x5, 5     //  x29 = i * 2^5(size of row of a)
add   x29, x29, x7   //  x29 = i * size(row) + k
slli  x29, x29, 3    //  x29 = byte offset of [i][k]
add   x29, x11, x29  //  x29 = byte address of a[i][k]
fld   f2, 0(x29)     //  f2 = 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 `f2` and `f1`, and then accumulate the sum in `f0`.

```
fmul.d  f1, f2, f1  // f1 = a[i][k] * b[k][j]
fadd.d  f0, f0, f1  // f0 = 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 `f0` into `c[i][j]`.

```
addi  x7, x7, 1    //  k = k + 1
bltu  x7, x28, L3  //  if (k < 32) go to L3
fsd   f0, 0(x30)   //  c[i][j] = f0
```

Similarly, these final six 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.

```
addi  x6, x6, 1    //  j = j + 1
bltu  x6, x28, L2  //  if (j < 32) go to L2
addi  x5, x5, 1    //  i = i + 1
bltu  x5, x28, L1  //  if (i < 32) go to L1
. . .
```

Looking ahead, Figure 3.20 below shows the x86 assembly language code for a slightly different version of DGEMM in Figure 3.19.

Elaboration: C and many other programming languages use the array layout discussed in the example, called *row-major order*. Fortran instead uses *column-major order*, whereby the array is stored column by column.

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 *coprocessor chips*. Since the early 1990s, microprocessors have integrated floating point (and just about everything else) on chip, and thus the term *coprocessor chip* 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 produce 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 accesses. It would also need to check that the object reference is not null.

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. It is also the name of the second of two extra bits kept on the right during intermediate floating-point calculations, which improves rounding accuracy.

Accurate Arithmetic

Unlike integers, which can represent exactly every number between the smallest and largest number, floating-point numbers are normally approximations for a number they can't really represent. The reason is that an infinite variety of real numbers exists between, say, 1 and 2, 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 intervening additions, called **guard** and **round**, respectively. Let's do a decimal example to illustrate their value.

Rounding with Guard Digits

EXAMPLE

Add $2.56_{ten} \times 10^0$ to $2.34_{ten} \times 10^2$, assuming that we have three significant decimal digits. Round to the nearest decimal number with three significant decimal digits, first with guard and round digits, and then without them.

ANSWER

First we must shift the smaller number to the right to align the exponents, so $2.56_{ten} \times 10^0$ becomes $0.0256_{ten} \times 10^2$. Since we have guard and round digits, we are able to represent the two least significant digits when we align exponents. The guard digit holds 5 and the round digit holds 6. The sum is

$$\begin{array}{r} 2.3400_{ten} \\ +0.0256_{ten} \\ \hline 2.3656_{ten} \end{array}$$

Thus the sum is $2.3656_{ten} \times 10^2$. Since we have two digits to round, we want values 0 to 49 to round down and 51 to 99 to round up, with 50 being the tiebreaker. Rounding the sum up with three significant digits yields $2.37_{ten} \times 10^2$.

Doing this *without* guard and round digits drops two digits from the calculation. The new sum is then

$$\begin{array}{r} 2.34_{ten} \\ +0.02_{ten} \\ \hline 2.36_{ten} \end{array}$$

The answer is $2.36_{ten} \times 10^2$, off by 1 in the last digit from the sum above.

Since the worst case for rounding would be when the actual number is halfway between two floating-point representations, accuracy in floating point is normally measured in terms of the number of bits in error in the least significant bits of the significand; the measure is called the number of **units in the last place**, or **ulp**. If a number were off by 2 in the least significant bits, it would be called off by 2 ulps. Provided there are 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 require 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.

sticky bit A bit used in rounding in addition to guard and round that is set whenever there are nonzero bits to the right of the round bit.

The 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 $0.50 \ldots 00_{ten}$ and $0.50 \ldots 01_{ten}$ when rounding.

The sticky bit may be set, for example, during addition, when the smaller number is shifted to the right. Suppose we added $5.01_{ten} \times 10^{-1}$ to $2.34_{ten} \times 10^{2}$ in the example above. Even with guard and round, we would be adding 0.0050 to 2.34, with a sum of 2.3450. The sticky bit would be set, since there are nonzero bits to the right. Without the sticky bit to remember whether any 1s were shifted off, we would assume the number is equal to 2.345000 … 00 and round to the nearest even of 2.34. With the sticky bit to remember that the number is larger than 2.345000 … 00, we round instead to 2.35.

fused multiply add A floating-point instruction that performs both a multiply and an add, but rounds only once after the add.

Elaboration: RISC-V, MIPS-64, PowerPC, SPARC64, AMD SSE5, and Intel AVX architectures all 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 revised IEEE 754-2008 standard (see **Section 3.11**).

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

$$(-1)^{5} \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

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 BIG Picture

Bit patterns have no inherent meaning. They may represent signed integers, unsigned integers, floating-point numbers, instructions, character strings, and so on. What is represented depends on the instruction that operates on the bits in the word.

The major difference between computer numbers and numbers in the real world is that computer numbers have limited size and hence limited precision; it's possible to calculate a number too big or too small to be represented in a computer word. Programmers must remember these limits and write programs accordingly.

C type	Java type	Data transfers	Operations
long long int	long	ld, sd	add, sub, addi, mul, mulh, mulhu, mulhsu, div, divu, rem, remu, and, andi, or, ori, xor, xori
unsigned long long int	–	ld, sd	add, sub, addi, mul, mulh, mulhu, mulhsu, div, divu, rem, remu, and, andi, or, ori, xor, xori
char	–	lb, sb	add, sub, addi, mul, div, divu, rem, remu, and, andi, or, ori, xor, xori
short	char	lh, sh	add, sub, addi, mul, div, divu, rem, remu, and, andi, or, ori, xor, xori
float	float	flw, fsw	fadd.s, fsub.s, fmul.s, fdiv.s, feq.s, flt.s, fle.s
double	double	fld, fsd	fadd.d, fsub.d, fmul.d, fdiv.d, feq.d, flt.d, fle.d

Hardware/ Software Interface

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 data transfer instructions, and instructions that operate on those types that appear in Chapter 2 and this chapter. Note that Java omits unsigned integers.

Check Yourself

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?

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, $\pm\ \infty$, NaN
3. $\pm 1.0000\ 0000\ 00 \times 2^{-14}$ to $\pm\ 1.1111\ 1111\ 11 \times 2^{15}$, $\pm$ 0, $\pm\ \infty$, NaN
4. $\pm 1.0000\ 0000\ 00 \times 2^{-15}$ to $\pm\ 1.1111\ 1111\ 11 \times 2^{14}$, $\pm$ 0, $\pm\ \infty$, NaN

Elaboration: To accommodate comparisons that may include NaNs, the standard includes *ordered* and *unordered* as options for compares. RISC-V does not provide instructions for unordered comparisons, but a careful sequence of ordered comparisons has the same effect. (Java does not support unordered compares.)

In an attempt to squeeze every 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

$$1.0000\,0000\,0000\,0000\,0000\,000_{two} \times 2^{-126}$$

but the smallest single precision denormalized number is

$$0.0000\,0000\,0000\,0000\,0000\,001_{two} \times 2^{-126}, \text{ or } 1.0_{two} \times 2^{-149}$$

For double precision, the denorm gap goes from 1.0×2^{-1022} to 1.0×2^{-1074}.

The possibility of an occasional unnormalized operand has given headaches to floating-point designers who are trying to build fast floating-point units. Hence, many computers cause an exception if an operand is denormalized, letting software complete the operation. Although software implementations are perfectly valid, their lower performance has lessened the popularity of denorms in portable floating-point software. Moreover, if programmers do not expect denorms, their programs may surprise them.

3.6 Parallelism and Computer Arithmetic: Subword Parallelism

Since every microprocessor in a phone, tablet, or laptop by definition has its own graphical display, 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 these 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 yet the speedups could be large.

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 are known as well as vector or SIMD, for single instruction, multiple data (see Section 6.6). The rising popularity of multimedia applications led to arithmetic instructions that support narrower operations that can easily compute in parallel. As of this writing, RISC-V does not have additional instructions to exploit subword parallelism, but the next section presents a real-world example of such an architecture.

3.7 Real Stuff: Streaming SIMD Extensions and Advanced Vector Extensions in x86

The original MMX (*MultiMedia eXtension*) for the x86 included instructions that operate on short vectors of integers. Later, SSE (*Streaming SIMD Extension*) provided instructions that operate on short vectors of single-precision floating-point numbers. 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 included 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 3.18 summarizes the SSE and SSE2 instructions.

Data transfer	Arithmetic	Compare
MOV[AU]{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[HL]{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 3.18 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. The table uses regular expressions to show the variations of instructions. Thus, MOV[AU]{SS|PS|SD|PD} represents the eight instructions MOVASS, MOVAPS, MOVASD, MOVAPD, MOVUSS, MOVUPS, MOVUSD, and MOVUPD. We use square brackets [] to show single-letter alternatives: 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. We use the curly brackets { } with a vertical bar | to show multiple letter 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.

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 compute simultaneously on four singles (PS) or two doubles (PD).

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 additions

```
addpd  %xmm0, %xmm4
```

becomes

```
vaddpd  %ymm0, %ymm4
```

which now produces four 64-bit floating-point multiplies. Intel has announced plans to widen the AVX registers to first 512 bits and later 1024 bits in later editions of the x86 architecture.

Elaboration: AVX also added three address instructions to x86. For example, `vaddpd` can now specify

```
vaddpd  %ymm0, %ymm1, %ymm4    // %ymm4 = %ymm0 + %ymm1
```

instead of the standard, two address version

```
addpd  %xmm0, %xmm4  // %xmm4 = %xmm4 + %xmm0
```

(Unlike RISC-V, 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 3.19 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

```
1.  void dgemm (size_t n, double* A, double* B, double* C)
2.  {
3.    for (size_t i = 0; i < n; ++i)
4.      for (size_t j = 0; j < n; ++j)
5.      {
6.        double cij = C[i+j*n]; /* cij = C[i][j] */
7.        for(size_t 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 3.19 Unoptimized C version of a double precision matrix multiply, widely known as DGEMM for Double-precision GEneral Matrix Multiply. 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.

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 3.20 shows the x86 assembly language output for the inner loop of Figure 3.19. 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.

While compiler writers may eventually be able to produce high-quality code routinely 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 3.21 shows the enhanced version of Figure 3.19 for which the Gnu C compiler produces AVX code. Figure 3.22 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 3.21 uses the `__m256d` data type, which tells the compiler the variable will hold four double-precision floating-point values. The intrinsic `_mm256_load_pd()` also on line 6 uses AVX instructions to load four 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 four double-precision floating-point numbers from `c0` into the matrix `C`. As we're going through four elements each iteration, the outer *for* loop on line 4 increments `i` by 4 instead of by 1 as on line 3 of Figure 3.19.

```
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,%r11                 // register %r11 = %r11 + 1
12. vmovsd %xmm0,(%r10)              // Store %xmm0 into C element
```

FIGURE 3.20 The x86 assembly language for the body of the nested loops generated by compiling the unoptimized C code in Figure 3.19. 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 (size_t n, double* A, double* B, double* C)
3.  {
4.    for ( size_t i = 0; i < n; i+=4 )
5.      for ( size_t j = 0; j < n; j++ ) {
6.        __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j] */
7.        for( size_t 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 3.21 Optimized C version of DGEMM using C intrinsics to generate the AVX subword-parallel instructions for the x86. Figure 3.22 shows the assembly language produced by the compiler for the inner loop.

```
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),%ymm1 // Make 4 copies of B element
5.  add    $0x8,%rax                // register %rax = %rax + 8
6.  vmulpd (%rcx),%ymm1,%ymm1       // Parallel mul %ymm1,4 A elements
7.  add    %r9,%rcx                 // register %rcx = %rcx + %r9
8.  cmp    %r10,%rax                // compare %r10 to %rax
9.  vaddpd %ymm1,%ymm0,%ymm0        // Parallel add %ymm1, %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 3.22 The x86 assembly language for the body of the nested loops generated by compiling the optimized C code in Figure 3.21. Note the similarities to Figure 3.20, 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 packed double precision instead of the sd version for scalar double precision.

Inside the loops, on line 9 we first load four 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 four 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 four products to the four sums in `c0`.

Figure 3.22 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 four of the five use `pd` for packed double precision—that correspond to the C intrinsics mentioned above. The code is very similar to that in Figure 3.20 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 *packed double* (`pd`) with YMM registers. The one exception is line 4 of Figure 3.22. 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 3.19 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 3.21 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 four 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

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

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 x, where xi means the ith bit, represents the number

$$\ldots + (x^3 \times 2^3) + (x^2 \times 2^2) + (x^1 \times 2^1) + (x^0 \times 2^0)$$

Shifting the bits of c right by n bits would seem to be the same as dividing by $2n$. 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

$$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111011_{two}$$

According to this fallacy, shifting right by two should divide by 4_{ten} (2^2):

$$00111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_{two}$$

With a 0 in the sign bit, this result is clearly wrong. The value created by the shift right is actually $4{,}611{,}686{,}018{,}427{,}387{,}902_{ten}$ 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

$$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_{two}$$

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 $c = -1.5_{ten} \times 10^{38}$, $a = 1.5_{ten} \times 10^{38}$, and $b = 1.0$, and that these are all single precision numbers.

$$\begin{aligned} c + (a + b) &= -1.5_{ten} \times 10^{38} + (1.5_{ten} \times 10^{38} + 1.0) \\ &= -1.5_{ten} \times 10^{38} + (1.5_{ten} \times 10^{38}) \\ &= 0.0 \\ c + (a + b) &= (-1.5_{ten} \times 10^{38} + 1.5_{ten} \times 10^{38}) + 1.0 \\ &= (0.0_{ten}) + 1.0 \\ &= 1.0 \end{aligned}$$

Since floating-point numbers have limited precision and result in approximations of real results, $1.5_{ten} \times 10^{38}$ is so much larger than 1.0_{ten} that $1.5_{ten} \times 10^{38} + 1.0$ is still $1.5_{ten} \times 10^{38}$. That is why the sum of 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 one 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

exact same 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.

Fallacy: Only theoretical mathematicians care about floating-point accuracy.

Newspaper headlines of November 1994 prove this statement is a fallacy (see Figure 3.23). The following is the inside story behind the headlines.

The Pentium uses a standard floating-point divide algorithm that generates multiple quotient bits per step, using the most significant bits of divisor and dividend to guess the next 2 bits of the quotient. The guess is taken from a lookup table containing −2, −1, 0, +1, or +2. The guess is multiplied by the divisor and 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 PLA to return 0 instead of 2 in these situations on the Pentium. Intel was wrong: while the first 11



FIGURE 3.23 A sampling of newspaper and magazine articles from November 1994, including the *New York Times*, *San Jose Mercury News*, *San Francisco Chronicle*, and *Infoworld*. The Pentium floating-point divide bug even made the "Top 10 List" of the *David Letterman Late Show* on television. Intel eventually took a $300 million write-off to replace the buggy chips.

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.

RISC-V Instruction	Name	Frequency	Cumulative
Add immediate	addi	14.36%	14.36%
Load doubleword	ld	8.27%	22.63%
Load fl. pt. double	fld	6.83%	29.46%
Add registers	add	6.23%	35.69%
Load word	lw	4.38%	40.07%
Store doubleword	sd	4.29%	44.36%
Branch if not equal	bne	4.14%	48.50%
Shift left immediate	slli	3.65%	52.15%
Fused mul-add double	fmadd.d	3.49%	55.64%
Branch if equal	beq	3.27%	58.91%
Add immediate word	addiw	2.86%	61.77%
Store fl. pt. double	fsd	2.24%	64.00%
Multiply fl. pt. double	fmul.d	2.02%	66.02%
Load upper immediate	lui	1.56%	67.59%
Store word	sw	1.52%	69.10%
Jump and link	jal	1.38%	70.49%
Branch if less than	blt	1.37%	71.86%
Add word	addw	1.34%	73.19%
Subtract fl. pt. double	fsub.d	1.28%	74.47%
Branch if greater/equal	bge	1.27%	75.75%

FIGURE 3.24 The frequency of the RISC-V instructions for the SPEC CPU2006 benchmarks. The 20 most popular instructions, which collectively account for 76% of all instructions executed, are included in the table. Pseudoinstructions are converted into RISC-V before execution, and hence do not appear here, explaining in part the popularity of `addi`.



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 switch to **parallelism** will shine 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 RISC-V instruction set.

Figure 3.24 ranks the popularity of the twenty most common RISC-V instructions the for SPEC CPU2006 integer and floating-point benchmarks. As you can see, a relatively small number of instructions dominate these rankings. This

observation has significant implications for the design of the processor, as we will see in Chapter 4.

No matter what the instruction set or its size—RISC-V, MIPS, 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.



Historical Perspective and Further Reading

Gresham's Law ("Bad money drives out Good") for computers would say, "The Fast drives out the Slow even if the Fast is wrong."

W. Kahan, *1992*

This section surveys the history of the floating point going back to von Neumann, including the surprisingly controversial IEEE standards effort, plus the rationale for the 80-bit stack architecture for floating point in the x86. See the rest of **Section 3.11** online.

3.12 Exercises

Never give in, never give in, never, never, never—in nothing, great or small, large or petty—never give in.

Winston Churchill, address at Harrow School, 1941

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?



Historical Perspective and Further Reading

Gresham's Law ("Bad money drives out Good") for computers would say, "The Fast drives out the Slow even if the Fast is wrong."

W. Kahan, 1992

This section surveys the history of the floating point going back to von Neumann, including the surprisingly controversial IEEE standards effort, the rationale for the 80-bit stack architecture for floating point in the IA-32, and an update on the next round of the standard.

At first it may be hard to imagine a subject of less interest than the correctness of computer arithmetic or its accuracy, and harder still to understand why a subject so old and mathematical should be so contentious. Computer arithmetic is as old as computing itself, and some of the subject's earliest notions, like the economical reuse of registers during serial multiplication and division, still command respect today. Maurice Wilkes [1985] recalled a conversation about that notion during his visit to the United States in 1946, before the earliest stored-program computer had been built:

> *… a project under von Neumann was to be set up at the Institute of Advanced Studies in Princeton.… Goldstine explained to me the principal features of the design, including the device whereby the digits of the multiplier were put into the tail of the accumulator and shifted out as the least significant part of the product was shifted in. I expressed some admiration at the way registers and shifting circuits were arranged … and Goldstine remarked that things of that nature came very easily to von Neumann.*

There is no controversy here; it can hardly arise in the context of exact integer arithmetic, so long as there is general agreement on what integer the correct result should be. However, as soon as approximate arithmetic enters the picture, so does controversy, as if one person's "negligible" must be another's "everything."

The First Dispute

Floating-point arithmetic kindled disagreement before it was ever built. John von Neumann was aware of Konrad Zuse's proposal for a computer in Germany in 1939 that was never built, probably because the floating point made it appear too complicated to finish before the Germans expected World War II to end. Hence, von Neumann refused to include it in the computer he built at Princeton. In an influential report coauthored in 1946 with H. H. Goldstine and A. W. Burks, he gave the arguments for and against floating point. In favor:

> *… to retain in a sum or product as many significant digits as possible and … to free the human operator from the burden of estimating and inserting into a problem "scale factors"—multiplication constants which serve to keep numbers within the limits of the machine.*

Floating point was excluded for several reasons:

> *There is, of course, no denying the fact that human time is consumed in arranging for the introduction of suitable scale factors. We only argue that the time consumed is a very small percentage of the total time we will spend in preparing an interesting problem for our machine. The first advantage of the floating point is, we feel, somewhat illusory. In order to have such a floating point, one must waste memory capacity which could otherwise be used for carrying more digits per word. It would therefore seem to us not at all clear whether the modest advantages of a floating binary point offset the loss of memory capacity and the increased complexity of the arithmetic and control circuits.*

The argument seems to be that most bits devoted to exponent fields would be bits wasted. Experience has proven otherwise.

One software approach to accommodate reals without floating-point hardware was called *floating vectors*; the idea was to compute at runtime one scale factor for a whole array of numbers, choosing the scale factor so that the array's biggest number would barely fill its field. By 1951, James H. Wilkinson had used this scheme extensively for matrix computations. The problem proved to be that a program might encounter a very large value, and hence the scale factor must accommodate these rare sizeable numbers. The common numbers would thus have many leading 0s, since all numbers had to use a single scale factor. Accuracy was sacrificed, because the least significant bits had to be lost on the right to accommodate leading 0s. This wastage became obvious to practitioners on early computers that displayed all their memory bits as dots on cathode ray tubes (like TV screens) because the loss of precision was visible. Where floating point deserved to be used, no practical alternative existed.

Thus, true floating-point hardware became popular because it was useful. By 1957, floating-point hardware was almost ubiquitous. A decimal floating-point unit was available for the IBM 650, and soon the IBM 704, 709, 7090, 7094 … series would offer binary floating-point hardware for double as well as single precision.

As a result, everybody had floating point, but every implementation was different.

Diversity versus Portability

Since roundoff introduces some error into almost all floating-point operations, to complain about another bit of error seems picayune. So for 20 years, nobody complained much that those operations behaved a little differently on different computers. If software required clever tricks to circumvent those idiosyncrasies and finally deliver results correct in all but the last several bits, such tricks were deemed part of the programmer's art. For a long time, matrix computations mystified most people who had no notion of error analysis; perhaps this continues to be true. That

may be why people are still surprised that numerically stable matrix computations depend upon the quality of arithmetic in so few places, far fewer than are generally supposed. Books by Wilkinson and widely used software packages like Linpack and Eispack sustained a false impression, widespread in the early 1970s, that a modicum of skill sufficed to produce *portable* numerical software.

"Portable" here means that the software is distributed as source code in some standard language to be compiled and executed on practically any commercially significant computer, and that it will then perform its task as well as any other program performs that task on that computer. Insofar as numerical software has often been thought to consist entirely of computer-independent mathematical formulas, its portability has commonly been taken for granted; the mistake in that presumption will become clear shortly.

Packages like Linpack and Eispack cost so much to develop—over a hundred dollars per line of Fortran delivered—that they could not have been developed without U.S. government subsidy; their portability was a precondition for that subsidy. But nobody thought to distinguish how various components contributed to their cost. One component was algorithmic—devise an algorithm that deserves to work on at least one computer despite its roundoff and over-/underflow limitations. Another component was the software engineering effort required to achieve and confirm portability to the diverse computers commercially significant at the time; this component grew more onerous as ever more diverse floating-point arithmetics blossomed in the 1970s. And yet scarcely anybody realized how much that diversity inflated the cost of such software packages.

A Backward Step

Early evidence that somewhat different arithmetics could engender grossly different software development costs was presented in 1964. It happened at a meeting of SHARE, the IBM mainframe users' group, at which IBM announced System/360, the successor to the 7094 series. One of the speakers described the tricks he had been forced to devise to achieve a level of quality for the S/360 library that was not quite so high as he had previously achieved for the 7094.

Von Neumann could have foretold part of the trouble, had he still been alive. In 1948, he and Goldstine had published a lengthy error analysis so difficult and so pessimistic that hardly anybody paid attention to it. It did predict correctly, however, that computations with larger arrays of data would probably fall prey to roundoff more often. IBM S/360s had bigger memories than 7094s, so data arrays could grow larger, and they did. To make matters worse, the S/360s had narrower single precision words (32 bits versus 36) and used a cruder arithmetic (hexadecimal or base 16 versus binary or base 2) with consequently poorer worst-case precision (21 significant bits versus 27) than the old 7094s. Consequently,

software that had almost always provided (barely) satisfactory accuracy on 7094s too often produced inaccurate results when run on S/360s. The quickest way to recover adequate accuracy was to replace old codes' single precision declarations with double precision before recompilation for the S/360. This practice exercised S/360 double precision far more than had been expected.

The early S/360's worst troubles were caused by lack of a guard digit in double precision. This lack showed up in multiplication as a failure of identities like $1.0^* x = x$ because multiplying x by 1.0 dropped x's last hexadecimal digit (4 bits). Similarly, if x and y were very close but had different exponents, subtraction dropped off the last digit of the smaller operand before computing $x - y$. This final aberration in double precision undermined a precious theorem that single precision then (and now) honored: If $1/2 \leq x/y \leq 2$, then no rounding error can occur when $x - y$ is computed; it must be computed exactly.

Innumerable computations had benefited from this minor theorem, most often unwittingly, for several decades before its first formal announcement and proof. We had been taking all this stuff for granted.

The identities and theorems about exact relationships that persisted, despite roundoff, with reasonable implementations of approximate arithmetic were not appreciated until they were lost. Previously, it had been thought that the things to matter were precision (how many significant digits were carried) and range (the spread between over-/underflow thresholds). Since the S/360's double precision had more precision and wider range than the 7094's, software was expected to continue to work at least as well as before. But it didn't.

Programmers who had matured into program managers were appalled at the cost of converting 7094 software to run on S/360s. A small subcommittee of SHARE proposed improvements to the S/360 floating point. This committee was surprised and grateful to get a fair part of what they asked for from IBM, including all-important guard digits. By 1968, these had been retrofitted to S/360s in the field at considerable expense; worse than that was customers' loss of faith in IBM's infallibility (a lesson learned by Intel 30 years later). IBM employees who can remember the incident still shudder.

The People Who Built the Bombs

Seymour Cray was associated for decades with the CDC and Cray computers that were, when he built them, the world's biggest and fastest. He always understood what his customers wanted most: *speed*. And he gave it to them even if, in so doing, he also gave them arithmetics more "interesting" than anyone else's. Among his customers have been the great government laboratories like those at Livermore and Los Alamos, where nuclear weapons were designed. The challenges of "interesting" arithmetics were pretty tame to people who had to overcome Mother Nature's challenges.

Perhaps all of us could learn to live with arithmetic idiosyncrasy if only one computer's idiosyncrasies had to be endured. Instead, when accumulating different computers' different anomalies, software dies the Death of a Thousand Cuts. Here is an example from Cray's computers:

```
if (x == 0.0)    y = 17.0 else y = z/x
```

Could this statement be stopped by a divide-by-zero error? On a CDC 6600 it could. The reason was a conflict between the 6600's adder, where `x` was compared with 0.0, and the multiplier and divider. The adder's comparison examined `x`'s leading 13 bits, which sufficed to distinguish zero from normal nonzero floating-point numbers `x`. The multiplier and divider examined only 12 leading bits. Consequently, tiny numbers existed that were nonzero to the adder but zero to the multiplier and divider! To avoid disasters with these tiny numbers, programmers learned to replace statements like the one above with

```
if (1.0 * x == 0.0)    y = 17.0 else y = z/x
```

But this statement is unsafe to use in would-be portable software because it malfunctions obscurely on other computers designed by Cray, the ones marketed by Cray Research, Inc. If `x` was so huge that 2.0 * `x` would overflow, then 1.0 * `x` might overflow too! Overflow happens because Cray computers check the product's exponent before the product's exponent has been normalized, just to save the delay of a single AND gate.

Rounding error anomalies that are far worse than the over-/underflow anomaly just discussed also affect Cray computers. The worst error came from the lack of a guard digit in add/subtract, an affliction of IBM S/360s. Further bad luck for software is occasioned by the way Cray economized his multiplier; about one-third of the bits that normal multiplier arrays generate have been left out of his multipliers, because they would contribute less than a unit to the last place of the final Cray-rounded product. Consequently, a Cray multiplier errs by almost a bit more than might have been expected. This error is compounded when division takes three multiplications to improve an approximate reciprocal of the divisor and then multiply the numerator by it. Square root compounds a few more multiplication errors.

The fast way drove out the slow, even though the fast was occasionally slightly wrong.

Making the World Safe for Floating Point, or Vice Versa

William Kahan was an undergraduate at the University of Toronto in 1953 when he learned to program its Ferranti-Manchester Mark-I computer. Because he entered the field early, Kahan became acquainted with a wide range of devices and a large proportion of the personalities active in computing; the numbers of both were small at that time. He has performed computations on slide rules, desktop

mechanical calculators, tabletop analog differential analyzers, and so on; he has used all but the earliest electronic computers and calculators mentioned in this book.

Kahan's desire to deliver reliable software led to an interest in error analysis that intensified during two years of postdoctoral study in England, where he became acquainted with Wilkinson. In 1960, he resumed teaching at Toronto, where an IBM 7090 had been acquired, and was granted free rein to tinker with its operating system, Fortran compiler, and runtime library. (He denies that he ever came near the 7090 hardware with a soldering iron but admits asking to do so.) One story from that time illuminates how misconceptions and numerical anomalies in computer systems can incur awesome hidden costs.

A graduate student in aeronautical engineering used the 7090 to simulate the wings he was designing for short takeoffs and landings. He knew such a wing would be difficult to control if its characteristics included an abrupt onset of stall, but he thought he could avoid that. His simulations were telling him otherwise. Just to be sure that roundoff was not interfering, he had repeated many of his calculations in double precision and gotten results much like those in single; his wings had stalled abruptly in both precisions. Disheartened, the student gave up.

Meanwhile Kahan replaced IBM's logarithm program (ALOG) with one of his own, which he hoped would provide better accuracy. While testing it, Kahan reran programs using the new version of ALOG. The student's results changed significantly; Kahan approached him to find out what had happened.

The student was puzzled. Much as the student preferred the results produced with the new ALOG—they predicted a gradual stall—he knew they must be wrong because they disagreed with his double precision results. The discrepancy between single and double precision results disappeared a few days later when a new release of IBM's double precision arithmetic software for the 7090 arrived. (The 7090 had no double precision hardware.) He went on to write a thesis about it and to build the wings; they performed as predicted. But that is not the end of the story.

In 1963, the 7090 was replaced by a faster 7094 with double precision floating-point hardware but with otherwise practically the same instruction set as the 7090. Only in double precision and only when using the new hardware did the wing stall abruptly again. A lot of time was spent to find out why. The 7094 hardware turned out, like the superseded 7090 software and the subsequent early S/360s, to lack a guard bit in double precision. Like so many programmers on those computers and on Cray's, the student discovered a trick to compensate for the lack of a guard digit; he wrote the expression `(0.5 − x) + 0.5` in place of `1.0 − x`. Nowadays we would blush if we had to explain why such a trick might be necessary, but it solved the student's problem.

Meanwhile the lure of California was working on Kahan and his family; they came to Berkeley and he to the University of California. An opportunity presented itself in 1974 when accuracy questions induced Hewlett-Packard's calculator designers to call in a consultant. The consultant was Kahan, and his work

dramatically improved the accuracy of HP calculators, but that is another story. Fruitful collaboration with congenial coworkers, however, fortified him for the next and crucial opportunity.



It came in 1976, when John F. Palmer at Intel was empowered to specify the "best possible" floating-point arithmetic for all of Intel's product line, as **Moore's Law** made it now possible to create a whole floating-point unit on a single chip. The floating-point standard was originally started for the iAPX-432, but when it was late, Intel started the 8086 as a short-term emergency stand-in until the iAPX-432 was ready. The iAPX-432 never became popular, so the emergency stand-in became the standard-bearer for Intel. The 8087 floating-point coprocessor for the 8086 was contemplated. (A *coprocessor* is simply an additional chip that accelerates a portion of the work of a processor; in this case, it accelerated floating-point computation.)

Palmer had obtained his Ph.D. at Stanford a few years before and knew whom to call for counsel of perfection—Kahan. They put together a design that obviously would have been impossible only a few years earlier and looked not quite possible at the time. But a new Israeli team of Intel employees led by Rafi Navé felt challenged to prove their prowess to Americans and leaped at an opportunity to put something impossible on a chip—the 8087.

By now, floating-point arithmetics that had been merely diverse among mainframes had become chaotic among microprocessors, one of which might be host to a dozen varieties of arithmetic in ROM firmware or software. Robert G. Stewart, an engineer prominent in IEEE activities, got fed up with this anarchy and proposed that the IEEE draft a decent floating-point standard. Simultaneously, word leaked out in Silicon Valley that Intel was going to put on one chip some awesome floating point well beyond anything its competitors had in mind. The competition had to find a way to slow Intel down, so they formed a committee to do what Stewart requested.

Meetings of this committee began in late 1977 with a plethora of competing drafts from innumerable sources and dragged on into 1985, when IEEE Standard 754 for Binary Floating Point was made official. The winning draft was very close to one submitted by Kahan, his student Jerome T. Coonen, and Harold S. Stone, a professor visiting Berkeley at the time. Their draft was based on the Intel design, with Intel's permission, of course, as simplified by Coonen. Their harmonious combination of features, almost none of them new, had at the outset attracted more support within the committee and from outside experts like Wilkinson than any other draft, but they had to win nearly unanimous support within the committee to win official IEEE endorsement, and that took time.

The First IEEE 754 Chips

In 1980, Intel became tired of waiting and released the 8087 for use in the IBM PC. The floating-point architecture of the companion 8087 had to be retrofitted into the 8086 opcode space, making it inconvenient to offer two operands per

instruction as found in the rest of the 8086. Hence the decision for one operand per instruction using a stack: "The designer's task was to make a Virtue of this Necessity." (Kahan's [1990] history of the stack architecture selection for the 8087 is entertaining reading.)

Rather than the classical stack architecture, which has no provision for avoiding common subexpressions from being pushed and popped from memory into the top of the stack found in registers, Intel tried to combine a flat register file with a stack. The reasoning was that the restriction of the top of stack as one operand was not so bad since it only required the execution of an `FXCH` instruction (which swapped registers) to get the same result as a two-operand instruction, and `FXCH` was much faster than the floating-point operations of the 8087.

Since floating-point expressions are not that complex, Kahan reasoned that eight registers meant that the stack would rarely overflow. Hence, he urged that the 8087 use this hybrid scheme with the provision that stack overflow or stack underflow would interrupt the 8086 so that interrupt software could give the illusion to the compiler writer of an unlimited stack for floating-point data.

The Intel 8087 was implemented in Israel, and 7500 miles and 10 time zones made communication from California difficult. According to Palmer and Morse (*The 8087 Primer*, J. Wiley, New York, 1984, p. 93):

> *Unfortunately, nobody tried to write a software stack manager until after the 8087 was built, and by then it was too late; what was too complicated to perform in hardware turned out to be even worse in software. One thing found lacking is the ability to conveniently determine if an invalid operation is indeed due to a stack overflow.… Also lacking is the ability to restart the instruction that caused the stack overflow …*

The result is that the stack exceptions are too slow to handle in software. As Kahan [1990] says:

> *Consequently, almost all higher-level languages' compilers emit inefficient code for the 80x87 family, degrading the chip's performance by typically 50% with spurious stores and loads necessary simply to preclude stack over/under-flow.…*
>
> *I still regret that the 8087's stack implementation was not quite so neat as my original intention.… If the original design had been realized, compilers today would use the 80x87 and its descendents more efficiently, and Intel's competitors could more easily market faster but compatible 80x87 imitations.*

In 1982, Motorola announced its 68881, which found a place in Sun 3s and Macintosh IIs; Apple had been a supporter of the proposal from the beginning. Another Berkeley graduate student, George S. Taylor, had soon designed a high-speed implementation of the proposed standard for an early superminicomputer (ELXSI 6400). The standard was becoming de facto before its final draft's ink was dry.

An early rush of adoptions gave the computing industry the false impression that IEEE 754, like so many other standards, could be implemented easily by following a standard recipe. Not true. Only the enthusiasm and ingenuity of its early implementors made it look easy.

In fact, to implement IEEE 754 correctly demands extraordinarily diligent attention to detail; to make it run fast demands extraordinarily competent ingenuity of design. Had the industry's engineering managers realized this, they might not have been so quick to affirm that, as a matter of policy, "We conform to all applicable standards."

IEEE 754 Today

Unfortunately, the compiler-writing community was not represented adequately in the wrangling, and some of the features didn't balance language and compiler issues against other points. That community has been slow to make IEEE 754's unusual features available to the applications programmer. Humane exception handling is one such unusual feature; directed rounding another. Without compiler support, these features have atrophied.

The successful parts of IEEE 754 are that it is a widely implemented standard with a common floating-point format, that it requires minimum accuracy to one-half ulp in the least significant bit, and that operations must be commutative.

The IEEE 754/854 has been implemented to a considerable degree of fidelity in at least part of the product line of every North American computer manufacturer. The only significant exceptions were the DEC VAX, IBM S/370 descendants, and Cray Research vector supercomputers, and all three have been replaced by compliant computers.

IEEE rules ask that a standard be revisited periodically for updating. A committee started in 2000, and drafts of the revised standards were circulated for voting, and these were approved in 2008. The revised standard, IEEE Std 754-2008 [2008], includes several new types: 16-bit floating point, called *half precision*; 128-bit floating point, called *quad precision*; and three decimal types, matching the length of the 32-bit, 64-bit, and 128-bit binary formats. In 1989, the Association for Computing Machinery, acknowledging the benefits conferred upon the computing industry by IEEE 754, honored Kahan with the Turing Award. On accepting it, he thanked his many associates for their diligent support, and his adversaries for their blunders. So . . . not all errors are bad.

Further Reading

If you are interested in learning more about floating point, two publications by David Goldberg [1991, 2002] are good starting points; they abound with pointers to further reading. Several of the stories told in this section come from Kahan [1972, 1983]. The latest word on the state of the art in computer arithmetic is often found in the *Proceedings* of the most recent IEEE-sponsored Symposium on Computer Arithmetic, held every two years; the 23rd was held in 2016.

Burks, A.W., H.H. Goldstine, and J. von Neumann [1946]. "Preliminary discussion of the logical design of an electronic computing instrument," *Report to the U.S. Army Ordnance Dept.*, p. 1; also in *Papers of John von Neumann*, W. Aspray and A. Burks (Eds.), MIT Press, Cambridge, MA, and Tomash Publishers, Los Angeles, 1987, 97–146.

This classic paper includes arguments against floating-point hardware.

Goldberg, D. [2002]. "Computer arithmetic". Appendix A of *Computer Architecture: A Quantitative Approach*, third edition, J. L. Hennessy and D. A. Patterson, Morgan Kaufmann Publishers, San Francisco.

A more advanced introduction to integer and floating-point arithmetic, with emphasis on hardware. It covers Sections 3.4–3.6 of this book in just 10 pages, leaving another 45 pages for advanced topics.

Goldberg, D. [1991]. "What every computer scientist should know about floating-point arithmetic", *ACM Computing Surveys* 23(1), 5–48.

Another good introduction to floating-point arithmetic by the same author, this time with emphasis on software.

Kahan, W. [1972]. "A survey of error-analysis," in *Info. Processing 71* (Proc. IFIP Congress 71 in Ljubljana), Vol. 2, North-Holland Publishing, Amsterdam, 1214–1239.

This survey is a source of stories on the importance of accurate arithmetic.

Kahan, W. [1983]. "Mathematics written in sand," *Proc. Amer. Stat. Assoc. Joint Summer Meetings of 1983, Statistical Computing Section*, 12–26.

The title refers to silicon and is another source of stories illustrating the importance of accurate arithmetic.

Kahan, W. [1990]. "On the advantage of the 8087's stack," unpublished course notes, Computer Science Division, University of California, Berkeley.

What the 8087 floating-point architecture could have been.

Kahan, W. [1997]. Available at *http://www.cims.nyu.edu/~dbindel/class/cs279/87stack.pdf*.

A collection of memos related to floating point, including "Beastly numbers" (another less famous Pentium bug), "Notes on the IEEE floating point arithmetic" (including comments on how some features are atrophying), and "The baleful effects of computing benchmarks" (on the unhealthy preoccupation on speed versus correctness, accuracy, ease of use, flexibility, …).

Koren, I. [2002]. *Computer Arithmetic Algorithms*, second edition, A. K. Peters, Natick, MA.

A textbook aimed at seniors and first-year graduate students that explains fundamental principles of basic arithmetic, as well as complex operations such as logarithmic and trigonometric functions.

Wilkes, M. V. [1985]. *Memoirs of a Computer Pioneer*, MIT Press, Cambridge, MA.

This computer pioneer's recollections include the derivation of the standard hardware for multiply and divide developed by von Neumann.

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.

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 four 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 four 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 four 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 three 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 $0 \times 0C000000$ represent if it is a two's complement integer? An unsigned integer?

3.21 [10] <§3.5> If the bit pattern $0 \times 0C000000$ is placed into the Instruction Register, what MIPS instruction will be executed?

3.22 [10] <§3.5> What decimal number does the bit pattern $0 \times 0C000000$ 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] <§3.10> Calculate $(3.984375 \times 10^{-1} + 3.4375 \times 10^{-1}) + 1.771 \times 10^{3}$ by hand, assuming each of the values is 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] <§3.10> Calculate $3.984375 \times 10^{-1} + (3.4375 \times 10^{-1} + 1.771 \times 10^{3})$ by hand, assuming each of the values is 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] <§3.10> Based on your answers to Exercises 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] <§3.10> 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 is 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] <§3.10> 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 is 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] <§3.10> Based on your answers to Exercises 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] <§3.10> Calculate $1.666015625 \times 10^{0} \times (1.9760 \times 10^{4} + -1.9744 \times 10^{4})$ by hand, assuming each of the values is 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] <§3.10> 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 is 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] <§3.10> Based on your answers to Exercises 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 four 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 of value 1/3 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 of value 1/3 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 of value 1/3 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 of 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.

Answers to Check Yourself

§3.2, page 177: 2.
§3.5, page 215: 3.

This page intentionally left blank

4

The Processor

4.1 **Introduction** 236
4.2 **Logic Design Conventions** 240
4.3 **Building a Datapath** 243
4.4 **A Simple Implementation Scheme** 251
4.5 **An Overview of Pipelining** 262
4.6 **Pipelined Datapath and Control** 276
4.7 **Data Hazards: Forwarding versus Stalling** 294
4.8 **Control Hazards** 307
4.9 **Exceptions** 315
4.10 **Parallelism via Instructions** 321

In a major matter, no details are small.

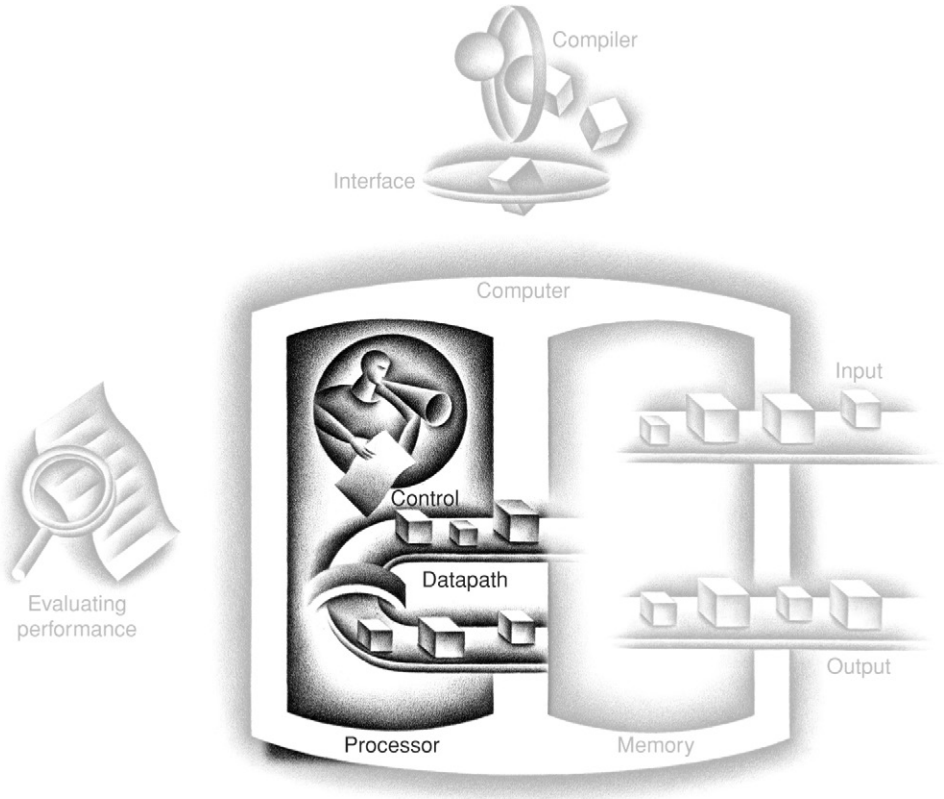
French Proverb

Computer Organization and Design. DOI: http://dx.doi.org/10.1016/B978-0-12-812275-4.00004-X

© 2018 Elsevier Inc. All rights reserved.

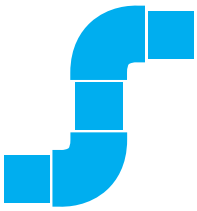
4.11 **Real Stuff: The ARM Cortex-A53 and Intel Core i7 Pipelines** 334
4.12 **Going Faster: Instruction-Level Parallelism and Matrix Multiply** 342
4.13 **Advanced Topic: An Introduction to Digital Design Using a Hardware Design Language to Describe and Model a Pipeline and More Pipelining Illustrations** 345
4.14 **Fallacies and Pitfalls** 345
4.15 **Concluding Remarks** 346
4.16 **Historical Perspective and Further Reading** 347
4.17 **Exercises** 347

The 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 RISC-V 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 RISC-V. The bulk of the chapter covers a more realistic **pipelined** RISC-V 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. Current trends are covered in Section 4.10, and Section 4.11 describes the recent Intel Core i7 and ARM Cortex-A53 architectures. Section 4.12 shows how to use instruction-level parallelism to more than double the performance of the matrix multiply from Section 3.9. 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 Sections 4.2, 4.7–4.9. 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 RISC-V Implementation

We will be examining an implementation that includes a subset of the core RISC-V instruction set:

- The memory-reference instructions *load doubleword* (`ld`) and *store doubleword* (`sd`)
- The arithmetic-logical instructions `add`, `sub`, `and`, and `or`
- The conditional branch instruction *branch if equal* (`beq`)

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 RISC-V 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 RISC-V 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 `ld` 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 RISC-V instruction set simplify the implementation by making the execution of many of the instruction classes similar.

For example, all instruction classes 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 conditional branches for the equality test. 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 conditional branch instruction, we may need to change the next instruction address based on the comparison; otherwise, the PC should be incremented by four to get the address of the subsequent instruction.

Figure 4.1 shows the high-level view of a RISC-V 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 A 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 on a load and write on a store. The register file must be written only on a load or

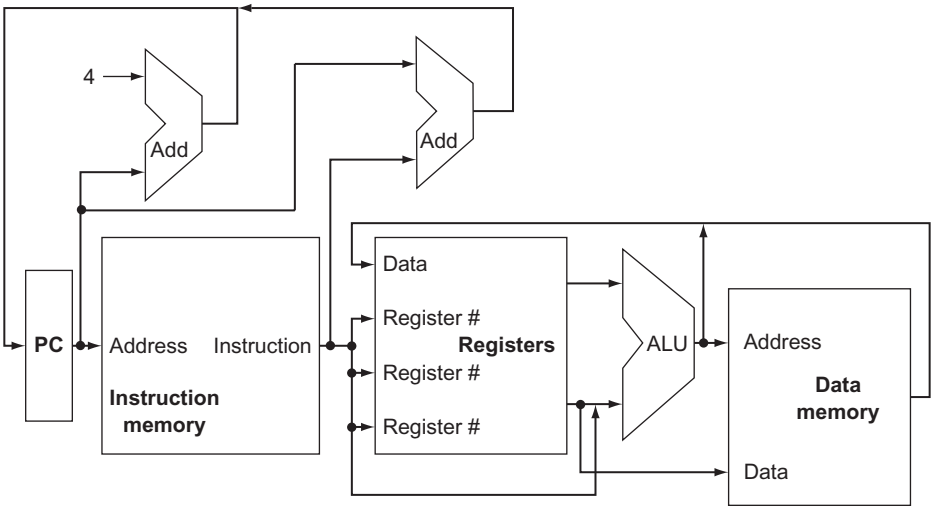


FIGURE 4.1 An abstract view of the implementation of the RISC-V subset showing the major functional units and the major connections between them. All instructions start by using the program counter to supply the instruction address to the instruction memory. After the instruction is fetched, the register 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 an equality check (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 adder (where the PC and branch offset are summed) or from an adder that increments the current PC by four. 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.

an arithmetic-logical instruction. And, of course, the ALU must perform one of several operations. (Appendix A describes the detailed design of the ALU.) Like the multiplexors, control lines that are set based on 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 top multiplexor, which

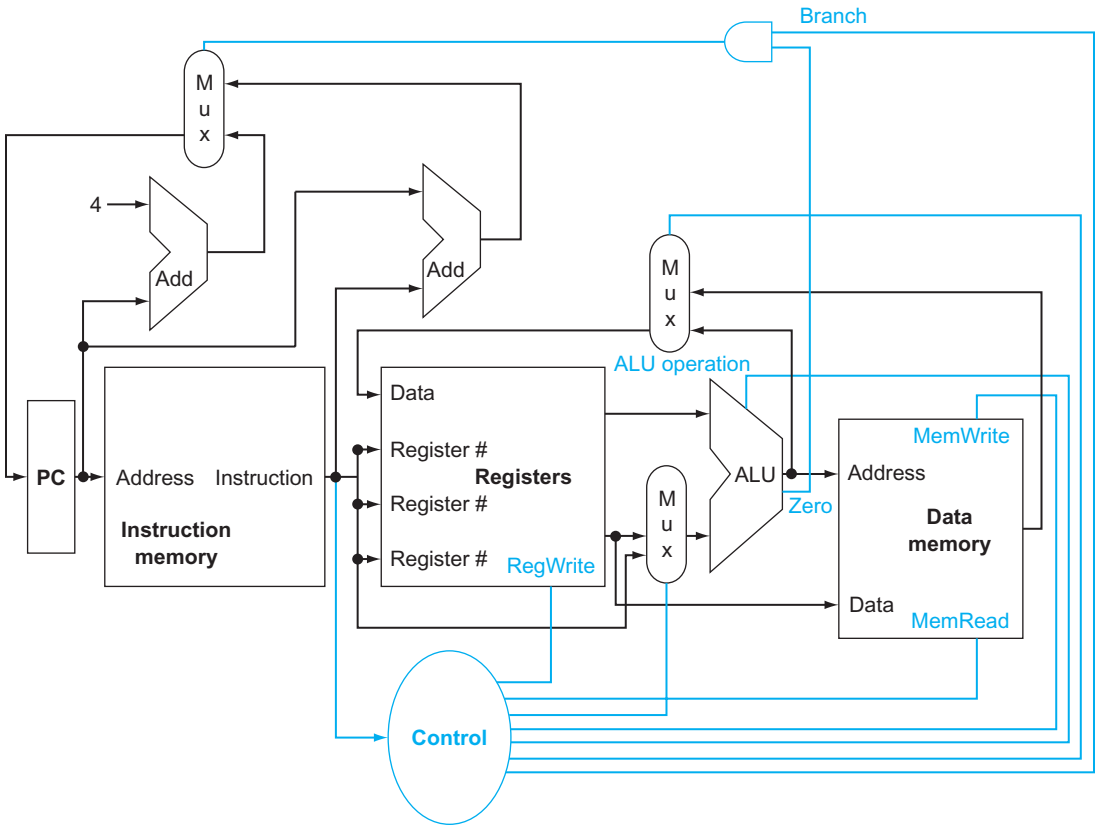


FIGURE 4.2 The basic implementation of the RISC-V subset, including the necessary multiplexors and control lines. The top multiplexor ("Mux") controls what value replaces the PC (PC + 4 or the branch destination address); the multiplexor is controlled by the gate that "ANDs" together the Zero output 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 bottom-most 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.

determines whether 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 RISC-V instruction set mean 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 235—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 A before continuing.

The datapath elements in the RISC-V 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 A is an example of a combinational element. Given a set of inputs, it always produces the same output because it has no internal storage.

combinational element An operational element, such as an AND gate or an ALU

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.

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 A), which has exactly these two inputs (a value and a clock) and one output. In addition to flip-flops, our RISC-V 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. Appendix A discusses the operation of both the combinational and sequential elements and their construction in more detail.

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 that 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.

clocking methodology The approach used to determine when data are valid and stable relative to the clock.

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.

edge-triggered clocking A clocking scheme in which all state changes occur on a clock edge.

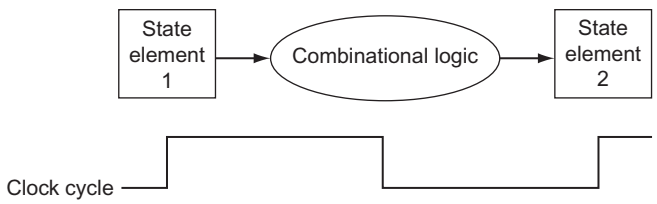


FIGURE 4.3 Combinational logic, state elements, and the clock are closely related. In a synchronous digital system, the clock determines when elements with state will write values into internal storage. Any inputs to a state element must reach a stable value (that is, have reached a value from which 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 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.

control signal A signal used for multiplexor selection or for directing the operation of a functional unit; contrasts with a *data signal*, which contains information that is operated on by a functional unit.

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.

asserted The signal is logically high or true.

deasserted The signal is logically low or false.

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 *deassert* 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 A, we briefly discuss additional timing constraints (such as setup and hold times) as well as other timing methodologies.

For the 64-bit RISC-V architecture, nearly all of these state and logic elements will have inputs and outputs that are 64 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 64 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 64-bit bus by combining two 32-bit buses. In such cases, labels on the bus lines

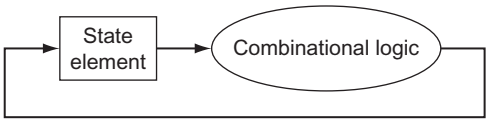


FIGURE 4.4 An edge-triggered methodology allows a state element to be read and written in the same clock cycle without creating a race that could lead to indeterminate data values. Of course, the clock cycle still must be long enough so that the input values are stable when the active clock edge occurs. 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.

will make it 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 contrary to a signal that carries data; this distinction will become clearer as we proceed through this chapter.

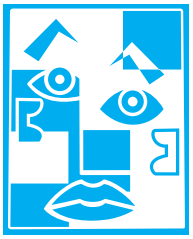
Check Yourself

True or false: Because the register file is both read and written on the same clock cycle, any RISC-V datapath using edge-triggered writes must have more than one copy of the register file.

Elaboration: There is also a 32-bit version of the RISC-V architecture, and, naturally enough, most paths in its implementation would be 32 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 RISC-V 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.



ABSTRACTION

datapath element A unit used to operate on or hold data within a processor. In the RISC-V 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 4.5a 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 A 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 4.5c, 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 2.19 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`, and `or`, which

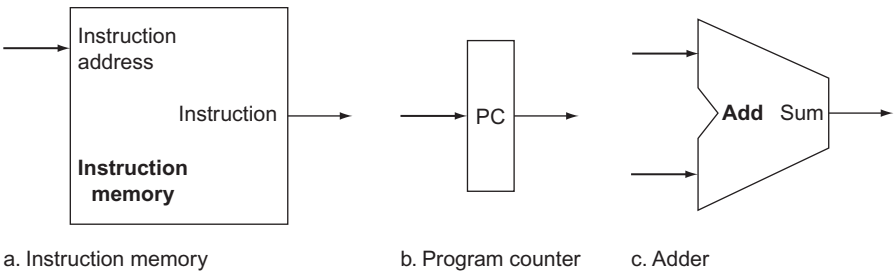


FIGURE 4.5 Two state elements are needed to store and access instructions, and an adder is needed to compute the next instruction address. The state elements are the instruction memory and the program counter. The instruction memory need only provide read access because the datapath does not write instructions. 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 64-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 64-bit inputs and place the sum on its output.

were introduced in Chapter 2. Recall that a typical instance of such an instruction is `add x1, x2, x3`, which reads `x2` and `x3` and writes the sum into `x1`.

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 three inputs (two 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 64 bits wide.

Figure 4.7b shows the ALU, which takes two 64-bit inputs and produces a 64-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 A; we will review the ALU control shortly when we need to know how to set it.

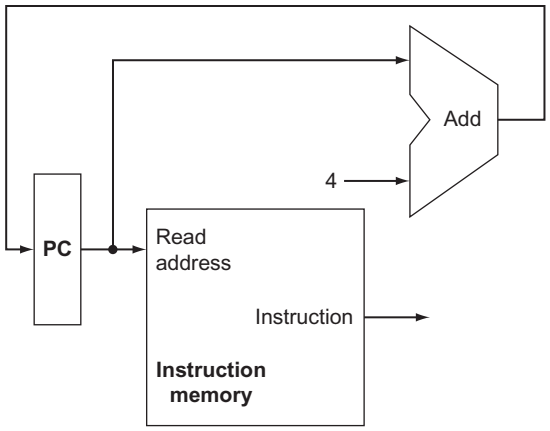


FIGURE 4.6 A portion of the datapath used for fetching instructions and incrementing the program counter. The fetched instruction is used by other parts of the datapath.

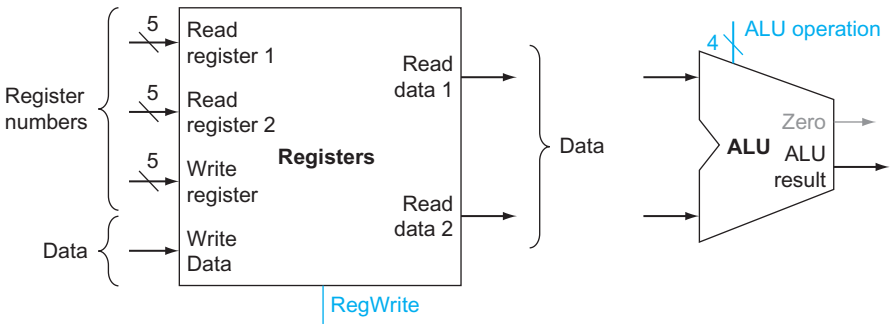


FIGURE 4.7 The two elements needed to implement R-format ALU operations are the register file and the ALU. The register file contains all the registers and has two read ports and one write port. The design of multiported register files is discussed in Section A.8 of Appendix A. The register file always 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 64 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 A. We will use the Zero detection output of the ALU shortly to implement conditional branches.

Next, consider the RISC-V load register and store register instructions, which have the general form `ld x1, offset(x2)` or `sd x1, offset(x2)`. These instructions compute a memory address by adding the base register, which is `x2`, to the 12-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 `x1`. If the instruction is a load, the value read from memory must be written into the register file in the specified register, which is `x1`. Thus, we will need both the register file and the ALU from Figure 4.7.

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.

In addition, we will need a unit to **sign-extend** the 12-bit offset field in the instruction to a 64-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.

branch target address The address specified in a branch, which becomes the new program counter (PC) if the branch is taken. In the RISC-V architecture, the branch target is given by the sum of the offset field of the instruction and the address of the branch.

The `beq` instruction has three operands, two registers that are compared for equality, and a 12-bit offset used to compute the **branch target address** relative to the branch instruction address. Its form is `beq x1, x2, 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 branch instruction.
- The architecture also states that the offset field is shifted left 1 bit so that it is a half word offset; this shift increases the effective range of the offset field by a factor of 2.

branch taken A branch where the branch condition is satisfied and the program counter (PC) becomes the branch target. All unconditional branches are taken branches.

To deal with the latter complication, we will need to shift the offset field by 1.

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., two operands are equal), the branch target address becomes the new PC, and we say that the **branch** is **taken**. If the operand is not zero, 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**.

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.

Thus, the branch datapath must do two operations: compute the branch target address and test 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 an immediate generation 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 two register operands (although we will not need to write into the register file). In addition, the equality comparison can be done using the ALU we designed in Appendix A. Since that ALU provides an output signal that indicates whether the result was 0, we can send both register operands to the ALU

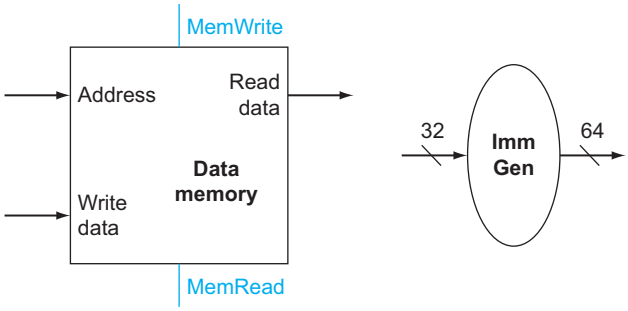


FIGURE 4.8 The two units needed to implement loads and stores, in addition to the register file and ALU of Figure 4.7, are the data memory unit and the immediate generation unit. The memory unit is a state element with inputs for the address and the write data, and a single output for the read result. 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 immediate generation unit (ImmGen) has a 32-bit instruction as input that selects a 12-bit field for load, store, and branch if equal that is sign-extended into a 64-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 A.8 of Appendix A for further discussion of how real memory chips work.

with the control set to subtract two values. If the Zero signal out of the ALU unit is asserted, we know that the register values are equal. Although the Zero output always signals if the result is 0, we will be using it only to implement the equality test of conditional branches. Later, we will show exactly how to connect the control signals of the ALU for use in the datapath.

The branch instruction operates by adding the PC with the 12 bits of the instruction shifted left by 1 bit. Simply concatenating 0 to the branch offset accomplishes this shift, as described in Chapter 2.

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 design 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.

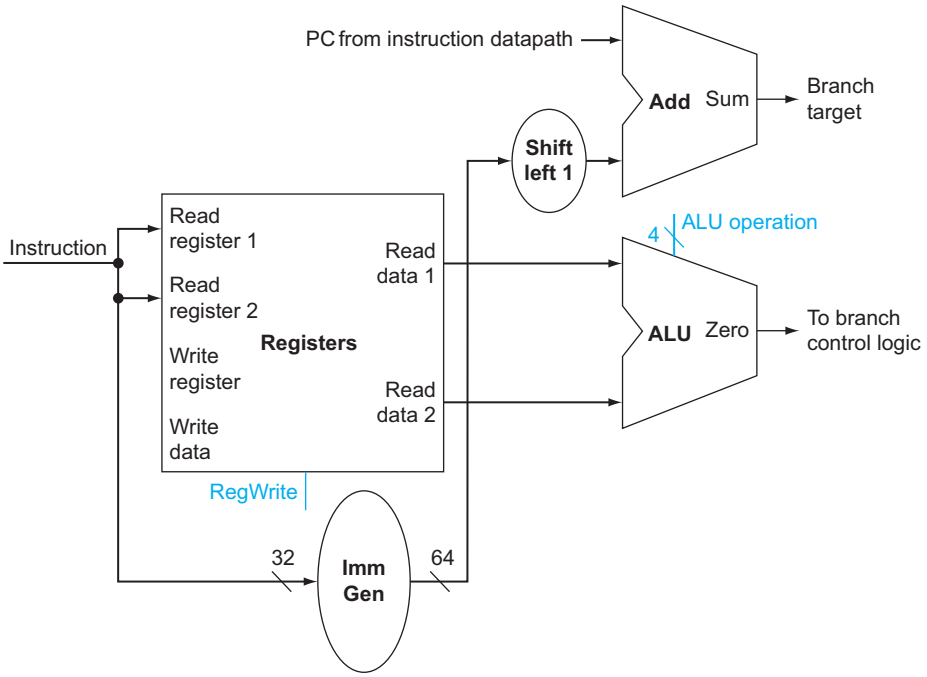


FIGURE 4.9 The datapath for a branch uses the ALU to evaluate the branch condition and a separate adder to compute the branch target as the sum of the PC and the sign-extended 12 bits of the instruction (the branch displacement), shifted left 1 bit. The unit labeled *Shift left 1* is simply a routing of the signals between input and output that adds 0_{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 12 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.

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.

EXAMPLE

Building a Datapath

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 12-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.

ANSWER

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.

Now we can combine all the pieces to make a simple datapath for the core RISC-V 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 to compare two register operands for equality, 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.

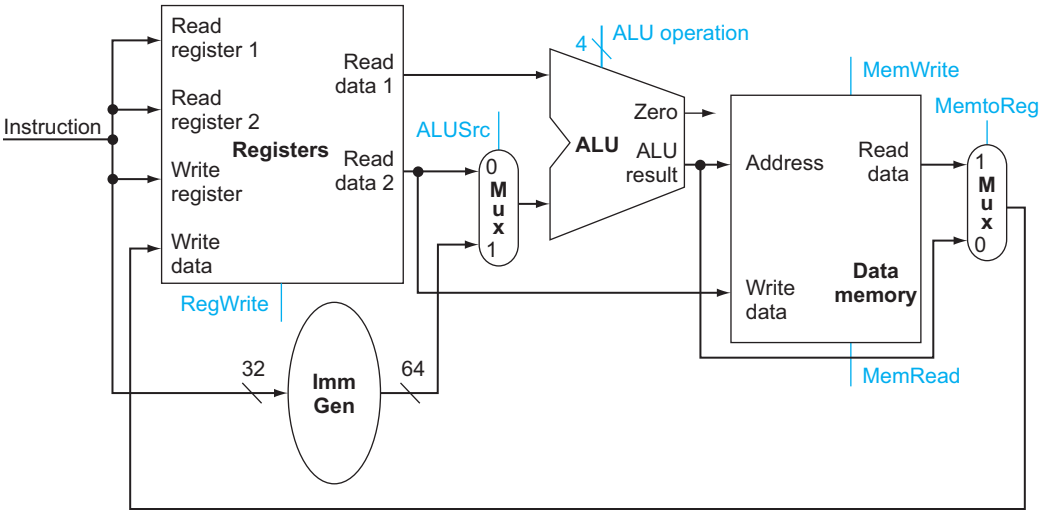


FIGURE 4.10 The datapath for the memory instructions and the R-type instructions. This example shows how a single datapath can be assembled from the pieces in Figures 4.7 and 4.8 by adding multiplexors. Two multiplexors are needed, as described in the example.

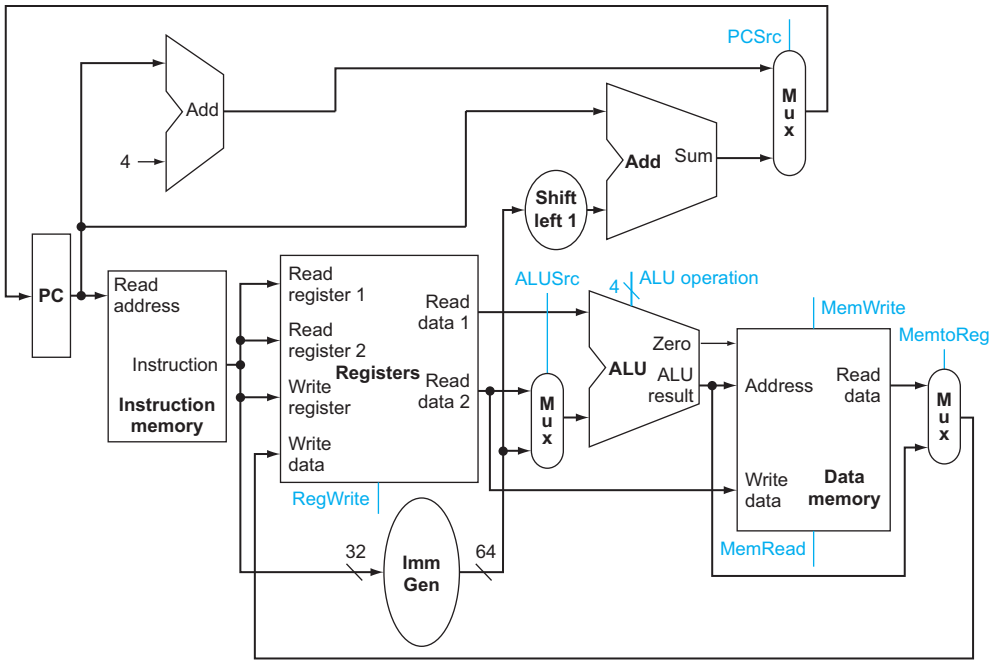


FIGURE 4.11 The simple datapath for the core RISC-V architecture combines the elements required by different instruction classes. The components come from Figures 4.6, 4.9, and 4.10. This datapath can execute the basic instructions (load-store register, ALU operations, and branches) in a single clock cycle. Just one additional multiplexor is needed to integrate branches.

Check Yourself

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.

   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 RISC-V, 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.

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.

Elaboration: The immediate generation logic must choose between sign-extending a 12-bit field in instruction bits 31:20 for load instructions, bits 31:25 and 11:7 for store instructions, or bits 31, 7, 30:25, and 11:8 for the conditional branch. Since the input is all 32 bits of the instruction, it can use the opcode bits of the instruction to select the proper field. RISC-V opcode bit 6 happens to be 0 for data transfer instructions and 1 for conditional branches, and RISC-V opcode bit 5 happens to be 0 for load instructions and 1 for store instructions. Thus, bits 5 and 6 can control a 3:1 multiplexor inside the immediate generation logic that selects the appropriate 12-bit field for load, store, and conditional branch instructions.

4.4 A Simple Implementation Scheme

In this section, we look at what might be thought of as a simple implementation of our RISC-V subset. We build this simple implementation using the datapath of the last section and adding a simple control function. This simple implementation covers *load doubleword* (`ld`), *store doubleword* (`sd`), *branch if equal* (`beq`), and the arithmetic-logical instructions `add`, `sub`, `and`, and `or`.

The ALU Control

The RISC-V ALU in Appendix A defines the four following combinations of four control inputs:

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract

Depending on the instruction class, the ALU will need to perform one of these four functions. For load and store 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 four actions (`AND`, `OR`, `add`, or `subtract`), depending on the value of the 7-bit funct7 field (bits 31:25) and 3-bit funct3 field (bits 14:12) in the instruction (see Chapter 2). For the conditional branch if equal instruction, the ALU subtracts two operands and tests to see if the result is 0.

We can generate the 4-bit ALU control input using a small control unit that has as inputs the funct7 and funct3 fields 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 and test if zero (01) for `beq`, or be determined by the operation encoded in the funct7 and funct3 fields (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, funct7, and funct3 fields. 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 reduce the latency of the control unit. Such optimizations are important, since the latency of the control unit is often a critical factor in determining the clock cycle time.

There are several different ways to implement the mapping from the 2-bit ALUOp field and the funct fields to the four ALU operation control bits. Because only a small number of the possible funct field values are of interest and funct fields are used only when the ALUOp bits equal 10, we can use a small piece of logic that recognizes the subset of possible values and generates the appropriate ALU control signals.

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract

Instruction opcode	ALUOp	Operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
ld	00	load doubleword	XXXXXXX	XXX	add	0010
sd	00	store doubleword	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

FIGURE 4.12 How the ALU control bits are set depends on the ALUOp control bits and the different opcodes for the R-type instruction. The instruction, 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 funct7 or funct3 fields; in this case, we say that we "don't care" about the value of the opcode, and the bits are shown as Xs. When the ALUOp value is 10, then the funct7 and funct3 fields are used to set the ALU control input. See Appendix A.

As a step in designing this logic, it is useful to create a *truth table* for the interesting combinations of funct fields and the ALUOp signals, as we've done in Figure 4.13; this **truth table** shows how the 4-bit ALU control is set depending on these input fields. Since the full truth table is very large, 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 C.2 of **Appendix C**.)

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.

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 funct fields. In this case, then, the funct 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 A for more information.

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.

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 C.2 of **Appendix C**.

Designing the Main Control Unit

Now that we have described how to design an ALU that uses the opcode 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		Funct7 field							Funct3 field			
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]	Operation
0	0	X	X	X	X	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	X	X	X	X	0110
1	X	0	0	0	0	0	0	0	0	0	0	0010
1	X	0	1	0	0	0	0	0	0	0	0	0110
1	X	0	0	0	0	0	0	0	1	1	1	0000
1	X	0	0	0	0	0	0	0	1	1	0	0001

FIGURE 4.13 The truth table for the 4 ALU control bits (called Operation). The inputs are the ALUOp and funct fields. 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. While we show all 10 bits of funct fields, note that the only bits with different values for the four R-format instructions are bits 30, 14, 13, and 12. Thus, we only need these four funct field bits as input for ALU control instead of all 10.

Name (Bit position)	Fields 31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
(d) SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode

FIGURE 4.14 The four instruction classes (arithmetic, load, store, and conditional branch) use four different instruction formats. (a) Instruction format for R-type arithmetic instructions (opcode = 51_{ten}), which have three register operands: rs1, rs2, and rd. Fields rs1 and rd are sources, and rd is the destination. The ALU function is in the funct3 and funct7 fields and is decoded by the ALU control design in the previous section. The R-type instructions that we implement are `add`, `sub`, `and`, and `or`. (b) Instruction format for I-type load instructions (opcode = 3_{ten}). The register rs1 is the base register that is added to the 12-bit immediate field to form the memory address. Field rd is the destination register for the loaded value. (c) Instruction format for S-type store instructions (opcode = 35_{ten}). The register rs1 is the base register that is added to the 12-bit immediate field to form the memory address. (The immediate field is split into a 7-bit piece and a 5-bit piece.) Field rs2 is the source register whose value should be stored into memory. (d) Instruction format for SB-type conditional branch instructions (opcode = 99_{ten}). The registers rs1 and rs2 compared. The 12-bit immediate address field is sign-extended, shifted left 1 bit, and added to the PC to compute the branch target address.

the formats of the four instruction classes: arithmetic, load, store, and conditional branch instructions. Figure 4.14 shows these formats.

There are several major observations about this instruction format that we will rely on:

opcode The field that denotes the operation and format of an instruction.

- The **opcode** field, which as we saw in Chapter 2, is always in bits 6:0. Depending on the opcode, the funct3 field (bits 14:12) and funct7 field (bits 31:25) serve as an extended opcode field.
- The first register operand is always in bit positions 19:15 (rs1) for R-type instructions and branch instructions. This field also specifies the base register for load and store instructions.
- The second register operand is always in bit positions 24:20 (rs2) for R-type instructions and branch instructions. This field also specifies the register operand that gets copied to memory for store instructions.
- Another operand can also be a 12-bit offset for branch or load-store instructions.
- The destination register is always in bit positions 11:7 (rd) for R-type instructions and load instructions.

The first design principle from Chapter 2—*simplicity favors regularity*—pays off here in specifying control.

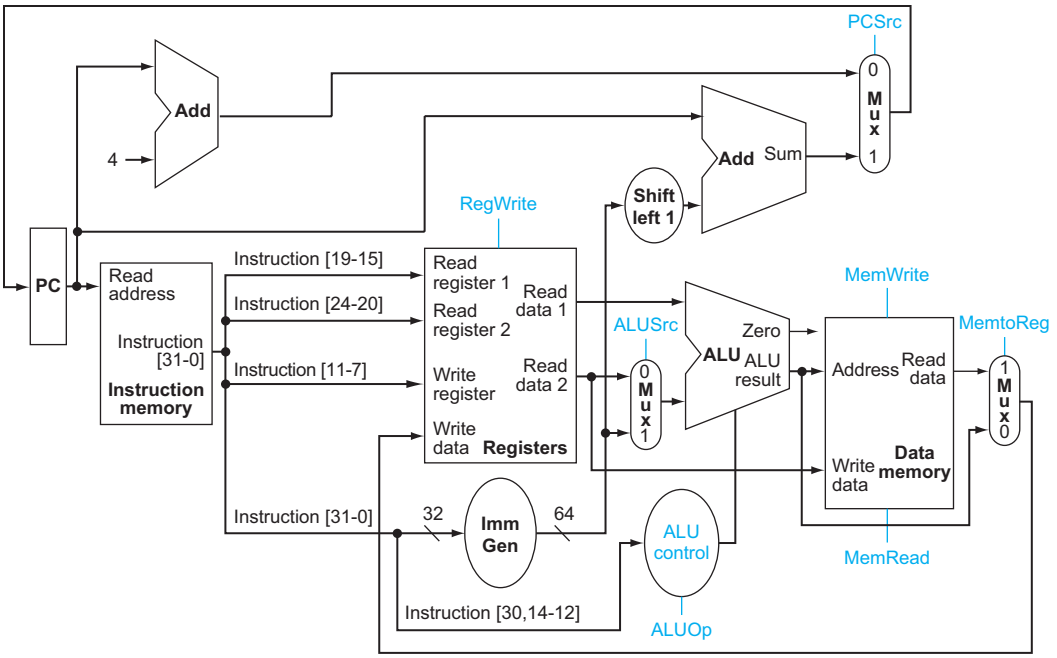


FIGURE 4.15 The datapath of Figure 4.11 with all necessary multiplexors and all control lines identified. The control lines are shown in color. The ALU control block has also been added, which depends on the funct3 field and part of the funct7 field. The PC does not require a write control, since it 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 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 six 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 six 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 six 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 and funct fields of the instruction. The PCSrc control line is the exception. That control line should be asserted if the instruction is branch if equal (a decision that the control unit can make) *and* the Zero output of the ALU, which is used for the equality test, 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
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, 12 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 six 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 A for further discussion of this problem.)

These eight control signals (six from Figure 4.16 and two for ALUOp) can now be set based on the input signals to the control unit, which are the opcode bits 6:0. 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 x1, x2, x3`. Although everything occurs in one clock cycle, we can think

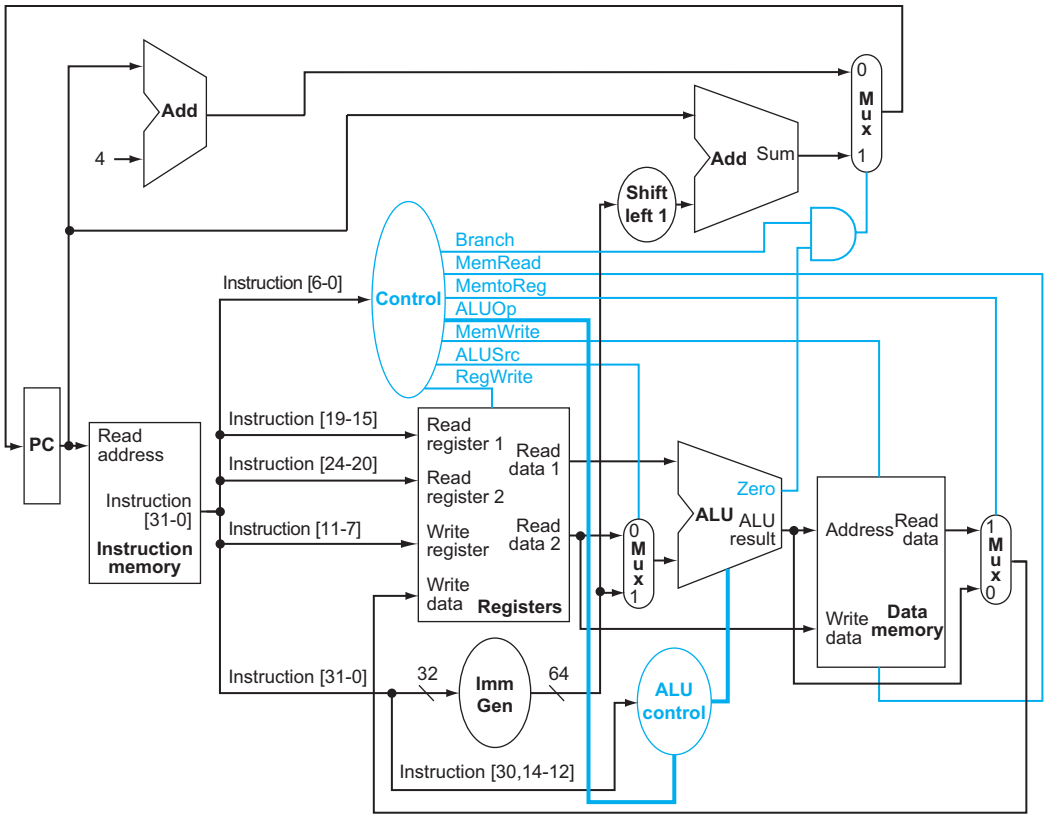


FIGURE 4.17 The simple datapath with the control unit. The input to the control unit is the 7-bit opcode field from the instruction. The outputs of the control unit consist of two 1-bit signals that are used to control multiplexors (ALUSrc and MemtoReg), three signals for controlling reads and writes 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.

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, `x2` and `x3`, 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 portions of the opcode to generate the ALU function.
4. The result from the ALU is written into the destination register (`x1`) in the register file.

Instruction	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	0	0	1	0	0	0	1	0
ld	1	1	1	1	0	0	0	0
sd	1	X	0	0	1	0	0	0
beq	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`, and `or`). For all these instructions, the source register fields are rs1 and rs2, and the destination register field is rd; this defines how the signals ALUSrc is set. Furthermore, an R-type instruction writes a register (RegWrite = 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 fields. The second and third rows of this table give the control signal settings for `ld` and `sd`. These ALUSrc and ALUOp fields are set to perform the address calculation. The MemRead and MemWrite are set to perform the memory access. Finally, RegWrite is set for a load to cause the result to be stored in the rd register. The ALUOp field for branch is set for 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. This type of don't care must be added by the designer, since it depends on knowledge of how the datapath works.

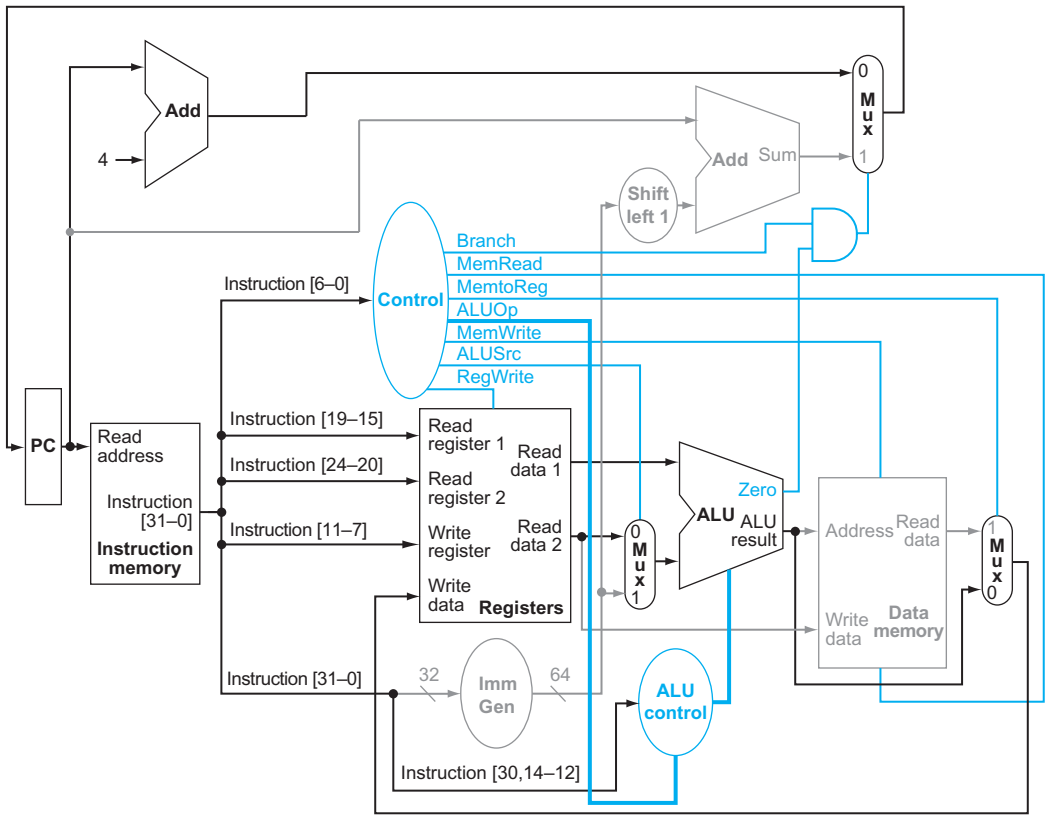


FIGURE 4.19 The datapath in operation for an R-type instruction, such as `add x1, x2, x3`. The control lines, datapath units, and connections that are active are highlighted.

Similarly, we can illustrate the execution of a load register, such as

```
ld  x1,  offset(x2)
```

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):

1. An instruction is fetched from the instruction memory, and the PC is incremented.
2. A register (`x2`) value is read from the register file.
3. The ALU computes the sum of the value read from the register file and the sign-extended 12 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 (`x1`).

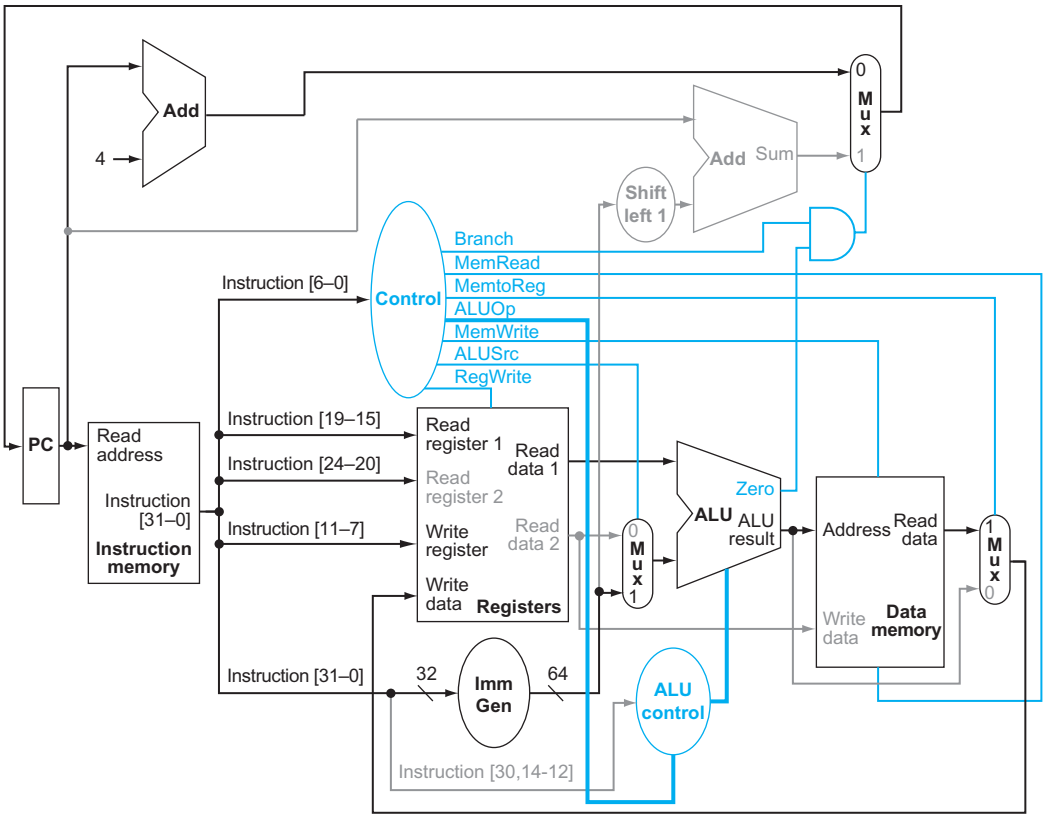


FIGURE 4.20 The datapath in operation for a load instruction. The control lines, datapath units, and connections that are active are highlighted. A store instruction would operate very similarly. The main difference would be that the memory control would indicate a write rather than a read, the second 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.

Finally, we can show the operation of the branch-if-equal instruction, such as `beq x1, x2, 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 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.
2. Two registers, `x1` and `x2`, are read from the register file.
3. The ALU subtracts one data value from the other data value, both read from the register file. The value of PC is added to the sign-extended, 12 bits of the instruction (`offset`) left shifted by one; the result is the branch target address.
4. The Zero status information from the ALU is used to decide which adder result to store in the PC.

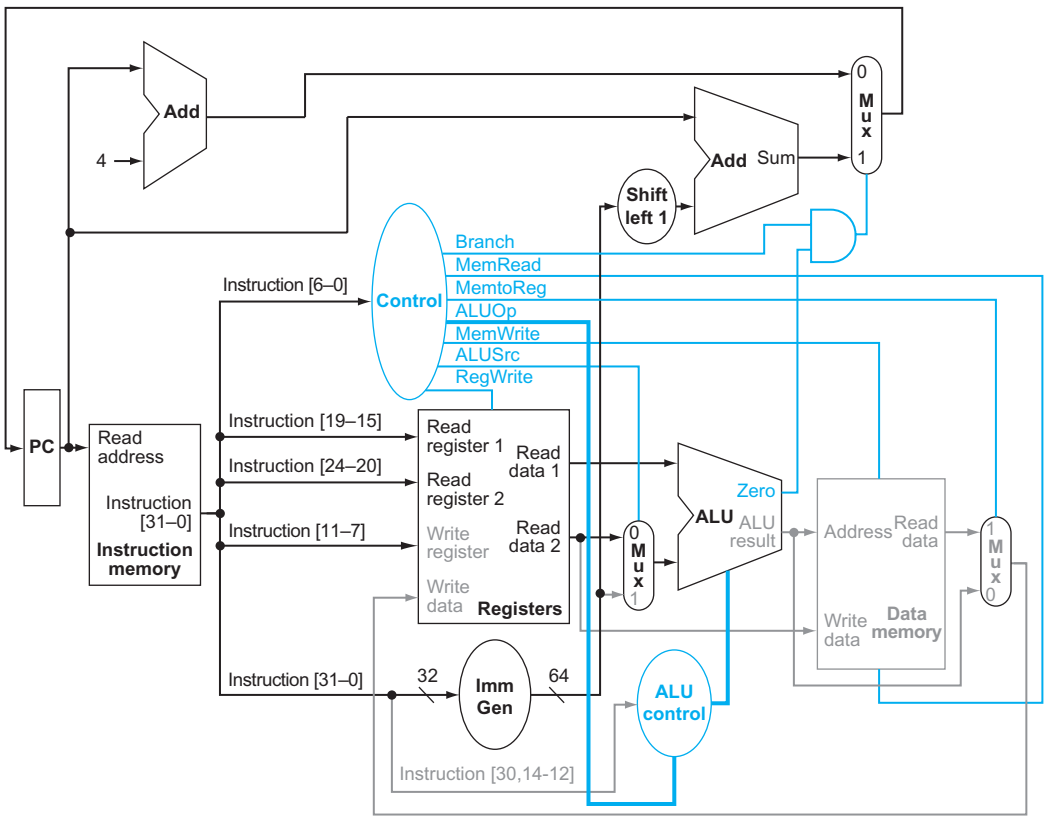


FIGURE 4.21 The datapath in operation for a branch-if-equal instruction. The control lines, datapath units, and connections that are active are highlighted. After using the register file and ALU to perform the compare, the Zero output is used to select the next program counter from between the two candidates.

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 inputs are the opcode bits. Thus, we can create a truth table for each of the outputs based on the binary encoding of the opcodes.

Figure 4.22 defines 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 C.2 in **Appendix C**.

Why a Single-Cycle Implementation is not Used Today

Although the single-cycle design will work correctly, it is too inefficient to be used in modern designs. 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 most likely a load instruction, which uses five functional units in series: the instruction memory,

Input or output	Signal name	R-format	ld	sd	beq
Inputs	I[6]	0	0	0	1
	I[5]	1	0	1	1
	I[4]	1	0	0	0
	I[3]	0	0	0	0
	I[2]	0	0	0	0
	I[1]	1	1	1	1
	I[0]	1	1	1	1
Outputs	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 instruction classes, one per column, that determine the control output settings. 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 Op4 · Op5, since this is sufficient to distinguish the R-format instructions from `ld`, `sd`, and `beq`. We do not take advantage of this simplification, since the rest of the RISC-V opcodes are used in a full implementation.

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 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.



COMMON CASE FAST

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?

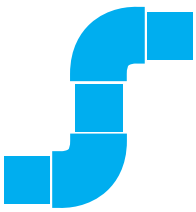
4.5 An Overview of Pipelining

Never waste time.
American proverb

pipelining An implementation technique in which multiple instructions are overlapped in execution, much like an assembly line.

Pipelining 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-A53. If you are curious about exploring the anatomy of a pipelined computer, this section is a good introduction to Sections 4.6 through 4.9.



PIPELINING

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 4.23 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,

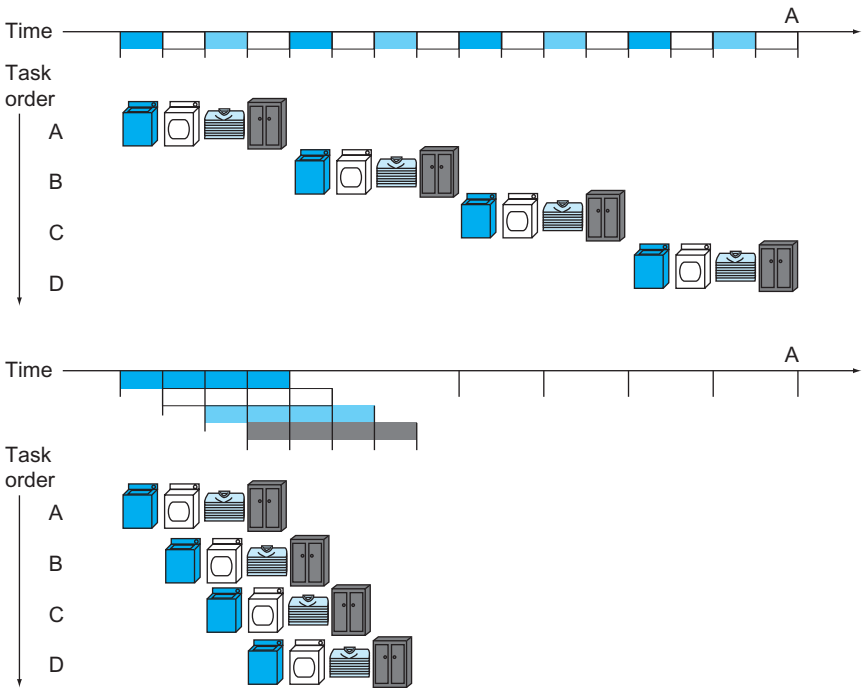


FIGURE 4.23 The laundry analogy for pipelining. Ann, Brian, Cathy, and Don each have dirty clothes to be washed, dried, folded, and put away. The washer, dryer, "folder," and "storer" each take 30 minutes for their task. Sequential laundry takes 8 hours for four loads of washing, while pipelined laundry 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.

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 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 five times as long as one load, while 20 loads of sequential laundry takes 20 times as long as one load. It's only 2.3 times faster in Figure 4.23, because we only show four loads. Notice that at the beginning and end of the workload in the pipelined version in Figure 4.23, 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 four, then the stages will be full most of the time and the increase in throughput will be very close to four.

The same principles apply to processors where we pipeline instruction execution. RISC-V instructions classically take five steps:

1. Fetch instruction from memory.
2. Read registers and decode the instruction.
3. Execute the operation or calculate an address.
4. Access an operand in data memory (if necessary).
5. Write the result into a register (if necessary).

Hence, the RISC-V 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 seven instructions: load doubleword (`ld`), store doubleword (`sd`), add (`add`), subtract (`sub`), AND (`and`), OR (`or`), and branch if equal (`beq`).

Contrast the average time between instructions of a single-cycle implementation, in which all instructions take one clock cycle, to a pipelined implementation. Assume that the operation times for the major functional units in this example are 200 ps for memory access for instructions or data, 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 4.24 shows the time required for each of the seven instructions. The single-cycle design must allow for the slowest instruction—in Figure 4.24 it is `ld`—so the time required for every instruction is 800 ps. Similarly to Figure 4.23, Figure 4.25 compares nonpipelined and pipelined execution of three load register instructions. Thus, the time between the first and fourth instructions in the nonpipelined design is 3 × 800 ps 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

$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of pipe stages}}$$

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

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load doubleword (ld)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store doubleword (sd)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, and, or)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

FIGURE 4.24 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.

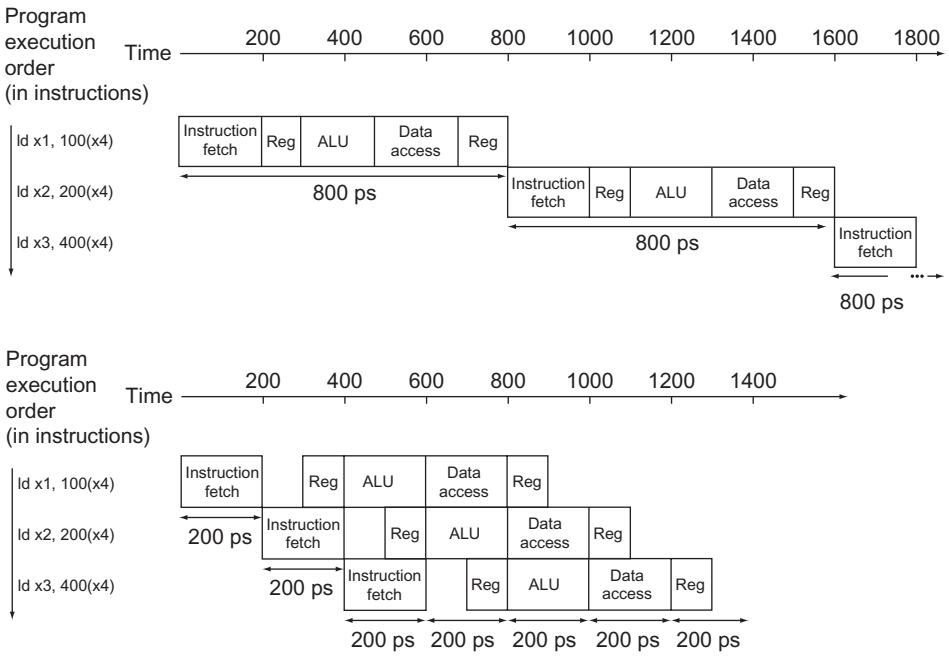


FIGURE 4.25 Single-cycle, nonpipelined execution (top) versus pipelined execution (bottom). Both use the same hardware components, whose time is listed in Figure 4.24. In this case, we see a fourfold speed-up on average time between instructions, from 800 ps down to 200 ps. Compare this figure to Figure 4.23. 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.

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.

However, 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 × 200 ps + 1400 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 × 800 ps + 2400 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:

$$\frac{800{,}002{,}400 \text{ ps}}{200{,}001{,}400 \text{ ps}} \simeq \frac{800 \text{ ps}}{200 \text{ ps}} \simeq 4.00$$

Pipelining improves performance by *increasing instruction throughput, in contrast 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 RISC-V instruction set, which was designed for pipelined execution.

First, all RISC-V 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. Modern implementations of the x86 architecture actually translate x86 instructions into simple operations that look like RISC-V instructions and then pipeline the simple operations rather than the native x86 instructions! (See Section 4.10.)

Second, RISC-V has just a few instruction formats, with the source and destination register fields being located in the same place in each instruction.

Third, memory operands only appear in loads or stores in RISC-V. 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. We will shortly see the downside of longer pipelines.

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.

Structural Hazard

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.

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.

As we said above, the RISC-V 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 4.25 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 are needed to execute the instruction are 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 that have completed drying are ready to fold and those that have finished washing 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 that sum (`x19`):

```
add  x19, x0, x1
sub  x2, x19, x3
```

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.

forwarding Also called **bypassing**. A method of resolving a data hazard by retrieving the missing data element from internal buffers rather than waiting for it to arrive from programmer-visible registers or memory.

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 far 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 4.26 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 4.23.

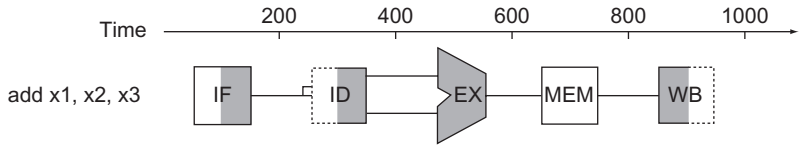


FIGURE 4.26 Graphical representation of the instruction pipeline, similar in spirit to the laundry pipeline in Figure 4.23. Here we use symbols representing the physical resources with the abbreviations for pipeline stages used throughout the chapter. The symbols for the five stages: *IF* for the instruction 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.

ANSWER

Figure 4.27 shows the connection to forward the value in `x1` after the execution stage of the `add` instruction as input to the execution stage of the `sub` instruction.

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 `x1` instead of an add. As we can imagine from looking at Figure 4.27, the

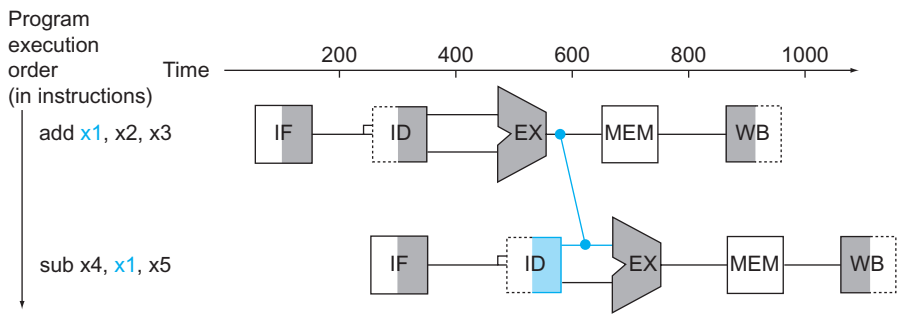


FIGURE 4.27 Graphical representation of forwarding. The connection shows the forwarding path from the output of the EX stage of `add` to the input of the EX stage for `sub`, replacing the value from register `x1` read in the second stage of `sub`.

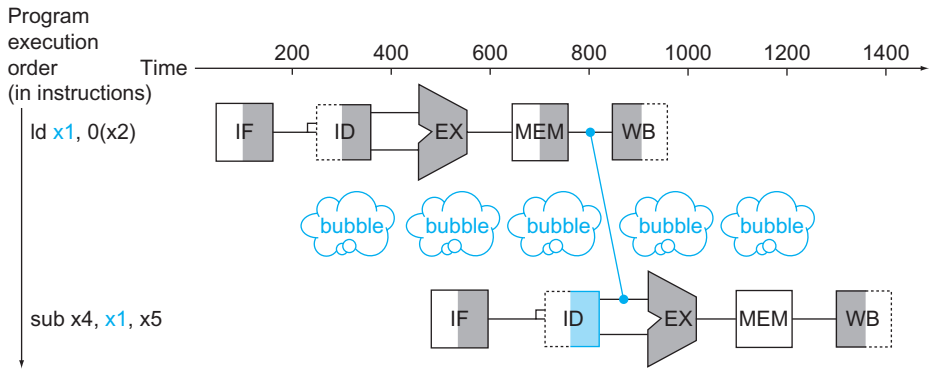


FIGURE 4.28 We need a stall even with forwarding when an R-format instruction following a load tries to use the data. Without the stall, the path from memory access stage output to execution stage input would be going backward in time, which is impossible. This figure is actually a simplification, since 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 have not yet become available when they are needed by another instruction.

pipeline stall Also called **bubble**. A stall initiated in order to resolve a hazard.

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 4.28 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.

EXAMPLE

Reordering Code to Avoid Pipeline Stalls

Consider the following code segment in C:

```
a = b + e;
c = b + f;
```

Here is the generated RISC-V code for this segment, assuming all variables are in memory and are addressable as offsets from `x31`:

```
ld      x1, 0(x31)   // Load b
ld      x2, 8(x31)   // Load e
add     x3, x1, x2   // b + e
sd      x3, 24(x31)  // Store a
ld      x4, 16(x31)  // Load f
add     x5, x1, x4   // b + f
sd      x5, 32(x31)  // Store c
```

Find the hazards in the preceding code segment and reorder the instructions to avoid any pipeline stalls.

ANSWER

Both `add` instructions have a hazard because of their respective dependence on the previous `ld` instruction. Notice that forwarding eliminates several other potential hazards, including the dependence of the first `add` on the first `ld` and any hazards for store instructions. Moving up the third `ld` instruction to become the third instruction eliminates both hazards:

```
ld   x1, 0(x31)
ld   x2, 8(x31)
ld   x4, 16(x31)
add  x3, x1, x2
sd   x3, 24(x31)
add  x5, x1, x4
sd   x5, 32(x31)
```

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 RISC-V architecture, in addition to the three mentioned on page 267. Each RISC-V 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.

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.

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 are 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 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.

The equivalent decision task in a computer is the conditional branch instruction. Notice that we must begin fetching the instruction following the branch on the following 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 a register, calculate the branch address, and update the PC during the second stage of the pipeline (see Section 4.8 for details). Even with this added hardware, the pipeline involving conditional branches would look like Figure 4.29. The instruction to be 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 3.28 in Chapter 3 shows that conditional branches are 17% of the instructions executed in SPECint2006. Since the other instructions run have a CPI of 1, and conditional 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.

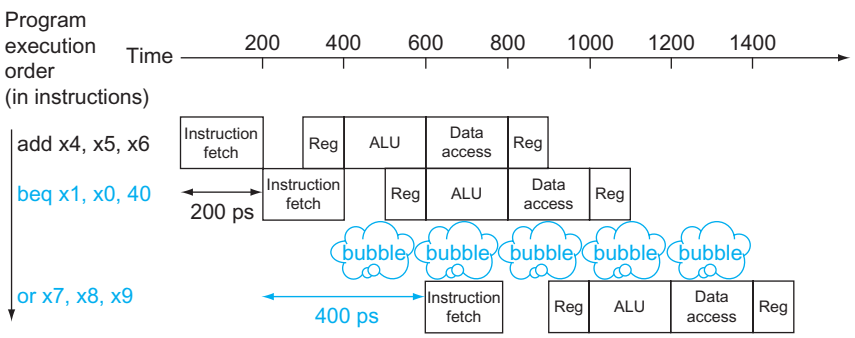


FIGURE 4.29 Pipeline showing stalling on every conditional branch as solution to control hazards. This example assumes the conditional branch is taken, and the instruction at the destination of the branch is the `or` instruction. There is a one-stage pipeline stall, or bubble, after the branch. In reality, 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 conditional 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 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 conditional branches. One simple approach is to predict always that conditional branches will be untaken. When you're right, the pipeline proceeds at full speed. Only when conditional branches are taken does the pipeline stall. Figure 4.30 shows such an example.

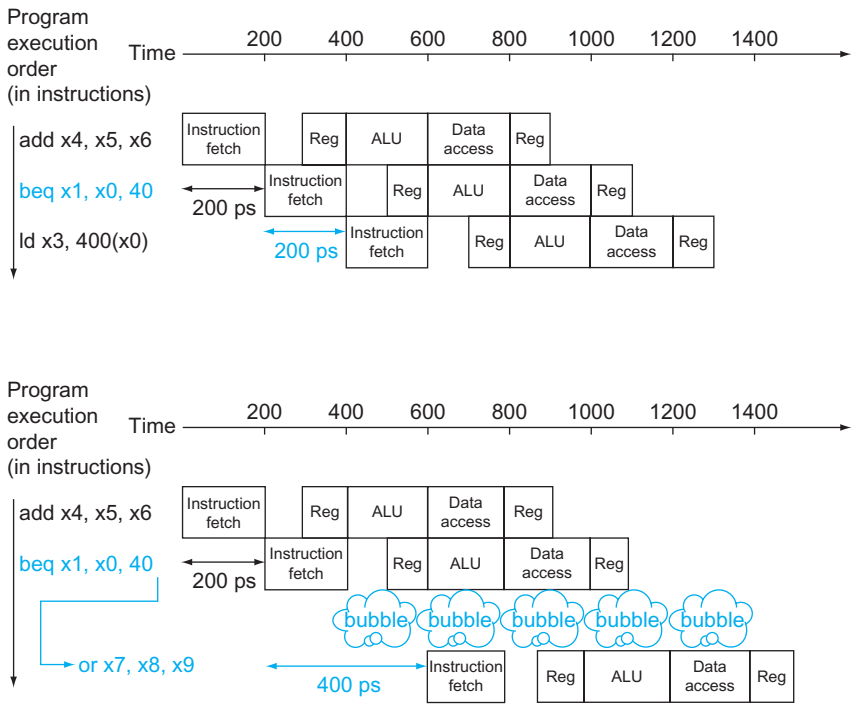


FIGURE 4.30 Predicting that branches are not taken as a solution to control hazard. The top drawing shows the pipeline when the branch is not taken. The bottom drawing shows the pipeline when the branch is taken. As we noted in Figure 4.29, the insertion of a bubble in this fashion simplifies what actually 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 conditional branch and proceeds from that assumption rather than waiting to ascertain the actual outcome.

A more sophisticated version of **branch prediction** would have some conditional 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 conditional branches that branch back to the top of the loop. Since they are likely to be taken and they branch backward, we could always predict taken for conditional branches that branch 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 conditional branch and may change predictions for a conditional 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.



PREDICTION

One popular approach to dynamic prediction of conditional branches is keeping a history for each conditional 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 conditional 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 conditional 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.

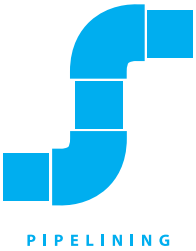
As in the case of all other solutions to control hazards, longer pipelines exacerbate the problem, in this case by raising the cost of misprediction. Solutions to control hazards are described in more detail in Section 4.8.

Elaboration: There is a third approach to the control hazard, called a *delayed decision*. In our analogy, whenever you are going to make such a decision about laundry, just place a load of non-football 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, 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 4.29 does not affect the branch and can be moved after the branch to hide the branch delay fully. Since delayed branches are useful when the branches are short, it is rare to see a processor with 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** between the instructions in a sequential instruction stream. It has the substantial advantage that, unlike programming a multiprocessor (see Chapter 6), it is fundamentally invisible to the programmer.

In the next few sections of this chapter, we cover the concept of pipelining using the RISC-V 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 next read Section 4.8 to learn more about solutions to branch hazards, and finally see how exceptions are handled in Section 4.9.

Check Yourself

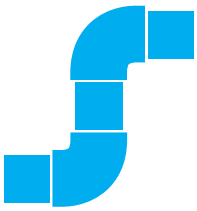
For each code sequence below, state whether it must stall, can avoid stalls using only forwarding, or can execute without stalling or forwarding.

Sequence 1	Sequence 2	Sequence 3
`ld x10, 0(x10)`	`add x11, x10, x10`	`addi x11, x10, 1`
`add x11, x10, x10`	`addi x12, x10, 5`	`addi x12, x10, 2`
	`addi x14, x11, 5`	`addi x13, x10, 3`
		`addi x14, x10, 4`
		`addi x15, x10, 5`

Understanding Program Performance

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 conditional branch frequencies as well as less predictable branches. Data hazards can be



PIPELINING

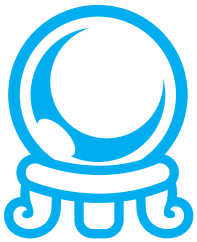
performance bottlenecks in both integer and floating-point programs. Often it is easier to deal with data hazards in floating-point programs because the lower conditional branch frequency and more regular memory access patterns allow the compiler to try to schedule instructions to avoid hazards. It is more difficult to perform such optimizations in integer programs that have less regular memory accesses, 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.

The BIG Picture

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 five 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 make life harder or simpler 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.

latency (pipeline) The number of stages in a pipeline or the number of stages between two instructions during execution.



PREDICTION

There is less in this than meets the eye.

Tallulah Bankhead, remark to Alexander Woollcott, 1922

4.6 Pipelined Datapath and Control

Figure 4.31 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 4.31, these five components correspond roughly to the way the datapath is drawn; instructions and data move generally from left to right through the

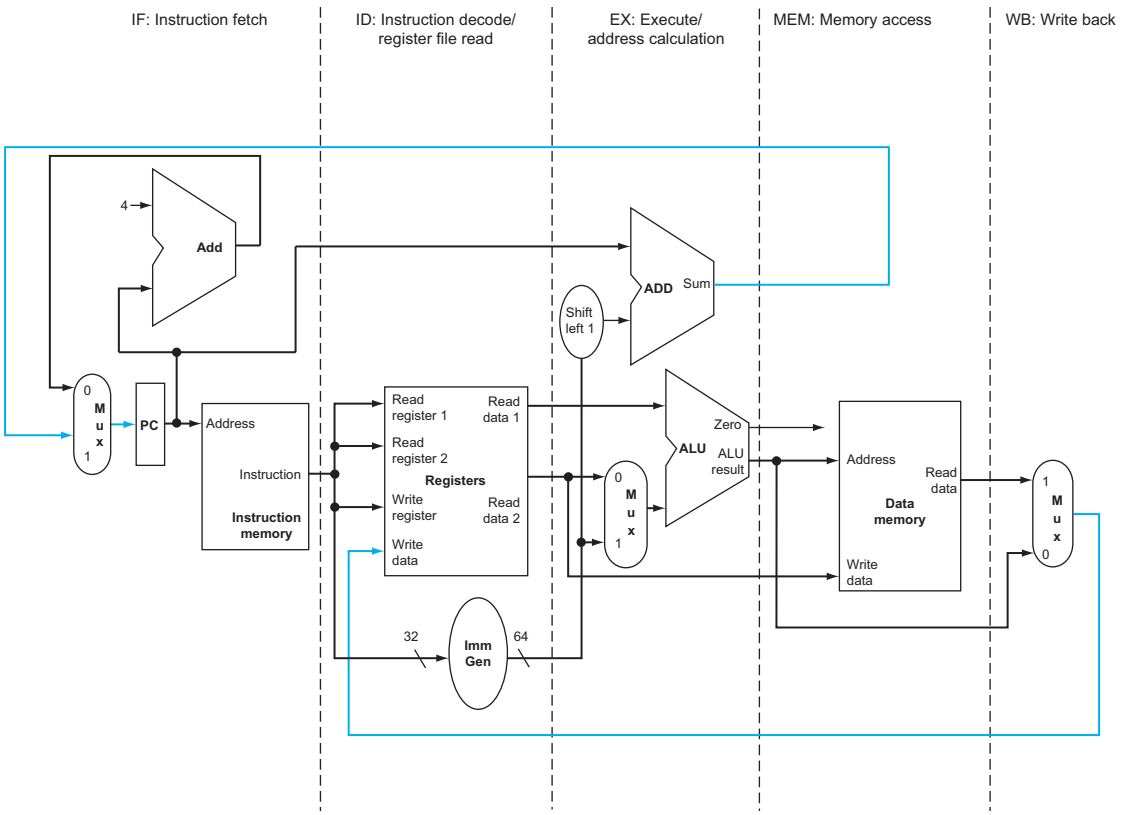


FIGURE 4.31 The single-cycle datapath from Section 4.4 (similar to Figure 4.17). Each step of the instruction can be mapped onto the datapath from left to right. The only exceptions are the update of the PC and the write-back step, shown in color, which sends either the ALU result or the data from memory 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 do 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 4.32 shows the execution of the instructions in Figure 4.25 by displaying their private datapaths on a common timeline. We use a stylized version of the datapath in Figure 4.31 to show the relationships in Figure 4.32.

Figure 4.32 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 4.32 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 4.31. Returning to our laundry analogy, we might have a basket between each pair of stages to hold the clothes for the next step.

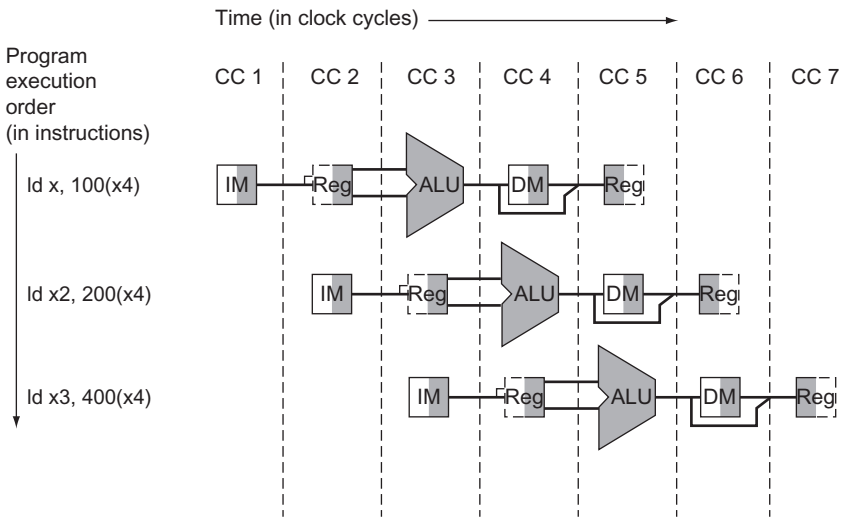


FIGURE 4.32 Instructions being executed using the single-cycle datapath in Figure 4.31, assuming pipelined execution. Similar to Figures 4.26 through 4.28, this figure pretends that each instruction has its own datapath, and shades each portion according to use. Unlike those figures, each stage is labeled by the physical resource used in that stage, corresponding to the portions of the datapath in Figure 4.31. *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 4.33 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 one 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 4.33, 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

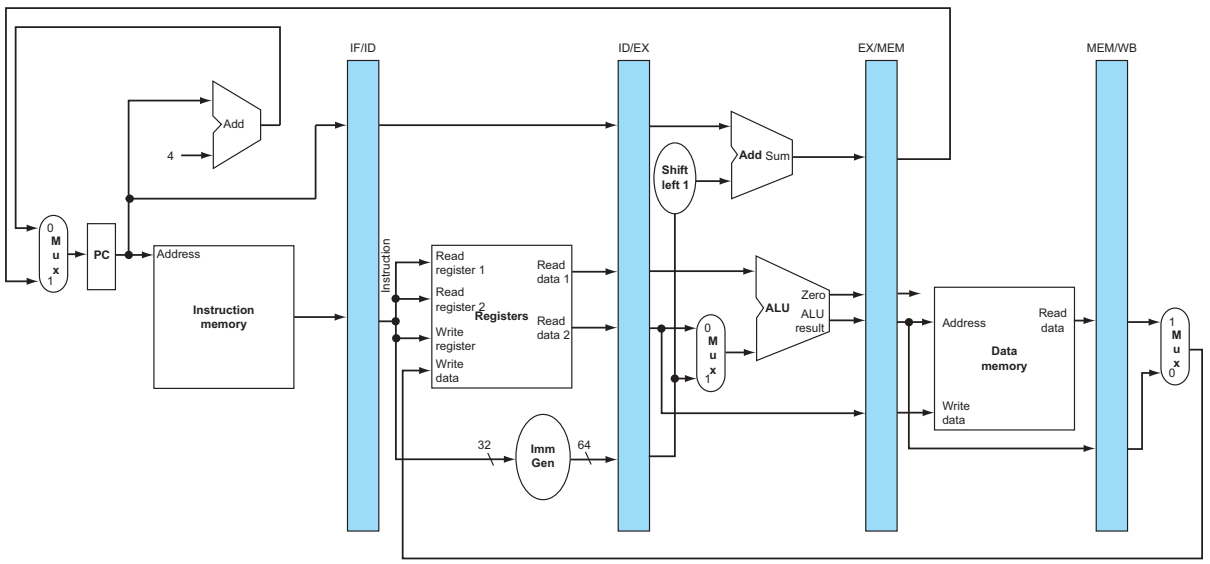


FIGURE 4.33 The pipelined version of the datapath in Figure 4.31. The pipeline registers, in color, separate each pipeline stage. They are labeled by the stages that they separate; for example, the first is labeled *IF/ID* because it separates the instruction fetch and instruction decode stages. The 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 96 bits wide, because it must hold both the 32-bit instruction fetched from memory and the incremented 64-bit PC address. We will expand these registers over the course of this chapter, but for now the other three pipeline registers contain 256, 193, and 128 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 4.34 through 4.37, 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 4.26 through 4.28, 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 `ld` 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 4.34 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 PC 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 4.34 shows the instruction portion of the IF/ID pipeline register supplying the immediate field, which is sign-extended to 64 bits, and the register numbers to read the two registers. All three values are stored in the ID/EX pipeline register, along with the PC address. We again transfer everything that might be needed by any instruction during a later clock cycle.

3. *Execute or address calculation:* Figure 4.35 shows that the load instruction reads the contents of a register 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 4.36 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 4.36 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:

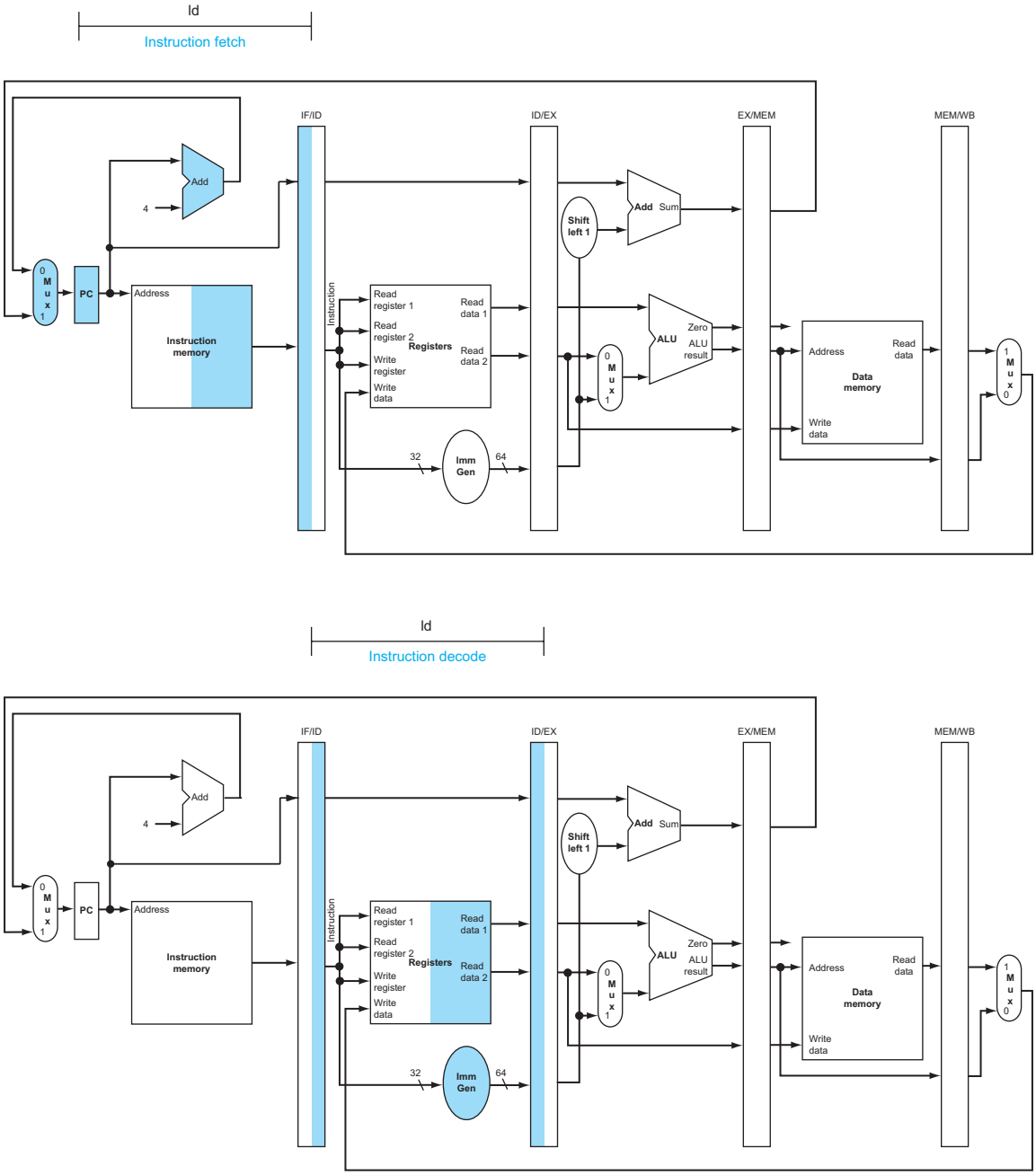


FIGURE 4.34 IF and ID: First and second pipe stages of an instruction, with the active portions of the datapath in Figure 4.33 highlighted. The highlighting convention is the same as that used in Figure 4.26. As in Section 4.2, there is no confusion when reading and writing registers, because the contents change only on the clock edge. Although the load needs only the top register in stage 2, it doesn't hurt to do potentially extra work, so it sign-extends the 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.

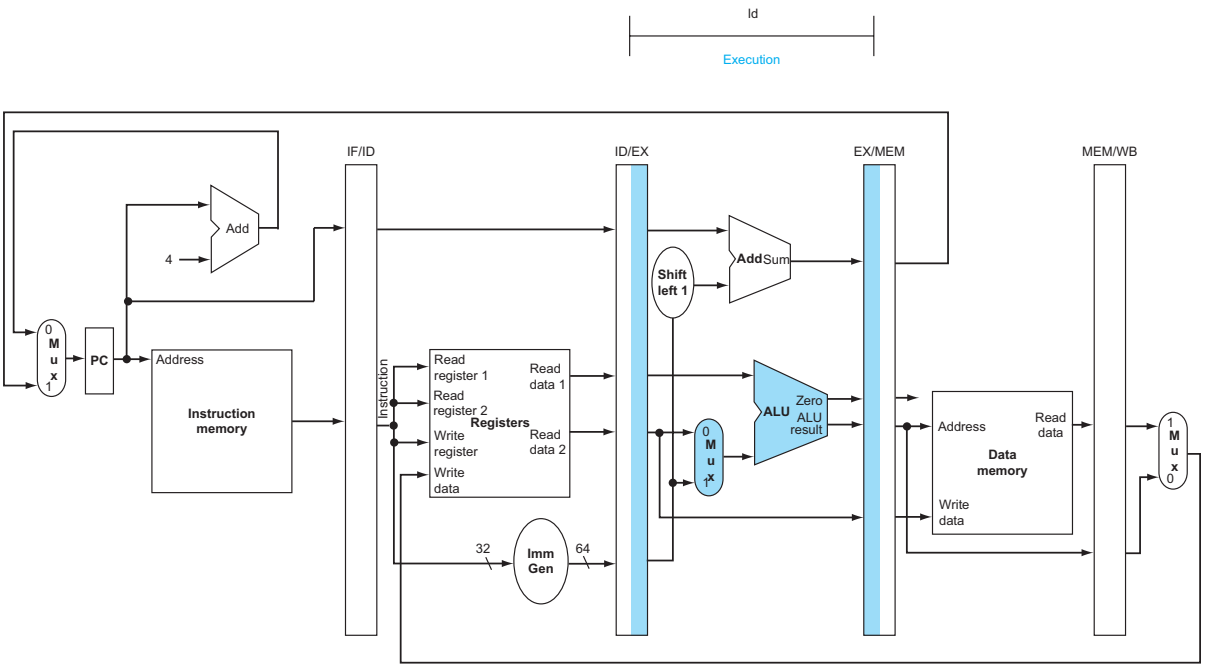


FIGURE 4.35 EX: The third pipe stage of a load instruction, highlighting the portions of the datapath in Figure 4.33 used in this pipe stage. The register is added to the sign-extended immediate, and the sum is placed in the EX/MEM pipeline register.

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 4.34 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 immediate operand. These three 64-bit values are all stored in the ID/EX pipeline register. The bottom portion of Figure 4.34 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. (While the store instruction uses the rs2 field to read the second register in this pipe stage, that detail is not shown in this pipeline diagram, so we can use the same figure for both.)
3. *Execute and address calculation:* Figure 4.37 shows the third step; the effective address is placed in the EX/MEM pipeline register.
4. *Memory access:* The top portion of Figure 4.38 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.

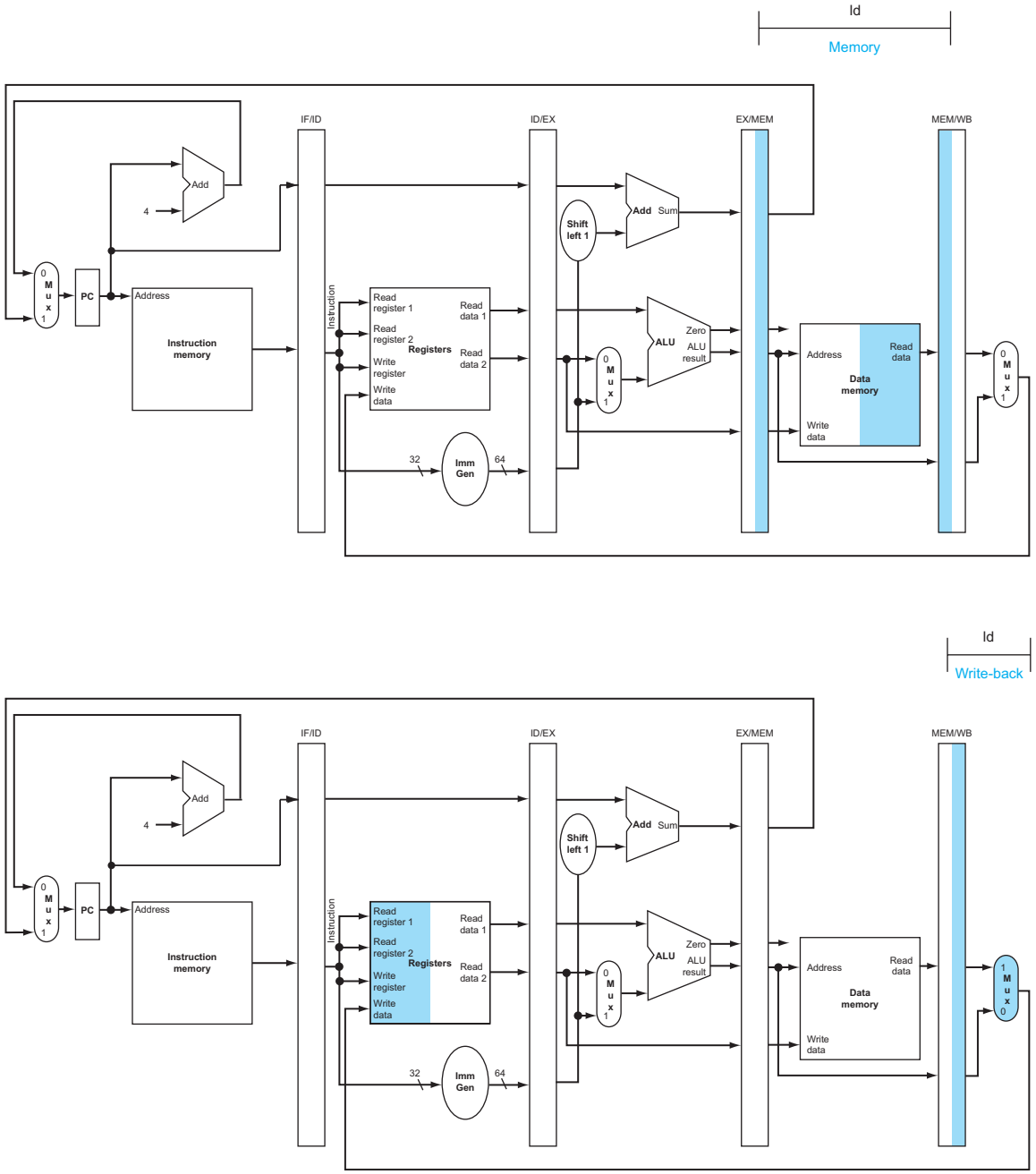


FIGURE 4.36 MEM and WB: The fourth and fifth pipe stages of a load instruction, highlighting the portions of the datapath in Figure 4.33 used in this pipe stage. Data memory is read using the address in the EX/MEM pipeline registers, and the data are placed in the MEM/WB pipeline register. Next, data are 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 4.39.

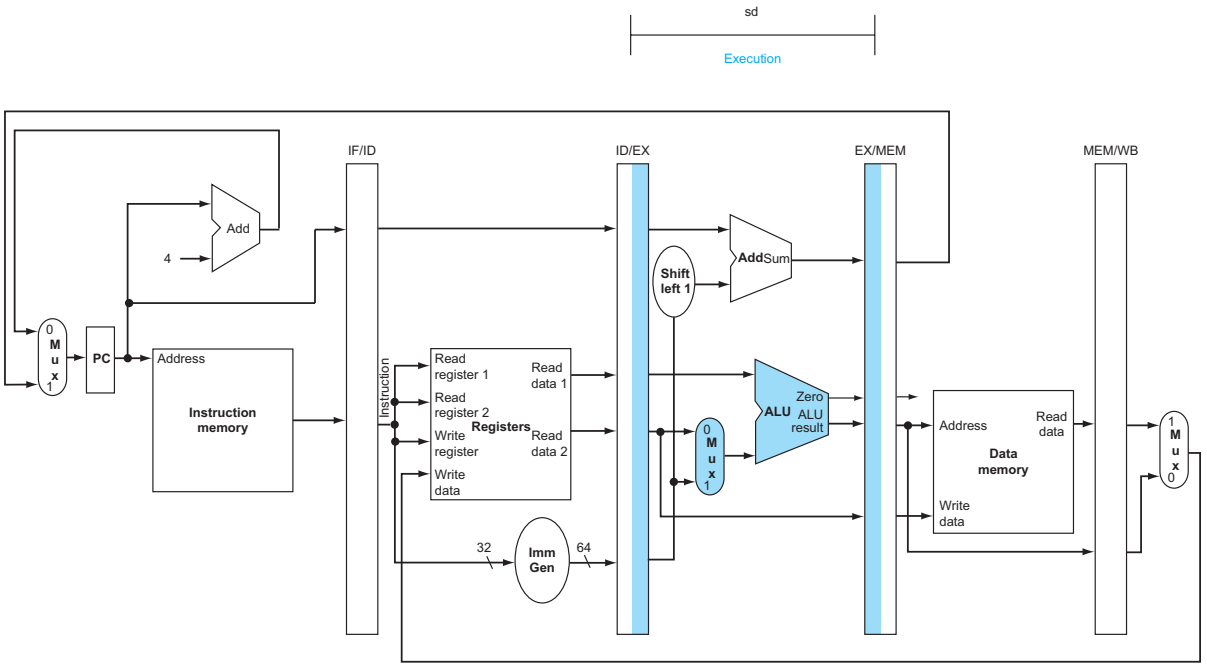


FIGURE 4.37 EX: The third pipe stage of a store instruction. Unlike the third stage of the load instruction in Figure 4.35, the second register value is loaded into the EX/MEM pipeline register to be used in the next stage. Although it wouldn't hurt to always write this second register into the EX/MEM 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 4.38 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 were 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 267). 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,

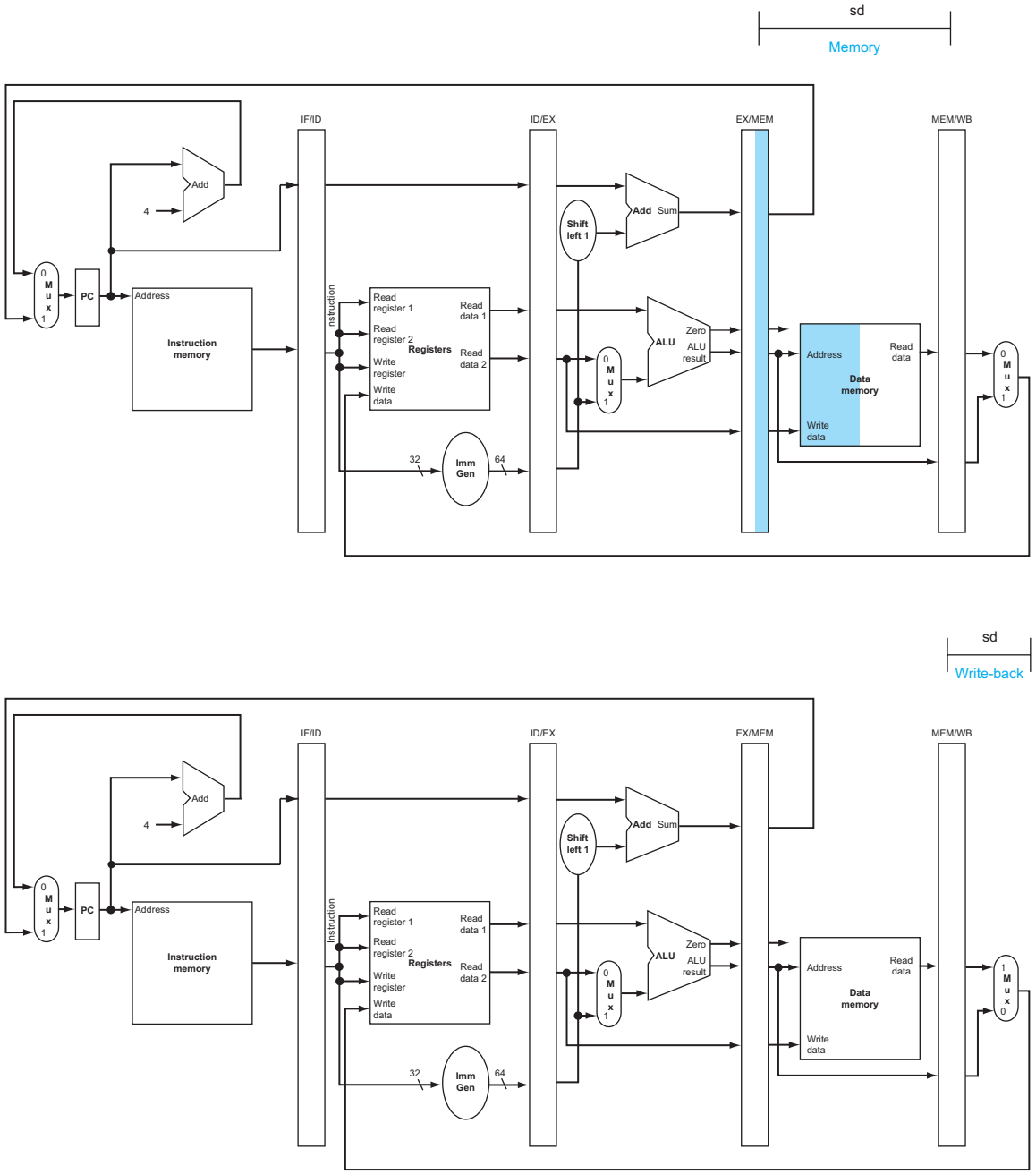


FIGURE 4.38 MEM and WB: The fourth and fifth pipe stages of a store instruction. In the fourth stage, the data are written into data memory for the store. Note that the data come from the EX/MEM pipeline register and that nothing is changed in the MEM/WB pipeline register. Once the data are written in memory, there is nothing left for the store instruction to do, so nothing happens in stage 5.

which 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 *value* 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 4.39 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 4.40 is a single drawing of the corrected datapath, highlighting the hardware used in all five stages of the load register instruction in Figures 4.34 through 4.36. 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 master, since many instructions are simultaneously executing in a single datapath in every clock cycle. To aid understanding, there are

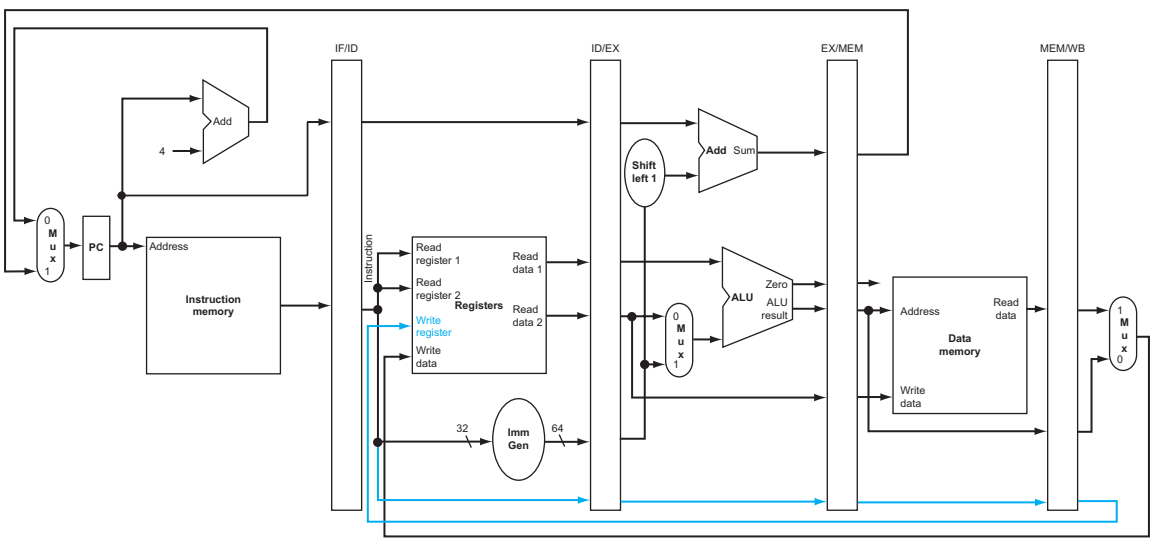


FIGURE 4.39 The corrected pipelined datapath to handle the load instruction properly. The write register number now comes from the MEM/WB pipeline register along with the data. The register number is passed from the ID pipe stage until it reaches the MEM/WB pipeline register, adding five more bits to 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 4.32 on page 278, and *single-clock-cycle pipeline diagrams*, such as Figures 4.34 through 4.38. The multiple-clock-cycle diagrams are simpler but do not contain all the details. For example, consider the following five-instruction sequence:

```
ld      x10, 40(x1)
sub     x11, x2, x3
add     x12, x3, x4
ld      x13, 48(x1)
add     x14, x5, x6
```

Figure 4.41 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 4.23. 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 4.42 shows the more traditional version of the multiple-clock-cycle pipeline diagram. Note that Figure 4.41 shows the physical resources used at each stage, while Figure 4.42 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

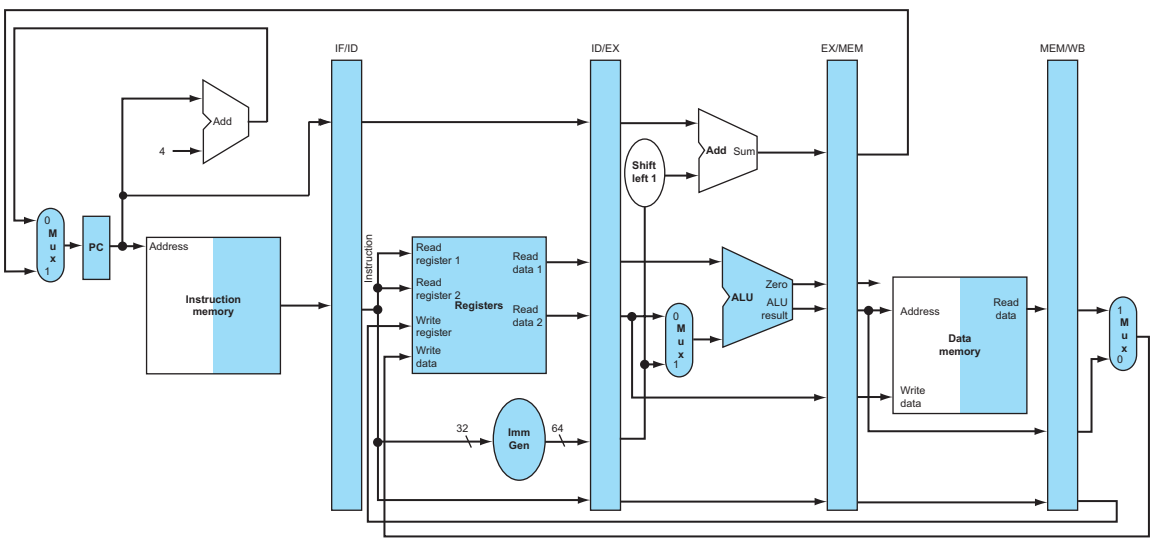


FIGURE 4.40 The portion of the datapath in Figure 4.39 that is used in all five stages of a load instruction.

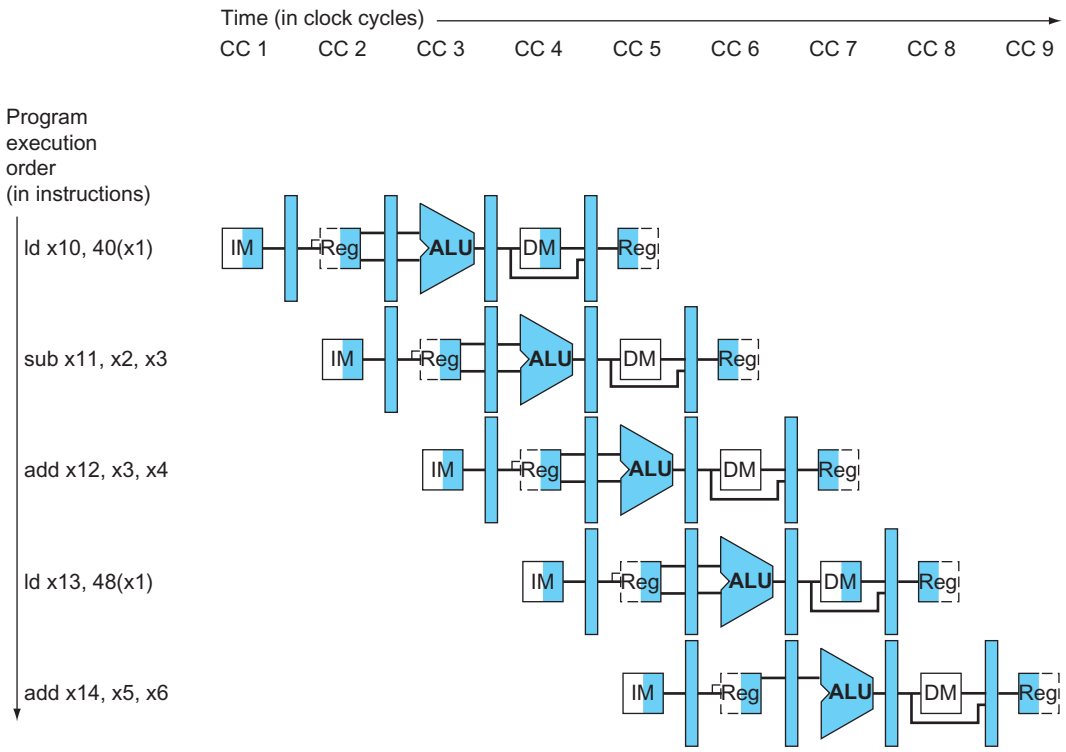


FIGURE 4.41 Multiple-clock-cycle pipeline diagram of five instructions. This style of pipeline representation shows the complete execution of instructions in a single figure. Instructions are listed in instruction execution order from top to bottom, and clock cycles move from left to right. Unlike Figure 4.26, here we show the pipeline registers between each stage. Figure 4.42 shows the traditional way to draw this diagram.

cycle; typically, 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 4.41.) A single-clock-cycle diagram represents a vertical slice of one clock cycle 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 4.43 shows the single-clock-cycle diagram corresponding to clock cycle 5 of Figures 4.41 and 4.42. 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?

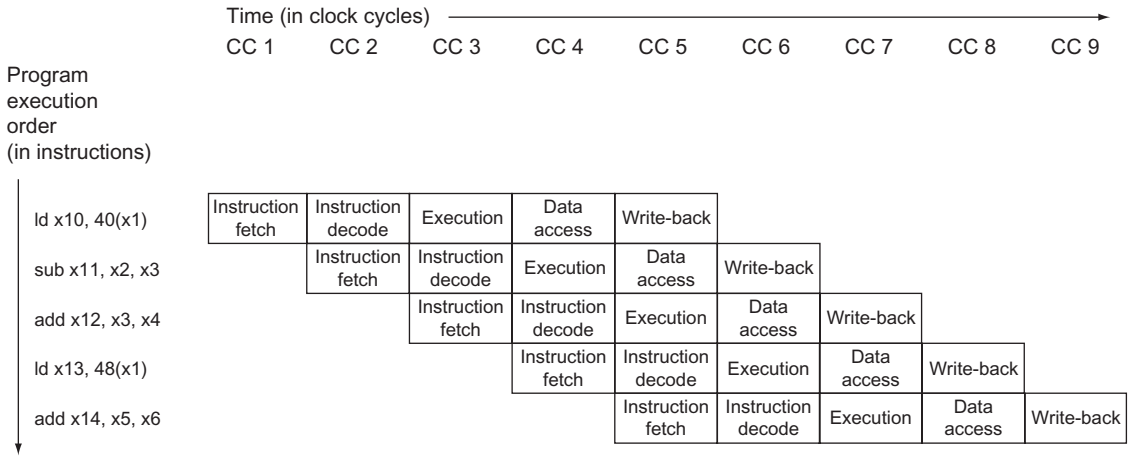


FIGURE 4.42 Traditional multiple-clock-cycle pipeline diagram of five instructions in Figure 4.41.

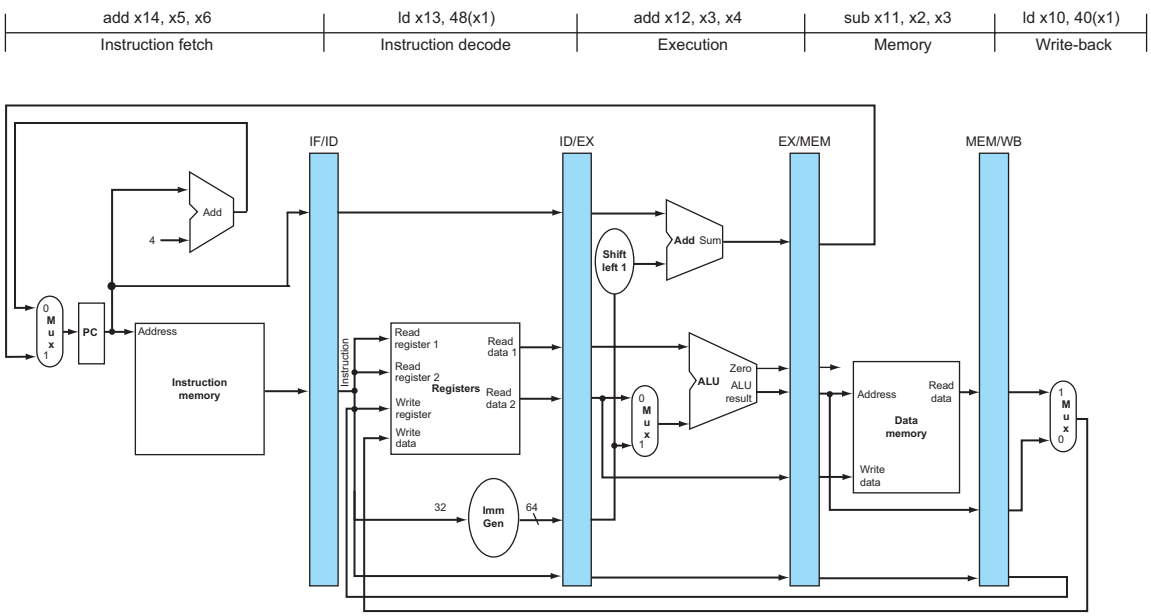


FIGURE 4.43 The single-clock-cycle diagram corresponding to clock cycle 5 of the pipeline in Figures 4.41 and 4.42. As you can see, a single-clock-cycle figure is a vertical slice through a multiple-clock-cycle diagram.

1. Allowing 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 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.

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

Pipelined Control

Just as we added control to the single-cycle datapath in Section 4.4, 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 4.44 shows those lines. We borrow as much as we can from the control for the simple

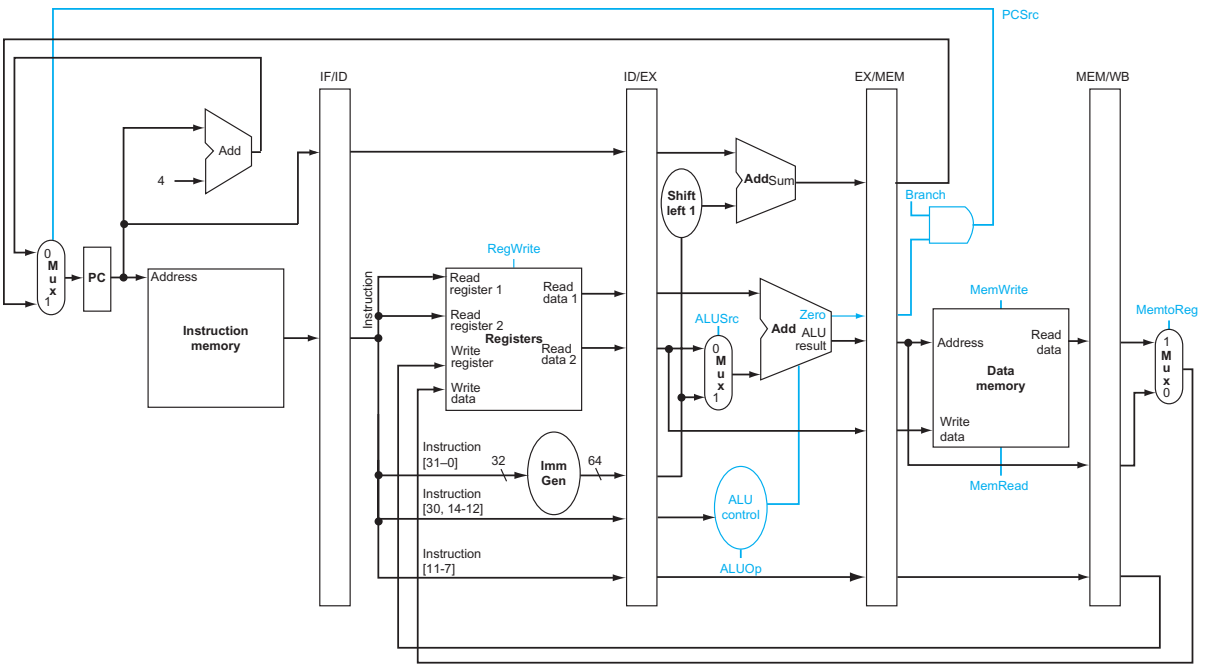


FIGURE 4.44 The pipelined datapath of Figure 4.39 with the control signals identified. This datapath borrows the control logic for PC source, register destination number, and ALU control from Section 4.4. Note that we now need funct fields of the instruction in the EX stage as input to ALU control, so these bits must also be included in the ID/EX pipeline register.

datapath in Figure 4.17. In particular, we use the same ALU control logic, branch logic, and control lines. These functions are defined in Figures 4.12, 4.16, and 4.18. We reproduce the key information in Figures 4.45 through 4.47 on a single page to make the following discussion easier to absorb.

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:* The two source registers are always in the same location in the RISC-V instruction formats, so there is nothing special to control in this pipeline stage.
3. *Execution/address calculation:* The signals to be set are ALUOp and ALUSrc (see Figures 4.45 and 4.46). The signals select the ALU operation and either Read data 2 or a sign-extended immediate as inputs to the ALU.
4. *Memory access:* The control lines set in this stage are Branch, MemRead, and MemWrite. The branch if equal, load, and store instructions set these signals, respectively. Recall that PCSrc in Figure 4.46 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 RegWrite, 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 4.47 has the same values as in Section 4.4, but now the seven control lines are grouped by pipeline stage.

Instruction	ALUOp	operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
ld	00	load doubleword	XXXXXXX	XXX	add	0010
sd	00	store doubleword	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

FIGURE 4.45 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 opcodes for the R-type instruction.

Signal name	Effect when deasserted	Effect when asserted
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, 12 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 4.46 A copy of Figure 4.16. The function of each of six control signals is defined. The ALU control lines (ALUOp) are defined in the second column of Figure 4.45. When a 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. Note that PCSrc is controlled by an AND gate in Figure 4.44. 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	
	ALUOp	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	10	0	0	0	0	1	0
ld	00	1	0	1	0	1	1
sd	00	1	0	0	1	0	X
beq	01	0	1	0	0	0	X

FIGURE 4.47 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.

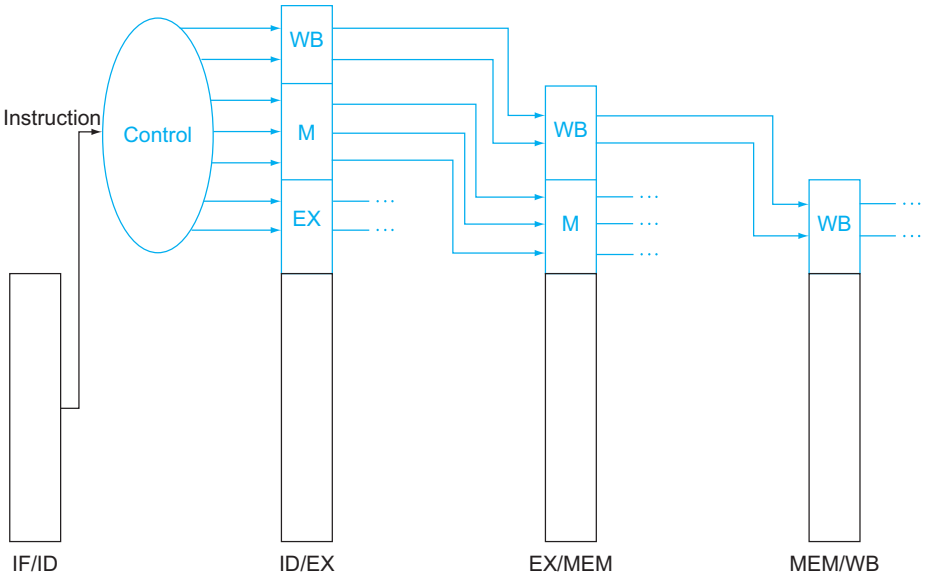


FIGURE 4.48 The seven control lines for the final three stages. Note that two of the seven control lines are used in the EX phase, with the remaining five control lines passed on to the EX/MEM pipeline register extended to hold the control lines; three are used during the MEM stage, and the last two are passed to MEM/WB for use in the WB stage.

Implementing control means setting the seven control lines to these values in each stage for each instruction.

Since the rest of the control lines starts with the EX stage, we can create the control information during instruction decode for the later stages. The simplest way to pass these control signals is to extend the pipeline registers to include control information. Figure 4.48 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 4.39. Figure 4.49 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 RISC-V code executing on pipelined hardware using single-clock diagrams, if you would like to see more details.)

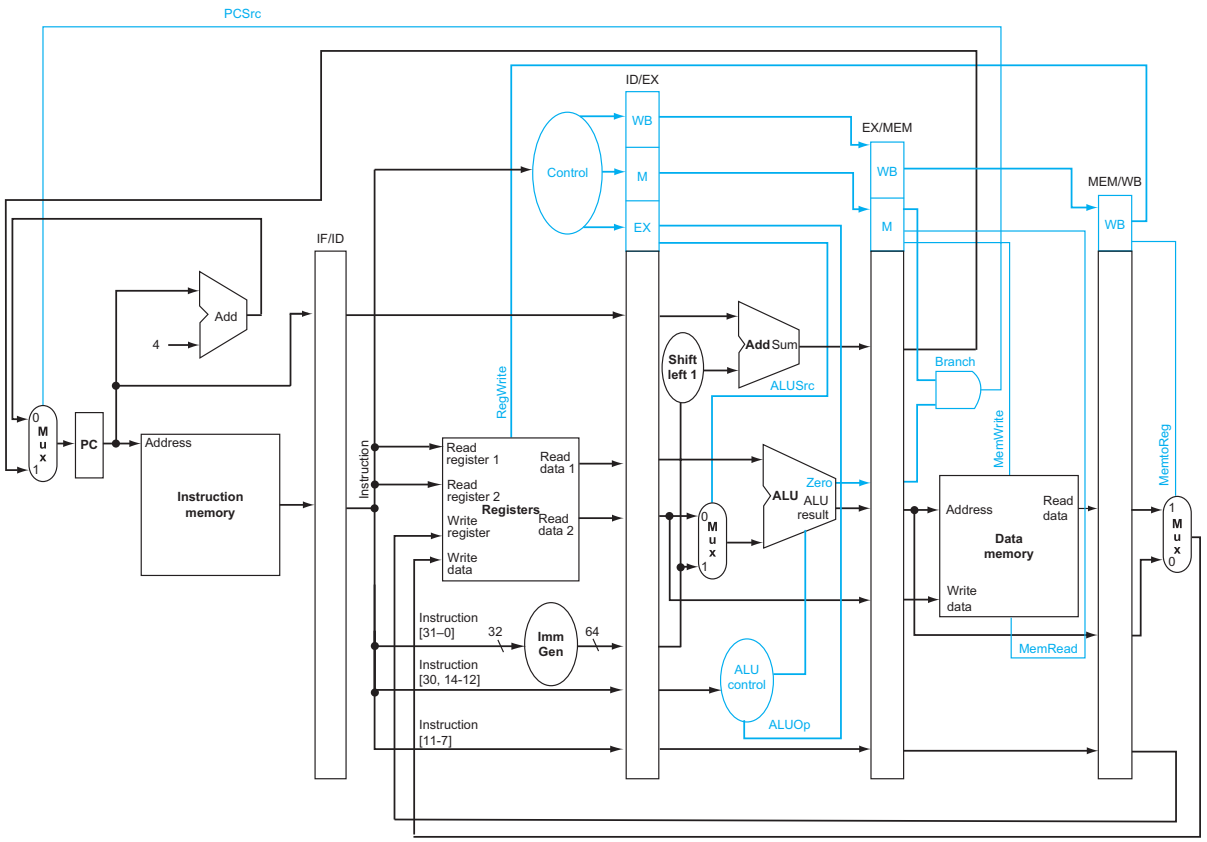


FIGURE 4.49 The pipelined datapath of Figure 4.44, with the control signals connected to the control portions of the pipeline registers. The control values for the last three stages are created during the instruction decode stage and then placed in the ID/EX pipeline register. The control lines for each pipe stage are used, and remaining control lines are then passed to the next pipeline stage.

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

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 RISC-V instructions in Figures 4.41 through 4.43 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.

Let's look at a sequence with many dependences, shown in color:

```
sub   x2, x1, x3    // Register z2 written by sub
and   x12, x2, x5   // 1st operand(x2) depends on sub
or    x13, x6, x2   // 2nd operand(x2) depends on sub
add   x14, x2, x2   // 1st(x2) & 2nd(x2) depend on sub
sd    x15, 100(x2)  // Base (x2) depends on sub
```

The last four instructions are all dependent on the result in register `x2` of the first instruction. If register `x2` 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 `x2`.

How would this sequence perform with our pipeline? Figure 4.50 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 4.50 shows the value of register `x2`, 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 4.50 shows that the values read for register `x2` 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 `sd`; the `and` and `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 of the `sub` instruction or clock cycle 3. When are the data actually needed by the `and` and `or` instructions? The answer is at the beginning of the EX stage of the `and` and `or` instructions, 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 ready 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.RegisterRs1" 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,

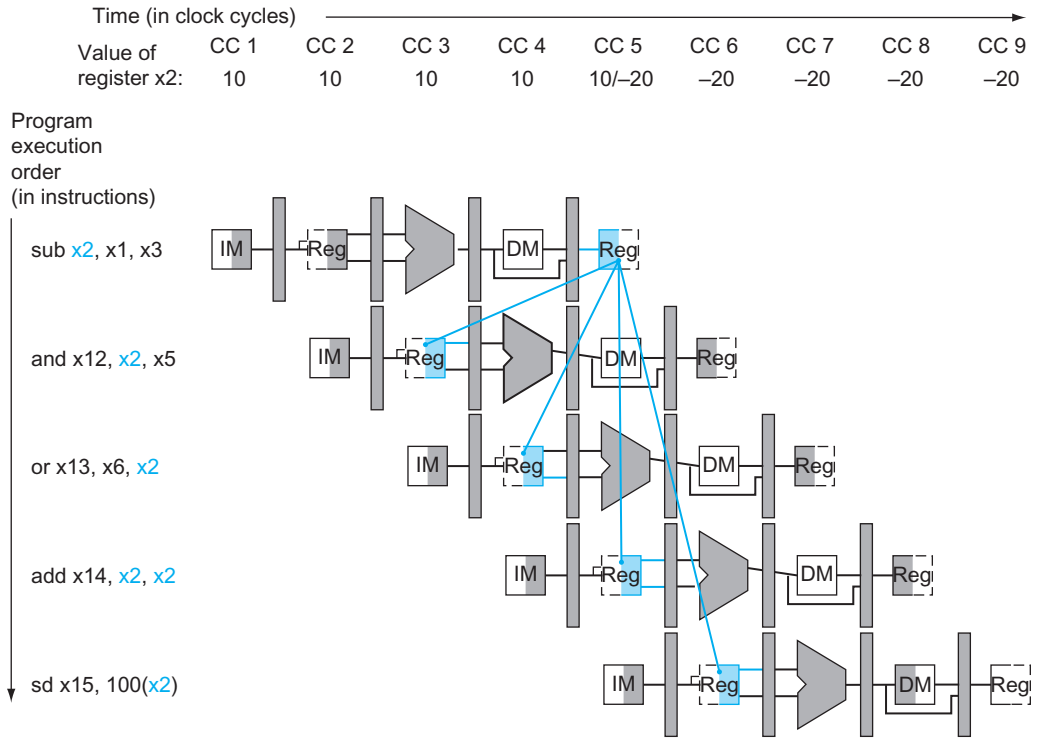


FIGURE 4.50 Pipelined dependences in a five-instruction sequence using simplified datapaths to show the dependences. All the dependent actions are shown in color, and "CC 1" at the top of the figure means clock cycle 1. The first instruction writes into `x2`, and all the following instructions read `x2`. 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*.

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.RegisterRs1

1b. EX/MEM.RegisterRd = ID/EX.RegisterRs2

2a. MEM/WB.RegisterRd = ID/EX.RegisterRs1

2b. MEM/WB.RegisterRd = ID/EX.RegisterRs2

The first hazard in the sequence on page 295 is on register `x2`, between the result of `sub x2, x1, x3` and the first read operand of `and x12, x2, x5`. 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:

$$\text{EX/MEM.RegisterRd} = \text{ID/EX.RegisterRs1} = \texttt{x2}$$

EXAMPLE

Dependence Detection

Classify the dependences in this sequence from page 295:

```
sub x2, x1, x3      // Register x2 set by sub
and x12, x2, x5     // 1st operand(z2) set by sub
or  x13, x6, x2     // 2nd operand(x2) set by sub
add x14, x2, x2     // 1st(x2) & 2nd(x2) set by sub
sd  x15, 100(x2)    // Index(x2) set by sub
```

ANSWER

As mentioned above, the `sub-and` is a type 1a hazard. The remaining hazards are as follows:

- The `sub-or` is a type 2b hazard:

  $$\text{MEM/WB.RegisterRd} = \text{ID/EX.RegisterRs2} = \texttt{x2}$$

- 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 `sd` because `sd` reads `x2` the clock cycle *after* `sub` writes `x2`.

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 RISC-V requires that every use of `x0` as an operand must yield an operand value of 0. If an instruction in the pipeline has `x0` as its destination (for example, `addi x0, x1, 2`), we want to avoid forwarding its possibly nonzero result value. Not forwarding results destined for `x0` frees the assembly programmer and the compiler of any requirement to avoid using `x0` as a destination. The conditions above thus work properly as long as 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 4.51 shows the dependences between the pipeline registers and the inputs to the ALU for the same code sequence as in Figure 4.50. 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 exist 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 correct 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 hazards.

For now, we will assume the only instructions we need to forward are the four R-format instructions: `add`, `sub`, `and`, and `or`. Figure 4.52 shows a close-up of the ALU and pipeline register before and after adding forwarding. Figure 4.53

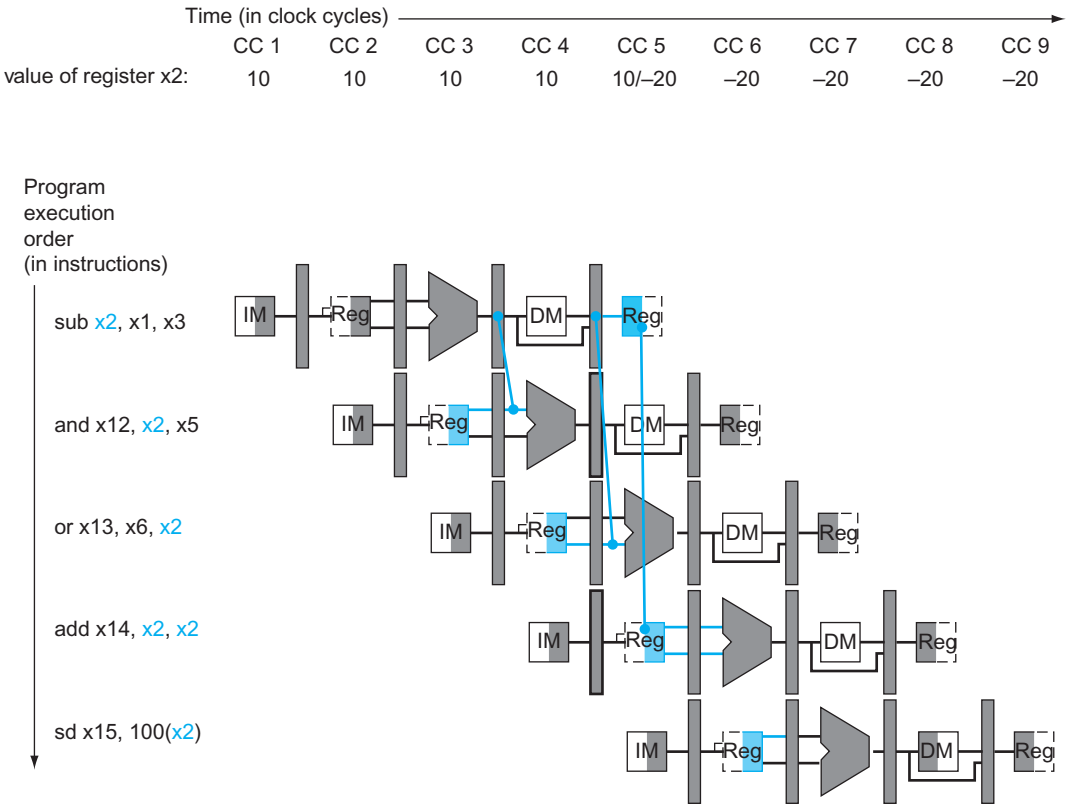
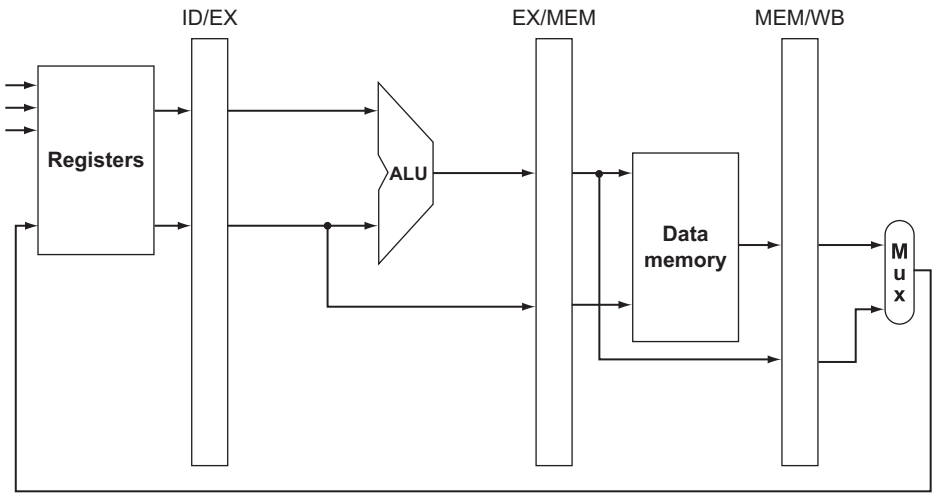
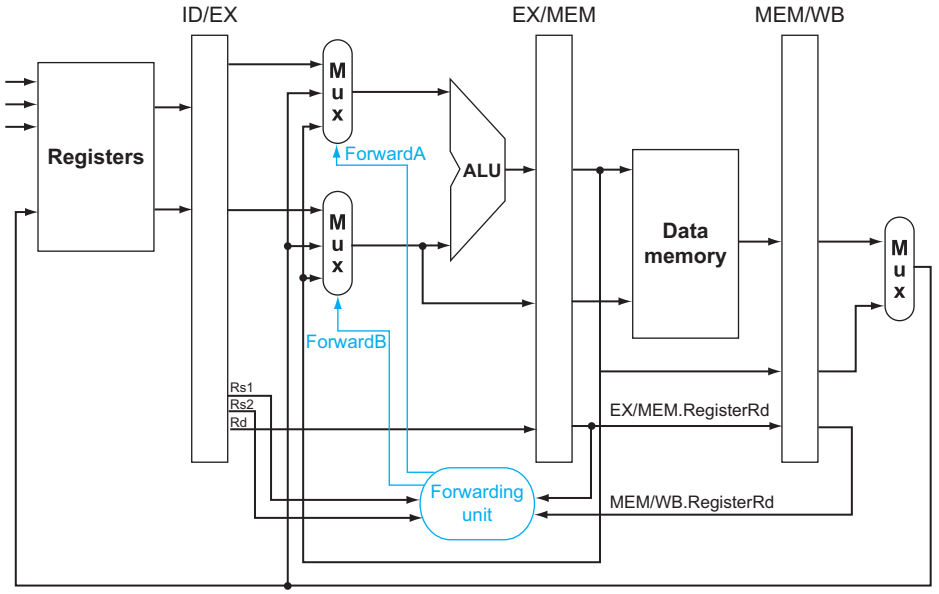


FIGURE 4.51 The dependences between the pipeline registers move forward in time, so it is possible to supply the inputs to the ALU needed by the `and` instruction and `or` instruction by forwarding the results found in the pipeline registers. The values in the pipeline registers show that the desired 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 `x2` having the value 10 at the beginning and –20 at the end of the clock cycle.



a. No forwarding



b. With forwarding

FIGURE 4.52 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.

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 4.53 The control values for the forwarding multiplexors in Figure 4.52. The signed immediate that is another input to the ALU is described in the *Elaboration* at the end of this section.

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. Before forwarding, the ID/EX register had no need to include space to hold the rs1 and rs2 fields. Hence, they were added to ID/EX.

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.RegisterRs1)) ForwardA = 10
```

```
if  (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 10
```

   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.RegisterRs1)) ForwardA = 01
```

```
if  (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs2)) 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 x1, x1, x2
add x1, x1, x3
add x1, x1, x4
. . .
```

In this case, the result should be 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.RegisterRs1))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) 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.RegisterRs2))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 01
```

Figure 4.54 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 or a load.

If you would like to see more illustrated examples using single-cycle pipeline drawings, **Section 4.13** has figures that show two pieces of RISC-V code with hazards that cause forwarding.

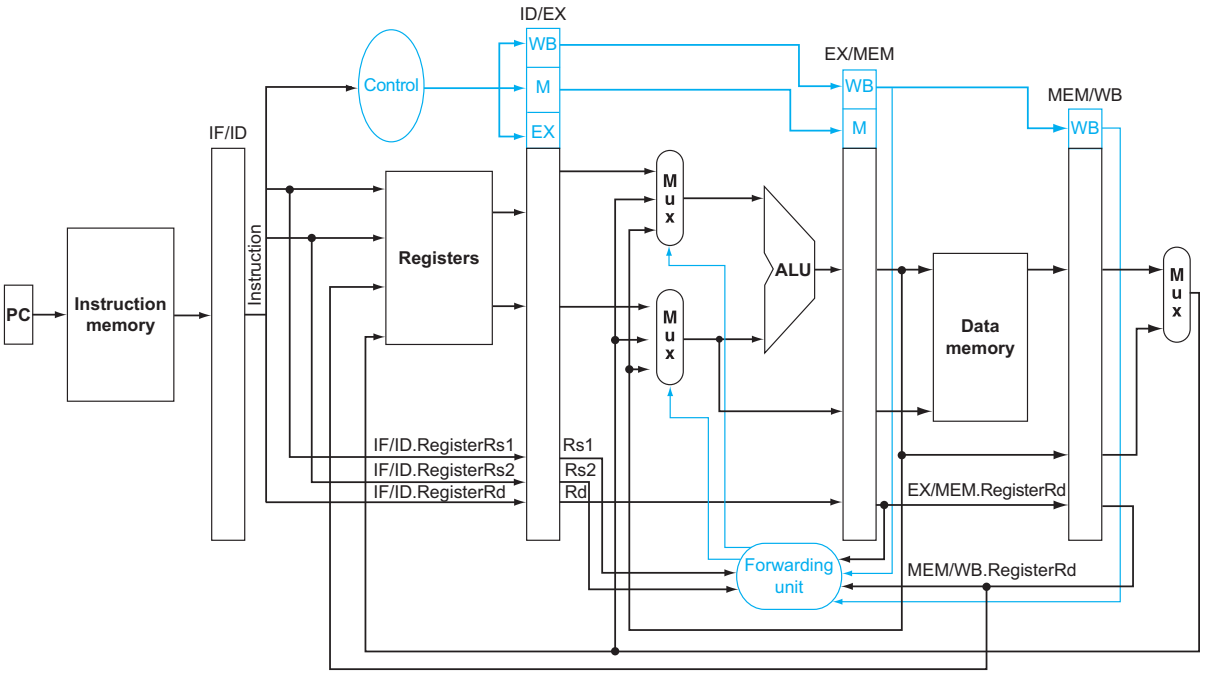


FIGURE 4.54 The datapath modified to resolve hazards via forwarding. Compared with the datapath in Figure 4.49, the additions are the multiplexors to the inputs to the ALU. This figure is a more stylized drawing, however, leaving out details from the full datapath, such as the branch hardware and the sign extension hardware.

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 RISC-V architecture. Since copies are frequent, we need to add more forwarding hardware to make them run faster. If we were to redraw Figure 4.51, replacing the `sub` and `and` instructions with `ld` and `sd`, we would see that it is possible to avoid a stall, since the data exist 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 4.54. Since central control decides between register and immediate, and since the forwarding unit chooses the pipeline register for a register 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 4.55 shows this addition.

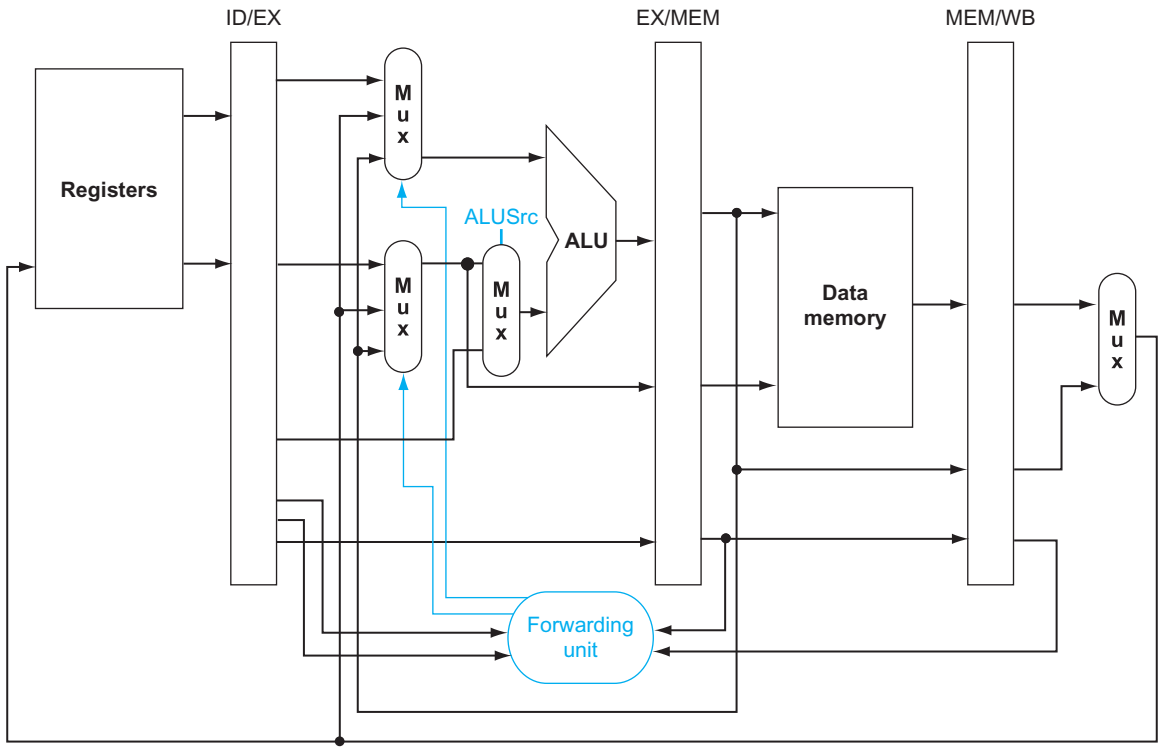


FIGURE 4.55 A close-up of the datapath in Figure 4.52 shows a 2:1 multiplexor, which has been added to select the signed immediate as an ALU input.

Data Hazards and Stalls

If at first you don't succeed, redefine success.

Anonymous

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 4.56 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 the instruction dependent on it. Checking for load instructions, the control for the hazard detection unit is this single condition:

```
if  (ID/EX.MemRead and
    ((ID/EX.RegisterRd = IF/ID.RegisterRs1) or
      (ID/EX.RegisterRd = IF/ID.RegisterRs2)))
      stall the pipeline
```

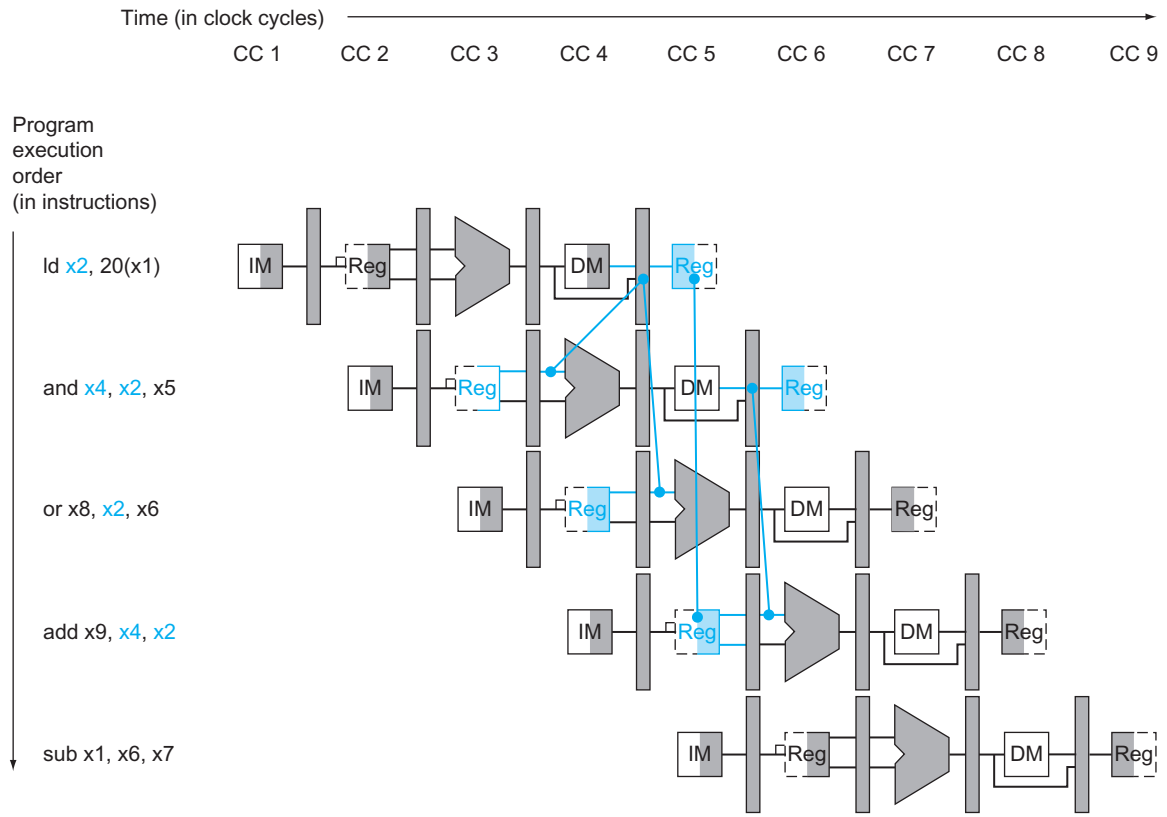


FIGURE 4.56 A pipelined sequence of instructions. Since the dependence between the load and the following instruction (`and`) goes backward in time, this hazard cannot be solved by forwarding. Hence, this combination must result in a stall by the hazard detection unit.

Recall that we are using the RegisterRd to refer the register specified in instruction bits 11:7 for both load and R-type instructions. 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 one-cycle stall, the forwarding logic can handle the dependence and execution proceeds. (If there were no forwarding, then the instructions in Figure 4.56 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**.

nops An instruction that does no operation to change state.

How can we insert these nops, which act like bubbles, into the pipeline? In Figure 4.47, we see that deasserting all seven 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 4.57 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

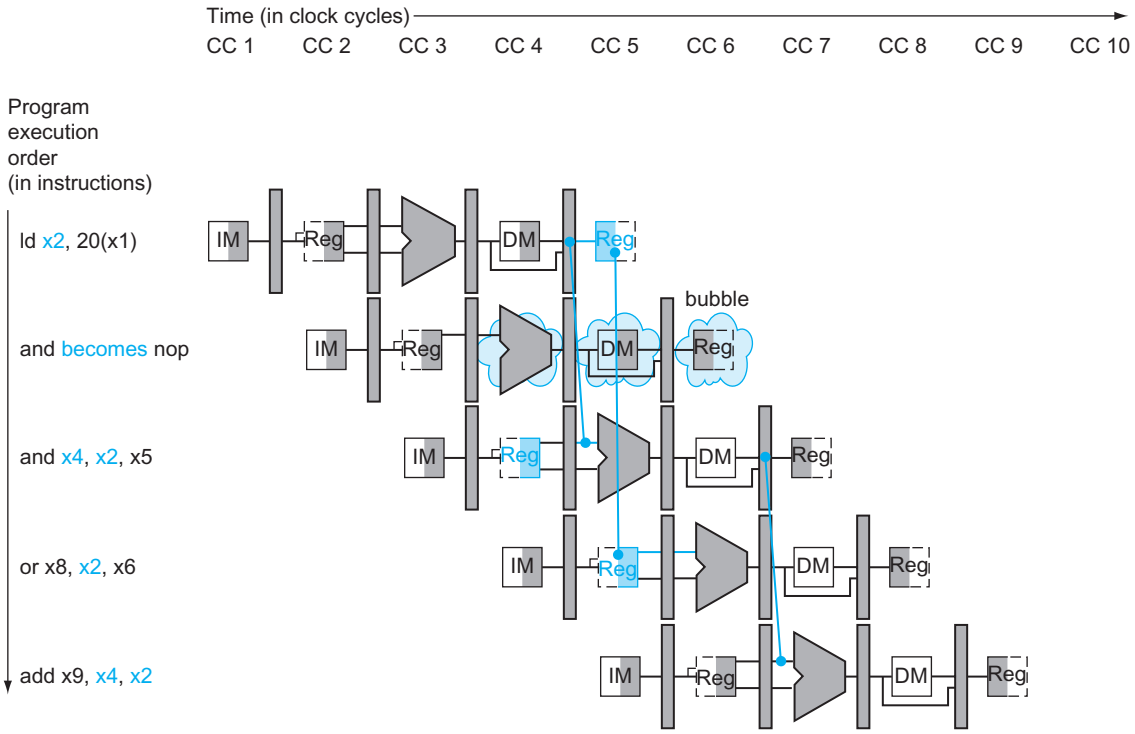


FIGURE 4.57 The way stalls are really inserted into the pipeline. A bubble is inserted beginning in clock cycle 4, by changing the `and` instruction to a nop. Note that the `and` instruction is really fetched and decoded in clock cycles 2 and 3, but its EX stage is delayed until clock cycle 5 (versus 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.

a water pipe, a stall bubble delays everything behind it and proceeds down the instruction pipe one stage each clock 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 4.58 highlights the pipeline connections for both the hazard detection unit and the forwarding unit. As before, the forwarding unit controls the ALU 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. If you would like to see more details, **Section 4.13** gives an example illustrated using single-clock pipeline diagrams of RISC-V code with hazards that cause stalling.

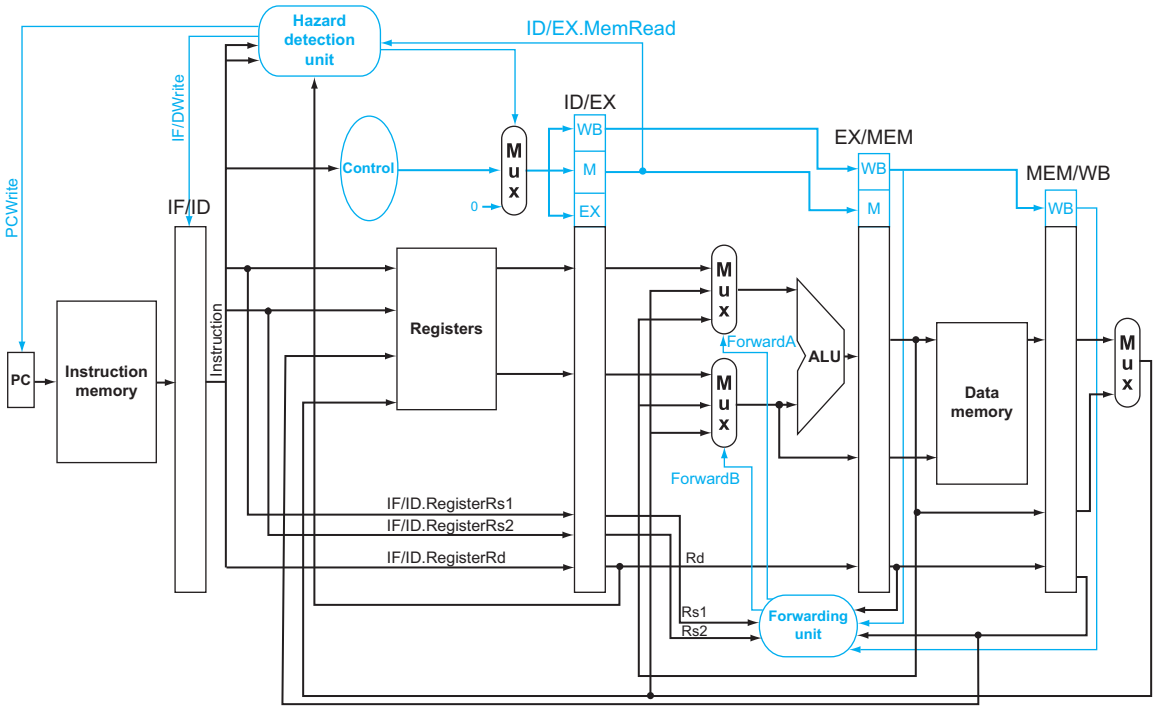


FIGURE 4.58 Pipelined control overview, showing the two multiplexors for forwarding, the hazard detection unit, and the forwarding unit. Although the ID and EX stages have been simplified—the sign-extended immediate and branch logic are missing—this drawing gives the essence of the forwarding hardware requirements.

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

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.

4.8 Control Hazards

There are a thousand hacking at the branches of evil to one who is striking at the root.

Henry David Thoreau, *Walden*, 1854

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 conditional branches. Figure 4.59 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, 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.

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 conditional branch will not be taken and thus continue execution down the sequential instruction stream. If the conditional branch is taken, the instructions that are being fetched and decoded must be discarded. Execution continues at the branch target. If conditional branches are untaken half the time, and if it costs little to discard the instructions, this optimization halves the cost of control hazards.

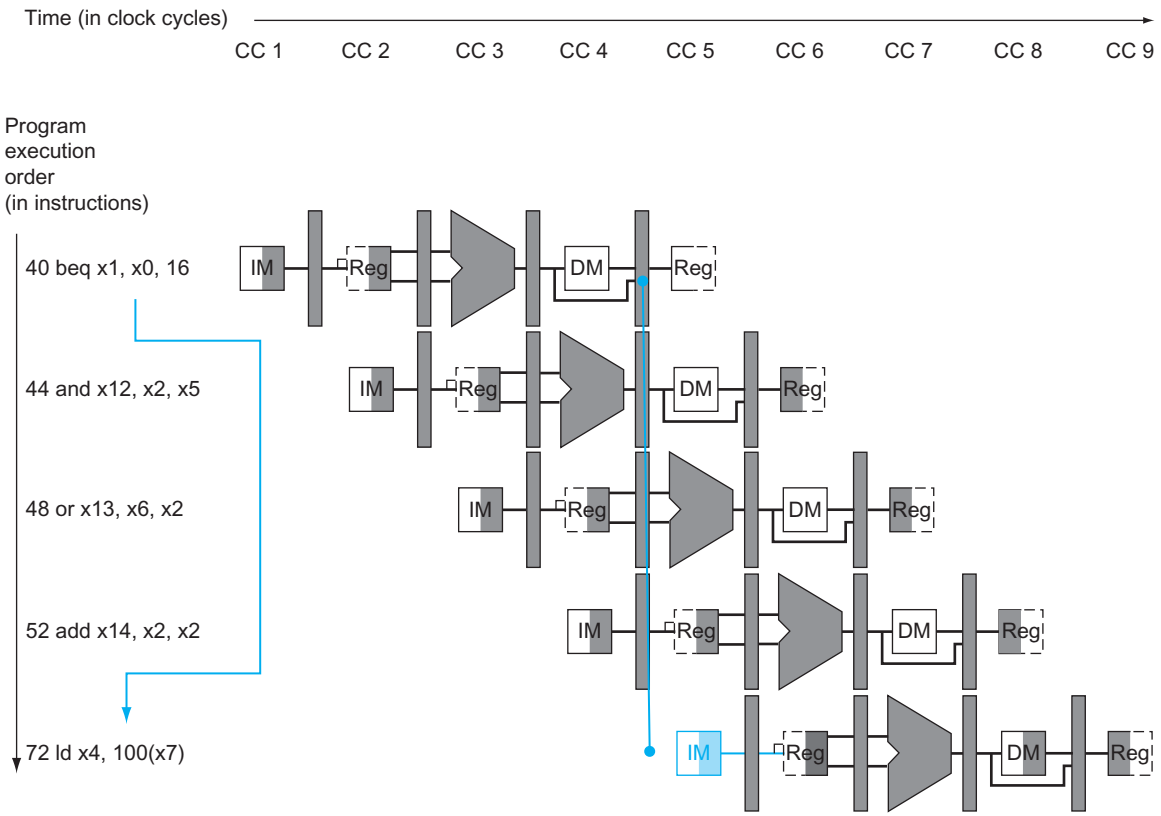


FIGURE 4.59 The impact of the pipeline on the branch instruction. The numbers to the left of the instruction (40, 44, …) are the addresses of the instructions. Since the branch instruction decides whether to branch in the MEM stage—clock cycle 4 for the `beq` instruction above—the three sequential instructions that follow the branch will be fetched and begin execution. Without intervention, those three following instructions will begin execution before `beq` branches to `ld` at location 72. (Figure 4.29 assumed extra hardware to reduce the control hazard to one clock cycle; this figure uses the nonoptimized datapath.)

flush To discard instructions in a pipeline, usually due to an unexpected event.

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 conditional 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 conditional branch execution earlier in the pipeline, then fewer instructions need be flushed. 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 address calculation for branch targets will be performed for all instructions, but only used when needed.

The harder part is the branch decision itself. For branch if equal, we would compare two register reads during the ID stage to see if they are equal. Equality can be tested by XORing individual bit positions of two registers and ORing the XORed result. 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 if 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 test unit is needed, and complete the equality test so that if the instruction is a branch, we can set the PC to the branch target address. Forwarding for the operand 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 EX/MEM or MEM/WB pipeline registers.

2. Because the value in a branch comparison is 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 the operand for the test in the conditional 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 depends 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 conditional 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.

EXAMPLE

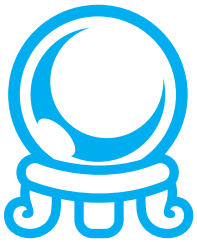
Pipelined Branch

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  x10, x4, x8
40  beq  x1,  x3, 16  // PC-relative branch to 40+16*2=72
44  and  x12, x2, x5
48  or   x13, x2, x6
52  add  x14, x4, x2
56  sub  x15, x6, x7
. . .
72  ld   x4, 50(x7)
```

ANSWER

Figure 4.60 shows what happens when a conditional branch is taken. Unlike Figure 4.59, there is only one pipeline bubble on a taken branch.



PREDICTION

Dynamic Branch Prediction

Assuming a conditional branch is not taken is one simple form of *branch prediction*. In that case, we predict that conditional 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.

dynamic branch prediction Prediction of branches at runtime using runtime information.

branch prediction buffer 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.

One approach is to look up the address of the instruction to see if the conditional 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 prediction uses 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 conditional 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

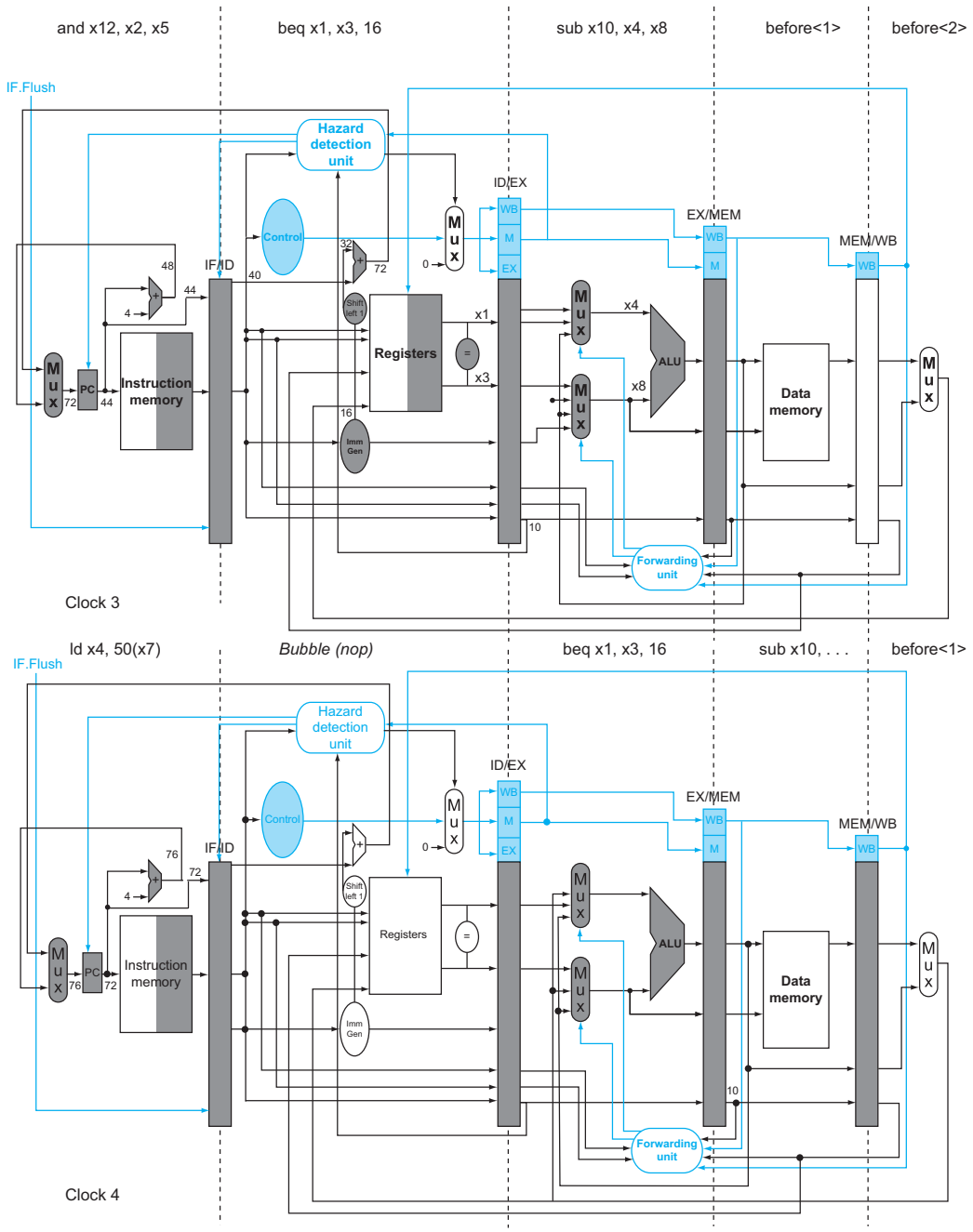


FIGURE 4.60 The ID stage of clock cycle 3 determines that a branch must be taken, so it selects 72 as the next PC address and zeros the instruction fetched for the next clock cycle. Clock cycle 4 shows the instruction at location 72 being fetched and the single bubble or `nop` instruction in the pipeline because of the taken branch.

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 conditional branch is almost always taken, we can predict incorrectly twice, rather than once, when it is not taken. The following example shows this dilemma.

EXAMPLE

Loops and Prediction

Consider a loop branch that branches nine times in a row, and 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?

ANSWER

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 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 4.61 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 308, 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 4.61.

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.

Elaboration: A branch predictor tells us whether a conditional 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 one-cycle penalty. One 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

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.

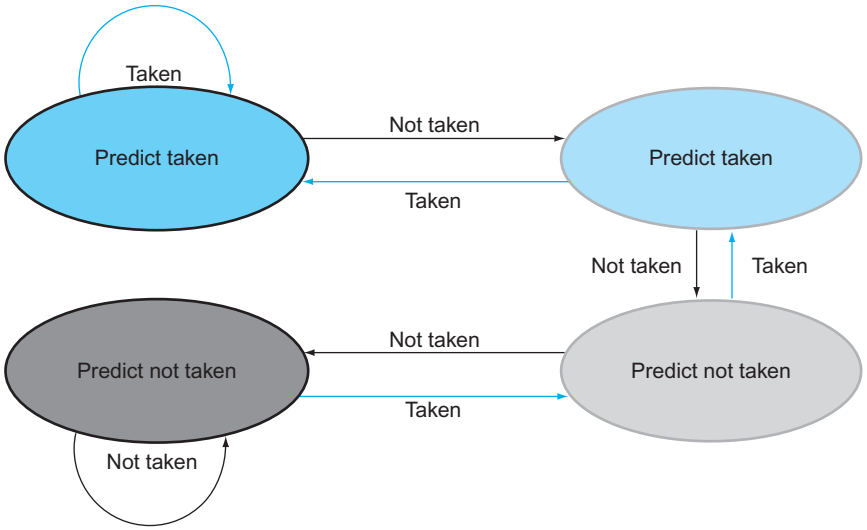


FIGURE 4.61 The states in a 2-bit prediction scheme. By using 2 bits rather than 1, a branch that strongly favors taken or not taken—as many branches do—will be mispredicted only once. The 2 bits are used to encode the four states in the system. The 2-bit scheme is a general instance of a counter-based 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.

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.

Another approach to branch prediction is the use of tournament predictors. A **tournament branch 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 ensemble predictors.

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: 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. For example, the ARMv8 instruction set architecture has a conditional select instruction called `CSEL`. It specifies a destination register, two source registers, and a condition. The destination register gets a value of the first operand if the condition is true and the second operand otherwise. Thus, `CSEL X8, X11, X4, NE` copies the contents of register 11 into register 8 if the condition codes say the result of the operation was not equal zero or a copy of register 4 into register 11 if it was zero. Hence, programs using the ARMv8 instruction set could have fewer conditional branches than programs written in RISC-V.

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 mispredicted branches or load-use data hazards. Figure 4.62 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 accuracy of the dynamic predictor is 90%. Which predictor is the best choice for the following branches?

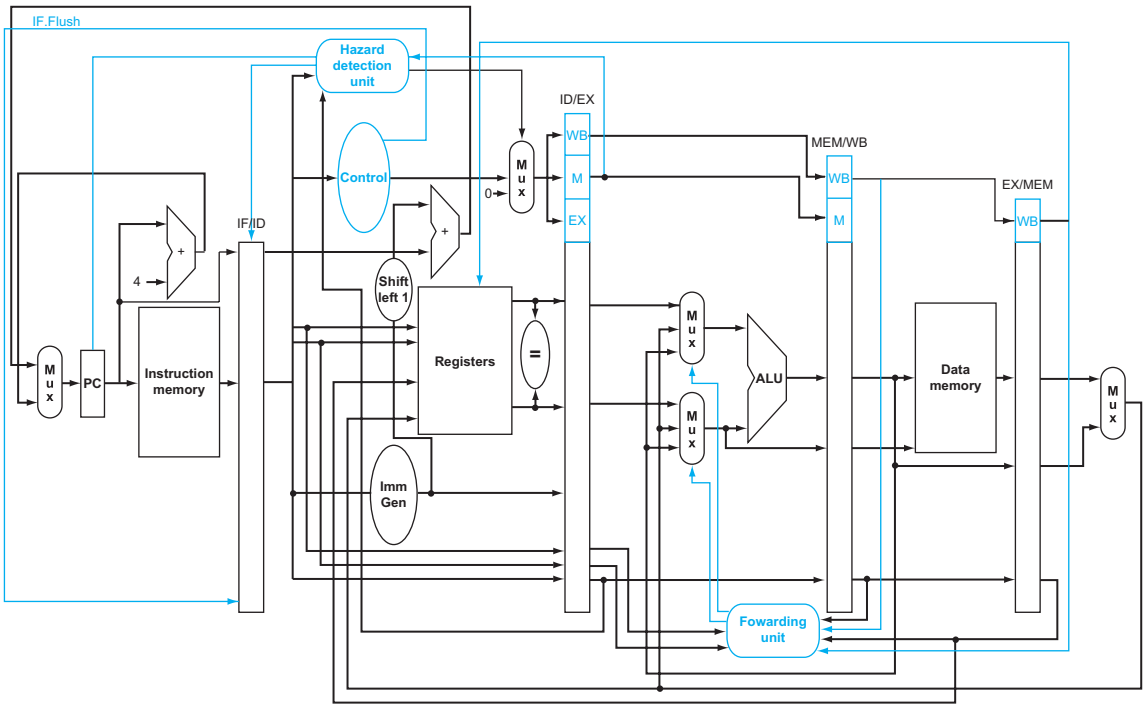


FIGURE 4.62 The final datapath and control for this chapter. Note that this is a stylized figure rather than a detailed datapath, so it's missing the ALUsrc Mux from Figure 4.55 and the multiplexor controls from Figure 4.49.

1. A conditional branch that is taken with 5% frequency
2. A conditional branch that is taken with 95% frequency
3. A conditional branch that is taken with 70% frequency

4.9 Exceptions

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

Control is the most challenging aspect of processor design: it is both the hardest part to get right and the toughest part to make fast. One of the demanding tasks of control is implementing **exceptions** and **interrupts**—events other than branches that change the normal flow of instruction execution. They were initially created to handle unexpected events from within the processor, like an undefined instruction. 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 either name to refer to both types of events. For example, the Intel x86 uses interrupt. We use 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 examples showing whether the situation is internally generated by the processor or externally generated and the name that RISC-V uses:

exception Also called **interrupt**. An unscheduled event that disrupts program execution; used to detect undefined instructions.

interrupt An exception that comes from outside of the processor. (Some architectures use the term *interrupt* for all exceptions.)

Type of event	From where?	RISC-V terminology
System reset	External	Exception
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Either

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 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 an intricate implementation can significantly reduce performance, as well as complicate the task of getting the design correct.

How Exceptions are Handled in the RISC-V Architecture

The only types of exceptions that our current implementation can generate are execution of an undefined instruction or a hardware malfunction. We'll assume a hardware malfunction occurs during the instruction `add x11, x12, x11` 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 unfortunate instruction in the *supervisor exception cause register* (SEPC) 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 a malfunction, 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 SEPC 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 RISC-V architecture is to include a register (called the *Supervisor Exception Cause Register* or *SCAUSE*), which holds a field that indicates the reason for the exception.

vectored interrupt An interrupt for which the address to which control is transferred is determined by the cause of 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, possibly added to a base register that points to memory range for vectored interrupts. For example, we might define the following exception vector addresses to accommodate these exception types:

Exception type	Exception vector address to be added to a Vector Table Base Register
Undefined instruction	$00\ 0100\ 0000_{two}$
System Error (hardware malfunction)	$01\ 1000\ 0000_{two}$

The operating system knows the reason for the exception by the address at which it is initiated. When the exception is not vectored, as in RISC-V, a single entry point for all exceptions can be used, and the operating system decodes the status register to find the cause. For architectures with vectored exceptions, the addresses might be separated by, say, 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.

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 with the single interrupt entry point being the address $0000\ 0000\ 1C09\ 0000_{hex}$. (Implementing

vectored exceptions is no more difficult.) We will need to add two additional registers to our current RISC-V implementation:

- *SEPC:* A 64-bit register used to hold the address of the affected instruction. (Such a register is needed even when exceptions are vectored.)
- *SCAUSE:* A register used to record the cause of the exception. In the RISC-V architecture, this register is 64 bits, although most bits are currently unused. Assume there is a field that encodes the two possible exception sources mentioned above, with 2 representing an undefined instruction and 12 representing hardware malfunction.

Exceptions in a Pipelined Implementation

A pipelined implementation treats exceptions as another form of control hazard. For example, suppose there is a hardware malfunction 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 misprediction, 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 0000 0000 1C09 0000_{hex}, which we are using as the RISC-V exception address, we simply add an additional input to the PC multiplexor that sends 0000 0000 1C09 0000_{hex} to the PC. Figure 4.63 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 `x1` because it will be clobbered as the destination register of the `add` instruction. If we assume the exception is detected during the EX stage, 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 *supervisor exception program counter* (SEPC). Figure 4.63 shows a stylized version of the datapath, including the branch hardware and necessary accommodations to handle exceptions.

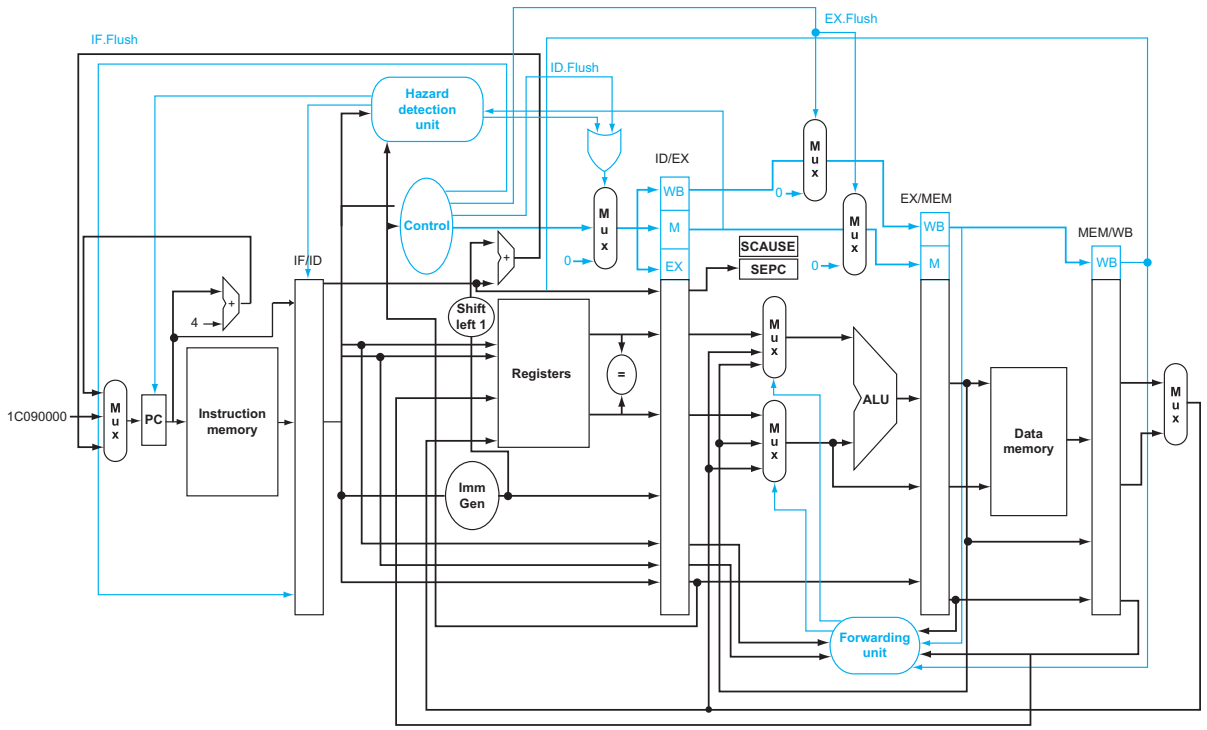


FIGURE 4.63 The datapath with controls to handle exceptions. The key additions include a new input with the value 0000 0000 1C09 0000_{hex} in the multiplexor that supplies the new PC value; an SCAUSE register to record the cause of the exception; and an SEPC register to save the address of the instruction that caused the exception. The 0000 0000 1C09 0000_{hex} input to the multiplexor is the initial address to begin fetching instructions in the event of an exception.

EXAMPLE

Exception in a Pipelined Computer

Given this instruction sequence,

```
40hex   sub    x11, x2, x4
44hex   and    x12, x2, x5
48hex   or     x13, x2, x6
4Chex   add    x1,  x2, x1
50hex   sub    x15, x6, x7
54hex   ld     x16, 100(x7)
. . .
```

assume the instructions to be invoked on an exception begin like this:

```
1C090000hex   sd    x26, 1000(x10)
1C090004hex   sd    x27, 1008(x10)
. . .
```

Show what happens in the pipeline if a hardware malfunction exception occurs in the `add` instruction.

ANSWER

Figure 4.64 shows the events, starting with the add instruction in the EX stage. Assume the hardware malfunction is detected during that phase, and 0000 0000 1C09 0000_{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-handling code is fetched. Note that the address of the `add` instruction is saved: $4C_{hex}$.

We mentioned several examples of exceptions on page 315, 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 RISC-V 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 SEPC register captures the address of the interrupted instructions, and the SCAUSE register records the highest priority exception in a clock cycle if more than one exception occurs.

Hardware/ Software Interface

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 branch to a prearranged address. The operating system contract is to look at the cause of the exception and act appropriately. For an undefined instruction or hardware failure, 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; Chapter 5 describes these exceptions and their handling in more detail.

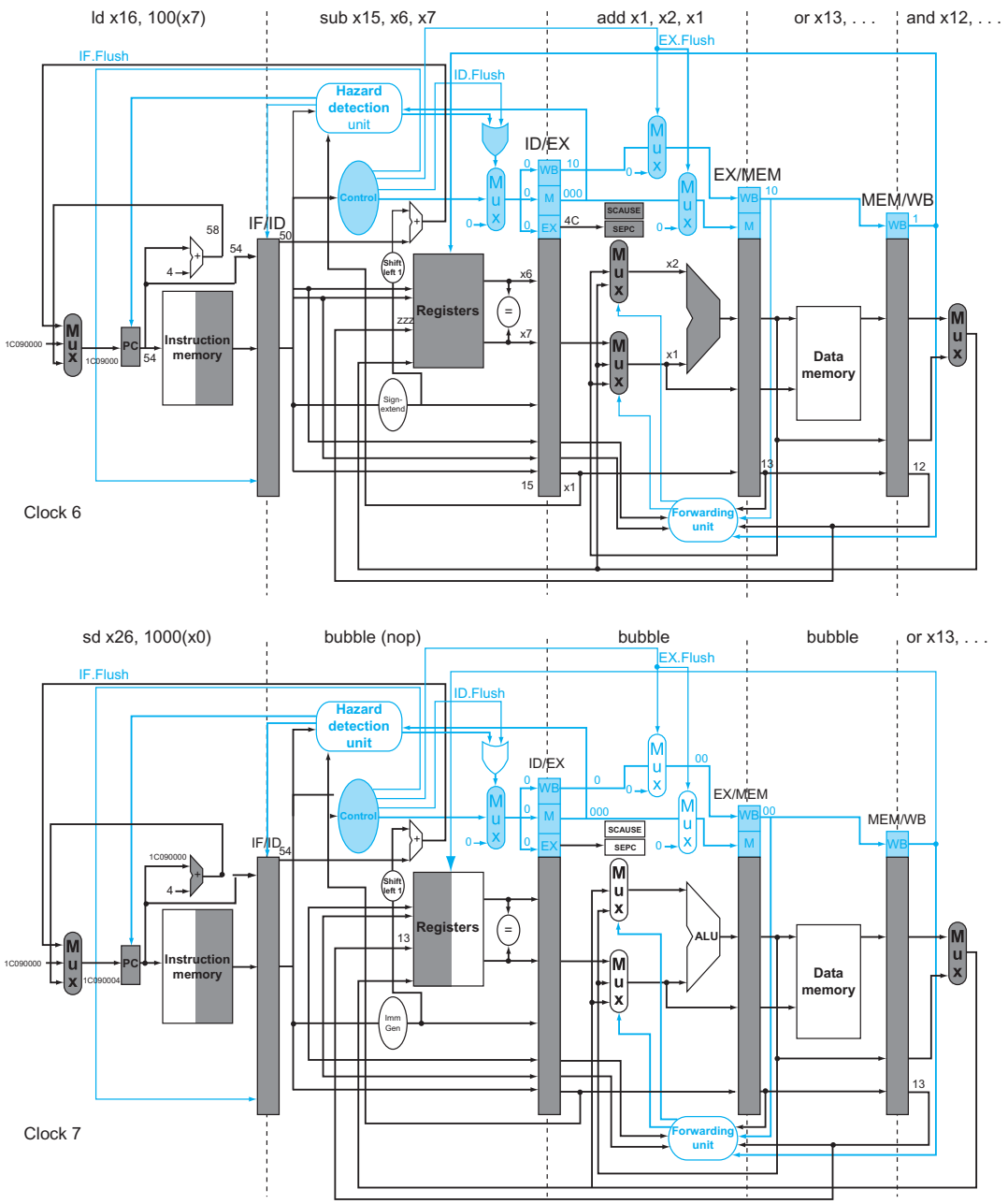


FIGURE 4.64 The result of an exception due to hardware malfunction in the add instruction. The exception is detected during the EX stage of clock 6, saving the address of the add instruction in the SEPC register ($4C_{hex}$). It causes all the Flush signals to be set near the end of this clock cycle, 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—`sd x26, 1000(x0)`—from instruction location $0000\ 0000\ 1C09\ 0000_{hex}$. Note that the `and` and `or` instructions, which are prior to the `add`, still complete.

Elaboration: The difficulty of always associating the proper 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 is at address $4C_{hex}$. A processor with imprecise exceptions might put 58_{hex} into SEPC and leave it up to the operating system to determine which instruction caused the problem. RISC-V and the vast majority of computers today support **precise interrupts** or **precise exceptions**. One reason is designers of a deeper pipeline processor might be tempted to record a different value in SEPC, which would create headaches for the OS. To prevent them, the deeper pipeline would likely be required to record the same PC that would have been recorded in the five-stage pipeline. It is simpler for everyone to just record the PC of the faulting instruction instead. (Another reason is to support virtual memory, which we shall see in Chapter 5.)

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.

precise interrupt Also called **precise exception**. An interrupt or exception that is always associated with the correct instruction in pipelined computers.

Elaboration: We show that RISC-V uses the exception entry address $0000\ 0000\ 1C09\ 0000_{hex}$, which is chosen somewhat arbitrarily. Many RISC-V computers store the exception entry address in a special register named *Supervisor Trap Vector* (STVEC), which the OS can load with a value of its choosing.

Check Yourself

Which exception should be recognized first in this sequence?

1. `xxx x11, x12, x11  // undefined instruction`
2. `sub x11, x12, x11  // hardware error`

4.10 Parallelism via Instructions



Be forewarned: this section is a brief overview of fascinating but complex 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, naturally enough, **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

instruction-level parallelism The parallelism among instructions.

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.

multiple issue A scheme whereby multiple instructions are launched in one clock cycle.

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 3-GHz four-way multiple-issue microprocessor can execute a peak rate of 12 billion instructions per second and have a best-case CPI of 0.33, or an IPC of 3. Assuming a five-stage pipeline, such a processor would have up to 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.

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.

There are two main 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.

Two primary and distinct responsibilities must be dealt with in a multiple-issue pipeline:

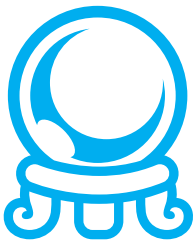
issue slots The positions from which instructions could issue in a given clock cycle; by analogy, these correspond to positions at the starting blocks for a sprint.

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 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



PREDICTION

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, 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. 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 An approach whereby the compiler or processor guesses the outcome of an instruction to remove it as a dependence in executing other instructions.

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 wrong. 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. Misspeculation typically requires the pipeline to be flushed, or at least stalled, and thus further reduces performance.

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 within bounds when the speculation is incorrect. The result would be that an exception that should not have occurred would occur. 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.

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)**.

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.

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.

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 an RISC-V processor, before we describe the use of these techniques in more aggressive processors.

An Example: Static Multiple Issue with the RISC-V ISA

To give a flavor of static multiple issue, we consider a simple two-issue RISC-V 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 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 4.65 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 packet. Even so, a hazard generally forces the entire issue packet containing the dependent 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

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	WB
Load or store instruction				IF	ID	EX	MEM	WB

FIGURE 4.65 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.

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 4.66). 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, 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.

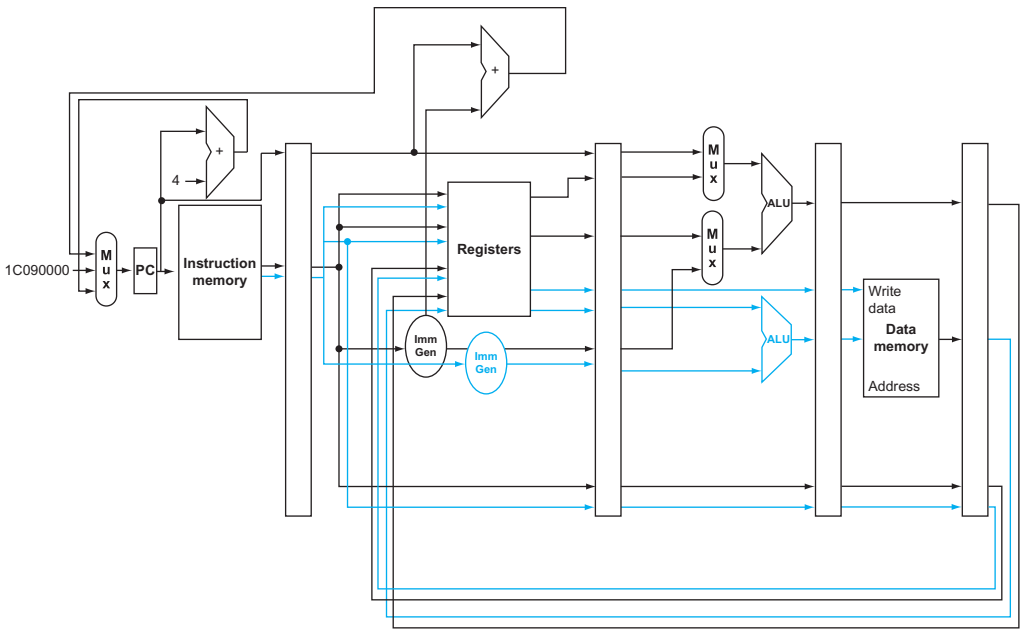


FIGURE 4.66 A static two-issue datapath. The additions needed for double issue are highlighted: another 32 bits from instruction memory, two more read ports and one more write port on the register file, and another ALU. Assume the bottom ALU handles address calculations for data transfers and the top ALU handles everything else.

EXAMPLE

Simple Multiple-Issue Code Scheduling

How would this loop be scheduled on a static two-issue pipeline for RISC-V?

```
Loop: ld   x31, 0(x20)     // x31=array element
      add  x31, x31, x21   // add scalar in x21
      sd   x31, 0(x20)     // store result
      addi x20, x20, -8    // decrement pointer
      blt  x22, x20, Loop  // compare to loop limit,
                           // branch if x20 > x22
```

Reorder the instructions to avoid as many pipeline stalls as possible. Assume branches are predicted, so that control hazards are handled by the hardware.

ANSWER

The first three instructions have data dependences, as do the next two. Figure 4.67 shows the best schedule for these instructions. Notice that just one pair of instructions has both issue slots used. It takes five 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!

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:		ld x31, 0(x20)	1
	addi x20, x20, -8		2
	add x31, x31, x21		3
	blt x22, x20, Loop	sd x31, 8(x20)	4

FIGURE 4.67 The scheduled code as it would look on a two-issue RISC-V pipeline. The empty slots are no-ops. Note that since we moved the `addi` before the `sd`, we had to adjust `sd`'s offset by 8.

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

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

EXAMPLE

See how well loop unrolling and scheduling work in the example above. For simplicity, assume that the loop index is a multiple of four.

ANSWER

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 `ld`, `add`, and `sd`, plus one `addi`, and one `blt`. Figure 4.68 shows the unrolled and scheduled code.

During the unrolling process, the compiler introduced additional registers (`x28`, `x29`, `x30`). 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 `x31`. There would be repeated instances of `ld x31, 0(x20)`, `add x31, x31, x21` followed by `sd x31, 8(x20)`, but these sequences, despite using `x31`, 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.

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.

Renaming the registers during the unrolling process allows the compiler to move these independent instructions subsequently to better schedule the code. The renaming process eliminates the name dependences, while preserving the true dependences.

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:	addi x20, x20, -32	ld x28, 0(x20)	1
		ld x29, 24(x20)	2
	add x28, x28, x21	ld x30, 16(x20)	3
	add x29, x29, x21	ld x31, 8(x20)	4
	add x30, x30, x21	sd x28, 32(x20)	5
	add x31, x31, x21	sd x29, 24(x20)	6
		sd x30, 16(x20)	7
	blt x22, x20, Loop	sd x31, 8(x20)	8

FIGURE 4.68 The unrolled and scheduled code of Figure 4.67 as it would look on a static two-issue RISC-V pipeline. The empty slots are no-ops. Since the first instruction in the loop decrements x20 by 32, the addresses loaded are the original value of x20, then that address minus 8, minus 16, and minus 24.

Notice now that 12 of the 14 instructions in the loop execute as pairs. It takes eight clocks for four loop iterations, which yields an IPC of 14/8 = 1.75. Loop unrolling and scheduling more than doubled performance—8 versus 20 clock cycles for 4 iterations—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 more than doubling the code size.

Dynamic Multiple-Issue Processors

superscalar An advanced pipelining technique that enables the processor to execute more than one instruction per clock cycle by selecting them during execution.

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.

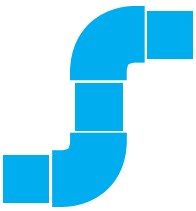
dynamic pipeline scheduling Hardware support for reordering the order of instruction execution to avoid stalls.

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:

```
ld    x31, 0(x21)
add   x1,  x31, x2
sub   x23, x23, x3
andi  x5,  x23, 20
```

Even though the `sub` instruction is ready to execute, it must wait for the `ld` and `add` 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.



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 (a dozen or more in high-end designs in 2015), and a **commit unit**. Figure 4.69 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. (In the next section, we will discuss 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.

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.

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 327. 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.

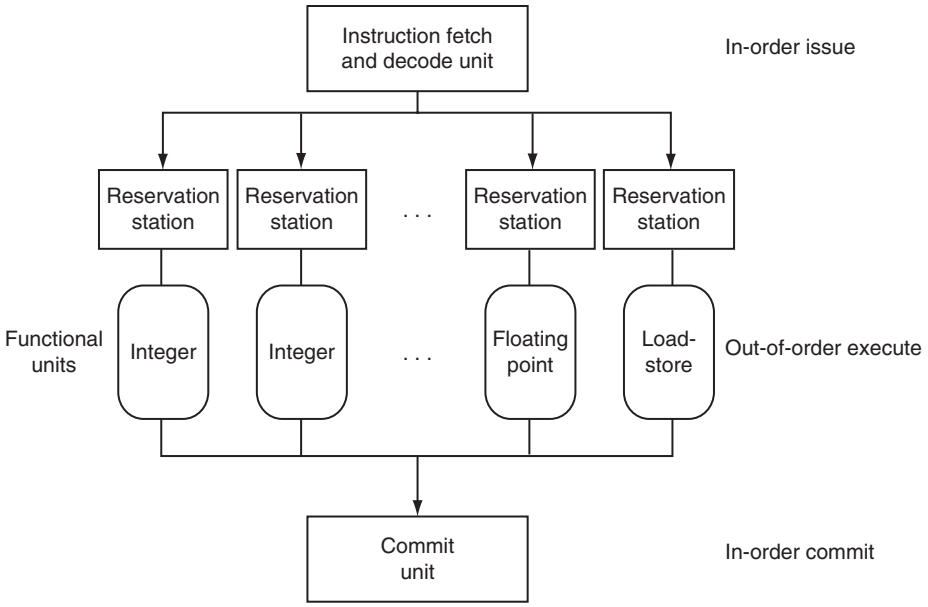


FIGURE 4.69 The three primary units of a dynamically scheduled pipeline. The final step of updating the state is also called retirement or graduation.

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.

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.

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 are 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 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.

Understanding Program Performance

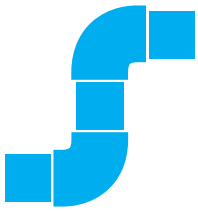


Given that compilers can also schedule code around data dependences, you might ask why a superscalar processor would use dynamic scheduling. There are three major reasons. First, not all stalls are predictable. In particular, cache misses (see Chapter 5) in the **memory hierarchy** cause unpredictable 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.

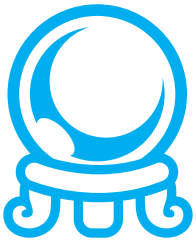
The BIG Picture



PIPELINING



PARALLELISM



PREDICTION

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.

Hardware/ Software Interface

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.



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 collided with 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 the number of transistors.

Figure 4.70 shows the number of pipeline stages, the issue width, speculation level, clock rate, cores per chip, and power of several past and recent Intel microprocessors. Note the drop in pipeline stages and power as companies switch to multicore designs.

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	2–4	87	W
Intel Core i5 Ivy Bridge	2012	3400 MHz	14	4	Yes	8	77	W

FIGURE 4.70 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: 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, which, as mentioned above, is 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.

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 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-A53 and Intel Core i7 Pipelines

Figure 4.71 describes the two microprocessors we examine in this section, whose targets are the two endpoints of the post-PC era.

The ARM Cortex-A53

The ARM Corxtex-A53 runs at 1.5 GHz with an eight-stage pipeline and executes the ARMv8 instruction set. 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 4.72 shows the overall pipeline.

Processor	ARM A53	Intel Core i7 920
Market	Personal Mobile Device	Server, Cloud
Thermal design power	100 milliWatts (1 core @ 1 GHz)	130 Watts
Clock rate	1.5 GHz	2.66 GHz
Cores/Chip	4 (configurable)	4
Floating point?	Yes	Yes
Multiple Issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline Stages	8	14
Pipeline schedule	Static In-order	Dynamic Out-of-order with Speculation
Branch prediction	Hybrid	2-level
1st level caches/core	16-64 KiB I, 16-64 KiB D	32 KiB I, 32 KiB D
2nd level cache/core	128–2048 KiB (shared)	256 KiB (per core)
3rd level cache (shared)	(platform dependent)	2–8 MiB

FIGURE 4.71 Specification of the ARM Cortex-A53 and the Intel Core i7 920.

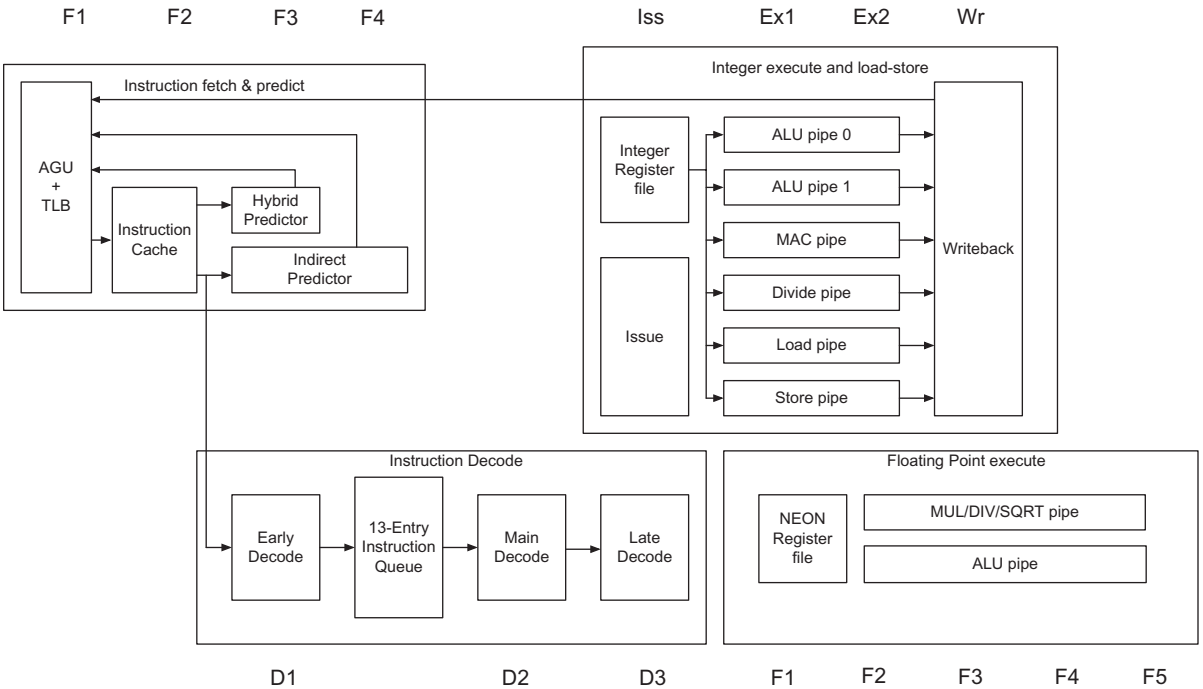


FIGURE 4.72 The Cortex-A53 pipeline. The first three stages fetch instructions into a 13-entry instruction queue. The *Address Generation Unit* (AGU) uses a *Hybrid Predictor, Indirect Predictor*, and a *Return Stack* to predict branches to try to keep the instruction queue full. Instruction decode is three stages and instruction execution is three stages. With two additional stages for floating point and SIMD operations.

The first three stages fetch two instructions at a time and try to keep a 13-entry instruction queue full. It uses a 6k-bit hybrid conditional branch predictor, a 256-entry indirect branch predictor, and an 8-entry return address stack to predict future function returns. The prediction of indirect branches takes an additional pipeline stage. This design choice will incur extra latency if the instruction queue cannot decouple the decode and execute stages from the fetch stage, primarily in the case of a branch misprediction or an instruction cache miss. When the branch prediction is wrong, it empties the pipeline, resulting in an eight-clock cycle misprediction penalty.

The decode stages of the 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 instruction execution section primarily occupies three pipeline stages and provides one pipeline for load instructions, one pipeline for store instructions, two pipelines for integer arithmetic operations, and separate pipelines for integer multiply and divide operations. Either instruction from the pair can be issued to the load or store pipelines. The execution stages have full forwarding between the pipelines.

Floating-point and SIMD operations add a two more pipeline stages to the instruction execution section and feature one pipeline for multiply/divide/square root operations and one pipeline for other arithmetic operations.

Figure 4.73 shows the CPI of the Cortex-A53 using the SPEC2006 benchmarks. While the ideal CPI is 0.5, the best case achieved is 1.0, the median case is 1.3, and

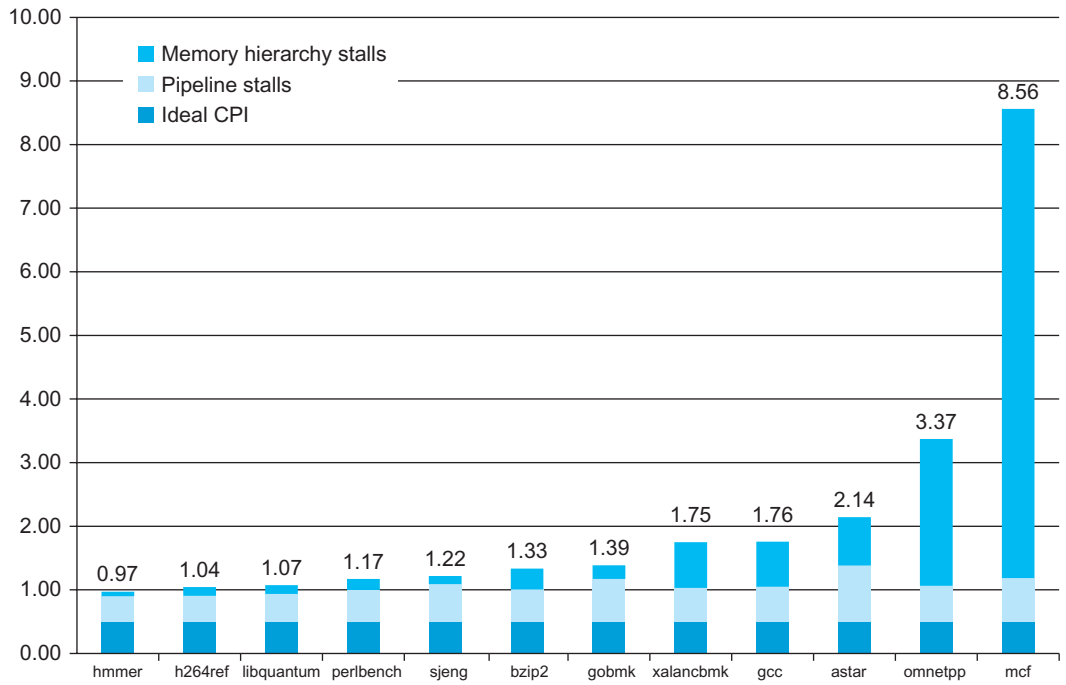


FIGURE 4.73 CPI on ARM Cortex-A53 for the SPEC2006 integer benchmarks.

the worst case is 8.6. For the median case, 60% of the stalls are due to the pipelining hazards and 40% are stalls due to the memory hierarchy. Pipeline stalls are caused by branch mispredictions, structural hazards, and data dependencies between pairs of instructions. Given the static pipeline of the Cortex-A53, it is up to the compiler to try to avoid structural hazards and data dependences.

Elaboration: The Cortex-A53 is a configurable core that supports the ARMv8 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 logically, the resultant chips have many differences. One parameter is the size of the L2 cache, which can vary by a factor of 16.

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 their pipelines. 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 RISC-V-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.

When we consider the design of such 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.

microarchitecture The organization of the processor, including the major functional units, their interconnection, and control.

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

architectural registers The instruction set of visible registers of a processor; for example, in RISC-V, these are the 32 integer and 32 floating-point registers.

speculated instruction. This undo 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 4.74 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 is nontrivial since the length of an x86 instruction can be from 1 to 15 bytes

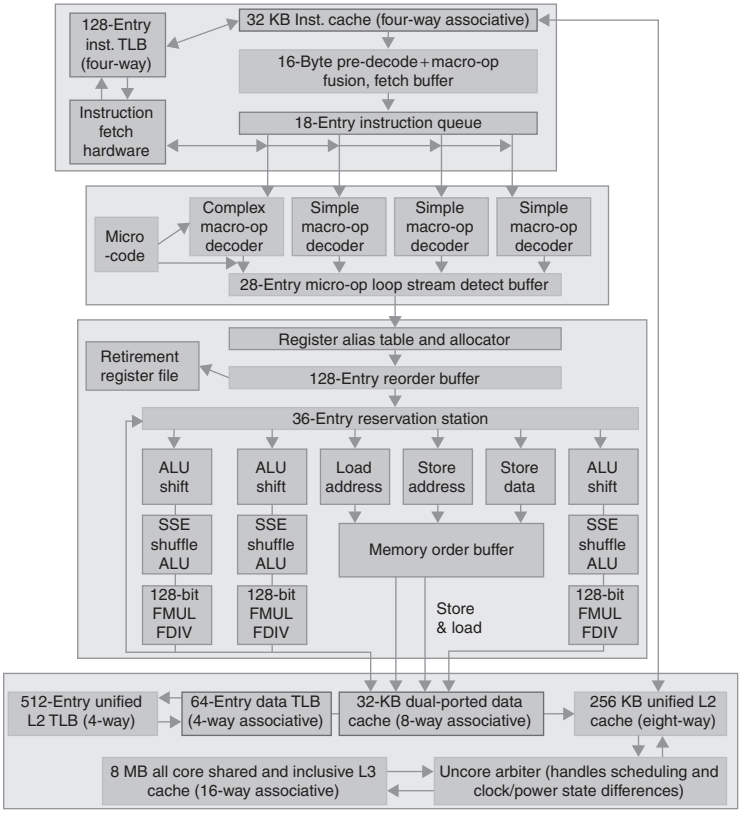


FIGURE 4.74 The Core i7 pipeline with memory components. The total pipeline depth is 14 stages, with branch mispredictions costing 17 clock cycles. This design can buffer 48 loads and 32 stores. The six independent units can begin execution of a ready micro-operation each clock cycle.

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 4.75 shows the CPI of the Intel Core i7 for each of the SPEC2006 benchmarks. While the ideal CPI is 0.25, the best case achieved is 0.44, the median case is 0.79, and the worst case is 2.67.

Although 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 4.76 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 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.

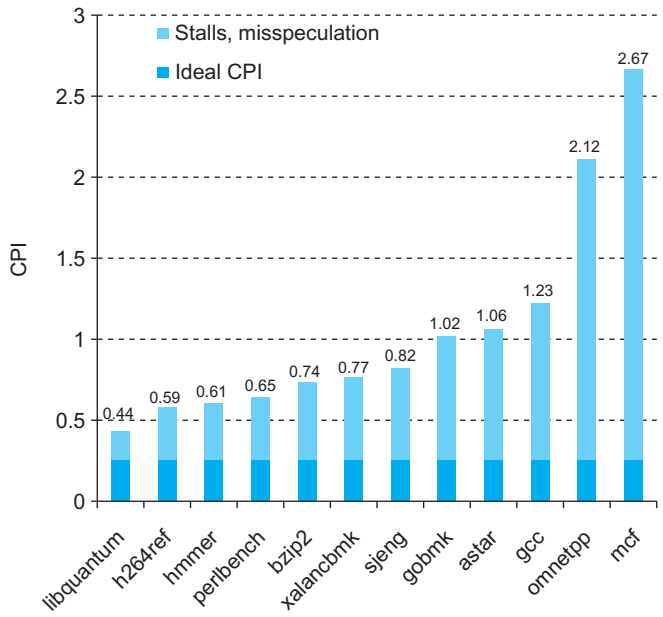


FIGURE 4.75 CPI of Intel Core i7 920 running SPEC2006 integer benchmarks.

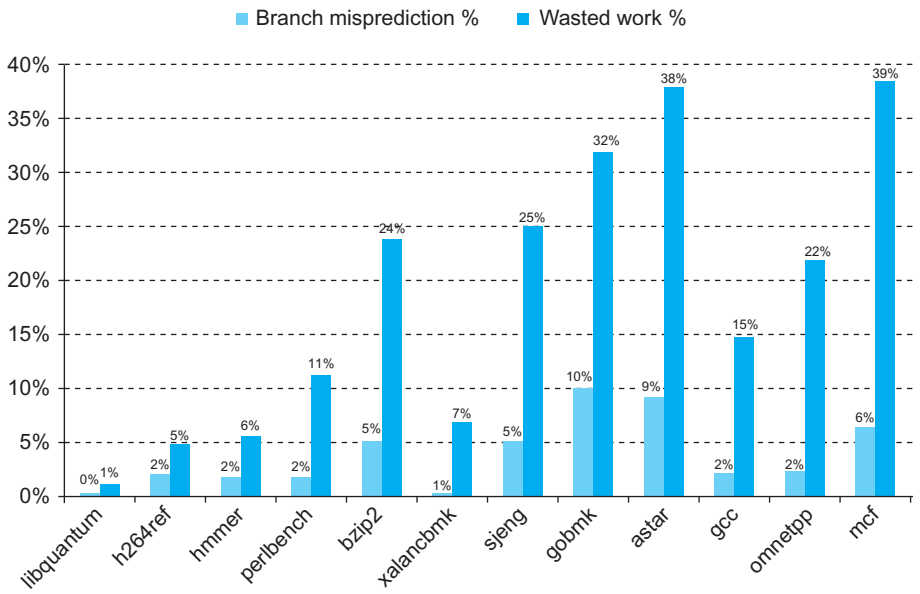


FIGURE 4.76 Percentage of branch mispredictions and wasted work due to unfruitful speculation of Intel Core i7 920 running SPEC2006 integer benchmarks.

Understanding Program Performance

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 possible 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



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 4.77 shows the unrolled version of Figure 3.22, which contains the C intrinsics to produce the AVX instructions.

Like the unrolling example in Figure 4.68 above, we are going to unroll the loop four times. Rather than manually unrolling the loop in C by making four copies of each of the intrinsics in Figure 3.22, we can rely on the gcc compiler to do the unrolling at −O3 optimization. (We use the constant UNROLL in the C code to control the amount of unrolling in case we want to try other values.) We surround each intrinsic with a simple *for* loop with four iterations (lines 9, 15, and 20) and replace the scalar `C0` in Figure 3.22 with a four-element array `c[]` (lines 8, 10, 16, and 21).

```
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 4.77 Optimized C version of DGEMM using C intrinsics to generate the AVX subword-parallel instructions for the x86 (Figure 3.22) and loop unrolling to create more opportunities for instruction-level parallelism. Figure 4.78 shows the assembly language produced by the compiler for the inner loop, which unrolls the three for-loop bodies to expose instruction-level parallelism.

Figure 4.78 shows the assembly language output of the unrolled code. As expected, in Figure 4.78 there are four versions of each of the AVX instructions in Figure 3.23, with one exception. We only need one copy of the `vbroadcastsd` instruction, since we can use the four copies of the B element in register `%ymm0` repeatedly throughout the loop. Thus, the five AVX instructions in Figure 3.23 become 17 in Figure 4.78, and the seven integer instructions appear in both, although the constants and addressing changes to account for the unrolling. Hence, despite unrolling four times, the number of instructions in the body of the loop only doubles: from 12 to 24.

```
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
10 vaddpd %ymm5,%ymm4,%ymm4      // Parallel add %ymm5, %ymm4
11 vmulpd 0x20(%rax),%ymm0,%ymm5 // Parallel mul %ymm1,4 A
12 vaddpd %ymm5,%ymm3,%ymm3      // Parallel add %ymm5, %ymm3
13 vmulpd 0x40(%rax),%ymm0,%ymm5 // Parallel mul %ymm1,4 A
14 vmulpd 0x60(%rax),%ymm0,%ymm0 // Parallel mul %ymm1,4 A
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>        // branch if %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 4.78 The x86 assembly language for the body of the nested loops generated by compiling the unrolled C code in Figure 4.77.

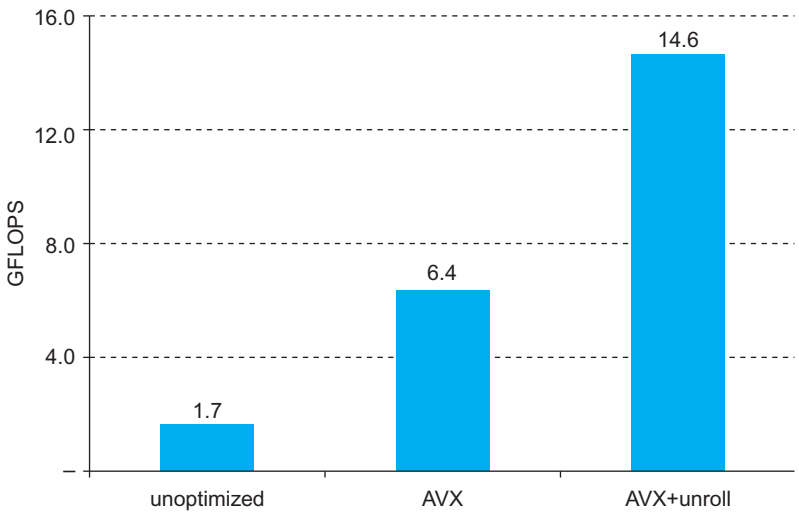


FIGURE 4.79 Performance of three versions of DGEMM for 32 × 32 matrices. Subword parallelism and instruction-level parallelism have led to speedup of almost a factor of 9 over the unoptimized code in Figure 3.21.



Figure 4.79 shows the performance increase DGEMM for 32 × 32 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 8.59 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.

Elaboration: There are no pipeline stalls despite the reuse of register `%ymm5` in lines 9 to 17 of Figure 4.78 because the Intel Core i7 pipeline renames the registers.

Check Yourself

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 Cortex-A53 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 4.70).



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 processor control both behaviorally and in a form suitable for hardware synthesis. It then provides a series of behavioral models in Verilog of the 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 RISC-V instructions.

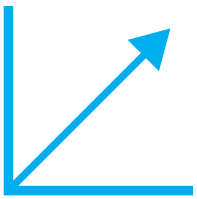
4.14 Fallacies and Pitfalls

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 hundreds of 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 274) 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 and more efficient designs.



Pitfall: Failure to consider instruction set design can adversely impact pipelining.



Advanced Topic: An Introduction to Digital Design Using a Hardware Design Language to Describe and Model a Pipeline and More Pipelining Illustrations

This online section covers hardware description languages and then gives a dozen examples of pipeline diagrams, starting on page 366.e18.

As mentioned in Appendix A, Verilog can describe processors for simulation or with the intention that the Verilog specification be synthesized. To achieve acceptable synthesis results in size and speed, and a behavioral specification intended for synthesis must carefully delineate the highly combinational portions of the design, such as a datapath, from the control. The datapath can then be synthesized using available libraries. A Verilog specification intended for synthesis is usually longer and more complex.

We start with a behavioral model of the five-stage pipeline. To illustrate the dichotomy between behavioral and synthesizable designs, we then give two Verilog descriptions of a multiple-cycle-per-instruction RISC-V processor: one intended solely for simulations and one suitable for synthesis.

Using Verilog for Behavioral Specification with Simulation for the Five-Stage Pipeline

Figure e4.13.1 shows a Verilog behavioral description of the pipeline that handles ALU instructions as well as loads and stores. It does not accommodate branches (even incorrectly!), which we postpone including until later in the chapter.

Because Verilog lacks the ability to define registers with named fields such as structures in C, we use several independent registers for each pipeline register. We name these registers with a prefix using the same convention; hence, IFIDIR is the IR portion of the IFID pipeline register.

This version is a behavioral description not intended for synthesis. Instructions take the same number of clock cycles as our hardware design, but the control is done in a simpler fashion by repeatedly decoding fields of the instruction in each pipe stage. Because of this difference, the instruction register (IR) is needed throughout the pipeline, and the entire IR is passed from pipe stage to pipe stage. As you read the Verilog descriptions in this chapter, remember that the actions in the `always` block all occur in parallel on every clock cycle. Since there are no blocking assignments, the order of the events within the `always` block is arbitrary.

```
module RISCVCPU (clock);
  // Instruction opcodes
  parameter LD = 7'b000_0011, SD = 7'b010_0011, BEQ = 7'b110_0011, NOP =
32'h0000_0013, ALUop = 7'b001_0011;
  input clock;

  reg [63:0] PC, Regs[0:31], IDEXA, IDEXB, EXMEMB, EXMEMALUOut,
MEMWBValue;
  reg [31:0] IMemory[0:1023], DMemory[0:1023], // separate memories
             IFIDIR, IDEXIR, EXMEMIR, MEMWBIR; // pipeline registers
  wire [4:0] IFIDrs1, IFIDrs2, MEMWBrd; // Access register fields
  wire [6:0] IDEXop, EXMEMop, MEMWBop; // Access opcodes
  wire [63:0] Ain, Bin; // the ALU inputs

  // These assignments define fields from the pipeline registers
  assign IFIDrs1  = IFIDIR[19:15];  // rs1 field
  assign IFIDrs2  = IFIDIR[24:20];  // rs2 field
  assign IDEXop   = IDEXIR[6:0];    // the opcode
  assign EXMEMop  = EXMEMIR[6:0];   // the opcode
  assign MEMWBop  = MEMWBIR[6:0];   // the opcode
  assign MEMWBrd  = MEMWBIR[11:7];  // rd field
  // Inputs to the ALU come directly from the ID/EX pipeline registers
  assign Ain = IDEXA;
  assign Bin = IDEXB;

  integer i; // used to initialize registers
  initial
  begin
    PC = 0;
    IFIDIR = NOP; IDEXIR = NOP; EXMEMIR = NOP; MEMWBIR = NOP; // put NOPs
in pipeline registers
    for (i=0;i<=31;i=i+1) Regs[i] = i; // initialize registers--just so
they aren't cares
  end

  // Remember that ALL these actions happen every pipe stage and with the
use of <= they happen in parallel!
  always @(posedge clock)
  begin
    // first instruction in the pipeline is being fetched
    // Fetch & increment PC
    IFIDIR <= IMemory[PC >> 2];
    PC <= PC + 4;

    // second instruction in pipeline is fetching registers
    IDEXA <= Regs[IFIDrs1]; IDEXB <= Regs[IFIDrs2]; // get two registers
    IDEXIR <= IFIDIR; // pass along IR--can happen anywhere, since this
affects next stage only!

    // third instruction is doing address calculation or ALU operation
    if (IDEXop == LD)
      EXMEMALUOut <= IDEXA + {{53{IDEXIR[31]}}, IDEXIR[30:20]};
    else if (IDEXop == SD)
      EXMEMALUOut <= IDEXA + {{53{IDEXIR[31]}}, IDEXIR [30:25],
IDEXIR[11:7]};
    else if (IDEXop == ALUop)
      case (IDEXIR[31:25]) // case for the various R-type instructions
        0: EXMEMALUOut <= Ain + Bin;  // add operation
```

FIGURE e4.13.1 A Verilog behavioral model for the RISC-V five-stage pipeline, ignoring branch and data hazards. As in the design earlier in Chapter 4, we use separate instruction and data memories, which would be implemented using separate caches as we describe in Chapter 5.

```
default: ; // other R-type operations: subtract, SLT, etc.
      endcase
    EXMEMIR <= IDEXIR; EXMEMB <= IDEXB; // pass along the IR & B register

    // Mem stage of pipeline
    if (EXMEMop == ALUop) MEMWBValue <= EXMEMALUOut; // pass along ALU
result
    else if (EXMEMop == LD) MEMWBValue <= DMemory[EXMEMALUOut >> 2];
    else if (EXMEMop == SD) DMemory[EXMEMALUOut >> 2] <= EXMEMB; //store
    MEMWBIR <= EXMEMIR; // pass along IR

    // WB stage
    if (((MEMWBop == LD) || (MEMWBop == ALUop)) && (MEMWBrd != 0)) //
update registers if load/ALU operation and destination not 0
      Regs[MEMWBrd] <= MEMWBValue;
  end
endmodule
```

FIGURE e4.13.1 A Verilog behavioral model for the RISC-V five-stage pipeline, ignoring branch and data hazards. (*Continued*)

Implementing Forwarding in Verilog

To extend the Verilog model further, Figure e4.13.2 shows the addition of forwarding logic for the case when the source and destination are ALU instructions. Neither load stalls nor branches are handled; we will add these shortly. The changes from the earlier Verilog description are highlighted.

Check Yourself

Someone has proposed moving the write for a result from an ALU instruction from the WB to the MEM stage, pointing out that this would reduce the maximum length of forwards from an ALU instruction by one cycle. Which of the following is accurate reasons *not to* consider such a change?

1. It would not actually change the forwarding logic, so it has no advantage.
2. It is impossible to implement this change under any circumstance since the write for the ALU result must stay in the same pipe stage as the write for a load result.
3. Moving the write for ALU instructions would create the possibility of writes occurring from two different instructions during the same clock cycle. Either an extra write port would be required on the register file or a structural hazard would be created.
4. The result of an ALU instruction is not available in time to do the write during MEM.

The Behavioral Verilog with Stall Detection

If we ignore branches, stalls for data hazards in the RISC-V pipeline are confined to one simple case: loads whose results are currently in the WB clock stage. Thus, extending the Verilog to handle a load with a destination that is either an ALU instruction or an effective address calculation is reasonably straightforward, and Figure e4.13.3 shows the few additions needed.

```
module RISCVCPU (clock);
  // Instruction opcodes
  parameter LD = 7'b000_0011, SD = 7'b010_0011, BEQ = 7'b110_0011, NOP =
32'h0000_0013, ALUop = 7'b001_0011;
  input clock;

  reg [63:0] PC, Regs[0:31], IDEXA, IDEXB, EXMEMB, EXMEMALUOut,
MEMWBValue;
  reg [31:0] IMemory[0:1023], DMemory[0:1023], // separate memories
             IFIDIR, IDEXIR, EXMEMIR, MEMWBIR; // pipeline registers
  wire [4:0] IFIDrs1, IFIDrs2, IDEXrs1, IDEXrs2, EXMEMrd, MEMWBrd; //
Access register fields
  wire [6:0] IDEXop, EXMEMop, MEMWBop; // Access opcodes
  wire [63:0] Ain, Bin; // the ALU inputs
  // declare the bypass signals
  wire bypassAfromMEM, bypassAfromALUinWB,
       bypassBfromMEM, bypassBfromALUinWB,
       bypassAfromLDinWB, bypassBfromLDinWB;

  assign IFIDrs1  = IFIDIR[19:15];
  assign IFIDrs2  = IFIDIR[24:20];
  assign IDEXop   = IDEXIR[6:0];
  assign IDEXrs1  = IDEXIR[19:15];
  assign IDEXrs2  = IDEXIR[24:20];
  assign EXMEMop  = EXMEMIR[6:0];
  assign EXMEMrd  = EXMEMIR[11:7];
  assign MEMWBop  = MEMWBIR[6:0];
  assign MEMWBrd  = MEMWBIR[11:7];

  // The bypass to input A from the MEM stage for an ALU operation
  assign bypassAfromMEM = (IDEXrs1 == EXMEMrd) && (IDEXrs1 != 0) &&
(EXMEMop == ALUop);
  // The bypass to input B from the MEM stage for an ALU operation
  assign bypassBfromMEM = (IDEXrs2 == EXMEMrd) && (IDEXrs2 != 0) &&
(EXMEMop == ALUop);
  // The bypass to input A from the WB stage for an ALU operation
  assign bypassAfromALUinWB = (IDEXrs1 == MEMWBrd) && (IDEXrs1 != 0) &&
(MEMWBop == ALUop);
  // The bypass to input B from the WB stage for an ALU operation
  assign bypassBfromALUinWB = (IDEXrs2 == MEMWBrd) && (IDEXrs2 != 0) &&
(MEMWBop == ALUop);
  // The bypass to input A from the WB stage for an LD operation
  assign bypassAfromLDinWB = (IDEXrs1 == MEMWBrd) && (IDEXrs1 != 0) &&
(MEMWBop == LD);
  // The bypass to input B from the WB stage for an LD operation
  assign bypassBfromLDinWB = (IDEXrs2 == MEMWBrd) && (IDEXrs2 != 0) &&
(MEMWBop == LD);
  // The A input to the ALU is bypassed from MEM if there is a bypass
there,
  // Otherwise from WB if there is a bypass there, and otherwise comes
from the IDEX register
  assign Ain = bypassAfromMEM ? EXMEMALUOut :
               (bypassAfromALUinWB || bypassAfromLDinWB) ? MEMWBValue :
IDEXA;
  // The B input to the ALU is bypassed from MEM if there is a bypass
there,
  // Otherwise from WB if there is a bypass there, and otherwise comes
from the IDEX register
```

FIGURE e4.13.2 A behavioral definition of the five-stage RISC-V pipeline with bypassing to ALU operations and address calculations. The code added to Figure e4.13.1 to handle bypassing is highlighted. Because these bypasses only require changing where the ALU inputs come from, the only changes required are in the combinational logic responsible for selecting the ALU inputs. (*continues on next page*)

```
  assign Bin = bypassBfromMEM ? EXMEMALUOut :
               (bypassBfromALUinWB || bypassBfromLDinWB) ? MEMWBValue:
IDEXB;

  integer i; // used to initialize registers
  initial
  begin
    PC = 0;
    IFIDIR = NOP; IDEXIR = NOP; EXMEMIR = NOP; MEMWBIR = NOP; // put NOPs
in pipeline registers
    for (i=0;i<=31;i=i+1) Regs[i] = i; // initialize registers--just so
they aren't cares
  end

  // Remember that ALL these actions happen every pipe stage and with the
use of <= they happen in parallel!
  always @(posedge clock)
  begin
    // first instruction in the pipeline is being fetched
    // Fetch & increment PC
    IFIDIR <= IMemory[PC >> 2];
    PC <= PC + 4;

    // second instruction in pipeline is fetching registers
    IDEXA <= Regs[IFIDrs1]; IDEXB <= Regs[IFIDrs2]; // get two registers
    IDEXIR <= IFIDIR; // pass along IR--can happen anywhere, since this
affects next stage only!

    // third instruction is doing address calculation or ALU operation
    if (IDEXop == LD)
      EXMEMALUOut <= IDEXA + {{53{IDEXIR[31]}}, IDEXIR[30:20]};
    else if (IDEXop == SD)
      EXMEMALUOut <= IDEXA + {{53{IDEXIR[31]}}, IDEXIR[30:25],
IDEXIR[11:7]};
    else if (IDEXop == ALUop)
      case (IDEXIR[31:25]) // case for the various R-type instructions
        0: EXMEMALUOut <= Ain + Bin;  // add operation
        default: ; // other R-type operations: subtract, SLT, etc.
      endcase
    EXMEMIR <= IDEXIR; EXMEMB <= IDEXB; // pass along the IR & B register

    // Mem stage of pipeline
    if (EXMEMop == ALUop) MEMWBValue <= EXMEMALUOut; // pass along ALU
result
    else if (EXMEMop == LD) MEMWBValue <= DMemory[EXMEMALUOut >> 2];
    else if (EXMEMop == SD) DMemory[EXMEMALUOut >> 2] <= EXMEMB; //store
    MEMWBIR <= EXMEMIR; // pass along IR

    // WB stage
    if (((MEMWBop == LD) || (MEMWBop == ALUop)) && (MEMWBrd != 0)) //
update registers if load/ALU operation and destination not 0
      Regs[MEMWBrd] <= MEMWBValue;
  end
endmodule
```

FIGURE e4.13.2 A behavioral definition of the five-stage RISC-V pipeline with bypassing to ALU operations and address calculations. (*Continued*)

```
module RISCVCPU (clock);
  // Instruction opcodes
  parameter LD = 7'b000_0011, SD = 7'b010_0011, BEQ = 7'b110_0011, NOP =
32'h0000_0013, ALUop = 7'b001_0011;
  input clock;

  reg [63:0] PC, Regs[0:31], IDEXA, IDEXB, EXMEMB, EXMEMALUOut,
MEMWBValue;
  reg [31:0] IMemory[0:1023], DMemory[0:1023], // separate memories
             IFIDIR, IDEXIR, EXMEMIR, MEMWBIR; // pipeline registers
  wire [4:0] IFIDrs1, IFIDrs2, IDEXrs1, IDEXrs2, EXMEMrd, MEMWBrd; //
Access register fields
  wire [6:0] IDEXop, EXMEMop, M EMWBop; // Access opcodes
  wire [63:0] Ain, Bin; // the ALU inputs
  // declare the bypass signals
  wire bypassAfromMEM, bypassAfromALUinWB,
       bypassBfromMEM, bypassBfromALUinWB,
       bypassAfromLDinWB, bypassBfromLDinWB;
  wire stall; // stall signal

  assign IFIDrs1  = IFIDIR[19:15];
  assign IFIDrs2  = IFIDIR[24:20];
  assign IDEXop   = IDEXIR[6:0];
  assign IDEXrs1  = IDEXIR[19:15];
  assign IDEXrs2  = IDEXIR[24:20];
  assign EXMEMop  = EXMEMIR[6:0];
  assign EXMEMrd  = EXMEMIR[11:7];
  assign MEMWBop  = MEMWBIR[6:0];
  assign MEMWBrd  = MEMWBIR[11:7];

  // The bypass to input A from the MEM stage for an ALU operation
  assign bypassAfromMEM = (IDEXrs1 == EXMEMrd) && (IDEXrs1 != 0) &&
(EXMEMop == ALUop);
  // The bypass to input B from the MEM  stage for an ALU operation
  assign bypassBfromMEM = (IDEXrs2 == EXMEMrd) && (IDEXrs2 != 0) &&
(EXMEMop == ALUop);
  // The bypass to input A from the WB stage for an ALU operation
  assign bypassAfromALUinWB = (IDEXrs1 == MEMWBrd) && (IDEXrs1 != 0) &&
(MEMWBop == ALUop);
  // The bypass to input B from the WB stage for an ALU operation
  assign bypassBfromALUinWB = (IDEXrs2 == MEMWBrd) && (IDEXrs2 != 0) &&
(MEMWBop == ALUop);
  // The bypass to input A from the WB stage for an LD operation
  assign bypassAfromLDinWB = (IDEXrs1 == MEMWBrd) && (IDEXrs1 != 0) &&
(MEMWBop == LD);
  // The bypass to input B from the WB stage for an LD operation
  assign bypassBfromLDinWB = (IDEXrs2 == MEMWBrd) && (IDEXrs2 != 0) &&
(MEMWBop == LD);
  // The A input to the ALU is  bypassed from MEM if there is a bypass
there,
  // Otherwise from WB if there is a bypass there, and otherwise comes
from the IDEX register
  assign Ain = bypassAfromMEM ? EXMEMALUOut :
               (bypassAfromALUinWB || bypassAfromLDinWB) ? MEMWBValue :
IDEXA;
  // The B input to the ALU is bypassed from MEM if there is a bypass
there,
 // Otherwise from WB if there is a bypass there, and otherwise comes
from the IDEX register
  assign Bin = bypassBfromMEM ? EXMEMALUOut :
               (bypassBfromALUinWB || bypassBfromLDinWB) ? MEMWBValue:
IDEXB;
```

FIGURE e4.13.3 A behavioral definition of the five-stage RISC-V pipeline with stalls for loads when the destination is an ALU instruction or effective address calculation. The changes from Figure e4.13.2 are highlighted. (*continues on next page*)

```
  // The signal for detecting a stall based on the use of a result from
LW
  assign stall = (MEMWBop == LD) && ( // source instruction is a load
                  (((IDEXop == LD) || (IDEXop == SD)) && (IDEXrs1 ==
MEMWBrd)) || // stall for address calc
                  ((IDEXop == ALUop) && ((IDEXrs1 == MEMWBrd) ||
(IDEXrs2 == MEMWBrd)))); // ALU use

  integer i; // used to initialize registers
  initial
  begin
    PC = 0;
    IFIDIR = NOP; IDEXIR = NOP; EXMEMIR = NOP; MEMWBIR = NOP; // put NOPs
in pipeline registers
    for (i=0;i<=31;i=i+1) Regs[i] = i; // initialize registers--just so
they aren't cares
  end

  // Remember that ALL these actions happen every pipe stage and with the
use of <= they happen in parallel!
  always @(posedge clock)
  begin
    if (~stall)
    begin // the first three pipeline stages stall if there is a load
hazard
      // first instruction in the pipeline is being fetched
      // Fetch & increment PC
      IFIDIR <= IMemory[PC >> 2];
      PC <= PC + 4;

      // second instruction in pipeline is fetching registers
      IDEXA <= Regs[IFIDrs1]; IDEXB <= Regs[IFIDrs2]; // get two
registers
      IDEXIR <= IFIDIR; // pass along IR--can happen anywhere, since this
affects next stage only!

      // third instruction is doing address calculation or ALU operation
      if (IDEXop == LD)
        EXMEMALUOut <= IDEXA + {{53{IDEXIR[31]}}, IDEXIR[30:20]};
      else if (IDEXop == SD)
        EXMEMALUOut <= IDEXA + {{53{IDEXIR[31]}}, IDEXIR[30:25],
IDEXIR[11:7]};
      else if (IDEXop == ALUop)
        case (IDEXIR[31:25]) // case for the various R-type instructions
          0: EXMEMALUOut <= Ain + Bin;  // add operation
          default: ; // other R-type operations: subtract, SLT, etc.
        endcase
      EXMEMIR <= IDEXIR; EXMEMB <= IDEXB; // pass along the IR & B
register
    end
    else EXMEMIR <= NOP; // Freeze first three stages of pipeline; inject
a nop into the EX output

// Mem stage of pipeline
    if (EXMEMop == ALUop) MEMWBValue <= EXMEMALUOut; // pass along ALU
result
    else if (EXMEMop == LD) MEMWBValue <= DMemory[EXMEMALUOut >> 2];
    else if (EXMEMop == SD) DMemory[EXMEMALUOut >> 2] <= EXMEMB; //store
    MEMWBIR <= EXMEMIR; // pass along IR

    // WB stage
    if (((MEMWBop == LD) || (MEMWBop == ALUop)) && (MEMWBrd != 0)) //
update registers if load/ALU operation and destination not 0
      Regs[MEMWBrd] <= MEMWBValue;
  end
endmodule
```

FIGURE e4.13.3 A behavioral definition of the five-stage RISC-V pipeline with stalls for loads when the destination is an ALU instruction or effective address calculation. (*Continued*)

Check Yourself

Someone has asked about the possibility of data hazards occurring through memory, contrary to through a register. Which of the following statements about such hazards is true?

1. Since memory accesses only occur in the MEM stage, all memory operations are done in the same order as instruction execution, making such hazards impossible in this pipeline.
2. Such hazards *are* possible in this pipeline; we just have not discussed them yet.
3. No pipeline can ever have a hazard involving memory, since it is the programmer's job to keep the order of memory references accurate.
4. Memory hazards may be possible in some pipelines, but they cannot occur in this particular pipeline.
5. Although the pipeline control would be obligated to maintain ordering among memory references to avoid hazards, it is impossible to design a pipeline where the references could be out of order.

Implementing the Branch Hazard Logic in Verilog

We can extend our Verilog behavioral model to implement the control for branches. We add the code to model branch equal using a "predict not taken" strategy. The Verilog code is shown in Figure e4.13.4. It implements the branch hazard by detecting a taken branch in ID and using that signal to squash the instruction in IF (by setting the IR to `0x00000013`, which is an effective `NOP` in RISC-V); in addition, the PC is assigned to the branch target. Note that to prevent an unexpected latch, it is important that the PC is clearly assigned on every path through the always block; hence, we assign the PC in a single *if* statement. Lastly, note that although Figure e4.13.4 incorporates the basic logic for branches and control hazards, supporting branches requires additional bypassing and data hazard detection, which we have not included.

Using Verilog for Behavioral Specification with Synthesis

To demonstrate the contrasting types of Verilog, we show two descriptions of a different, nonpipelined implementation style of RISC-V that uses multiple clock cycles per instruction. (Since some instructors make a synthesizable description of the RISC-V pipeline project for a class, we chose not to include it here. It would also be long.)

Figure e4.13.5 gives a behavioral specification of a multicycle implementation of the RISC-V processor. Because of the use of behavioral operations, it would be difficult to synthesize a separate datapath and control unit with any reasonable efficiency. This version demonstrates another approach to the control by using a Mealy finite-state machine (see discussion in Section A.10 of Appendix A). The use of a Mealy machine, which allows the output to depend both on inputs and the current state, allows us to decrease the total number of states.

```
module RISCVCPU (clock);
  // Instruction opcodes
  parameter LD = 7'b000_0011, SD = 7'b010_0011, BEQ = 7'b110_0011, NOP =
32'h0000_0013, ALUop = 7'b001_0011;
  input clock;

  reg [63:0] PC, Regs[0:31], IDEXA, IDEXB, EXMEMB, EXMEMALUOut,
MEMWBValue;
  reg [31:0] IMemory[0:1023], DMemory[0:1023], // separate memories
             IFIDIR, IDEXIR, EXMEMIR, MEMWBIR; // pipeline registers
  wire [4:0] IFIDrs1, IFIDrs2, IDEXrs1, IDEXrs2, EXMEMrd, MEMWBrd; //
Access register fields
  wire [6:0] IFIDop, IDEXop, EXMEMop, MEMWBop; // Access opcodes
  wire [63:0] Ain, Bin; // the ALU inputs
  // declare the bypass signals
  wire bypassAfromMEM, bypassAfromALUinWB,
       bypassBfromMEM, bypassBfromALUinWB,
       bypassAfromLDinWB, bypassBfromLDinWB;
  wire stall; // stall signal
  wire takebranch;

  assign IFIDop   = IFIDIR[6:0];
  assign IFIDrs1  = IFIDIR[19:15];
  assign IFIDrs2  = IFIDIR[24:20];
  assign IDEXop   = IDEXIR[6:0];
  assign IDEXrs1  = IDEXIR[19:15];
  assign IDEXrs2  = IDEXIR[24:20];
  assign EXMEMop  = EXMEMIR[6:0];
  assign EXMEMrd  = EXMEMIR[11:7];
  assign MEMWBop  = MEMWBIR[6:0];
  assign MEMWBrd  = MEMWBIR[11:7];

  // The bypass to input A from the MEM stage for an ALU operation
  assign bypassAfromMEM = (IDEXrs1 == EXMEMrd) && (IDEXrs1 != 0) &&
(EXMEMop == ALUop);
  // The bypass to input B from the MEM stage for an ALU operation
  assign bypassBfromMEM = (IDEXrs2 == EXMEMrd) && (IDEXrs2 != 0) &&
(EXMEMop == ALUop);
  // The bypass to input A from the WB stage for an ALU operation
  assign bypas sAfromALUinWB = (IDEXrs1 == MEMWBrd) && (IDEXrs1 != 0) &&
(MEMWBop == ALUop);
  // The bypass to input B from the WB stage for an ALU operation
  assign bypassBfromALUinWB = (IDEXrs2 == MEMWBrd) && (IDEXrs2 != 0) &&
(MEMWBop == ALUop);
  // The bypass to input A from the WB stage for an LD operation
  assign bypassAfromLDinWB = (IDEXrs1 == MEMWBrd) && (IDEXrs1 != 0) &&
(MEMWBop == LD);
  // The bypass to input B from the WB stage for an LD operation
  assign bypassBfromLDinWB = (IDEXrs2 == MEMWBrd) && (IDEX rs2 != 0) &&
(MEMWBop == LD);
  // The A input to the ALU is bypassed from MEM if there is a bypass
there,
  // Otherwise from WB if there is a bypass there, and otherwise comes
from the IDEX register
  assign Ain = bypassAfromMEM ? EXMEMALUOut :
               (bypassAfromALUinWB || bypassAfromLDinWB) ? MEMWBValue :
IDEXA;

 // The B input to the ALU is bypassed from MEM if there is a bypass
there,
  // Otherwise from WB if there is a bypass there, and otherwise comes
from the IDEX register
  assign Bin = bypassBfromMEM ? EXMEMALUOut :
               (bypassBfromALUinWB || bypassBfromLDinWB) ? MEMWBValue:
```

FIGURE e4.13.4 A behavioral definition of the five-stage RISC-V pipeline with stalls for loads when the destination is an ALU instruction or effective address calculation. The changes from Figure e4.13.2 are highlighted. (*continues on next page*)

```
IDEXB;
  // The signal for detecting a stall based on the use of a result from
LW
  assign stall = (MEMWBop == LD) && ( // source instruction is a load
                    (((IDEXop == LD) || (IDEXop == SD)) && (IDEXrs1 ==
MEMWBrd)) || // stall for address calc
                    ((IDEXop == ALUop) && ((IDEXrs1 == MEMWBrd) ||
(IDEXrs2 == MEMWBrd)))); // ALU use
  // Signal for a taken branch: instruction is BEQ and registers are
equal
  assign takebranch = (IFIDop == BEQ) && (Regs[IFIDrs1] ==
Regs[IFIDrs2]);

  integer i; // used to initialize registers
  initial
  begin
    PC = 0;
    IFIDIR = NOP; IDEXIR = NOP; EXMEMIR = NOP; MEMWBIR = NOP; // put NOPs
in pipeline registers
    for (i=0;i<=31;i=i+1) Regs[i] = i; // initialize registers--just so
they aren't cares
  end

  // Remember that ALL these actions happen every pipe stage and with the
use of <= they happen in parallel!
  always @(posedge clock)
  begin
    if (~stall)
    begin // the first three pipeline stages stall if there is a load
hazard
      if (~takebranch)
      begin // first instruction in the pipeline is being fetched
normally
        IFIDIR <= IMemory[PC >> 2];
        PC <= PC + 4;
      end
      else
      begin // a taken branch is in ID; instruction in IF is wrong;
insert a NOP and reset the PC
        IFIDIR <= NOP;
        PC <= PC + {{52{IFIDIR[31]}}, IFIDIR[7], IFIDIR[30:25],
IFIDIR[11:8], 1'b0};
      end

      // second instruction in pipeline is fetching registers
      IDEXA <= Regs[IFIDrs1]; IDEXB <= Regs[IFIDrs2]; // get two
registers
      IDEXIR <= IFIDIR; // pass along IR--can happen anywhere, since this
affects next stage only!

      // third instruction is doing addre ss calculation or ALU operation
      if (IDEXop == LD)


 EXMEMALUOut <= IDEXA + {{53{IDEXIR[31]}}, IDEXIR[30:20]};
      else if (IDEXop == SD)
        EXMEMALUOut <= IDEXA + {{53{IDEXIR[31]}}, IDEXIR[30:25],
IDEXIR[11:7]};
      else if (IDEXop == ALUop)
        case (IDEXIR[31:25]) // case for the various R-type instructions
          0: EXMEMALUOut <= Ain + Bin;  // add operation
```

FIGURE e4.13.4 A behavioral definition of the five-stage RISC-V pipeline with stalls for loads when the destination is an ALU instruction or effective address calculation. (*Continued*)

```
default: ; // other R-type operations: subtract, SLT, etc.
        endcase
      EXMEMIR <= IDEXIR; EXMEMB <= IDEXB; // pass along the IR & B
register
    end
    else EXMEMIR <= NOP; // Freeze first three stages of pipeline; inject
a nop into the EX output

    // Mem stage of pipeline
    if (EXMEMop == ALUop) MEMWBValue <= EXMEMALUOut; // pass along ALU
result
    else if (EXMEMop == LD) MEMWBValue <= DMemory[EXMEMALUOut >> 2];
    else if (EXMEMop == SD) DMemory[EXMEMALUOut >> 2] <= EXMEMB; //store
    MEMWBIR <= EXMEMIR; // pass along IR

    // WB stage
    if (((MEMWBop == LD) || (MEMWBop == ALUop)) && (MEM WBrd != 0)) //
update registers if load/ALU operation and destination not 0
      Regs[MEMWBrd] <= MEMWBValue;
  end
endmodule
```

FIGURE e4.13.4 A behavioral definition of the five-stage RISC-V pipeline with stalls for loads when the destination is an ALU instruction or effective address calculation. (*Continued*)

Since a version of the RISC-V design intended for synthesis is considerably more complex, we have relied on a number of Verilog modules that were specified in Appendix A, including the following:

- The 4-to-1 multiplexor shown in Figure A.4.2, and the 2-to-1 multiplexor that can be trivially derived based on the 4-to-1 multiplexor.
- The RISC-V ALU shown in Figure A.5.15.
- The RISC-V ALU control defined in Figure A.5.16.
- The RISC-V register file defined in Figure A.8.11.

Now, let's look at a Verilog version of the RISC-V processor intended for synthesis. Figure e4.13.6 shows the structural version of the RISC-V datapath. Figure e4.13.7 uses the datapath module to specify the RISC-V CPU. This version also demonstrates another approach to implementing the control unit, as well as some optimizations that rely on relationships between various control signals. Observe that the state machine specification only provides the sequencing actions.

The setting of the control lines is done with a series of `assign` statements that depend on the state as well as the opcode field of the instruction register. If one were to fold the setting of the control into the state specification, this would look like a Mealy-style finite-state control unit. Because the setting of the control lines is specified using `assign` statements outside of the `always` block, most logic synthesis systems will generate a small implementation of a finite-state machine

```
module RISCVCPU (clock);
  parameter LD = 7'b000_0011, SD = 7'b010_0011, BEQ = 7'b110_0011, ALUop
= 7'b001_0011;
  input clock; //the clock is an external input

  // The architecturally visible registers and scratch registers for
implementation
  reg [63:0] PC, Regs[0:31], ALUOut, MDR, A, B;
  reg [31:0] Memory [0:1023], IR;
  reg [2:0] state; // processor state
  wire [6:0] opcode; // use to get opcode easily
  wire [63:0] ImmGen; // used to generate immediate

  assign opcode = IR[6:0]; // opcode is lower 7 bits
  assign ImmGen = (opcode == LD) ? {{53{IR[31]}}, IR[30:20]} :
              /* (opcode == SD) */{{53{IR[31]}}, IR[30:25], IR[11:7]};
  assign PCOffset = {{52{IR[31]}}, IR[7], IR[30:25], IR[11:8], 1'b0};

  // set the PC to 0 and start the control in state 1
  initial begin PC = 0; state = 1; end

  // The state machine--triggered on a rising clock
  always @(posedge clock)
  begin
    Regs[0] <= 0; // shortcut way to make sure R0 is always 0
    case (state) //action depends on the state
      1: begin // first step: fetch the instruction, increment PC, go to
next state
        IR <= Memory[PC >> 2];
        PC <= PC + 4;
        state <= 2; // next state
      end
      2: begin // second step: Instruction decode, register fetch, also
compute branch address
        A <= Regs[IR[19:15]];
        B <= Regs[IR[24:20]];
        ALUOut <= PC + PCOffset; // compute PC-relative branch target
        state <= 3;
      end
      3: begin // third step: Load-store execution, ALU execution, Branch
completion
        if ((opcode == LD) || (opcode == SD))
        begin
          ALUOut <= A + ImmGen; // compute effective address
          state <= 4;
        end
        else if (opcode == ALUop)
        begin
          case (IR[31:25]) // case for the various R-type instructions
            0: ALUOut <= A + B; // add operation
            default: ; // other R-type operations: subtract, SLT, etc.
          endcase
          state <= 4;
        end
        else if (opcode == BEQ)
        begin
          if (A == B) begin
            PC <= ALUOut; // branch taken--update PC

    end
```

FIGURE e4.13.5 A behavioral specification of the multicycle RISC-V design. This has the same cycle behavior as the multicycle design, but is purely for simulation and specification. It cannot be used for synthesis. (*continues on next page*)

```
          state <= 1;
        end
        else ; // other opcodes or exception for undefined instruction
  would go here
      end
      4: begin
        if (opcode == ALUop)
        begin // ALU Operation
          Regs[IR[11:7]] <= ALUOut; // write the result
          state <= 1;
        end // R-type finishes
        else if (opcode == LD)
        begin // load instruction
           MDR <= Memory[ALUOut >> 2]; // read the memory
           state <= 5; // next state
        end
        else if (opcode == SD)
        begin // store instruction
          Memory[ALUOut >> 2] <= B; // write the memory
          state <= 1; // return to state 1
        end
        else ; // other instructions go here
      end
      5: begin // LD is the only instruction still in execution
        Regs[IR[11:7]] <= MDR; // write the MDR to the register
        state <= 1;
      end // complete an LD instruction
    endcase
  end
endmodule
```

FIGURE e4.13.5 A behavioral specification of the multicycle RISC-V design. (*Continued*)

that determines the setting of the state register and then uses external logic to derive the control inputs to the datapath.

In writing this version of the control, we have also taken advantage of a number of insights about the relationship between various control signals as well as situations where we don't care about the control signal value; some examples of these are given in the following elaboration.

More Illustrations of Instruction Execution on the Hardware

To reduce the cost of this book, starting with the third edition, we moved sections and figures that were used by a minority of instructors online. This subsection recaptures those figures for readers who would like more supplemental material to understand pipelining better. These are all single-clock-cycle pipeline diagrams, which take many figures to illustrate the execution of a sequence of instructions.

The three examples are respectively for code with no hazards, an example of forwarding on the pipelined implementation, and an example of bypassing on the pipelined implementation.

```
module Datapath (ALUOp, MemtoReg, MemRead, MemWrite, IorD, RegWrite,
IRWrite,
                 PCWrite, PCWriteCond, ALUSrcA, ALUSrcB, PCSource,
opcode, clock); // the control inputs + clock
  parameter LD = 7'b000_0011, SD = 7'b010_0011;
  input [1:0] ALUOp, ALUSrcB; // 2-bit control signals
  input MemtoReg, MemRead, MemWrite, IorD, RegWrite, IRWrite, PCWrite,
PCWriteCond,
        ALUSrcA, PCSource, clock; // 1-bit control signals
  output [6:0] opcode; // opcode is needed as an output by control
  reg [63:0] PC, MDR, ALUOut; // CPU state + some temporaries
  reg [31:0] Memory[0:1023], IR; // CPU state + some temporaries
  wire [63:0] A, B, SignExtendOffset, PCOffset, ALUResultOut, PCValue,
JumpAddr, Writedata, ALUAin,
              ALUBin, MemOut; // these are signals derived from registers
  wire [3:0] ALUCtl; // the ALU control lines
  wire Zero; // the Zero out signal from the ALU

  initial PC = 0; //start the PC at 0
  //Combinational signals used in the datapath
  // Read using word address with either ALUOut or PC as the address
source
  assign MemOut = MemRead ? Memory[(IorD ? ALUOut : PC) >> 2] : 0;
  assign opcode = IR[6:0]; // opcode shortcut
  // Get the write register data either from the ALUOut or from the MDR
  assign Writedata = MemtoReg ? MDR : ALUOut;
  // Generate immediate
  assign ImmGen = (opcode == LD) ? {{53{IR[31]}}, IR[30:20]} :
               /* (opcode == SD) */{{53{IR[31]}}, IR[30:25], IR[11:7]};
  // Generate pc offset for branches
  assign PCOffset = {{52{IR[31]}}, IR[7], IR[30:25], IR[11:8], 1'b0};
  // The A input to the ALU is either the rs register or the PC
  assign ALUAin = ALUSrcA ? A : PC; // ALU input is PC or A

  // Creates an instance of the ALU control unit (see the module defined
in Figure B.5.16
    // Input ALUOp is control-unit set and used to describe the
instruction class as in Chapter 4
    // Input IR[31:25] is the function code field for an ALU instruction
    // Output ALUCtl are the actual ALU control bits as in Chapter 4
    ALUControl alucontroller (ALUOp, IR[31:25], ALUCtl); // ALU control
unit

  // Creates a 2-to-1 multiplexor used to select the source of the next
PC
    // Inputs are ALUResultOut (the incremented PC), ALUOut (the branch
address)
    // PCSource is the selector input and PCValue is the multiplexor
output
    Mult2to1 PCdatasrc (ALUResultOut, ALUOut, PCSource, PCValue);

  // Creates a 4-to-1 multiplexor used to select the B input of the ALU
    // Inputs are register B, constant 4, generated immediate, PC offset
// ALUSrcB is the select or input
    // ALUBin is the multiplexor output
    Mult4to1 ALUBinput (B, 64'd4, ImmGen, PCOffset, ALUSrcB, ALUBin);

  // Creates a RISC-V ALU
    // Inputs are ALUCtl (the ALU control), ALU value inputs (ALUAin,
ALUBin)
    // Outputs are ALUResultOut (the 64-bit output) and Zero (zero
detection output)
    RISCVALU ALU (ALUCtl, ALUAin, ALUBin, ALUResultOut, Zero); // the ALU
```

FIGURE e4.13.6 A Verilog version of the multicycle RISC-V datapath that is appropriate for synthesis. This datapath relies on several units from Appendix A. Initial statements do not synthesize, and a version used for synthesis would have to incorporate a reset signal that had this effect. Also note that resetting R0 to 0 on every clock is not the best way to ensure that R0 stays at 0; instead, modifying the register file module to produce 0 whenever R0 is read and to ignore writes to R0 would be a more efficient solution. (*continues on next page*)

```
  // Creates a RISC-V register file
    // Inputs are the rs1 and rs2 fields of the IR used to specify which
registers to read,
    // Writereg (the write register number), Writedata (the data to be
written),
    // RegWrite (indicates a write), the clock
    // Outputs are A and B, the registers read
    registerfile regs (IR[19:15], IR[24:20], IR[11:7], Writedata,
RegWrite, A, B, clock); // Register file

  // The clock-triggered actions of the datapath
  always @(posedge clock)
  begin
    if (MemWrite) Memory[ALUOut >> 2] <= B; // Write memory--must be a
store
    ALUOut <= ALUResultOut; // Save the ALU result for use on a later
clock cycle
    if (IRWrite) IR <= MemOut; // Write the IR if an instruction fetch
    MDR <= MemOut; // Always save the memory read value
    // The PC is written both conditionally (controlled by PCWrite) and
unconditionally
  end
endmodule
```

FIGURE e4.13.6 A Verilog version of the multicycle RISC-V datapath that is appropriate for synthesis. (*Continued*)

No Hazard Illustrations

On page 285, we gave the example code sequence

```
ld    x10, 40(x1)
sub   x11, x2, x3
add   x12, x3, x4
ld    x13, 48(x1)
add   x14, x5, x6
```

Figures e4.42 and e4.43 showed the multiple-clock-cycle pipeline diagrams for this two-instruction sequence executing across six clock cycles. Figures e4.13.8 through e4.13.10 show the corresponding single-clock-cycle pipeline diagrams for these two instructions. Note that the order of the instructions differs between these two types of diagrams: the newest instruction is at the *bottom and to the right* of the multiple-clock-cycle pipeline diagram, and it is on the *left* in the single-clock-cycle pipeline diagram.

More Examples

To understand how pipeline control works, let's consider these five instructions going through the pipeline:

```
ld    x10, 40(x1)
sub   x11, x2, x3
and   x12, x4, x5
or    x13, x6, x7
add   x14, x8, x9
```

```
module RISCVCPU (clock);
  parameter LD = 7'b000_0011, SD = 7'b010_0011, BEQ = 7'b110_0011, ALUop
= 7'b001_0011;
  input clock;

  reg [2:0] state;
  wire [1:0] ALUOp, ALUSrcB;
  wire [6:0] opcode;
  wire MemtoReg, MemRead, MemWrite, IorD, RegWrite, IRWrite,
       PCWrite, PCWriteCond, ALUSrcA, PCSource, MemoryOp;

  // Create an instance of the RISC-V datapath, the inputs are the
control signals; opcode is only output
  Datapath RISCVDP (ALUOp, MemtoReg, MemRead, MemWrite, IorD, RegWrite,
IRWrite,
                    PCWrite, PCWriteCond, ALUSrcA, ALUSrcB, PCSource,
opcode, clock);

  initial begin state = 1; end // start the state machine in state 1
  // These are the definitions of the control signals
  assign MemoryOp = (opcode == LD) || (opcode == SD); // a memory
operation
  assign ALUOp = ((state == 1) || (state == 2) || ((state == 3) &&
MemoryOp)) ? 2'b00 : // add
                ((state == 3) && (opcode == BEQ)) ? 2'b01 : 2'b10; //
subtract or use function code
  assign MemtoReg = ((state == 4) && (opcode == ALUop)) ? 0 : 1;
  assign MemRead = (state == 1) || ((state == 4) && (opcode == LD));
  assign MemWrite = (state == 4) && (opcode == SD);
  assign IorD = (state == 1) ? 0 : 1;
  assign RegWrite = (state == 5) || ((state == 4) && (opcode == ALUop));
  assign IRWrite = (state == 1);
  assign PCWrite = (state == 1);
  assign PCWriteCond = (state == 3) && (opcode == BEQ);
  assign ALUSrcA = ((state == 1) || (state == 2)) ? 0 : 1;
  assign ALUSrcB = ((state == 1) || ((state == 3) && (opcode == BEQ)))?
2'b01 :
                   (state == 2) ? 2'b11 :
                   ((state == 3) && MemoryOp) ? 2'b10 : 2'b00; // memory
operation or other
  assign PCSource = (state == 1) ? 0 : 1;

  // Here is the state machine, which only has to sequence states
  always @(posedge clock)
  begin // all state updates on a positive clock edge
    case (state)
      1: state <= 2; // unconditional next state
      2: state <= 3; // unconditional next state
      3: state <= (opcode == BEQ) ? 1 : 4; // branch go back else next
state
      4: state <= (opcode == LD) ? 5 : 1; // R-type and SD finish
      5: state <= 1; // go back
    endcase
  end
endmodule
```

FIGURE e4.13.7 The RISC-V CPU using the datapath from Figure e4.13.6.

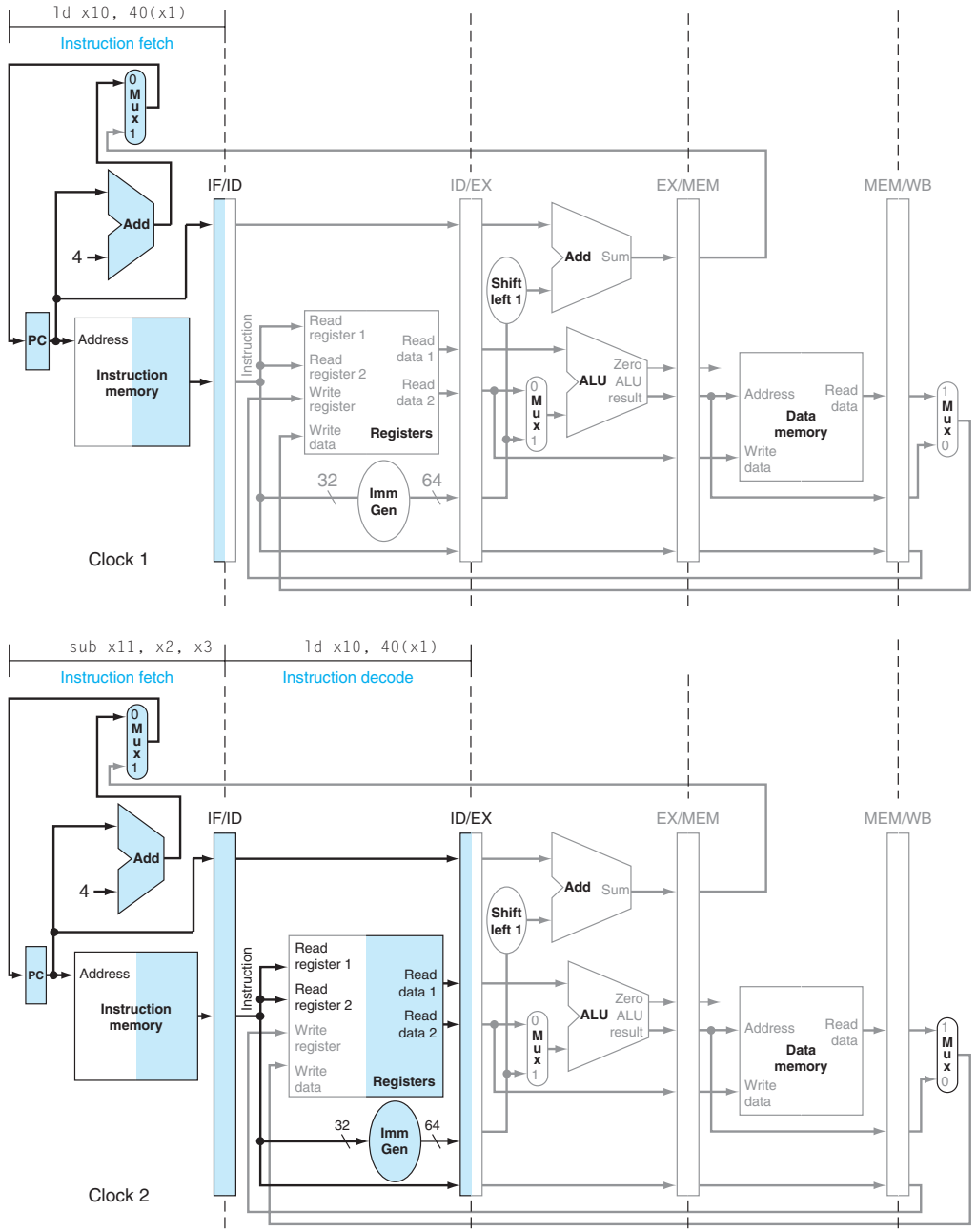


FIGURE e4.13.8 Single-cycle pipeline diagrams for clock cycles 1 (top diagram) and 2 (bottom diagram). This style of pipeline representation is a snapshot of every instruction executing during one clock cycle. Our example has but two instructions, so at most two stages are identified in each clock cycle; normally, all five stages are occupied. The highlighted portions of the datapath are active in that clock cycle. The load is fetched in clock cycle 1 and decoded in clock cycle 2, with the subtract fetched in the second clock cycle. To make the figures easier to understand, the other pipeline stages are empty, but normally there is an instruction in every pipeline stage.

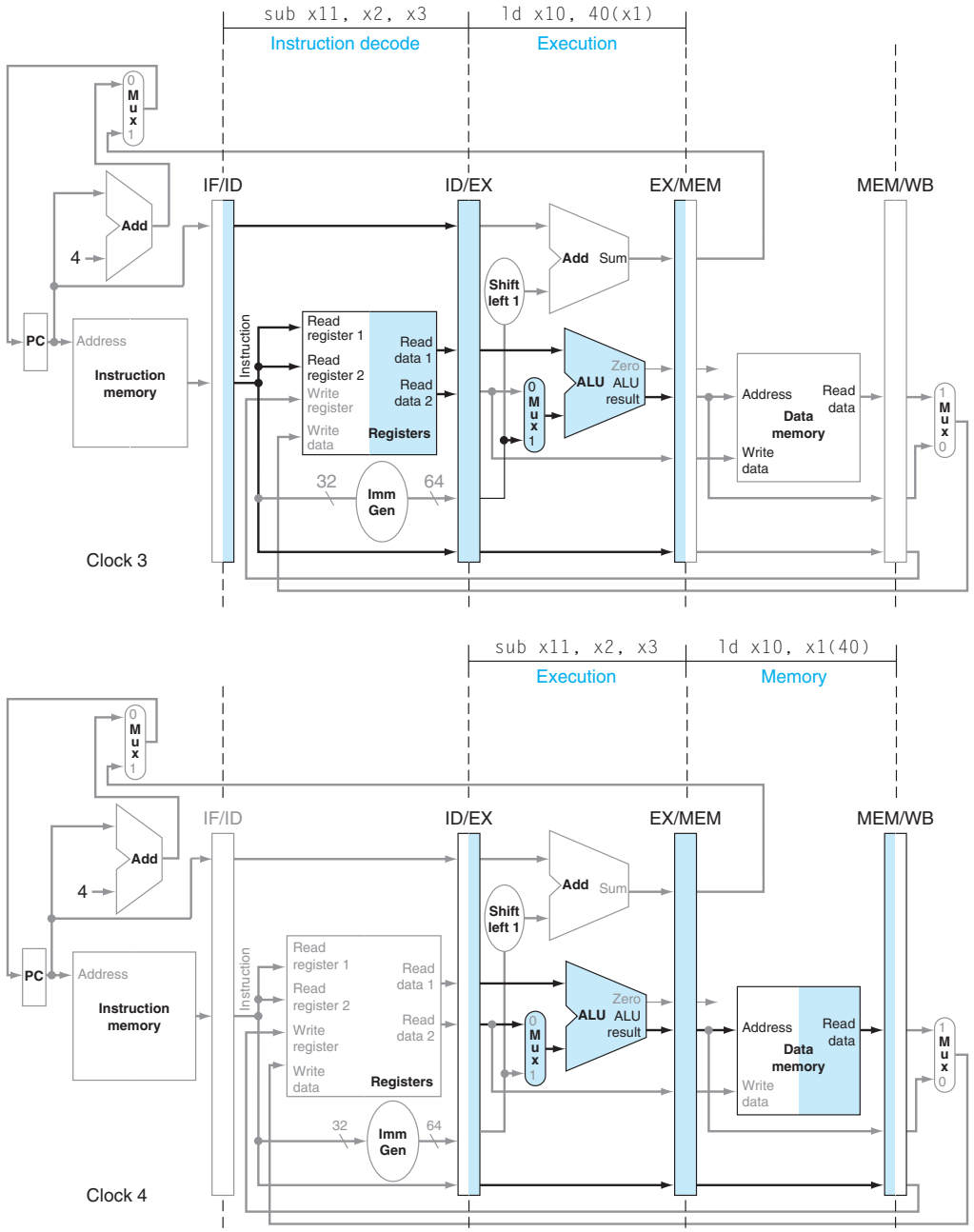


FIGURE e4.13.9 Single-cycle pipeline diagrams for clock cycles 3 (top diagram) and 4 (bottom diagram). In the third clock cycle in the top diagram, `ld` enters the EX stage. At the same time, `sub` enters ID. In the fourth clock cycle (bottom datapath), `ld` moves into MEM stage, reading memory using the address found in EX/MEM at the beginning of clock cycle 4. At the same time, the ALU subtracts and then places the difference into EX/MEM at the end of the clock cycle.

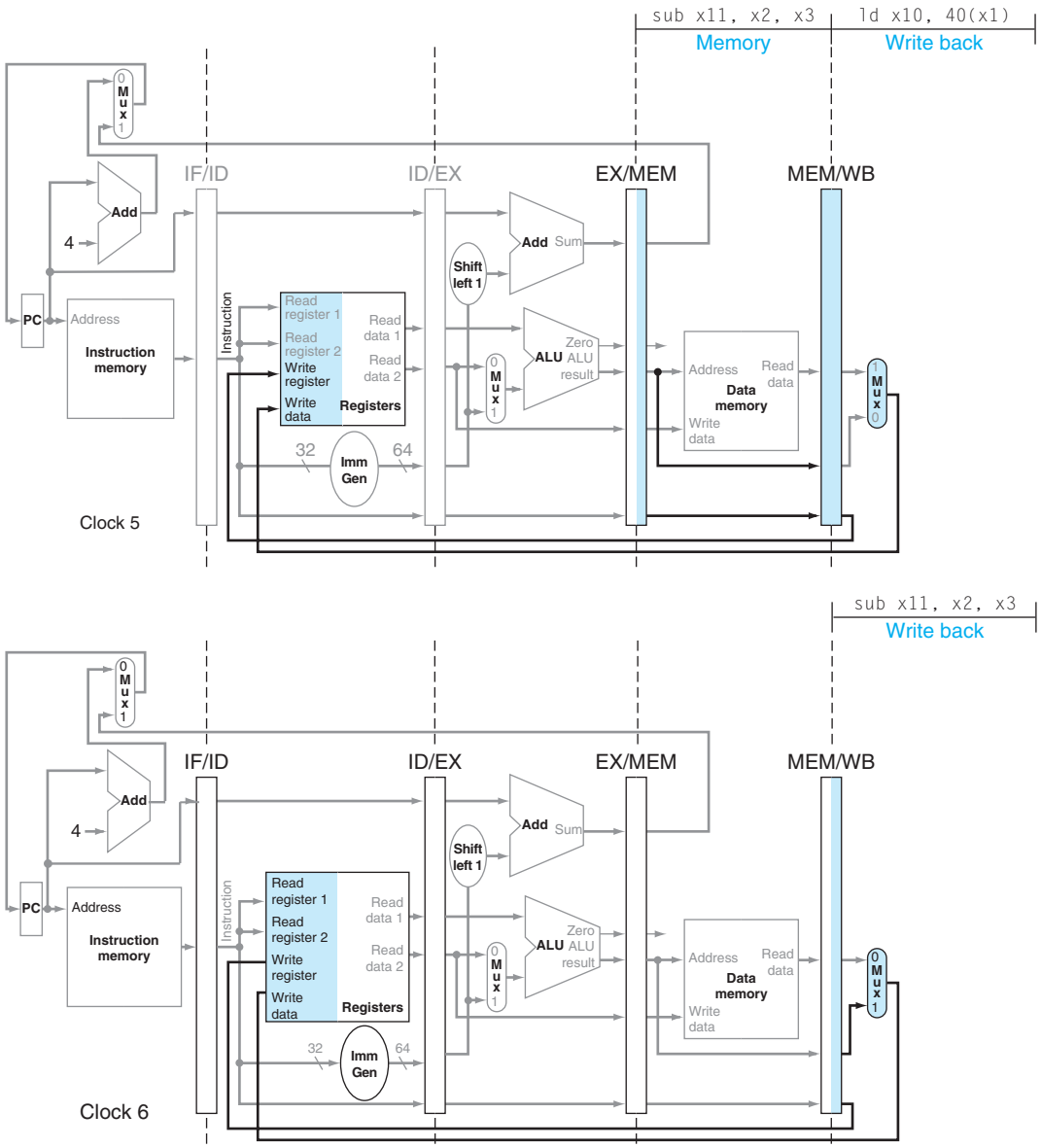


FIGURE e4.13.10 Single-cycle pipeline diagrams for clock cycles 5 (top diagram) and 6 (bottom diagram). In clock cycle 5, `ld` completes by writing the data in MEM/WB into register 10, and `sub` sends the difference in EX/MEM to MEM/WB. In the next clock cycle, `sub` writes the value in MEM/WB to register 11.

Figures e4.13.11 through e4.13.15 show these instructions proceeding through the nine clock cycles it takes them to complete execution, highlighting what is active in a stage and identifying the instruction associated with each stage during a clock cycle. If you examine them carefully, you may notice:

- In Figure e4.13.13 you can see the sequence of the destination register numbers from left to right at the bottom of the pipeline registers. The numbers advance to the right during each clock cycle, with the MEM/WB pipeline register supplying the number of the register written during the WB stage.
- When a stage is inactive, the values of control lines that are deasserted are shown as 0 or X (for don't care).
- Sequencing of control is embedded in the pipeline structure itself. First, all instructions take the same number of clock cycles, so there is no special control for instruction duration. Second, all control information is computed during instruction decode and then passed along by the pipeline registers.

Forwarding Illustrations

We can use the single-clock-cycle pipeline diagrams to show how forwarding operates, as well as how the control activates the forwarding paths. Consider the following code sequence in which the dependences have been highlighted:

```
sub  x2, x1, x3
and  x4, x2, x5
or   x4, x4, x2
add  x9, x4, x2
```

Figures e4.13.16 and e4.13.17 show the events in clock cycles 3–6 in the execution of these instructions.

Thus, in clock cycle 5, the forwarding unit selects the EX/MEM pipeline register for the upper input to the ALU and the MEM/WB pipeline register for the lower input to the ALU. The following `add` instruction reads both register `x4`, the target of the `and` instruction, and register `x2`, which the `sub` instruction has already written. Notice that the prior two instructions both write register `x4`, so the forwarding unit must pick the immediately preceding one (MEM stage).

In clock cycle 6, the forwarding unit thus selects the EX/MEM pipeline register, containing the result of the `or` instruction, for the upper ALU input but uses the non-forwarding register value for the lower input to the ALU.

Illustrating Pipelines with Stalls and Forwarding

We can use the single-clock-cycle pipeline diagrams to show how the control for stalls works. Figures e4.13.18 through e4.13.20 show the single-cycle diagram for clocks 2 through 7 for the following code sequence (dependences highlighted):

```
ld   x2, 40(x1)
and  x4, x2, x5
or   x4, x4, x2
add  x9, x4, x2
```

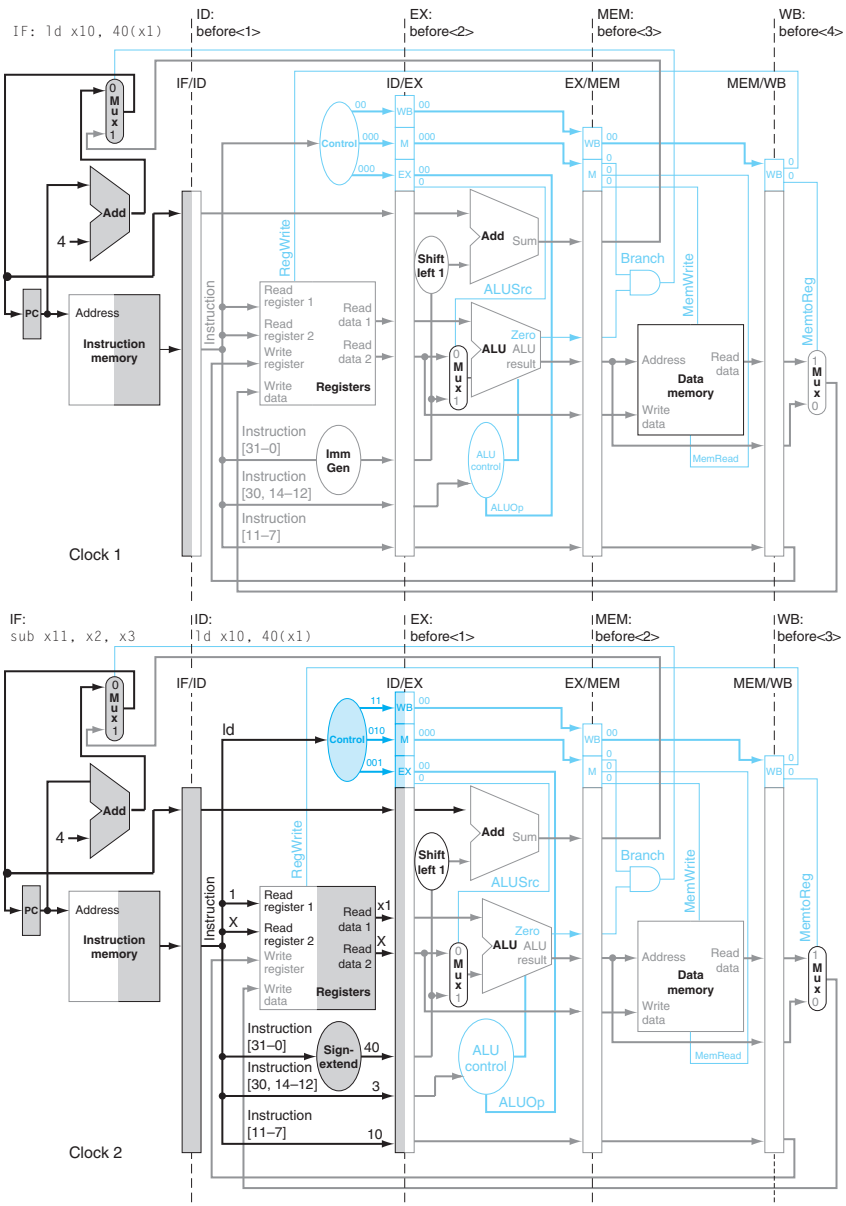


FIGURE e4.13.11 Clock cycles 1 and 2. The phrase "*before <i>*" means the *i*th instruction before `ld`. The `ld` instruction in the top datapath is in the IF stage. At the end of the clock cycle, the `ld` instruction is in the IF/ID pipeline registers. In the second clock cycle, seen in the bottom datapath, the `ld` moves to the ID stage, and `sub` enters in the IF stage. Note that the values of the instruction fields and the selected source registers are shown in the ID stage. Hence, register `x1` and the constant 40, the operands of `ld`, are written into the ID/EX pipeline register. The number 10, representing the destination register number of `ld`, is also placed in ID/EX. The top of the ID/EX pipeline register shows the control values for `ld` to be used in the remaining stages. These control values can be read from the `ld` row of the table in Figure e4.18.

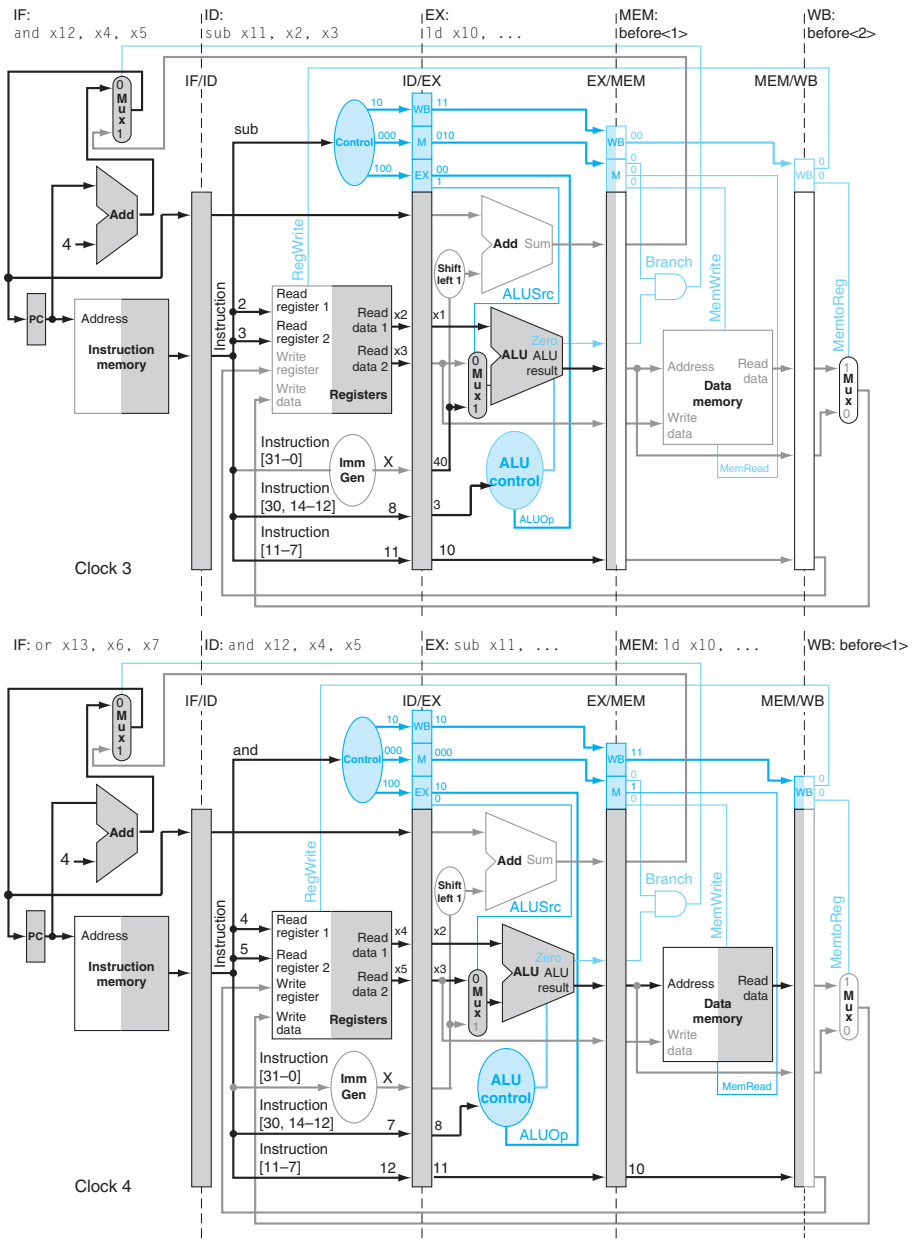


FIGURE e4.13.12 Clock cycles 3 and 4. In the top diagram, `ld` enters the EX stage in the third clock cycle, adding `x1` and 40 to form the address in the EX/MEM pipeline register. (The `ld` instruction is written `ld x10, ...` upon reaching EX, because the identity of instruction operands is not needed by EX or the subsequent stages. In this version of the pipeline, the actions of EX, MEM, and WB depend only on the instruction and its destination register or its target address.) At the same time, `sub` enters ID, reading registers `x2` and `x3`, and the `and` instruction starts IF. In the fourth clock cycle (bottom datapath), `ld` moves into MEM stage, reading memory using the value in EX/MEM as the address. In the same clock cycle, the ALU subtracts `x3` from `x2` and places the difference into EX/MEM, reads registers `x4` and `x5` during ID, and the `or` instruction enters IF. The two diagrams show the control signals being created in the ID stage and peeled off as they are used in subsequent pipe stages.

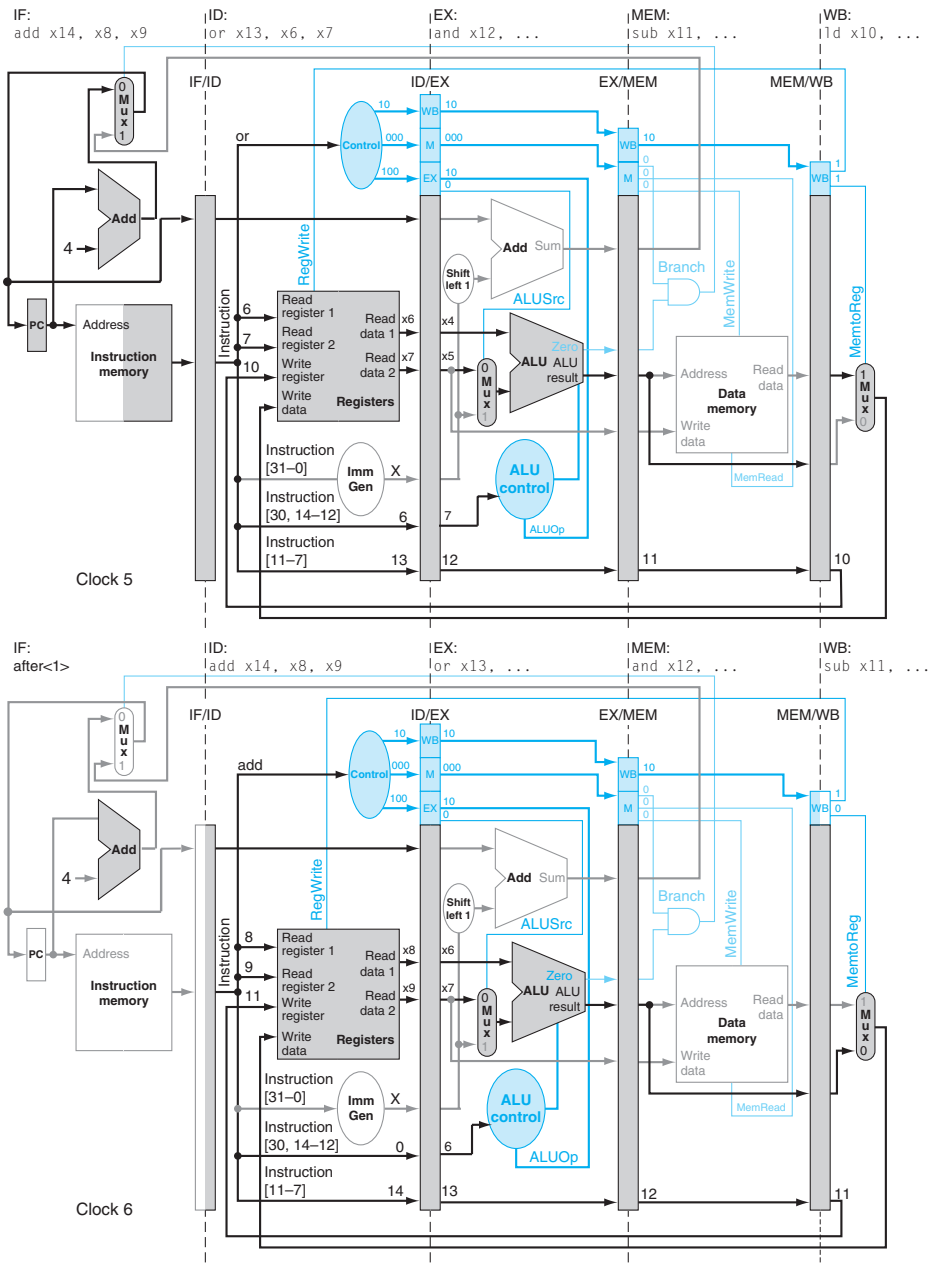


FIGURE e4.13.13 Clock cycles 5 and 6. With `add`, the final instruction in this example, entering IF in the top datapath, all instructions are engaged. By writing the data in MEM/WB into register 10, `ld` completes; both the data and the register number are in MEM/WB. In the same clock cycle, `sub` sends the difference in EX/MEM to MEM/WB, and the rest of the instructions move forward. In the next clock cycle, `sub` selects the value in MEM/WB to write to register number 11, again found in MEM/WB. The remaining instructions play follow-the-leader: the ALU calculates the OR of `x6` and `x7` for the `or` instruction in the EX stage, and registers `x8` and `x9` are read in the ID stage for the `add` instruction. The instructions after `add` are shown as inactive just to emphasize what occurs for the five instructions in the example. The phrase "after <i>" means the *i*th instruction after `add`.

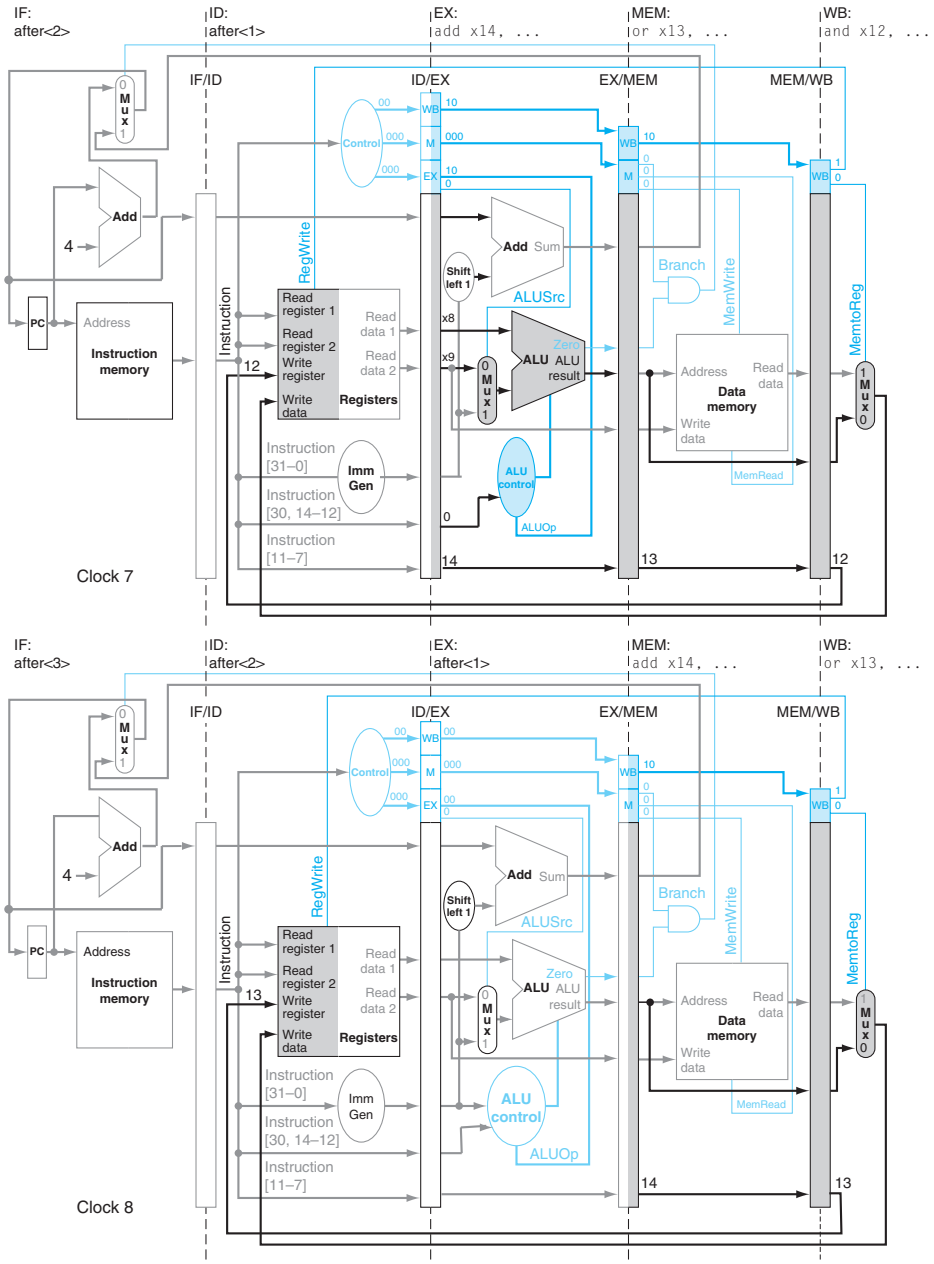


FIGURE e4.13.14 Clock cycles 7 and 8. In the top datapath, the `add` instruction brings up the rear, adding the values corresponding to registers `x8` and `x9` during the EX stage. The result of the `or` instruction is passed from EX/MEM to MEM/WB in the MEM stage, and the WB stage writes the result of the `and` instruction in MEM/WB to register `x12`. Note that the control signals are deasserted (set to 0) in the ID stage, since no instruction is being executed. In the following clock cycle (lower drawing), the WB stage writes the result to register `x13`, thereby completing `or`, and the MEM stage passes the sum from the `add` in EX/MEM to MEM/WB. The instructions after `add` are shown as inactive for pedagogical reasons.

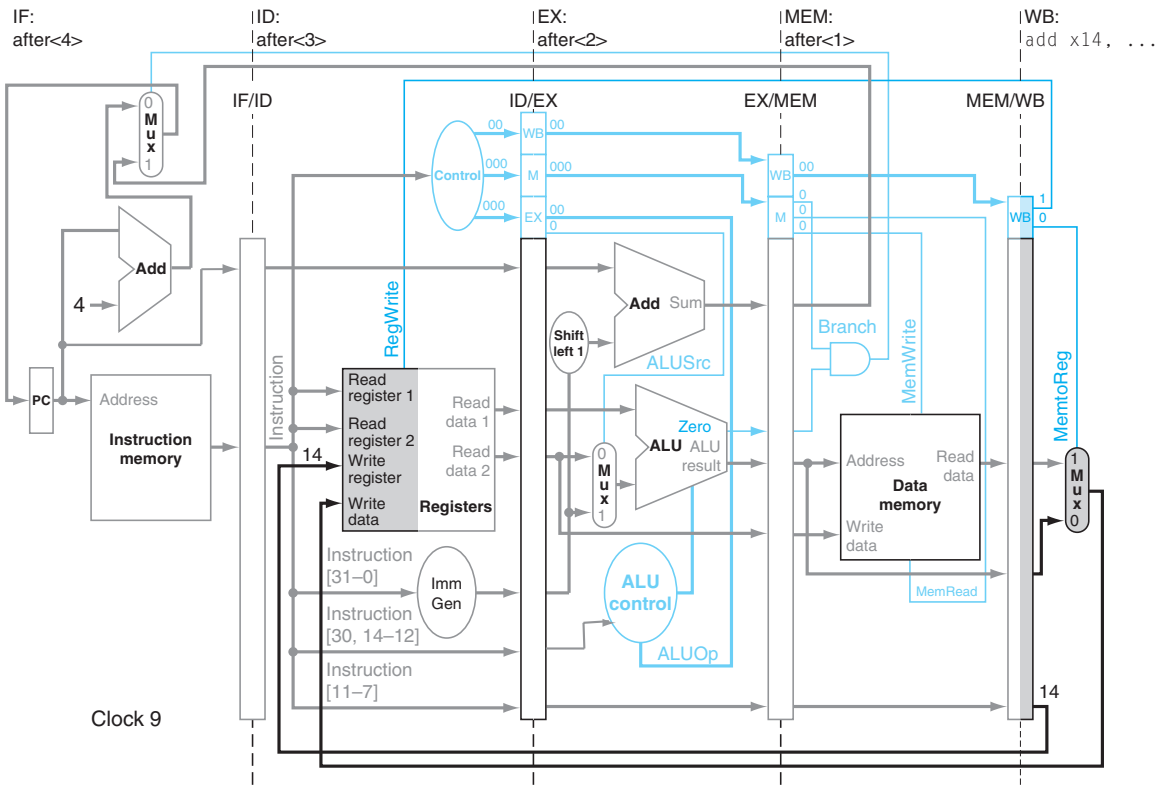


FIGURE e4.13.15 Clock cycle 9. The WB stage writes the ALU result in MEM/WB into register `x14`, completing `add` and the five-instruction sequence. The instructions after `add` are shown as inactive for pedagogical reasons.

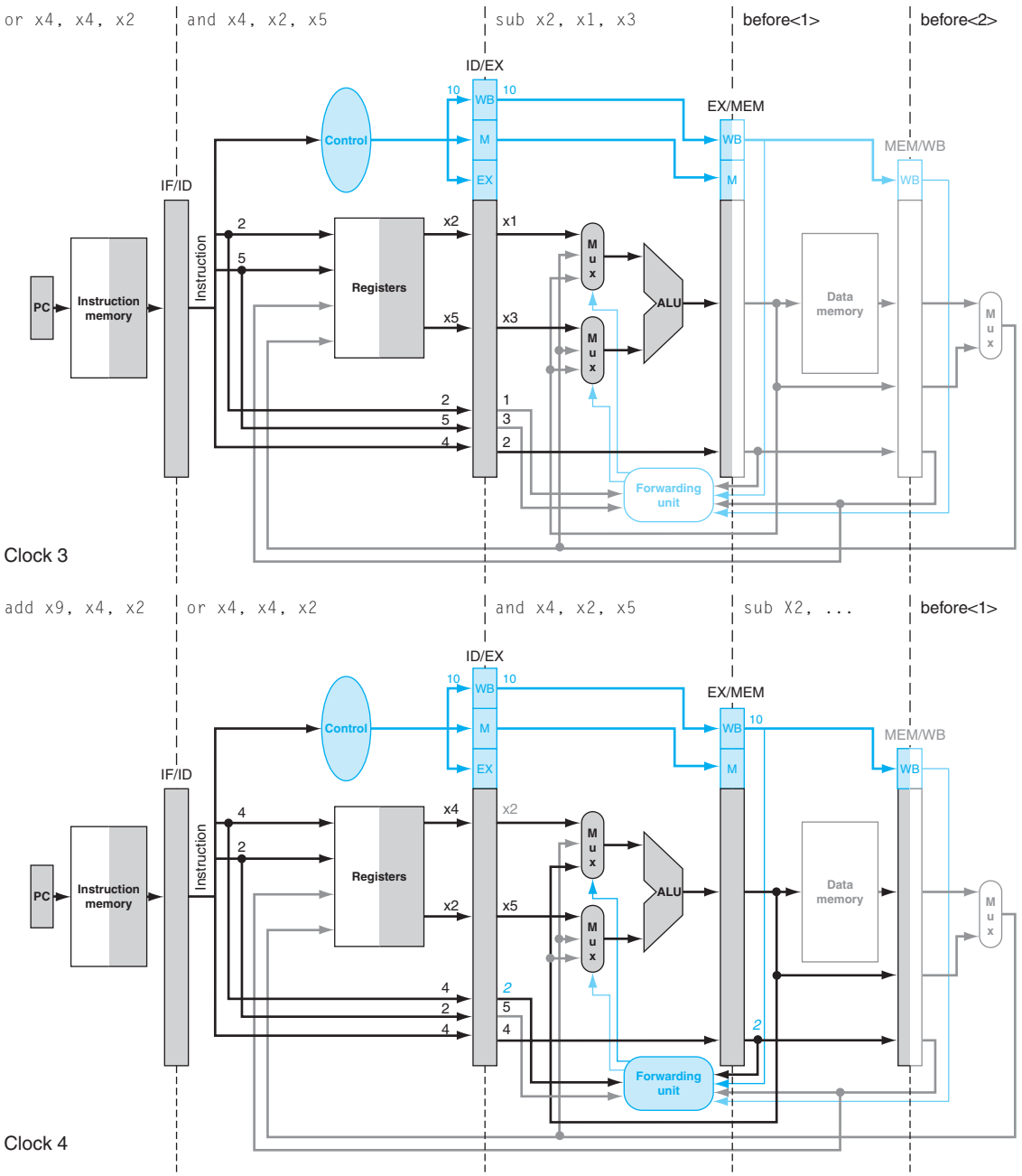


FIGURE e4.13.16 Clock cycles 3 and 4 of the instruction sequence on page 366.e26. The bold lines are those active in a clock cycle, and the italicized register numbers in color indicate a hazard. The forwarding unit is highlighted by shading it when it is forwarding data to the ALU. The instructions before `sub` are shown as inactive just to emphasize what occurs for the four instructions in the example. Operand names are used in EX for control of forwarding; thus they are included in the instruction label for EX. Operand names are not needed in MEM or WB, so … is used. Compare this with Figures e4.13.12 through e4.13.15, which show the datapath without forwarding where ID is the last stage to need operand information.

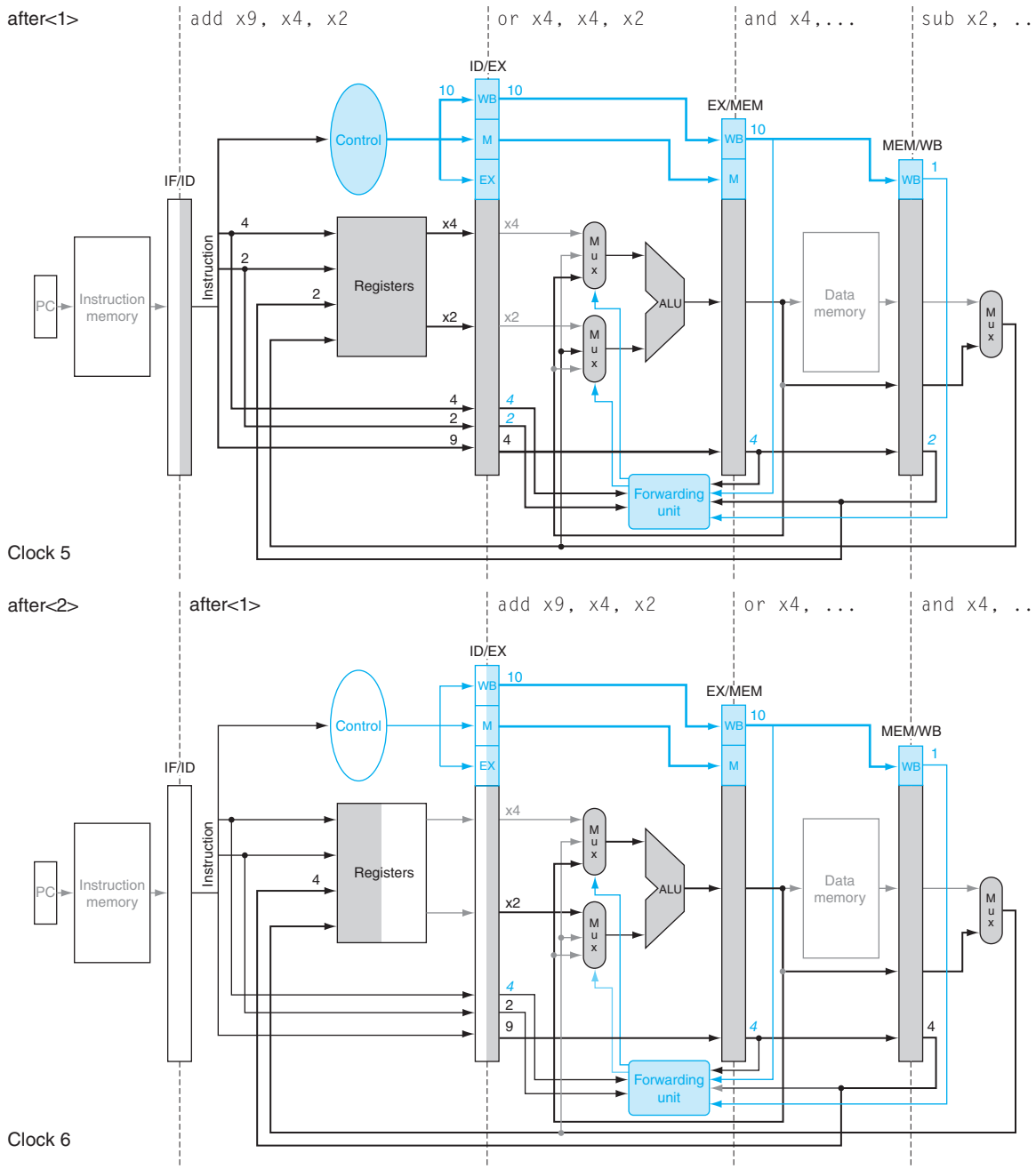


FIGURE e4.13.17 Clock cycles 5 and 6 of the instruction sequence on page 366.e26. The forwarding unit is highlighted when it is forwarding data to the ALU. The two instructions after `add` are shown as inactive just to emphasize what occurs for the four instructions in the example. The bold lines are those active in a clock cycle, and the italicized register numbers in color indicate a hazard.

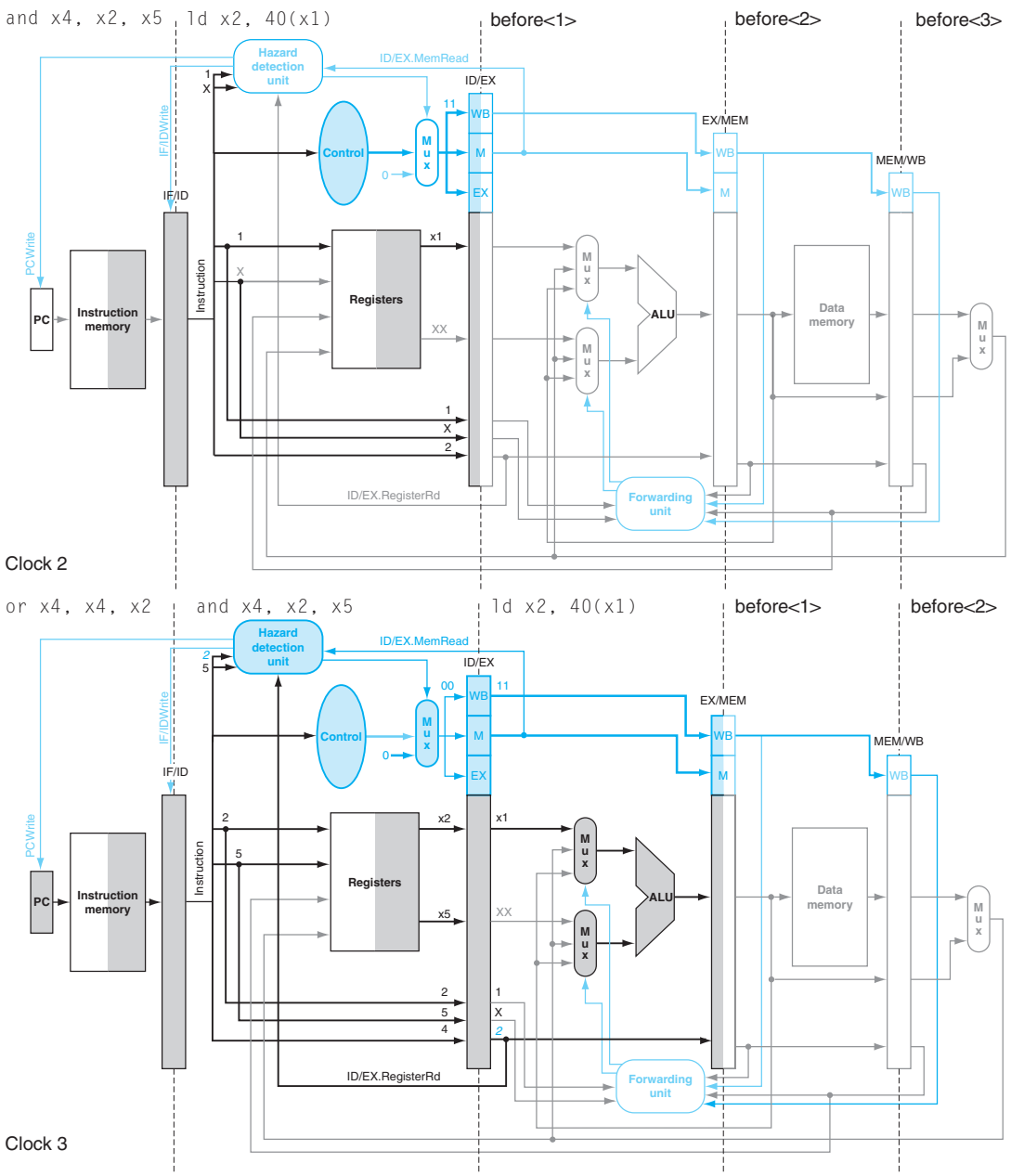


FIGURE e4.13.18 Clock cycles 2 and 3 of the instruction sequence on page 366.e26 with a load replacing `sub`. The bold lines are those active in a clock cycle, the italicized register numbers in color indicate a hazard, and the … in the place of operands means that their identity is information not needed by that stage. The values of the significant control lines, registers, and register numbers are labeled in the figures. The `and` instruction wants to read the value created by the `ld` instruction in clock cycle 3, so the hazard detection unit stalls the `and` and `or` instructions. Hence, the hazard detection unit is highlighted.

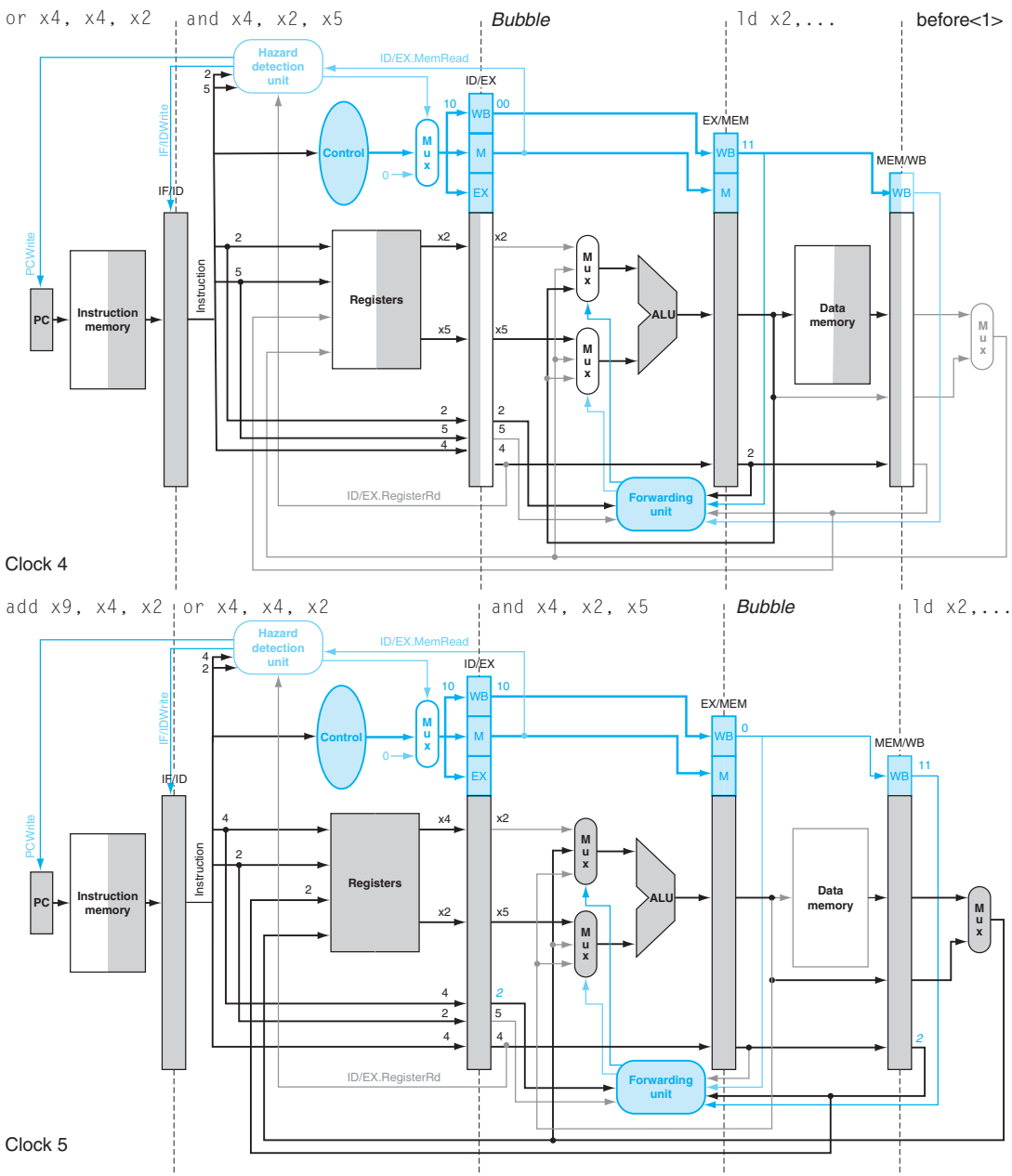


FIGURE e4.13.19 Clock cycles 4 and 5 of the instruction sequence on page 366.e26 with a load replacing `sub`. The bubble is inserted in the pipeline in clock cycle 4, and then the `and` instruction is allowed to proceed in clock cycle 5. The forwarding unit is highlighted in clock cycle 5 because it is forwarding data from `ld` to the ALU. Note that in clock cycle 4, the forwarding unit forwards the address of the `ld` as if it were the contents of register `x2`; this is rendered harmless by the insertion of the bubble. The bold lines are those active in a clock cycle, and the italicized register numbers in color indicate a hazard.

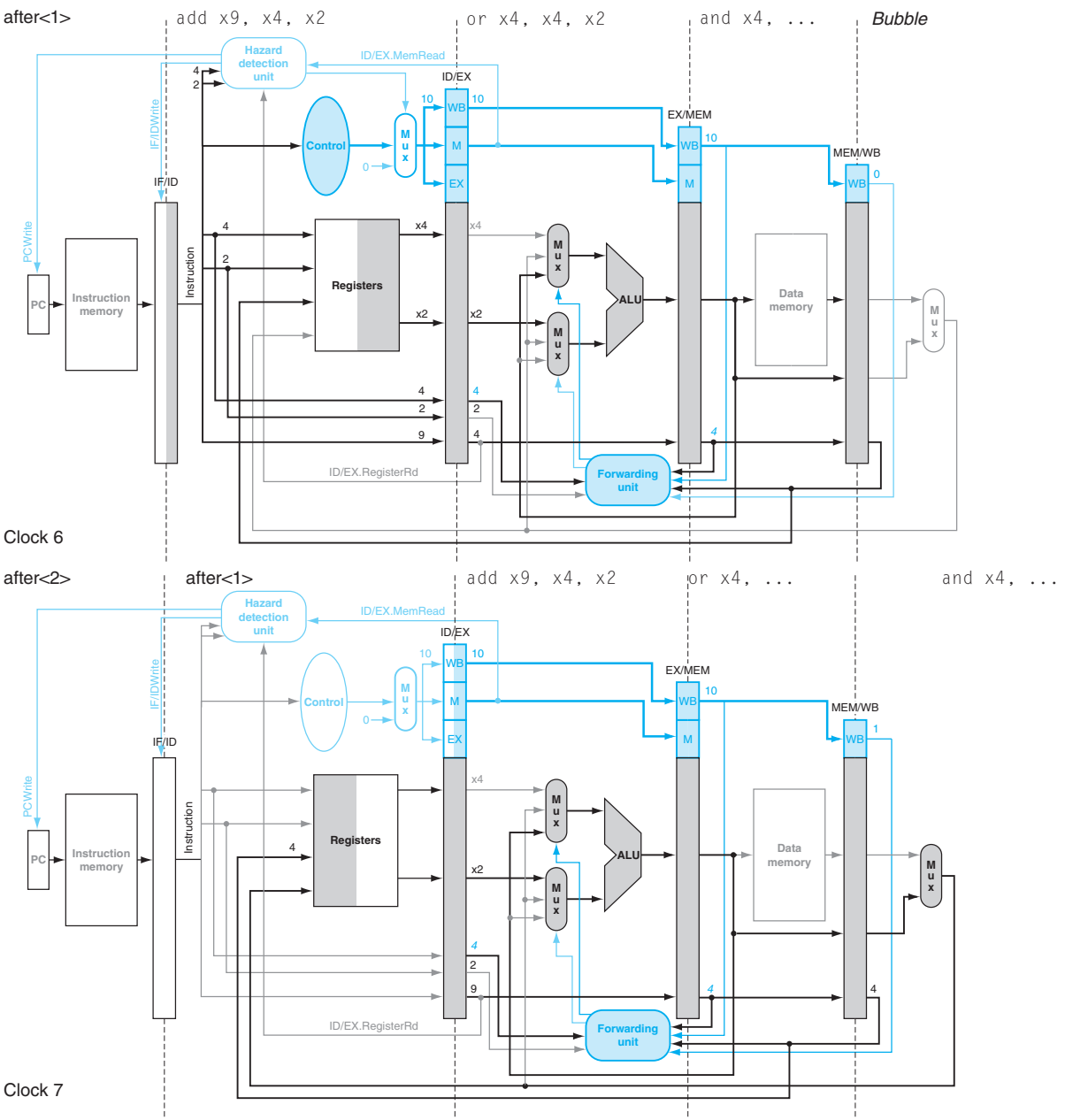


FIGURE e4.13.20 Clock cycles 6 and 7 of the instruction sequence on page 366.e26 with a load replacing `sub`. Note that unlike in Figure e4.13.17, the stall allows the `ld` to complete, and so there is no forwarding from MEM/WB in clock cycle 6. Register `x4` for the `add` in the EX stage still depends on the result from or in EX/MEM, so the forwarding unit passes the result to the ALU. The bold lines show ALU input lines active in a clock cycle, and the italicized register numbers indicate a hazard. The instructions after `add` are shown as inactive for pedagogical reasons.

Many of the difficulties of pipelining arise because of instruction set complications. Here are some examples:

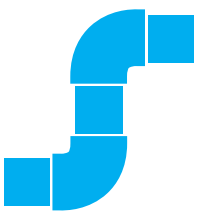
- Widely variable instruction lengths and running times can lead to imbalance among pipeline stages and severely complicate hazard detection in a design pipelined at the instruction set level. This problem was overcome, initially in the DEC VAX 8500 in the late 1980s, using the micro-operations and micropipelined 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 instructions, the VAX on average executes 2.7 times as many clock cycles, so the MIPS is faster.

Nine-tenths of wisdom consists of being wise in time.

American proverb

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 an RISC-V 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

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).

instruction latency The inherent execution time for an instruction.

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 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 was 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, rs1, rs2`

Interpretation: `Reg[rd] = Reg[rs1] AND Reg[rs2]`

4.1.1 [5] <§4.3> What are the values of control signals generated by the control in Figure 4.10 for this instruction?



Historical Perspective and Further Reading

supercomputer: *Any machine still on the drawing board.*

Stan Kelly-Bootle, *The Devil's DP Dictionary*, 1981

This section discusses the history of the original 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.

It is generally agreed that one of the first general-purpose pipelined computers was Stretch, the IBM 7030 (Figure e4.16.1). Stretch followed the IBM 704 and had a goal of being 100 times faster than the 704. The goals were a "stretch" of the state of the art at that time—hence the nickname. The plan was to obtain a factor of 1.6 from overlapping fetch, decode, and execute by using a four-stage pipeline. Apparently, the rest was to come from much more hardware and faster logic. Stretch was also a training ground for both the architects of the IBM 360, Gerrit Blaauw and Fred Brooks, Jr., and the architect of the IBM RS/6000, John Cocke.



FIGURE e4.16.1 The Stretch computer, one of the first pipelined computers.

Control Data Corporation (CDC) delivered what is considered to be the first supercomputer, the CDC 6600, in 1964 (Figure e4.16.2). The core instructions of Cray's subsequent computers have many similarities to those of the original CDC 6600. The CDC 6600 was unique in many ways. The interaction between pipelining and instruction set design was understood, and the instruction set was kept simple to promote pipelining. The CDC 6600 also used an advanced packaging technology. James Thornton's book [1970] provides an excellent description of the entire computer, from technology to architecture, and includes a foreword by Seymour Cray. (Unfortunately, this book is currently out of print.) Jim Smith, then working at CDC, developed the original 2-bit branch prediction scheme and explored several techniques for enhancing instruction issue for the CDC Cyber 180/990. Cray, Thornton, and Smith have each won the ACM Eckert-Mauchly Award (in 1989, 1994, and 1999, respectively).

The IBM 360/91 introduced many new concepts, including dynamic detection of memory hazards, generalized forwarding, and reservation stations (Figure e4.16.3). The approach is normally named *Tomasulo's algorithm*, after an engineer who worked on the project. The team that created the 360/91 was led by Michael Flynn, who was given the 1992 ACM Eckert-Mauchly Award, in part for his contributions to the IBM 360/91; in 1997, the same award went to Robert Tomasulo for his pioneering work on out-of-order processing.

The internal organization of the 360/91 shares many features with the Pentium III and Pentium 4, as well as with several other microprocessors. One major



FIGURE e4.16.2 The CDC 6600, the first supercomputer.



FIGURE e4.16.3 The IBM 360/91 pushed the state of the art in pipelined execution when it was unveiled in 1966.

difference was that there was no branch prediction in the 360/91 and hence no speculation. Another major difference was that there was no commit unit, so once the instructions finished execution, they updated the registers. Out-of-order instruction commit led to *imprecise interrupts*, which proved to be unpopular and led to the commit units in dynamically scheduled pipelined processors since that time. Although the 360/91 was not a success, its key ideas were resurrected later and exist in some form in the majority of microprocessors of the last decade.

Improving Pipelining Effectiveness and Adding Multiple Issue

The RISC processors refined the notion of compiler-scheduled pipelines in the early 1980s. The concepts of delayed branches and delayed loads—common in microprogramming—were extended into the high-level architecture. In fact, the Stanford processor that led to the commercial MIPS architecture was called "Microprocessor without Interlocked Pipelined Stages" because it was up to the assembler or compiler to avoid data hazards.

In addition to its contribution to the development of the RISC concepts, IBM did pioneering work on multiple issue. In the 1960s, a project called ACS was underway. It included multiple-instruction issue concepts and the notion of integrated compiler and architecture design, but it never reached product stage. The earliest proposal for a superscalar processor that dynamically makes issue decisions was

by John Cocke; he described the key ideas in several talks in the mid-1980s and, with Tilak Agarwala, coined the name *superscalar*. This original design was a two-issue machine named Cheetah, which was followed by a more widely discussed four-issue machine named America. The IBM Power-1 architecture, used in the RS/6000 line, is based on these ideas, and the PowerPC is a variation of the Power-1 architecture. Cocke won the Turing Award, the highest award in computer science and engineering, for his architecture work.

Static multiple issue, as exemplified by the *long instruction word* (LIW) or sometimes *very long instruction word* (VLIW) approaches, appeared in real designs before the superscalar approach. In fact, the earliest multiple-issue machines were special-purpose attached processors designed for scientific applications. Culler Scientific and Floating Point Systems were two of the most prominent manufacturers of such computers. Another inspiration for the use of multiple operations per instruction came from those working on microcode compilers. Such inspiration led to a research project at Yale led by Josh Fisher, who coined the term VLIW. Cydrome and Multiflow were two early companies involved in building mini-supercomputers using processors with multiple-issue capability. These processors, built with bit-slice and multiple-chip gate array implementations, arrived on the market at the same time as the initial RISC microprocessors. Despite some promising performance on high-end scientific codes, the much better cost/performance of the microprocessor-based computers doomed the first generation of VLIW computers. Bob Rau and Josh Fisher won the Eckert-Mauchly Award in 2002 and 2003, respectively, for their contributions to the development of multiple processors and software techniques to exploit ILP.

The very beginning of the 1990s saw the first superscalar processors using static scheduling and no speculation, including versions of the MIPS and PowerPC architectures. The early 1990s also saw important research at a number of universities, including Wisconsin, Stanford, Illinois, and Michigan, focused on techniques for exploiting additional ILP through multiple issue with and without speculation. These research insights were used to build dynamically scheduled, speculative processors, including the Motorola 88110, MIPS R10000, DEC Alpha 21264, PowerPC 603, and the Intel Pentium Pro, Pentium III, and Pentium 4.

In 2001, Intel introduced the IA-64 architecture and its first implementation, Itanium. Itanium represented a return to a more compiler-intensive approach that they called EPIC. EPIC represented a considerable enhancement over the early VLIW architectures, removing many of their drawbacks. It has had modest sales. In 2013, the IA-64 architecture is used only in low-volume, high-end servers and is outnumbered by x86 processors by more than 100:1.

Compiler Technology for Exploiting ILP

Successful development of processors to exploit ILP has depended on progress in compiler technology. The concept of loop unrolling was understood early, and a number of companies and researchers—including Floating Point Systems, Cray, and the Stanford MIPS project—developed compilers that made use of loop

unrolling and pipeline scheduling to improve instruction throughput. A special-purpose processor called WARP, designed at Carnegie Mellon University, inspired the development of software pipelining, an approach that symbolically unrolls loops.

To exploit higher levels of ILP, more aggressive compiler technology was needed. The VLIW project at Yale developed the concept of trace scheduling that Multiflow implemented in their compilers. Trace scheduling relies on aggressive loop unrolling and path prediction to compile favored execution traces efficiently. The Cydrome designers created early versions of predication and support for software pipelining. Hwu at Illinois worked on extended versions of loop unrolling, called *superblocks*, and techniques for compiling with predication. The concepts from Multiflow, Cydrome, and the research group at Illinois served as the architectural and compiler basis for the IA-64 architecture.

Further Reading

Bhandarkar, D. and D.W. Clark [1991]. "Performance from architecture: Comparing a RISC and a CISC with similar hardware organizations," *Proc. Fourth Conf. on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (April), Palo Alto, CA, 310–19.

A quantitative comparison of RISC and CISC written by scholars who argued for CISCs as well as built them; they conclude that MIPS is between 2 and 4 times faster than a VAX built with similar technology, with a mean of 2.7.

Fisher, J.A. and B.R. Rau [1993]. *Journal of Supercomputing* (January), Kluwer.

This entire issue is devoted to the topic of exploiting ILP. It contains papers on both the architecture and software and is a wonderful source for further references.

Hennessy, J. L. and D. A. Patterson [2001]. *Computer Architecture: A Quantitative Approach*, fourth edition, Morgan Kaufmann, San Francisco.

Chapter 2 and Appendix A go into considerably more detail about pipelined processors (almost 200 pages), including superscalar processors and VLIW processors. Appendix G describes Itanium.

Jouppi, N.P. and D.W. Wall [1989]. "Available instruction-level parallelism for superscalar and superpipelined processors," *Proc. Third Conf. on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (April), Boston, 272–82.

A comparison of deeply pipelined (also called superpipelined) and superscalar systems.

Kogge, P. M. [1981]. *The Architecture of Pipelined Computers*, McGraw-Hill, New York.

A formal text on pipelined control, with emphasis on underlying principles.

Russell, R. M. [1978]. "The CRAY-1 computer system", *Comm. of the ACM* 21:1 (January), 63–72.

A short summary of a classic computer that uses vectors of operations to remove pipeline stalls.

Smith, A. and J. Lee [1984]. "Branch prediction strategies and branch target buffer design", *Computer* 17:1 (January), 6–22.

An early survey on branch prediction.

Smith, J. E. and A. R. Plezkun [1988]. "Implementing precise interrupts in pipelined processors", *IEEE Trans. on Computers* 37:5 (May), 562–73.

Covers the difficulties in interrupting pipelined computers.

Thornton, J. E. [1970]. *Design of a Computer. The Control Data 6600*, Glenview, IL: Scott, Foresman.

A classic book describing a classic computer, considered the first supercomputer.

4.1.2 [5] <§4.3> Which resources (blocks) perform a useful function for this instruction?

4.1.3 [10] <§4.3> Which resources (blocks) produce no output for this instruction? Which resources produce output that is not used?

4.2 [10] <§4.4> Explain each of the "don't cares" in Figure 4.18.

4.3 Consider the following instruction mix:

R-type	I-type (non-ld)	Load	Store	Branch	Jump
24%	28%	25%	10%	11%	2%

4.3.1 [5] <§4.4> What fraction of all instructions use data memory?

4.3.2 [5] <§4.4> What fraction of all instructions use instruction memory?

4.3.3 [5] <§4.4> What fraction of all instructions use the sign extend?

4.3.4 [5] <§4.4> What is the sign extend doing during cycles in which its output is not needed?

4.4 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 signal wire to get "broken" and always register a logical 0. This is often called a "stuck-at-0" fault.

4.4.1 [5] <§4.4> Which instructions fail to operate correctly if the `MemToReg` wire is stuck at 0?

4.4.2 [5] <§4.4> Which instructions fail to operate correctly if the `ALUSrc` wire is stuck at 0?

4.5 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: `0x00c6ba23`.

4.5.1 [10] <§4.4> What are the values of the ALU control unit's inputs for this instruction?

4.5.2 [5] <§4.4> What is the new PC address after this instruction is executed? Highlight the path through which this value is determined.

4.5.3 [10] <§4.4> For each mux, show the values of its inputs and outputs during the execution of this instruction. List values that are register outputs at `Reg [xn]`.

4.5.4 [10] <§4.4> What are the input values for the ALU and the two add units?

4.5.5 [10] <§4.4> What are the values of all inputs for the registers unit?

4.6 Section 4.4 does not discuss I-type instructions like `addi` or `andi`.

4.6.1 [5] <§4.4> What additional logic blocks, if any, are needed to add I-type instructions to the CPU shown in Figure 4.21? Add any necessary logic blocks to Figure 4.21 and explain their purpose.

4.6.2 [10] <§4.4> List the values of the signals generated by the control unit for `addi`. Explain the reasoning for any "don't care" control signals.

4.7 Problems in this exercise assume that the logic blocks used to implement a processor's datapath have the following latencies:

I-Mem / D-Mem	Register File	Mux	ALU	Adder	Single gate	Register Read	Register Setup	Sign extend	Control
250 ps	150 ps	25 ps	200 ps	150 ps	5 ps	30 ps	20 ps	50 ps	50 ps

"Register read" is the time needed after the rising clock edge for the new register value to appear on the output. This value applies to the PC only. "Register setup" is the amount of time a register's data input must be stable before the rising edge of the clock. This value applies to both the PC and Register File.

4.7.1 [5] <§4.4> What is the latency of an R-type instruction (i.e., how long must the clock period be to ensure that this instruction works correctly)?

4.7.2 [10] <§4.4> What is the latency of `ld`? (Check your answer carefully. Many students place extra muxes on the critical path.)

4.7.3 [10] <§4.4> What is the latency of `sd`? (Check your answer carefully. Many students place extra muxes on the critical path.)

4.7.4 [5] <§4.4> What is the latency of `beq`?

4.7.5 [5] <§4.4> What is the latency of an I-type instruction?

4.7.6 [5] <§4.4> What is the minimum clock period for this CPU?

4.8 [10] <§4.4> Suppose you could build a CPU where the clock cycle time was different for each instruction. What would the speedup of this new CPU be over the CPU presented in Figure 4.21 given the instruction mix below?

R-type/I-type (non-ld)	ld	sd	beq
52%	25%	11%	12%

4.9 Consider the addition of a multiplier to the CPU shown in Figure 4.21. This addition will add 300 ps to the latency of the ALU, but will reduce the number of instructions by 5% (because there will no longer be a need to emulate the multiply instruction).

4.9.1 [5] <§4.4> What is the clock cycle time with and without this improvement?

4.9.2 [10] <§4.4> What is the speedup achieved by adding this improvement?

4.9.3 [10] <§4.4> What is the slowest the new ALU can be and still result in improved performance?

4.10 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 beginning with the datapath from Figure 4.21, the latencies from Exercise 4.7, and the following costs:

I-Mem	Register File	Mux	ALU	Adder	D-Mem	Single Register	Sign extend	Single gate	Control
1000	200	10	100	30	2000	5	100	1	500

Suppose doubling the number of general purpose registers from 32 to 64 would reduce the number of `ld` and `sd` instruction by 12%, but increase the latency of the register file from 150 ps to 160 ps and double the cost from 200 to 400. (Use the instruction mix from Exercise 4.8 and ignore the other effects on the ISA discussed in Exercise 2.18.)

4.10.1 [5] <§4.4> What is the speedup achieved by adding this improvement?

4.10.2 [10] <§4.4> Compare the change in performance to the change in cost.

4.10.3 [10] <§4.4> Given the cost/performance ratios you just calculated, describe a situation where it makes sense to add more registers and describe a situation where it doesn't make sense to add more registers.

4.11 Examine the difficulty of adding a proposed `lwi.d rd, rs1, rs2` ("*L*oad *W*ith *I*ncrement") instruction to RISC-V.

Interpretation: `Reg[rd]=Mem[Reg[rs1]+Reg[rs2]]`

4.11.1 [5] <§4.4> Which new functional blocks (if any) do we need for this instruction?

4.11.2 [5] <§4.4> Which existing functional blocks (if any) require modification?

4.11.3 [5] <§4.4> Which new data paths (if any) do we need for this instruction?

4.11.4 [5] <§4.4> What new signals do we need (if any) from the control unit to support this instruction?

4.12 Examine the difficulty of adding a proposed `swap rs1, rs2` instruction to RISC-V.

Interpretation: `Reg[rs2]=Reg[rs1]; Reg[rs1]=Reg[rs2]`

4.12.1 [5] <§4.4> Which new functional blocks (if any) do we need for this instruction?

4.12.2 [10] <§4.4> Which existing functional blocks (if any) require modification?

4.12.3 [5] <§4.4> What new data paths do we need (if any) to support this instruction?

4.12.4 [5] <§4.4> What new signals do we need (if any) from the control unit to support this instruction?

4.12.5 [5] <§4.4> Modify Figure 4.21 to demonstrate an implementation of this new instruction.

4.13 Examine the difficulty of adding a proposed `ss rs1, rs2, imm` (Store Sum) instruction to RISC-V.

Interpretation: `Mem[Reg[rs1]]=Reg[rs2]+immediate`

4.13.1 [10] <§4.4> Which new functional blocks (if any) do we need for this instruction?

4.13.2 [10] <§4.4> Which existing functional blocks (if any) require modification?

4.13.3 [5] <§4.4> What new data paths do we need (if any) to support this instruction?

4.13.4 [5] <§4.4> What new signals do we need (if any) from the control unit to support this instruction?

4.13.5 [5] <§4.4> Modify Figure 4.21 to demonstrate an implementation of this new instruction.

4.14 [5] <§4.4> For which instructions (if any) is the Imm Gen block on the critical path?

4.15 `ld` is the instruction with the longest latency on the CPU from Section 4.4. If we modified `ld` and `sd` so that there was no offset (i.e., the address to be loaded from/stored to must be calculated and placed in `rs1` before calling `ld/sd`), then no instruction would use both the ALU and Data memory. This would allow us to reduce the clock cycle time. However, it would also increase the number of instructions, because many `ld` and `sd` instructions would need to be replaced with `ld/add` or `sd/add` combinations.

4.15.1 [5] <§4.4> What would the new clock cycle time be?

4.15.2 [10] <§4.4> Would a program with the instruction mix presented in Exercise 4.7 run faster or slower on this new CPU? By how much? (For simplicity, assume every `ld` and `sd` instruction is replaced with a sequence of two instructions.)

4.15.3 [5] <§4.4> What is the primary factor that influences whether a program will run faster or slower on the new CPU?

4.15.4 [5] <§4.4> Do you consider the original CPU (as shown in Figure 4.21) a better overall design; or do you consider the new CPU a better overall design? Why?

4.16 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
250 ps	350 ps	150 ps	300 ps	200 ps

Also, assume that instructions executed by the processor are broken down as follows:

ALU/Logic	Jump/Branch	Load	Store
45%	20%	20%	15%

4.16.1 [5] <§4.5> What is the clock cycle time in a pipelined and non-pipelined processor?

4.16.2 [10] <§4.5> What is the total latency of an `ld` instruction in a pipelined and non-pipelined processor?

4.16.3 [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?

4.16.4 [10] <§4.5> Assuming there are no stalls or hazards, what is the utilization of the data memory?

4.16.5 [10] <§4.5> Assuming there are no stalls or hazards, what is the utilization of the write-register port of the "Registers" unit?

4.17 [10] <§4.5> What is the minimum number of cycles needed to completely execute `n` instructions on a CPU with a `k` stage pipeline? Justify your formula.

4.18 [5] <§4.5> Assume that `x11` is initialized to 11 and `x12` is initialized to 22. Suppose you executed the code below on a version of the pipeline from Section 4.5 that does not handle data hazards (i.e., the programmer is responsible for

addressing data hazards by inserting NOP instructions where necessary). What would the final values of registers x13 and x14 be?

```
addi   x11, x12, 5
add    x13, x11, x12
addi   x14, x11, 15
```

4.19 [10] <§4.5> Assume that x11 is initialized to 11 and x12 is initialized to 22. Suppose you executed the code below on a version of the pipeline from Section 4.5 that does not handle data hazards (i.e., the programmer is responsible for addressing data hazards by inserting NOP instructions where necessary). What would the final values of register x15 be? Assume the register file is written at the beginning of the cycle and read at the end of a cycle. Therefore, an ID stage will return the results of a WB state occurring during the same cycle. See Section 4.7 and Figure 4.51 for details.

```
addi   x11, x12, 5
add    x13, x11, x12
addi   x14, x11, 15
add    x15, x11, x11
```

4.20 [5] <§4.5> Add NOP instructions to the code below so that it will run correctly on a pipeline that does not handle data hazards.

```
addi   x11, x12, 5
add    x13, x11, x12
addi   x14, x11, 15
add    x15, x13, x12
```

4.21 Consider a version of the pipeline from Section 4.5 that does not handle data hazards (i.e., the programmer is responsible for addressing data hazards by inserting NOP instructions where necessary). Suppose that (after optimization) a typical n-instruction program requires an additional 4*n NOP instructions to correctly handle data hazards.

4.21.1 [5] <§4.5> Suppose that the cycle time of this pipeline without forwarding is 250 ps. Suppose also that adding forwarding hardware will reduce the number of NOPs from .4*n to .05*n, but increase the cycle time to 300 ps. What is the speedup of this new pipeline compared to the one without forwarding?

4.21.2 [10] <§4.5> Different programs will require different amounts of NOPs. How many NOPs (as a percentage of code instructions) can remain in the typical program before that program runs slower on the pipeline with forwarding?

4.21.3 [10] <§4.5> Repeat 4.21.2; however, this time let x represent the number of NOP instructions relative to n. (In 4.21.2, x was equal to .4.) Your answer will be with respect to x.

4.21.4 [10] <§4.5> Can a program with only `.075*n NOPs` possibly run faster on the pipeline with forwarding? Explain why or why not.

4.21.5 [10] <§4.5> At minimum, how many `NOP`s (as a percentage of code instructions) must a program have before it can possibly run faster on the pipeline with forwarding?

4.22 [5] <§4.5> Consider the fragment of RISC-V assembly below:

```
sd   x29, 12(x16)
ld   x29, 8(x16)
sub  x17, x15, x14
beqz x17, label
add  x15, x11, x14
sub  x15, x30, x14
```

Suppose we modify the pipeline so that it has only one memory (that handles both instructions and data). In this case, there will be a structural hazard every time a program needs to fetch an instruction during the same cycle in which another instruction accesses data.

4.22.1 [5] <§4.5> Draw a pipeline diagram to show were the code above will stall.

4.22.2 [5] <§4.5> In general, is it possible to reduce the number of stalls/`NOP`s resulting from this structural hazard by reordering code?

4.22.3 [5] <§4.5> Must this structural hazard be handled in hardware? We have seen that data hazards can be eliminated by adding `NOP`s to the code. Can you do the same with this structural hazard? If so, explain how. If not, explain why not.

4.22.4 [5] <§4.5> Approximately how many stalls would you expect this structural hazard to generate in a typical program? (Use the instruction mix from Exercise 4.8.)

4.23 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. (See Exercise 4.15.) As a result, the MEM and EX stages can be overlapped and the pipeline has only four stages.

4.23.1 [10] <§4.5> How will the reduction in pipeline depth affect the cycle time?

4.23.2 [5] <§4.5> How might this change improve the performance of the pipeline?

4.23.3 [5] <§4.5> How might this change degrade the performance of the pipeline?

4.24 [10] <§4.7> Which of the two pipeline diagrams below better describes the operation of the pipeline's hazard detection unit? Why?

Choice 1:

```
ld x11, 0(x12):     IF ID EX ME WB
add x13, x11, x14:   IF ID EX..ME WB
or x15, x16, x17:     IF ID..EX ME WB
```

Choice 2:

```
ld x11, 0(x12):   IF ID EX ME WB
add x13, x11, x14:  IF ID..EX ME WB
or x15, x16, x17:    IF..ID EX ME WB
```

4.25 Consider the following loop.

```
LOOP: ld    x10, 0(x13)
      ld    x11, 8(x13)
      add   x12, x10, x11
      subi  x13, x13, 16
      bnez  x12, LOOP
```

Assume that perfect branch prediction is used (no stalls due to control hazards), that there are no delay slots, that the pipeline has full forwarding support, and that branches are resolved in the EX (as opposed to the ID) stage.

4.25.1 [10] <§4.7> Show a pipeline execution diagram for the first two iterations of this loop.

4.25.2 [10] <§4.7> Mark pipeline stages that do not perform useful work. How often while the pipeline is full do we have a cycle in which all five pipeline stages are doing useful work? (Begin with the cycle during which the `subi` is in the IF stage. End with the cycle during which the `bnez` is in the IF stage.)

4.26 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 4.53. These problems assume that, of all the instructions executed in a processor, the following fraction of these instructions has 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 next 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. We also assume that branches are resolved in the EX stage (as opposed to the ID stage), and that the CPI of the processor is 1 if there are no data hazards.

EX to 1st Only	MEM to 1st Only	EX to 2nd Only	MEM to 2nd Only	EX to 1st and EX to 2nd
5%	20%	5%	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
120 ps	100 ps	110 ps	130 ps	120 ps	120 ps	120 ps	100 ps

4.26.1 [5] <§4.7> For each RAW dependency listed above, give a sequence of at least three assembly statements that exhibits that dependency.

4.26.2 [5] <§4.7> For each RAW dependency above, how many `NOP`s would need to be inserted to allow your code from 4.26.1 to run correctly on a pipeline with no forwarding or hazard detection? Show where the `NOP`s could be inserted.

4.26.3 [10] <§4.7> Analyzing each instruction independently will over-count the number of `NOP`s needed to run a program on a pipeline with no forwarding or hazard detection. Write a sequence of three assembly instructions so that, when you consider each instruction in the sequence independently, the sum of the stalls is larger than the number of stalls the sequence actually needs to avoid data hazards.

4.26.4 [5] <§4.7> Assuming no other hazards, what is the CPI for the program described by the table above when run on a pipeline with no forwarding? What percent of cycles are stalls? (For simplicity, assume that all necessary cases are listed above and can be treated independently.)

4.26.5 [5] <§4.7> What is the CPI if we use full forwarding (forward all results that can be forwarded)? What percent of cycles are stalls?

4.26.6 [10] <§4.7> Let us assume that we cannot afford to have three-input multiplexors 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). What is the CPI for each option?

4.26.7 [5] <§4.7> For the given hazard probabilities and pipeline stage latencies, what is the speedup achieved by each type of forwarding (EX/MEM, MEM/WB, for full) as compared to a pipeline that has no forwarding?

4.26.8 [5] <§4.7> What would be the additional speedup (relative to the fastest processor from 4.26.7) be 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.

4.26.9 [5] <§4.7> The table of hazard types has separate entries for "EX to 1st" and "EX to 1st and EX to 2nd". Why is there no entry for "MEM to 1st and MEM to 2nd"?

4.27 Problems in this exercise refer to the following sequence of instructions, and assume that it is executed on a five-stage pipelined datapath:

```
add  x15, x12, x11
ld   x13, 4(x15)
ld   x12, 0(x2)
or   x13, x15, x13
sd   x13, 0(x15)
```

4.27.1 [5] <§4.7> If there is no forwarding or hazard detection, insert `NOP`s to ensure correct execution.

4.27.2 [10] <§4.7> Now, change and/or rearrange the code to minimize the number of `NOP`s needed. You can assume register `x17` can be used to hold temporary values in your modified code.

4.27.3 [10] <§4.7> If the processor has forwarding, but we forgot to implement the hazard detection unit, what happens when the original code executes?

4.27.4 [20] <§4.7> If there is forwarding, for the first seven cycles during the execution of this code, specify which signals are asserted in each cycle by hazard detection and forwarding units in Figure 4.59.

4.27.5 [10] <§4.7> If there is no forwarding, what new input and output signals do we need for the hazard detection unit in Figure 4.59? Using this instruction sequence as an example, explain why each signal is needed.

4.27.6 [20] <§4.7> For the new hazard detection unit from 4.26.5, specify which output signals it asserts in each of the first five cycles during the execution of this code.

4.28 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	beqz/bnez	jal	ld	sd
40%	25%	5%	25%	5%

Also, assume the following branch predictor accuracies:

Always-Taken	Always-Not-Taken	2-Bit
45%	55%	85%

4.28.1 [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 ID stage and applied in the EX stage that there are no data hazards, and that no delay slots are used.

4.28.2 [10] <§4.8> Repeat 4.28.1 for the "always-not-taken" predictor.

4.28.3 [10] <§4.8> Repeat 4.28.1 for the 2-bit predictor.

4.28.4 [10] <§4.8> With the 2-bit predictor, what speedup would be achieved if we could convert half of the branch instructions to some ALU instruction? Assume that correctly and incorrectly predicted instructions have the same chance of being replaced.

4.28.5 [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.

4.28.6 [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?

4.29 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`.

4.29.1 [5] <§4.8> What is the accuracy of always-taken and always-not-taken predictors for this sequence of branch outcomes?

4.29.2 [5] <§4.8> What is the accuracy of the 2-bit predictor for the first four branches in this pattern, assuming that the predictor starts off in the bottom left state from Figure 4.61 (predict not taken)?

4.29.3 [10] <§4.8> What is the accuracy of the 2-bit predictor if this pattern is repeated forever?

4.29.4 [30] <§4.8> Design a predictor that would achieve a perfect accuracy if this pattern is repeated forever. You 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.

4.29.5 [10] <§4.8> What is the accuracy of your predictor from 4.29.4 if it is given a repeating pattern that is the exact opposite of this one?

4.29.6 [20] <§4.8> Repeat 4.29.4, 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.

4.30 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
beqz x11, LABEL	ld x11, 0(x12)

4.30.1 [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.

4.30.2 [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.

4.30.3 [10] <§4.9> If the second instruction is fetched immediately after the first instruction, describe what happens in the pipeline when the first instruction causes the first exception you listed in Exercise 4.30.1. 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.

4.30.4 [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 Exercise 4.30.3 using this modified pipeline and vectored exception handling.

4.30.5 [15] <§4.9> We want to emulate vectored exception handling (described in Exercise 4.30.4) 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.

4.31 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];
```

A compiler doing little or no optimization might produce the following RISC-V assembly code:

```
        li    x12, 0
        jal   ENT
TOP:    slli  x5, x12, 3
        add   x6, x10, x5
        ld    x7, 0(x6)
        ld    x29, 8(x6)
        sub   x30, x7, x29
        add   x31, x11, x5
        sd    x30, 0(x31)
        addi  x12, x12, 2
ENT:    bne   x12, x13, TOP
```

The code above uses the following registers:

i	j	a	b	Temporary values
x12	x13	x10	x11	x5–x7, x29–x31

Assume the two-issue, statically scheduled processor for this exercise has the following properties:

1. One instruction must be a memory operation; the other must be an arithmetic/logic instruction or a branch.
2. The processor has all possible forwarding paths between stages (including paths to the ID stage for branch resolution).
3. The processor has perfect branch prediction.
4. Two instruction may not issue together in a packet if one depends on the other. (See page 324.)
5. If a stall is necessary, both instructions in the issue packet must stall. (See page 324.)

As you complete these exercises, notice how much effort goes into generating code that will produce a near-optimal speedup.

4.31.1 [30] <§4.10> Draw a pipeline diagram showing how RISC-V code given above executes on the two-issue processor. Assume that the loop exits after two iterations.

4.31.2 [10] <§4.10> What is the speedup of going from a one-issue to a two-issue processor? (Assume the loop runs thousands of iterations.)

4.31.3 [10] <§4.10> Rearrange/rewrite the RISC-V code given above to achieve better performance on the one-issue processor. Hint: Use the instruction "`beqz x13,DONE`" to skip the loop entirely if `j = 0`.

4.31.4 [20] <§4.10> Rearrange/rewrite the RISC-V code given above to achieve better performance on the two-issue processor. (Do not unroll the loop, however.)

4.31.5 [30] <§4.10> Repeat Exercise 4.31.1, but this time use your optimized code from Exercise 4.31.4.

4.31.6 [10] <§4.10> What is the speedup of going from a one-issue processor to a two-issue processor when running the optimized code from Exercises 4.31.3 and 4.31.4.

4.31.7 [10] <§4.10> Unroll the RISC-V code from Exercise 4.31.3 so that each iteration of the unrolled loop handles two iterations of the original loop. Then, rearrange/rewrite your unrolled code to achieve better performance on the one-issue processor. You may assume that `j` is a multiple of 4.

4.31.8 [20] <§4.10> Unroll the RISC-V code from Exercise 4.31.4 so that each iteration of the unrolled loop handles two iterations of the original loop. Then, rearrange/rewrite your unrolled code to achieve better performance on the two-issue processor. You may assume that `j` is a multiple of 4. (Hint: Re-organize the loop so that some calculations appear both outside the loop and at the end of the loop. You may assume that the values in temporary registers are not needed after the loop.)

4.31.9 [10] <§4.10> What is the speedup of going from a one-issue processor to a two-issue processor when running the unrolled, optimized code from Exercises 4.31.7 and 4.31.8?

4.31.10 [30] <§4.10> Repeat Exercises 4.31.8 and 4.31.9, but this time assume the two-issue processor can run two arithmetic/logic instructions together. (In other words, the first instruction in a packet can be any type of instruction, but the second must be an arithmetic or logic instruction. Two memory operations cannot be scheduled at the same time.)

4.32 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 consume a negligible amount of energy. ("Register Read" and "Register Write" refer to the register file only.)

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
200 ps	150 ps	90 ps	90 ps	250 ps

4.32.1 [5] <§§4.3, 4.6, 4.14> How much energy is spent to execute an `add` instruction in a single-cycle design and in the five-stage pipelined design?

4.32.2 [10] <§§4.6, 4.14> What is the worst-case RISC-V instruction in terms of energy consumption? What is the energy spent to execute it?

4.32.3 [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 `ld` instruction after this change?

4.32.4 [10] <§§4.6, 4.14> What other instructions can potentially benefit from the change discussed in Exercise 4.32.3?

4.32.5 [10] <§§4.6, 4.14> How do your changes from Exercise 4.32.3 affect the performance of a pipelined CPU?

4.32.6 [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. If 25% of instructions are loads, what is the effect of this change on clock frequency and energy consumption?

4.33 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). These faults, where the affected signal always has a logical value of either 0 or 1 are called "stuck-at-0" or "stuck-at-1" faults. The following problems refer to bit 0 of the Write Register input on the register file in Figure 4.21.

4.33.1 [10] <§§4.3, 4.4> Let us assume that processor testing is done by (1) filling the PC, registers, and data and instruction memories with some values (you can choose which values), (2) letting a single instruction execute, then (3) 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?

4.33.2 [10] <§§4.3, 4.4> Repeat Exercise 4.33.1 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.

4.33.3 [10] <§§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 RISC-V 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.

4.33.4 [10] <§§4.3, 4.4> Repeat Exercise 4.33.1; but now the fault to test for is whether the `MemRead` control signal becomes 0 if the `branch` control signal is 0, no fault otherwise.

4.33.5 [10] <§§4.3, 4.4> Repeat Exercise 4.33.1; but now the fault to test for is whether the `MemRead` control signal becomes 1 if `RegRd` control signal is 1, no fault otherwise. Hint: This problem requires knowledge of operating systems. Consider what causes segmentation faults.

Answers to Check Yourself

§4.1, page 240: 3 of 5: Control, Datapath, Memory. Input and Output are missing.
§4.2, page 243: false. Edge-triggered state elements make simultaneous reading and writing both possible and unambiguous.
§4.3, page 250: I. a. II. c.
§4.4, page 262: Yes, Branch and ALUOp0 are identical. In addition, you can use the flexibility of the don't care bits to combine other signals together. ALUSrc and MemtoReg can be made the same by setting the two don't care bits of MemtoReg to 1 and 0. ALUOp1 and MemtoReg can be made to be inverses of one another by setting the don't care bit of MemtoReg to 1. You don't need an inverter; simply use the other signal and flip the order of the inputs to the MemtoReg multiplexor!
§4.5, page 275: 1. Stall due to a load-use data hazard of the `ld` result. 2. Avoid stalling in the third instruction for the read-after-write data hazard on `x11` by forwarding the `add` result. 3. It need not stall, even without forwarding.
§4.6, page 288: Statements 2 and 4 are correct; the rest are incorrect.
§4.8, page 314: 1. Predict not taken. 2. Predict taken. 3. Dynamic prediction.
§4.9, page 321: The first instruction, since it is logically executed before the others.
§4.10, page 334: 1. Both. 2. Both. 3. Software. 4. Hardware. 5. Hardware. 6. Hardware. 7. Both. 8. Hardware. 9. Both.
§4.12, page 344: First two are false and the last two are true.

5

Ideally one would desire an indefinitely large memory capacity such that any particular … word would be immediately available. … We are … forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible.

A. W. Burks, H. H. Goldstine, and J. von Neumann,
Preliminary Discussion of the Logical Design of an Electronic Computing Instrument, 1946

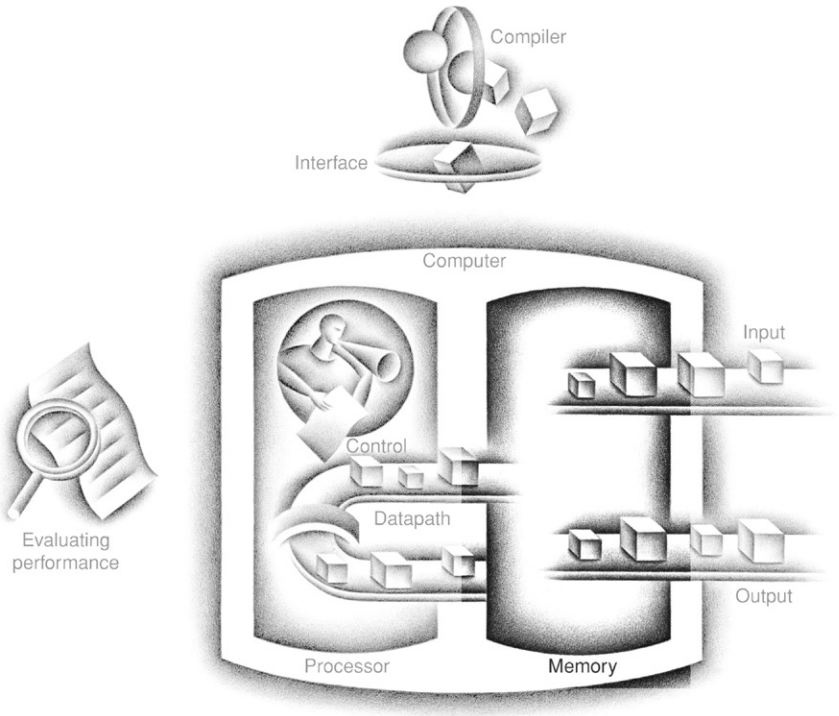
Large and Fast: Exploiting Memory Hierarchy

5.1 **Introduction** 366
5.2 **Memory Technologies** 370
5.3 **The Basics of Caches** 375
5.4 **Measuring and Improving Cache Performance** 390
5.5 **Dependable Memory Hierarchy** 410
5.6 **Virtual Machines** 416
5.7 **Virtual Memory** 419

Computer Organization and Design. DOI: http://dx.doi.org/10.1016/B978-0-12-812275-4.00005-1
© 2018 Elsevier Inc. All rights reserved.

5.8 **A Common Framework for Memory Hierarchy** 443
5.9 **Using a Finite-State Machine to Control a Simple Cache** 449
5.10 **Parallelism and Memory Hierarchy: Cache Coherence** 454
5.11 **Parallelism and Memory Hierarchy: Redundant Arrays of Inexpensive Disks** 458
5.12 **Advanced Material: Implementing Cache Controllers** 459
5.13 **Real Stuff: The ARM Cortex-A53 and Intel Core i7 Memory Hierarchies** 459
5.14 **Real Stuff: The Rest of the RISC-V System and Special Instructions** 464
5.15 **Going Faster: Cache Blocking and Matrix Multiply** 465
5.16 **Fallacies and Pitfalls** 468
5.17 **Concluding Remarks** 472
5.18 **Historical Perspective and Further Reading** 473
5.19 **Exercises** 473

The Five Classic Components of a Computer



5.1 Introduction

From the earliest days of computing, programmers have wanted unlimited amounts of fast memory. The topics in this chapter aid programmers by creating that illusion. Before we look at creating the illusion, let's consider a simple analogy that illustrates the key principles and mechanisms that we use.

Suppose you were a student writing a term paper on important historical developments in computer hardware. You are sitting at a desk in a library with a collection of books that you have pulled from the shelves and are examining. You find that several of the important computers that you need to write about are described in the books you have, but there is nothing about the EDSAC. Therefore, you go back to the shelves and look for an additional book. You find a book on early British computers that covers the EDSAC. Once you have a good selection of books on the desk in front of you, there is a high probability that many of the topics you need can be found in them, and you may spend most of your time just using the books on the desk without returning to the shelves. Having several books on the desk in front of you saves time compared to having only one book there and constantly having to go back to the shelves to return it and take out another.

The same principle allows us to create the illusion of a large memory that we can access as fast as a very small memory. Just as you did not need to access all the books in the library at once with equal probability, a program does not access all of its code or data at once with equal probability. Otherwise, it would be impossible to make most memory accesses fast and still have large memory in computers, just as it would be impossible for you to fit all the library books on your desk and still find what you wanted quickly.

This *principle of locality* underlies both the way in which you did your work in the library and the way that programs operate. The principle of locality states that programs access a relatively small portion of their address space at any instant of time, just as you accessed a very small portion of the library's collection. There are two different types of locality:

temporal locality The locality principle stating that if a data location is referenced then it will tend to be referenced again soon.

spatial locality The locality principle stating that if a data location is referenced, data locations with nearby addresses will tend to be referenced soon.

- **Temporal locality** (locality in time): if an item is referenced, it will tend to be referenced again soon. If you recently brought a book to your desk to look at, you will probably need to look at it again soon.
- **Spatial locality** (locality in space): if an item is referenced, items whose addresses are close by will tend to be referenced soon. For example, when you brought out the book on early English computers to learn about the EDSAC, you also noticed that there was another book shelved next to it about early mechanical computers, so you likewise brought back that book and, later on, found something useful in that book. Libraries put books on the same topic together on the same shelves to increase spatial locality. We'll see how memory hierarchies use spatial locality a little later in this chapter.

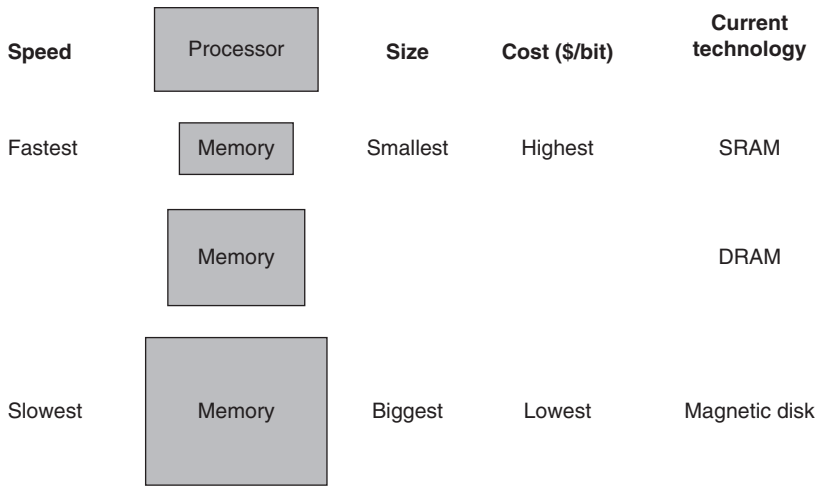


FIGURE 5.1 The basic structure of a memory hierarchy. By implementing the memory system as a hierarchy, the user has the illusion of a memory that is as large as the largest level of the hierarchy, but can be accessed as if it were all built from the fastest memory. Flash memory has replaced disks in many personal mobile devices, and may lead to a new level in the storage hierarchy for desktop and server computers; see Section 5.2.

Just as accesses to books on the desk naturally exhibit locality, locality in programs arises from simple and natural program structures. For example, most programs contain loops, so instructions and data are likely to be accessed repeatedly, showing large temporal locality. Since instructions are normally accessed sequentially, programs also show high spatial locality. Accesses to data also exhibit a natural spatial locality. For example, sequential accesses to elements of an array or a record will naturally have high degrees of spatial locality.

We take advantage of the principle of locality by implementing the memory of a computer as a **memory hierarchy**. A memory hierarchy consists of multiple levels of memory with different speeds and sizes. The faster memories are more expensive per bit than the slower memories and thus are smaller.

memory hierarchy A structure that uses multiple levels of memories; as the distance from the processor increases, the size of the memories and the access time both increase.

Figure 5.1 shows the faster memory is close to the processor and the slower, less expensive memory is below it. The goal is to present the user with as much memory as is available in the cheapest technology, while providing access at the speed offered by the fastest memory.

The data are similarly hierarchical: a level closer to the processor is generally a subset of any level further away, and all the data are stored at the lowest level. By analogy, the books on your desk form a subset of the library you are working in, which is in turn a subset of all the libraries on campus. Furthermore, as we move away from the processor, the levels take progressively longer to access, just as we might encounter in a hierarchy of campus libraries.

A memory hierarchy can consist of multiple levels, but data are copied between only two adjacent levels at a time, so we can focus our attention on just two levels.

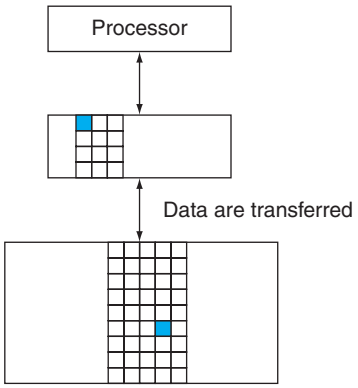


FIGURE 5.2 Every pair of levels in the memory hierarchy can be thought of as having an upper and lower level. Within each level, the unit of information that is present or not is called a *block* or a *line*. Usually we transfer an entire block when we copy something between levels.

block (or line) The minimum unit of information that can be either present or not present in a cache.

hit rate The fraction of memory accesses found in a level of the memory hierarchy.

miss rate The fraction of memory accesses not found in a level of the memory hierarchy.

hit time The time required to access a level of the memory hierarchy, including the time needed to determine whether the access is a hit or a miss.

miss penalty The time required to fetch a block into a level of the memory hierarchy from the lower level, including the time to access the block, transmit it from one level to the other, insert it in the level that experienced the miss, and then pass the block to the requestor.

The upper level—the one closer to the processor—is smaller and faster than the lower level, since the upper level uses technology that is more expensive. Figure 5.2 shows that the minimum unit of information that can be either present or not present in the two-level hierarchy is called a **block** or a **line**; in our library analogy, a block of information is one book.

If the data requested by the processor appear in some block in the upper level, this is called a *hit* (analogous to your finding the information in one of the books on your desk). If the data are not found in the upper level, the request is called a *miss*. The lower level in the hierarchy is then accessed to retrieve the block containing the requested data. (Continuing our analogy, you go from your desk to the shelves to find the desired book.) The **hit rate**, or *hit ratio*, is the fraction of memory accesses found in the upper level; it is often used as a measure of the performance of the memory hierarchy. The **miss rate** (1−hit rate) is the fraction of memory accesses not found in the upper level.

Since performance is the major reason for having a memory hierarchy, the time to service hits and misses is important. **Hit time** is the time to access the upper level of the memory hierarchy, which includes the time needed to determine whether the access is a hit or a miss (that is, the time needed to look through the books on the desk). The **miss penalty** is the time to replace a block in the upper level with the corresponding block from the lower level, plus the time to deliver this block to the processor (or the time to get another book from the shelves and place it on the desk). Because the upper level is smaller and built using faster memory parts, the hit time will be much smaller than the time to access the next level in the hierarchy, which is the major component of the miss penalty. (The time to examine the books on the desk is much smaller than the time to get up and get a new book from the shelves.)

As we will see in this chapter, the concepts used to build memory systems affect many other aspects of a computer, including how the operating system manages memory and I/O, how compilers generate code, and even how applications use the computer. Of course, because all programs spend much of their time accessing memory, the memory system is necessarily a major factor in determining performance. The reliance on memory hierarchies to achieve performance has meant that programmers, who used to be able to think of memory as a flat, random access storage device, now need to understand that memory is a hierarchy to get good performance. We show how important this understanding is in later examples, such as Figure 5.18 on page 400, and Section 5.14, which shows how to double matrix multiply performance.

Since memory systems are critical to performance, computer designers devote a great deal of attention to these systems and develop sophisticated mechanisms for improving the performance of the memory system. In this chapter, we discuss the major conceptual ideas, although we use many simplifications and abstractions to keep the material manageable in length and complexity.

The BIG Picture

Programs exhibit both temporal locality, the tendency to reuse recently accessed data items, and spatial locality, the tendency to reference data items that are close to other recently accessed items. Memory hierarchies take advantage of temporal locality by keeping more recently accessed data items closer to the processor. Memory hierarchies take advantage of spatial locality by moving blocks consisting of multiple contiguous words in memory to upper levels of the hierarchy.

Figure 5.3 shows that a memory hierarchy uses smaller and faster memory technologies close to the processor. Thus, accesses that hit in the highest level of the hierarchy can be processed quickly. Accesses that miss go to lower levels of the hierarchy, which are larger but slower. If the hit rate is high enough, the memory hierarchy has an effective access time close to that of the highest (and fastest) level and a size equal to that of the lowest (and largest) level.

In most systems, the memory is a true hierarchy, meaning that data cannot be present in level i unless they are also present in level $i + 1$.

Check Yourself

Which of the following statements are generally true?

1. Memory hierarchies take advantage of temporal locality.
2. On a read, the value returned depends on which blocks are in the cache.
3. Most of the cost of the memory hierarchy is at the highest level.
4. Most of the capacity of the memory hierarchy is at the lowest level.

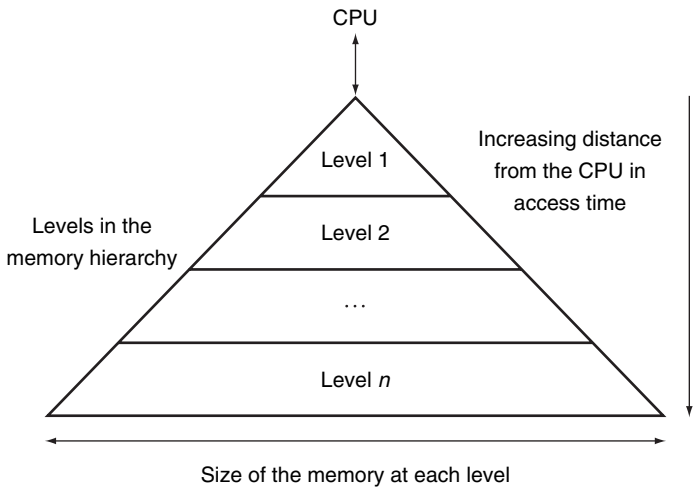


FIGURE 5.3 This diagram shows the structure of a memory hierarchy: as the distance from the processor increases, so does the size. This structure, with the appropriate operating mechanisms, allows the processor to have an access time that is determined primarily by level 1 of the hierarchy and yet have a memory as large as level *n*. Maintaining this illusion is the subject of this chapter. Although the local disk is normally the bottom of the hierarchy, some systems use tape or a file server over a local area network as the next levels of the hierarchy.

5.2 Memory Technologies

There are four primary technologies used today in memory hierarchies. Main memory is implemented from DRAM (*dynamic random access memory*), while levels closer to the processor (caches) use SRAM (*static random access memory*). DRAM is less costly per bit than SRAM, although it is substantially slower. The price difference arises because DRAM uses significantly less area per bit of memory, and DRAMs thus have larger capacity for the same amount of silicon; the speed difference arises from several factors described in Section A.9 of Appendix A. The third technology is flash memory. This nonvolatile memory is the secondary memory in Personal Mobile Devices. The fourth technology, used to implement the largest and slowest level in the hierarchy in servers, is magnetic disk. The access time and price per bit vary widely among these technologies, as the table below shows, using typical values for 2012.

Memory technology	Typical access time	$ per GiB in 2012
SRAM semiconductor memory	0.5–2.5 ns	$500–$1000
DRAM semiconductor memory	50–70 ns	$10–$20
Flash semiconductor memory	5,000–50,000 ns	$0.75–$1.00
Magnetic disk	5,000,000–20,000,000 ns	$0.05–$0.10

We describe each memory technology in the remainder of this section.

SRAM Technology

SRAMs are simply integrated circuits that are memory arrays with (usually) a single access port that can provide either a read or a write. SRAMs have a fixed access time to any datum, though the read and write access times may differ.

SRAMs don't need to refresh and so the access time is very close to the cycle time. SRAMs typically use six to eight transistors per bit to prevent the information from being disturbed when read. SRAM needs only minimal power to retain the charge in standby mode.

In the past, most PCs and server systems used separate SRAM chips for either their primary, secondary, or even tertiary caches. Today, thanks to **Moore's Law**, all levels of caches are integrated onto the processor chip, so the market for independent SRAM chips has nearly evaporated.



DRAM Technology

In a SRAM, as long as power is applied, the value can be kept indefinitely. In a *dynamic RAM* (DRAM), the value kept in a cell is stored as a charge in a capacitor. A single transistor is then used to access this stored charge, either to read the value or to overwrite the charge stored there. Because DRAMs use only one transistor per bit of storage, they are much denser and cheaper per bit than SRAM. As DRAMs store the charge on a capacitor, it cannot be kept indefinitely and must periodically be refreshed. That is why this memory structure is called dynamic, in contrast to the static storage in an SRAM cell.

To refresh the cell, we merely read its contents and write it back. The charge can be kept for several milliseconds. If every bit had to be read out of the DRAM and then written back individually, we would constantly be refreshing the DRAM, leaving no time for accessing it. Fortunately, DRAMs use a two-level decoding structure, and this allows us to refresh an entire *row* (which shares a word line) with a read cycle followed immediately by a write cycle.

Figure 5.4 shows the internal organization of a DRAM, and Figure 5.5 shows how the density, cost, and access time of DRAMs have changed over the years.

The row organization that helps with refresh also helps with performance. To improve performance, DRAMs buffer rows for repeated access. The buffer acts like an SRAM; by changing the address, random bits can be accessed in the buffer until the next row access. This capability improves the access time significantly, since the access time to bits in the row is much lower. Making the chip wider also improves the memory bandwidth of the chip. When the row is in the buffer, it can be transferred by successive addresses at whatever the width of the DRAM is (typically 4, 8, or 16 bits), or by specifying a block transfer and the starting address within the buffer.

To improve the interface to processors further, DRAMs added clocks and are properly called synchronous DRAMs or SDRAMs. The advantage of SDRAMs is that the use of a clock eliminates the time for the memory and processor to synchronize. The speed advantage of synchronous DRAMs comes from the ability to transfer the bits in the burst without having to specify additional address bits.

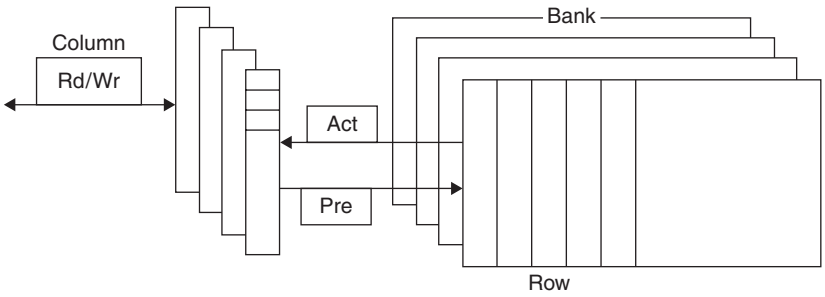


FIGURE 5.4 Internal organization of a DRAM. Modern DRAMs are organized in banks, typically four for DDR3. Each bank consists of a series of rows. Sending a PRE (precharge) command opens or closes a bank. A row address is sent with an Act (activate), which causes the row to transfer to a buffer. When the row is in the buffer, it can be transferred by successive column addresses at whatever the width of the DRAM is (typically 4, 8, or 16 bits in DDR3) or by specifying a block transfer and the starting address. Each command, as well as block transfers, is synchronized with a clock.

Year introduced	Chip size	$ per GiB	Total access time to a new row/column	Average column access time to existing row
1980	64 Kibibit	$1,500,000	250 ns	150 ns
1983	256 Kibibit	$500,000	185 ns	100 ns
1985	1 Mebibit	$200,000	135 ns	40 ns
1989	4 Mebibit	$50,000	110 ns	40 ns
1992	16 Mebibit	$15,000	90 ns	30 ns
1996	64 Mebibit	$10,000	60 ns	12 ns
1998	128 Mebibit	$4,000	60 ns	10 ns
2000	256 Mebibit	$1,000	55 ns	7 ns
2004	512 Mebibit	$250	50 ns	5 ns
2007	1 Gibibit	$50	45 ns	1.25 ns
2010	2 Gibibit	$30	40 ns	1 ns
2012	4 Gibibit	$1	35 ns	0.8 ns

FIGURE 5.5 DRAM size increased by multiples of four approximately once every 3 years until 1996, and thereafter considerably slower. The improvements in access time have been slower but continuous, and cost roughly tracks density improvements, although cost is often affected by other issues, such as availability and demand. The cost per gibibyte is not adjusted for inflation.

Instead, the clock transfers the successive bits in a burst. The fastest version is called *Double Data Rate (DDR)* SDRAM. The name means data transfers on both the rising *and* falling edge of the clock, thereby getting twice as much bandwidth as you might expect based on the clock rate and the data width. The latest version of this technology is called DDR4. A DDR4-3200 DRAM can do 3200 million transfers per second, which means it has a 1600-MHz clock.

Sustaining that much bandwidth requires clever organization *inside* the DRAM. Instead of just a faster row buffer, the DRAM can be internally organized to read or

write from multiple *banks*, with each having its own row buffer. Sending an address to several banks permits them all to read or write simultaneously. For example, with four banks, there is just one access time and then accesses rotate between the four banks to supply four times the bandwidth. This rotating access scheme is called *address interleaving*.

Although personal mobile devices like the iPad (see Chapter 1) use individual DRAMs, memory for servers is commonly sold on small boards called *dual inline memory modules* (DIMMs). DIMMs typically contain 4–16 DRAMs, and they are normally organized to be 8 bytes wide for server systems. A DIMM using DDR4-3200 SDRAMs could transfer at $8 \times 3200 = 25{,}600$ megabytes per second. Such DIMMs are named after their bandwidth: PC25600. Since a DIMM can have so many DRAM chips that only a portion of them are used for a particular transfer, we need a term to refer to the subset of chips in a DIMM that share common address lines. To avoid confusion with the internal DRAM names of row and banks, we use the term *memory rank* for such a subset of chips in a DIMM.

Elaboration: One way to measure the performance of the memory system behind the caches is the Stream benchmark [McCalpin, 1995]. It measures the performance of long vector operations. They have no temporal locality and they access arrays that are larger than the cache of the computer being tested.

Flash Memory

Flash memory is a type of *electrically erasable programmable read-only memory* (EEPROM).

Unlike disks and DRAM, but like other EEPROM technologies, writes can wear out flash memory bits. To cope with such limits, most flash products include a controller to spread the writes by remapping blocks that have been written many times to less trodden blocks. This technique is called *wear leveling*. With wear leveling, personal mobile devices are very unlikely to exceed the write limits in the flash. Such wear leveling lowers the potential performance of flash, but it is needed unless higher-level software monitors block wear. Flash controllers that perform wear leveling can also improve yield by mapping out memory cells that were manufactured incorrectly.

Disk Memory

As Figure 5.6 shows, a magnetic hard disk consists of a collection of platters, which rotate on a spindle at 5400 to 15,000 revolutions per minute. The metal platters are covered with magnetic recording material on both sides, similar to the material found on a cassette or videotape. To read and write information on a hard disk, a movable *arm* containing a small electromagnetic coil called a *read-write head* is located just above each surface. The entire drive is permanently sealed to control the environment inside the drive, which, in turn, allows the disk heads to be much closer to the drive surface.

track One of thousands of concentric circles that make up the surface of a magnetic disk.

Each disk surface is divided into concentric circles, called **tracks**. There are typically tens of thousands of tracks per surface. Each track is in turn divided into

sector One of the segments that make up a track on a magnetic disk; a sector is the smallest amount of information that is read or written on a disk.

sectors that contain the information; each track may have thousands of sectors. Sectors are typically 512 to 4096 bytes in size. The sequence recorded on the magnetic media is a sector number, a gap, the information for that sector including error correction code (see Section 5.5), a gap, the sector number of the next sector, and so on.

The disk heads for each surface are connected together and move in conjunction, so that every head is over the same track of every surface. The term *cylinder* is used to refer to all the tracks under the heads at a given point on all surfaces.

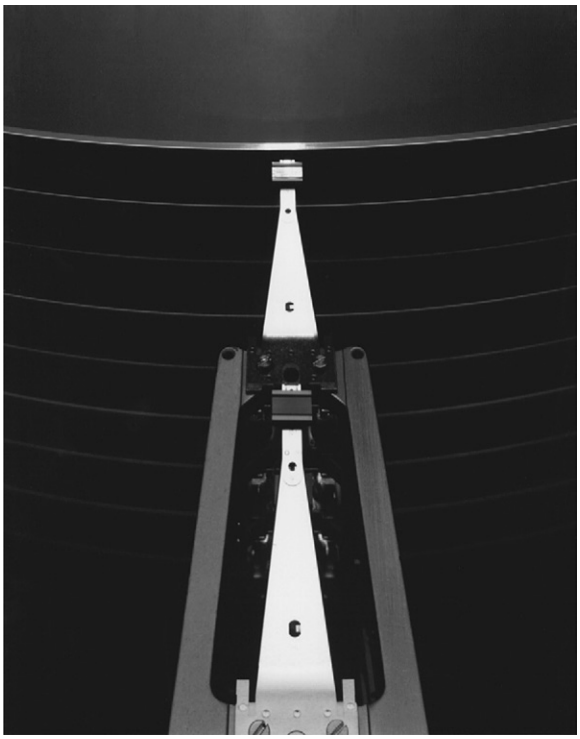


FIGURE 5.6 A disk showing 10 disk platters and the read/write heads. The diameter of today's disks is 2.5 or 3.5 inches, and there are typically one or two platters per drive today.

seek The process of positioning a read/write head over the proper track on a disk.

To access data, the operating system must direct the disk through a three-stage process. The first step is to position the head over the proper track. This operation is called a **seek**, and the time to move the head to the desired track is called the *seek time*.

Disk manufacturers report minimum seek time, maximum seek time, and average seek time in their manuals. The first two are easy to measure, but the average is open to wide interpretation because it depends on the seek distance. The industry calculates average seek time as the sum of the time for all possible seeks divided by the number of possible seeks. Average seek times are usually advertised as 3 ms to 13 ms, but, depending on the application and scheduling of disk requests, the actual average seek time may be only 25% to 33% of the advertised number because of the locality of disk

references. This locality arises both because of successive accesses to the same file and because the operating system tries to schedule such accesses together.

Once the head has reached the correct track, we must wait for the desired sector to rotate under the read/write head. This time is called the **rotational latency** or **rotational delay**. The average latency to the desired information is halfway around the disk. Disks rotate at 5400 RPM to 15,000 RPM. The average rotational latency at 5400 RPM is

rotational latency Also called **rotational delay**. The time required for the desired sector of a disk to rotate under the read/write head; usually assumed to be half the rotation time.

$$\text{Average rotational latency} = \frac{0.5 \text{ rotation}}{5400 \text{ RPM}} = \frac{0.5 \text{ rotation}}{5400 \text{ RPM}/\left(60 \frac{\text{seconds}}{\text{minute}}\right)}$$

$$= 0.0056 \text{ seconds} = 5.6 \text{ ms}$$

The last component of a disk access, *transfer time*, is the time to transfer a block of bits. The transfer time is a function of the sector size, the rotation speed, and the recording density of a track. Transfer rates in 2012 were between 100 and 200 MB/sec.

One complication is that most disk controllers have a built-in cache that stores sectors as they are passed over; transfer rates from the cache are typically higher, and were up to 750 MB/sec (6 Gbit/sec) in 2012.

Alas, where block numbers are located is no longer intuitive. The assumptions of the sector-track-cylinder model above are that nearby blocks are on the same track, blocks in the same cylinder take less time to access since there is no seek time, and some tracks are closer than others. The reason for the change was the raising of the level of the disk interfaces. To speed-up sequential transfers, these higher-level interfaces organize disks more like tapes than like random access devices. The logical blocks are ordered in serpentine fashion across a single surface, trying to capture all the sectors that are recorded at the same bit density to try to get best performance. Hence, sequential blocks may be on different tracks.

In summary, the two primary differences between magnetic disks and semiconductor memory technologies are that disks have a slower access time because they are mechanical devices—flash is 1000 times as fast and DRAM is 100,000 times as fast—yet they are cheaper per bit because they have very high storage capacity at a modest cost—disks are 10 to 100 times cheaper. Magnetic disks are nonvolatile like flash, but unlike flash there is no write wear-out problem. However, flash is much more rugged and hence a better match to the jostling inherent in personal mobile devices.

5.3 The Basics of Caches

Cache: a safe place for hiding or storing things.

Webster's New World Dictionary of the American Language, Third College Edition, 1988

In our library example, the desk acted as a cache—a safe place to store things (books) that we needed to examine. *Cache* was the name chosen to represent the level of the memory hierarchy between the processor and main memory in the first commercial computer to have this extra level. The memories in the datapath in Chapter 4 are

simply replaced by caches. Today, although this remains the dominant use of the word *cache*, the term is also used to refer to any storage managed to take advantage of locality of access. Caches first appeared in research computers in the early 1960s and in production computers later in that same decade; every general-purpose computer built now from servers to low-power embedded processors, includes caches.

In this section, we begin by looking at a very simple cache in which the processor requests are each one word, and the blocks also consist of a single word. (Readers already familiar with cache basics may want to skip to Section 5.4.) Figure 5.7 shows such a simple cache, before and after requesting a data item that is not initially in the cache. Before the request, the cache contains a collection of recent references X_1, X_2, …, X_{n-1}, and the processor requests a word X_n that is not in the cache. This request results in a miss, and the word X_n is brought from memory into the cache.

In looking at the scenario in Figure 5.7, there are two questions to answer: How do we know if a data item is in the cache? Moreover, if it is, how do we find it? The answers are related. If each word can go in exactly one place in the cache, then it is straightforward to find the word if it is in the cache. The simplest way to assign a location in the cache for each word in memory is to assign the cache location based on the *address* of the word in memory. This cache structure is called **direct mapped**, since each memory location is mapped directly to exactly one location in the cache. The typical mapping between addresses and cache locations for a direct-mapped cache is usually simple. For example, almost all direct-mapped caches use this mapping to find a block:

direct-mapped cache A cache structure in which each memory location is mapped to exactly one location in the cache.

(Block address) modulo (Number of blocks in the cache)

If the number of entries in the cache is a power of 2, then modulo can be computed simply by using the low-order $\log_2$ (cache size in blocks) bits of the address. Thus, an 8-block cache uses the three lowest bits ($8 = 2^3$) of the block address. For example, Figure 5.8 shows how the memory addresses between 1_{ten} (00001_{two}) and 29_{ten} (11101_{two}) map to locations 1_{ten} (001_{two}) and 5_{ten} (101_{two}) in a direct-mapped cache of eight words.

Because each cache location can contain the contents of a number of different memory locations, how do we know whether the data in the cache corresponds to a requested word? That is, how do we know whether a requested word is in the cache or not? We answer this question by adding a set of **tags** to the cache. The tags contain the address information required to identify whether a word in the cache corresponds to the requested word. The tag needs just to contain the upper portion of the address, corresponding to the bits that are not used as an index into the cache. For example, in Figure 5.8 we need only have the upper two of the five address bits in the tag, since the lower 3-bit index field of the address selects the block. Architects omit the index bits because they are redundant, since by definition, the index field of any address of a cache block must be that block number.

tag A field in a table used for a memory hierarchy that contains the address information required to identify whether the associated block in the hierarchy corresponds to a requested word.

We also need a way to recognize that a cache block does not have valid information. For instance, when a processor starts up, the cache does not have good data, and the tag fields will be meaningless. Even after executing many instructions,

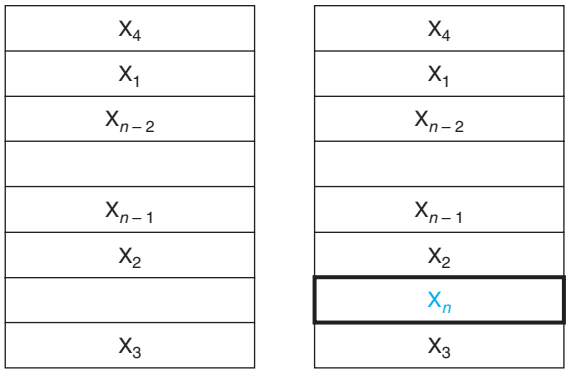


FIGURE 5.7 The cache just before and just after a reference to a word X_n that is not initially in the cache. This reference causes a miss that forces the cache to fetch X_n from memory and insert it into the cache.

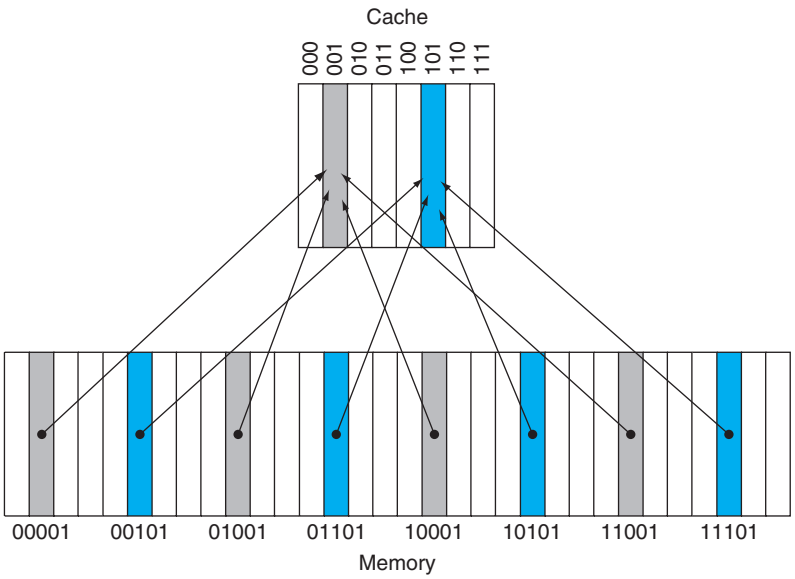


FIGURE 5.8 A direct-mapped cache with eight entries showing the addresses of memory words between 0 and 31 that map to the same cache locations. Because there are eight words in the cache, an address X maps to the direct-mapped cache word X modulo 8. That is, the low-order $\log_2(8) = 3$ bits are used as the cache index. Thus, addresses 00001_{two}, 01001_{two}, 10001_{two}, and 11001_{two} all map to entry 001_{two} of the cache, while addresses 00101_{two}, 01101_{two}, 10101_{two}, and 11101_{two} all map to entry 101_{two} of the cache.

some of the cache entries may still be empty, as in Figure 5.7. Thus, we need to know that the tag should be ignored for such entries. The most common method is to add a **valid bit** to indicate whether an entry contains a valid address. If the bit is not set, there cannot be a match for this block.

valid bit A field in the tables of a memory hierarchy that indicates that the associated block in the hierarchy contains valid data.

For the rest of this section, we will focus on explaining how a cache deals with reads. In general, handling reads is a little simpler than handling writes, since reads do not have to change the contents of the cache. After seeing the basics of how reads work and how cache misses can be handled, we'll examine the cache designs for real computers and detail how these caches handle writes.

The BIG Picture



PREDICTION

Caching is perhaps the most important example of the big idea of **prediction**. It relies on the principle of locality to try to find the desired data in the higher levels of the memory hierarchy, and provides mechanisms to ensure that when the prediction is wrong it finds and uses the proper data from the lower levels of the memory hierarchy. The hit rates of the cache prediction on modern computers are often above 95% (see Figure 5.46).

Accessing a Cache

Below is a sequence of nine memory references to an empty eight-block cache, including the action for each reference. Figure 5.9 shows how the contents of the cache change on each miss. Since there are eight blocks in the cache, the low-order 3 bits of an address give the block number:

Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110_{two}	miss (5.9b)	$(10110_{two} \bmod 8) = 110_{two}$
26	11010_{two}	miss (5.9c)	$(11010_{two} \bmod 8) = 010_{two}$
22	10110_{two}	hit	$(10110_{two} \bmod 8) = 110_{two}$
26	11010_{two}	hit	$(11010_{two} \bmod 8) = 010_{two}$
16	10000_{two}	miss (5.9d)	$(10000_{two} \bmod 8) = 000_{two}$
3	00011_{two}	miss (5.9e)	$(00011_{two} \bmod 8) = 011_{two}$
16	10000_{two}	hit	$(10000_{two} \bmod 8) = 000_{two}$
18	10010_{two}	miss (5.9f)	$(10010_{two} \bmod 8) = 010_{two}$
16	10000_{two}	hit	$(10000_{two} \bmod 8) = 000_{two}$

Since the cache is empty, several of the first references are misses; the caption of Figure 5.9 describes the actions for each memory reference. On the eighth reference

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

a. The initial state of the cache after power-on

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

b. After handling a miss of address (10110_{two})

Index	V	Tag	Data
000	N		
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

c. After handling a miss of address (11010_{two})

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

d. After handling a miss of address (10000_{two})

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

e. After handling a miss of address (00011_{two})

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	10_{two}	Memory (10010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

f. After handling a miss of address (10010_{two})

FIGURE 5.9 The cache contents are shown after each reference request that misses, with the index and tag fields shown in binary for the sequence of addresses on page 379. The cache is initially empty, with all valid bits (V entry in cache) turned off (N). The processor requests the following addresses: 10110_{two} (miss), 11010_{two} (miss), 10110_{two} (hit), 11010_{two} (hit), 10000_{two} (miss), 00011_{two} (miss), 10000_{two} (hit), 10010_{two} (miss), and 10000_{two} (hit). The figures show the cache contents after each miss in the sequence has been handled. When address 10010_{two} (18) is referenced, the entry for address 11010_{two} (26) must be replaced, and a reference to 11010_{two} will cause a subsequent miss. The tag field will contain only the upper portion of the address. The full address of a word contained in cache block *i* with tag field *j* for this cache is $j \times 8 + i$, or equivalently the concatenation of the tag field *j* and the index *i*. For example, in cache *f* above, index 010_{two} has tag 10_{two} and corresponds to address 10010_{two}.

we have conflicting demands for a block. The word at address 18 (10010_{two}) should be brought into cache block 2 (010_{two}). Hence, it must replace the word at address 26 (11010_{two}), which is already in cache block 2 (010_{two}). This behavior allows a cache to take advantage of temporal locality: recently referenced words replace less recently referenced words.

This situation is directly analogous to needing a book from the shelves and having no more space on your desk—some book already on your desk must be returned to the shelves. In a direct-mapped cache, there is only one place to put the newly requested item and hence just one choice of what to replace.

We know where to look in the cache for each possible address: the low-order bits of an address can be used to find the unique cache entry to which the address could map. Figure 5.10 shows how a referenced address is divided into

- A *tag field*, which is used to compare with the value of the tag field of the cache
- A *cache index*, which is used to select the block

The index of a cache block, together with the tag contents of that block, uniquely specifies the memory address of the word contained in the cache block. Because the index field is used as an address to reference the cache, and because an n-bit field has 2^n values, the total number of entries in a direct-mapped cache must be a power of 2. Since words are aligned to multiples of four bytes, the least significant two bits of every address specify a byte within a word. Hence, if the words are aligned in memory, the least significant two bits can be ignored when selecting a word in the block. For this chapter, we'll assume that data are aligned in memory, and discuss how to handle unaligned cache accesses in an Elaboration.

The total number of bits needed for a cache is a function of the cache size and the address size, because the cache includes both the storage for the data and the tags. The size of the block above was one word (4 bytes), but normally it is several. For the following situation:

- 64-bit addresses
- A direct-mapped cache
- The cache size is 2^n blocks, so n bits are used for the index
- The block size is 2^m words (2^{m+2} bytes), so m bits are used for the word within the block, and two bits are used for the byte part of the address

The size of the tag field is

$$64 - (n + m + 2).$$

The total number of bits in a direct-mapped cache is

$$2^n \times (\text{block size} + \text{tag size} + \text{valid field size}).$$

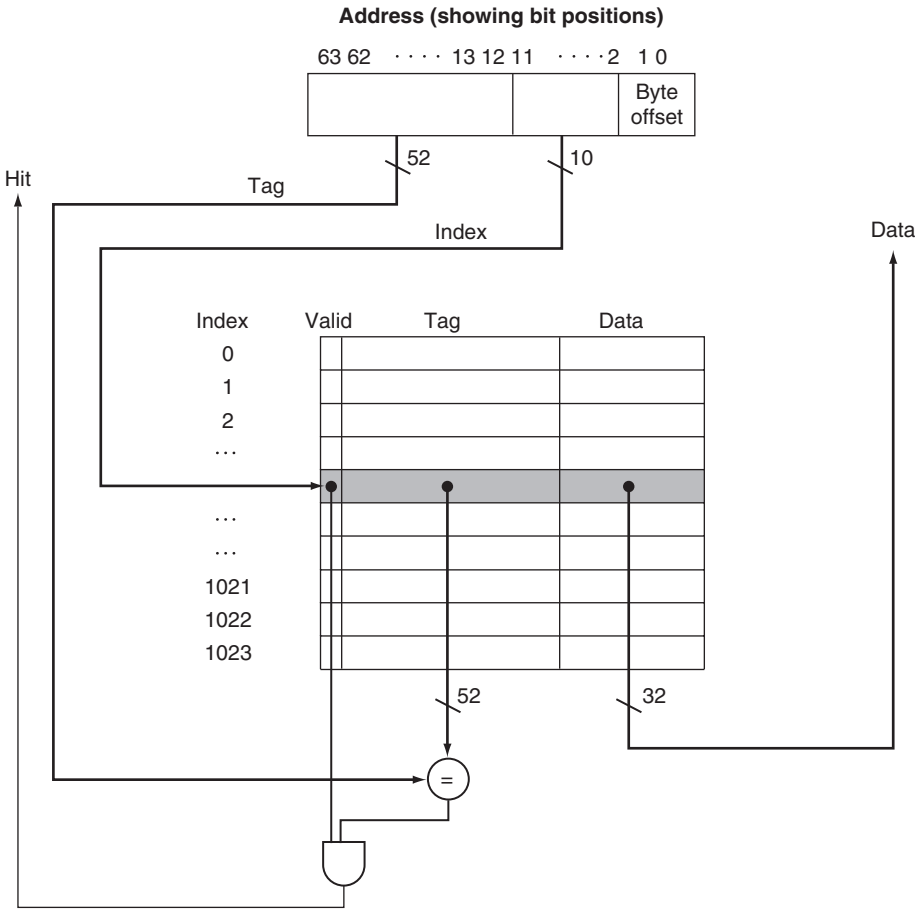


FIGURE 5.10 For this cache, the lower portion of the address is used to select a cache entry consisting of a data word and a tag. This cache holds 1024 words or 4 KiB. Unless noted otherwise, we assume 64-bit addresses in this chapter. The tag from the cache is compared against the upper portion of the address to determine whether the entry in the cache corresponds to the requested address. Because the cache has 2^{10} (or 1024) words and a block size of one word, 10 bits are used to index the cache, leaving $64 - 10 - 2 = 52$ bits to be compared against the tag. If the tag and upper 52 bits of the address are equal and the valid bit is on, then the request hits in the cache, and the word is supplied to the processor. Otherwise, a miss occurs.

Since the block size is 2^m words (2^{m+5} bits), and we need 1 bit for the valid field, the number of bits in such a cache is

$$2^n \times (2^m \times 32 + (64 - n - m - 2) + 1) = 2^n \times (2^m \times 32 + 63 - n - m).$$

Although this is the actual size in bits, the naming convention is to exclude the size of the tag and valid field and to count only the size of the data. Thus, the cache in Figure 5.10 is called a 4 KiB cache.

EXAMPLE

Bits in a Cache

How many total bits are required for a direct-mapped cache with 16 KiB of data and four-word blocks, assuming a 64-bit address?

ANSWER

We know that 16 KiB is 4096 (2^{12}) words. With a block size of four words (2^2), there are 1024 (2^{10}) blocks. Each block has 4×32 or 128 bits of data plus a tag, which is $64 - 10 - 2 - 2$ bits, plus a valid bit. Thus, the complete cache size is

$$2^{10} \times (4 \times 32 + (64 - 10 - 2 - 2) + 1) = 2^{10} \times 179 = 179 \text{ Kibibits}$$

or 22.4 KiB for a 16 KiB cache. For this cache, the total number of bits in the cache is about 1.4 times as many as needed just for the storage of the data.

EXAMPLE

Mapping an Address to a Multiword Cache Block

Consider a cache with 64 blocks and a block size of 16 bytes. To what block number does byte address 1200 map?

ANSWER

We saw the formula on page 376. The block is given by

$$\text{(Block address) modulo (Number of blocks in the cache)}$$

where the address of the block is

$$\frac{\text{Byte address}}{\text{Bytes per block}}$$

Notice that this block address is the block containing all addresses between

$$\left\lfloor \frac{\text{Byte address}}{\text{Bytes per block}} \right\rfloor \times \text{Bytes per block}$$

and

$$\left\lfloor \frac{\text{Byte address}}{\text{Bytes per block}} \right\rfloor \times \text{Bytes per block} + (\text{Bytes per block} - 1)$$

Thus, with 16 bytes per block, byte address 1200 is block address

$$\left\lfloor \frac{1200}{16} \right\rfloor = 75$$

which maps to cache block number (75 modulo 64) = 11. In fact, this block maps all addresses between 1200 and 1215.

Larger blocks exploit spatial locality to lower miss rates. As Figure 5.11 shows, increasing the block size usually decreases the miss rate. The miss rate may go up eventually if the block size becomes a significant fraction of the cache size, because the number of blocks that can be held in the cache will become small, and there will be a great deal of competition for those blocks. As a result, a block will be bumped out of the cache before many of its words are accessed. Stated alternatively, spatial locality among the words in a block decreases with a very large block; consequently, the benefits to the miss rate become smaller.

A more serious problem associated with just increasing the block size is that the cost of a miss rises. The miss penalty is determined by the time required to fetch

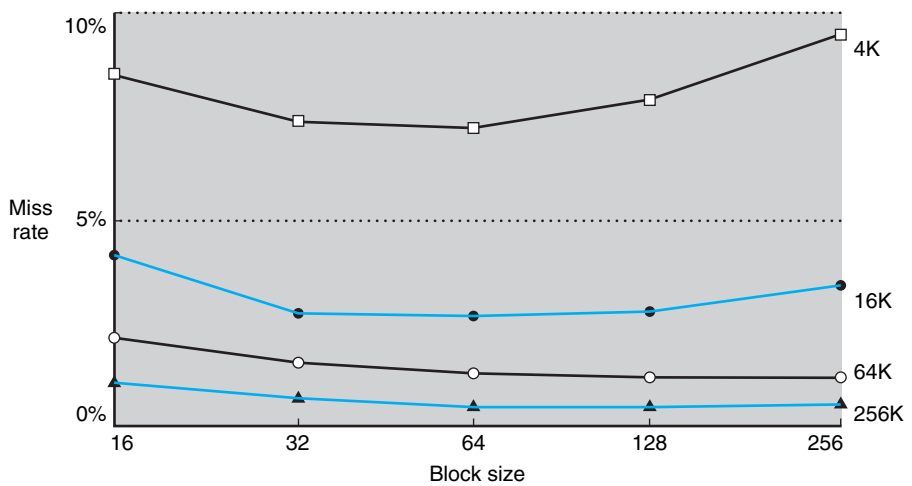


FIGURE 5.11 Miss rate versus block size. Note that the miss rate actually goes up if the block size is too large relative to the cache size. Each line represents a cache of different size. (This figure is independent of associativity, discussed soon.) Unfortunately, SPEC CPU2000 traces would take too long if block size were included, so these data are based on SPEC92.

the block from the next lower level of the hierarchy and load it into the cache. The time to fetch the block has two parts: the latency to the first word and the transfer time for the rest of the block. Clearly, unless we change the memory system, the transfer time—and hence the miss penalty—will likely increase as the block size expands. Furthermore, the improvement in the miss rate starts to decrease as the blocks become larger. The result is that the increase in the miss penalty overwhelms the decrease in the miss rate for blocks that are too large, and cache performance thus decreases. Of course, if we design the memory to transfer larger blocks more efficiently, we can increase the block size and obtain further improvements in cache performance. We discuss this topic in the next section.

Elaboration: Although it is hard to do anything about the longer latency component of the miss penalty for large blocks, we may be able to hide some of the transfer time so that the miss penalty is effectively smaller. The easiest method for doing this, called *early restart*, is simply to resume execution as soon as the requested word of the block is returned, rather than wait for the entire block. Many processors use this technique for instruction access, where it works best. Instruction accesses are largely sequential, so if the memory system can deliver a word every clock cycle, the processor may be able to restart operation when the requested word is returned, with the memory system delivering new instruction words just in time. This technique is usually less effective for data caches because it is likely that the words will be requested from the block in a less predictable way, and the probability that the processor will need another word from a different cache block before the transfer completes is high. If the processor cannot access the data cache because a transfer is ongoing, then it must stall.

An even more sophisticated scheme is to organize the memory so that the requested word is transferred from the memory to the cache first. The remainder of the block is then transferred, starting with the address after the requested word and wrapping around to the beginning of the block. This technique, called *requested word first* or *critical word first*, can be slightly faster than early restart, but it is limited by the same properties that restrain early restart.

Handling Cache Misses

cache miss A request for data from the cache that cannot be filled because the data are not present in the cache.

Before we look at the cache of a real system, let's see how the control unit deals with **cache misses**. (We describe a cache controller in detail in Section 5.9.) The control unit must detect a miss and process the miss by fetching the requested data from memory (or, as we shall see, a lower-level cache). If the cache reports a hit, the computer continues using the data as if nothing happened.

Modifying the control of a processor to handle a hit is trivial; misses, however, require some extra work. The cache miss handling is done in collaboration with the processor control unit and with a separate controller that initiates the memory access and refills the cache. The processing of a cache miss creates a pipeline stall (Chapter 4) in contrast to an exception or interrupt, which would require saving the state of all registers. For a cache miss, we can stall the entire processor, essentially freezing the contents of the temporary and programmer-visible registers, while we wait for memory. More sophisticated out-of-order processors can allow execution

of instructions while waiting for a cache miss, but we'll assume in-order processors that stall on cache misses in this section.

Let's look a little more closely at how instruction misses are handled; the same approach can be easily extended to handle data misses. If an instruction access results in a miss, then the content of the Instruction register is invalid. To get the proper instruction into the cache, we must be able to tell the lower level in the memory hierarchy to perform a read. Since the program counter is incremented in the first clock cycle of execution, the address of the instruction that generates an instruction cache miss is equal to the value of the program counter minus 4. Once we have the address, we need to instruct the main memory to perform a read. We wait for the memory to respond (since the access will take multiple clock cycles), and then write the words containing the desired instruction into the cache.

We can now define the steps to be taken on an instruction cache miss:

1. Send the original PC value to the memory.
2. Instruct main memory to perform a read and wait for the memory to complete its access.
3. Write the cache entry, putting the data from memory in the data portion of the entry, writing the upper bits of the address (from the ALU) into the tag field, and turning the valid bit on.
4. Restart the instruction execution at the first step, which will refetch the instruction, this time finding it in the cache.

The control of the cache on a data access is essentially identical: on a miss, we simply stall the processor until the memory responds with the data.

Handling Writes

Writes work somewhat differently. Suppose on a store instruction, we wrote the data into only the data cache (without changing main memory); then, after the write into the cache, memory would have a different value from that in the cache. In such a case, the cache and memory are said to be *inconsistent*. The simplest way to keep the main memory and the cache consistent is always to write the data into both the memory and the cache. This scheme is called **write-through**.

write-through
A scheme in which writes always update both the cache and the next lower level of the memory hierarchy, ensuring that data are always consistent between the two.

The other key aspect of writes is what occurs on a write miss. We first fetch the words of the block from memory. After the block is fetched and placed into the cache, we can overwrite the word that caused the miss into the cache block. We also write the word to main memory using the full address.

Although this design handles writes very simply, it would not provide good performance. With a write-through scheme, every write causes the data to be written to main memory. These writes will take a long time, likely at least 100 processor clock cycles, and could slow down the processor considerably. For example, suppose 10% of the instructions are stores. If the CPI without cache

misses was 1.0, spending 100 extra cycles on every write would lead to a CPI of $1.0 + 100 \times 10\% = 11$, reducing performance by more than a factor of 10.

write buffer A queue that holds data while the data are waiting to be written to memory.

One solution to this problem is to use a **write buffer**. A write buffer stores the data while they are waiting to be written to memory. After writing the data into the cache and into the write buffer, the processor can continue execution. When a write to main memory completes, the entry in the write buffer is freed. If the write buffer is full when the processor reaches a write, the processor must stall until there is an empty position in the write buffer. Of course, if the rate at which the memory can complete writes is less than the rate at which the processor is generating writes, no amount of buffering can help, because writes are being generated faster than the memory system can accept them.

The rate at which writes are generated may also be *less* than the rate at which the memory can accept them, and yet stalls may still occur. This can happen when the writes occur in bursts. To reduce the occurrence of such stalls, processors usually increase the depth of the write buffer beyond a single entry.

write-back A scheme that handles writes by updating values only to the block in the cache, then writing the modified block to the lower level of the hierarchy when the block is replaced.

The alternative to a write-through scheme is a scheme called **write-back**. In a write-back scheme, when a write occurs, the new value is written only to the block in the cache. The modified block is written to the lower level of the hierarchy when it is replaced. Write-back schemes can improve performance, especially when processors can generate writes as fast or faster than the writes can be handled by main memory; a write-back scheme is, however, more complex to implement than write-through.

In the rest of this section, we describe caches from real processors, and we examine how they handle both reads and writes. In Section 5.8, we will describe the handling of writes in more detail.

Elaboration: Writes introduce several complications into caches that are not present for reads. Here we discuss two of them: the policy on write misses and efficient implementation of writes in write-back caches.

Consider a miss in a write-through cache. The most common strategy is to allocate a block in the cache, called *write allocate*. The block is fetched from memory and then the appropriate portion of the block is overwritten. An alternative strategy is to update the portion of the block in memory but not put it in the cache, called *no write allocate*. The motivation is that sometimes programs write entire blocks of data, such as when the operating system zeros a page of memory. In such cases, the fetch associated with the initial write miss may be unnecessary. Some computers allow the write allocation policy to be changed on a per-page basis.

Actually implementing stores efficiently in a cache that uses a write-back strategy is more complex than in a write-through cache. A write-through cache can write the data into the cache and read the tag; if the tag mismatches, then a miss occurs. Because the cache is write-through, the overwriting of the block in the cache is not catastrophic, since memory has the correct value. In a write-back cache, we must first write the block back to memory if the data in the cache are modified and we have a cache miss. If we simply overwrote the block on a store instruction before we knew whether the store had hit in the cache (as we could for a write-through cache), we would destroy the contents of the block, which is not backed up in the next lower level of the memory hierarchy.

In a write-back cache, because we cannot overwrite the block, stores either require two cycles (a cycle to check for a hit followed by a cycle to actually perform the write) or require a write buffer to hold that data—effectively allowing the store to take only one cycle by pipelining it. When a store buffer is used, the processor does the cache lookup and places the data in the store buffer during the normal cache access cycle. Assuming a cache hit, the new data are written from the store buffer into the cache on the next unused cache access cycle.

By comparison, in a write-through cache, writes can always be done in one cycle. We read the tag and write the data portion of the selected block. If the tag matches the address of the block being written, the processor can continue normally, since the correct block has been updated. If the tag does not match, the processor generates a write miss to fetch the rest of the block corresponding to that address.

Many write-back caches also include write buffers that are used to reduce the miss penalty when a miss replaces a modified block. In such a case, the modified block is moved to a write-back buffer associated with the cache while the requested block is read from memory. The write-back buffer is later written back to memory. Assuming another miss does not occur immediately, this technique halves the miss penalty when a dirty block must be replaced.

An Example Cache: The Intrinsity FastMATH Processor

The Intrinsity FastMATH is an embedded microprocessor that uses the MIPS architecture and a simple cache implementation. Near the end of the chapter, we will examine the more complex cache designs of ARM and Intel microprocessors, but we start with this simple, yet real, example for pedagogical reasons. Figure 5.12 shows the organization of the Intrinsity FastMATH data cache. Note that the address size for this computer is just 32 bits, not 64 as in the rest of the book.

This processor has a 12-stage pipeline. When operating at peak speed, the processor can request both an instruction word and a data word on every clock. To satisfy the demands of the pipeline without stalling, separate instruction and data caches are used. Each cache is 16 KiB, or 4096 words, with 16-word blocks.

Read requests for the cache are straightforward. Because there are separate data and instruction caches, we need separate control signals to read and write each cache. (Remember that we need to update the instruction cache when a miss occurs.) Thus, the steps for a read request to either cache are as follows:

1. Send the address to the appropriate cache. The address comes either from the PC (for an instruction) or from the ALU (for data).
2. If the cache signals hit, the requested word is available on the data lines. Since there are 16 words in the desired block, we need to select the right one. A block index field is used to control the multiplexor (shown at the bottom of the figure), which selects the requested word from the 16 words in the indexed block.

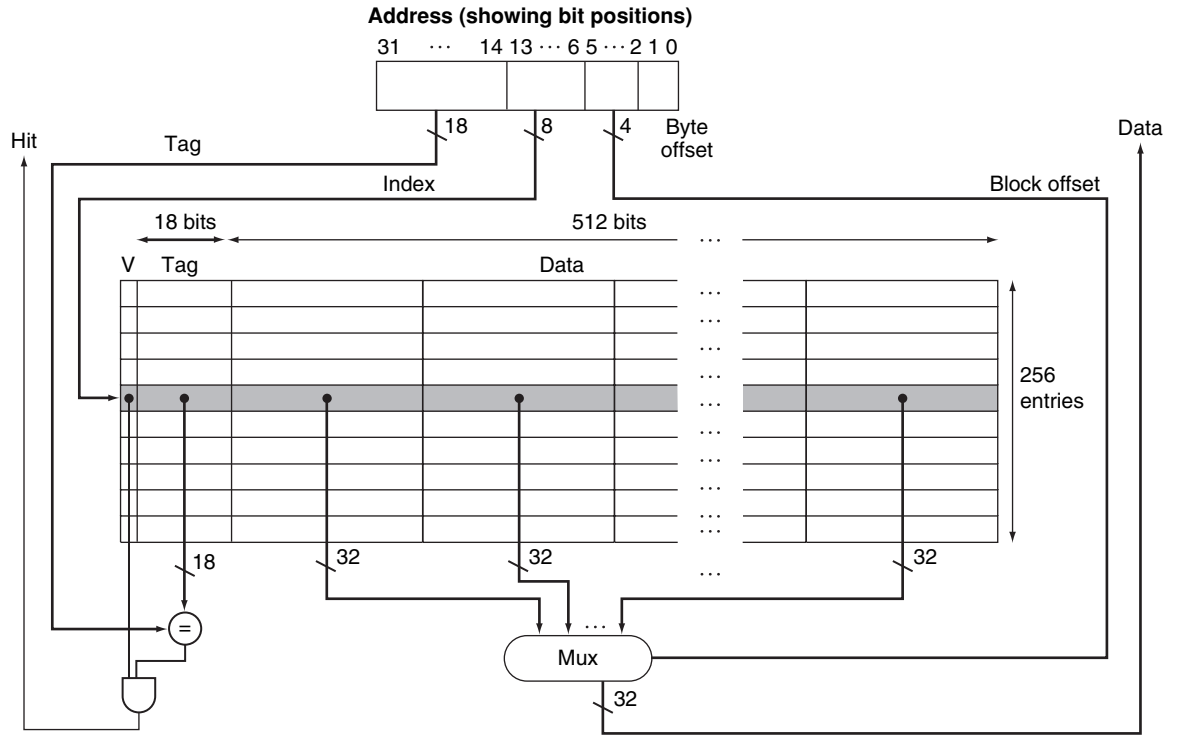


FIGURE 5.12 The 16 KiB caches in the Intrinsity FastMATH each contain 256 blocks with 16 words per block. Note that the address size for this computer is just 32 bits. The tag field is 18 bits wide and the index field is 8 bits wide, while a 4-bit field (bits 5–2) is used to index the block and select the word from the block using a 16-to-1 multiplexor. In practice, to eliminate the multiplexor, caches use a separate large RAM for the data and a smaller RAM for the tags, with the block offset supplying the extra address bits for the large data RAM. In this case, the large RAM is 32 bits wide and must have 16 times as many words as blocks in the cache.

3. If the cache signals miss, we send the address to the main memory. When the memory returns with the data, we write it into the cache and then read it to fulfill the request.

For writes, the Intrinsity FastMATH offers both write-through and write-back, leaving it up to the operating system to decide which strategy to use for an application. It has a one-entry write buffer.

What cache miss rates are attained with a cache structure like that used by the Intrinsity FastMATH? Figure 5.13 shows the miss rates for the instruction and data caches. The combined miss rate is the effective miss rate per reference for each program after accounting for the differing frequency of instruction and data accesses.

Instruction miss rate	Data miss rate	Effective combined miss rate
0.4%	11.4%	3.2%

FIGURE 5.13 Approximate instruction and data miss rates for the Intrinsity FastMATH processor for SPEC CPU2000 benchmarks. The combined miss rate is the effective miss rate seen for the combination of the 16 KiB instruction cache and 16 KiB data cache. It is obtained by weighting the instruction and data individual miss rates by the frequency of instruction and data references.

Although miss rate is an important characteristic of cache designs, the ultimate measure will be the effect of the memory system on program execution time; we'll see how miss rate and execution time are related shortly.

Elaboration: A combined cache with a total size equal to the sum of the two **split caches** will usually have a better hit rate. This higher rate occurs because the combined cache does not rigidly divide the number of entries that may be used by instructions from those that may be used by data. Nonetheless, almost all processors today use split instruction and data caches to increase cache *bandwidth* to match what modern pipelines expect. (There may also be fewer conflict misses; see Section 5.8.)

split cache A scheme in which a level of the memory hierarchy is composed of two independent caches that operate in parallel with each other, with one handling instructions and one handling data.

Here are miss rates for caches the size of those found in the Intrinsity FastMATH processor, and for a combined cache whose size is equal to the sum of the two caches:

- Total cache size: 32 KiB
- Split cache effective miss rate: 3.24%
- Combined cache miss rate: 3.18%

The miss rate of the split cache is only slightly worse.

The advantage of doubling the cache bandwidth, by supporting both an instruction and data access simultaneously, easily overcomes the disadvantage of a slightly increased miss rate. This observation cautions us that we cannot use miss rate as the sole measure of cache performance, as Section 5.4 shows.

Summary

We began the previous section by examining the simplest of caches: a direct-mapped cache with a one-word block. In such a cache, both hits and misses are simple, since a word can go in exactly one location and there is a separate tag for every word. To keep the cache and memory consistent, a write-through scheme can be used, so that every write into the cache also causes memory to be updated. The alternative to write-through is a write-back scheme that copies a block back to memory when it is replaced; we'll discuss this scheme further in upcoming sections.

To take advantage of spatial locality, a cache must have a block size larger than one word. The use of a bigger block decreases the miss rate and improves the efficiency of the cache by reducing the amount of tag storage relative to the amount of data storage in the cache. Although a larger block size decreases the miss rate, it can also increase the miss penalty. If the miss penalty increased linearly with the block size, larger blocks could easily lead to lower performance.

To avoid performance loss, the bandwidth of main memory is increased to transfer cache blocks more efficiently. Common methods for increasing bandwidth external to the DRAM are making the memory wider and interleaving. DRAM designers have steadily improved the interface between the processor and memory to increase the bandwidth of burst mode transfers to reduce the cost of larger cache block sizes.

Check Yourself

The speed of the memory system affects the designer's decision on the size of the cache block. Which of the following cache designer guidelines is generally valid?

1. The shorter the memory latency, the smaller the cache block
2. The shorter the memory latency, the larger the cache block
3. The higher the memory bandwidth, the smaller the cache block
4. The higher the memory bandwidth, the larger the cache block

5.4 Measuring and Improving Cache Performance

In this section, we begin by examining ways to measure and analyze cache performance. We then explore two different techniques for improving cache performance. One focuses on reducing the miss rate by reducing the probability that two distinct memory blocks will contend for the same cache location. The second technique reduces the miss penalty by adding an additional level to the hierarchy. This technique, called *multilevel caching*, first appeared in high-end computers selling for more than $100,000 in 1990; since then it has become common on personal mobile devices selling for a few hundred dollars!

CPU time can be divided into the clock cycles that the CPU spends executing the program and the clock cycles that the CPU spends waiting for the memory system. Normally, we assume that the costs of cache accesses that are hits are part of the normal CPU execution cycles. Thus,

$$\text{CPU time} = (\text{CPU execution clock cycles} + \text{Memory-stall clock cycles}) \times \text{Clock cycle time}$$

The memory-stall clock cycles come primarily from cache misses, and we make that assumption here. We also restrict the discussion to a simplified model of the memory system. In real processors, the stalls generated by reads and writes can be quite complex, and accurate performance prediction usually requires very detailed simulations of the processor and memory system.

Memory-stall clock cycles can be defined as the sum of the stall cycles coming from reads plus those coming from writes:

$$\text{Memory-stall clock cycles} = (\text{Read-stall cycles} + \text{Write-stall cycles})$$

The read-stall cycles can be defined in terms of the number of read accesses per program, the miss penalty in clock cycles for a read, and the read miss rate:

$$\text{Read-stall cycles} = \frac{\text{Reads}}{\text{Program}} \times \text{Read miss rate} \times \text{Read miss penalty}$$

Writes are more complicated. For a write-through scheme, we have two sources of stalls: write misses, which usually require that we fetch the block before continuing the write (see the *Elaboration* on page 386 for more details on dealing with writes), and write buffer stalls, which occur when the write buffer is full when a write happens. Thus, the cycles stalled for writes equal the sum of these two:

$$\text{Write-stall cycles} = \left(\frac{\text{Writes}}{\text{Program}} \times \text{Write miss rate} \times \text{Write miss penalty} \right) + \text{Write buffer stalls}$$

Because the write buffer stalls depend on the proximity of writes, and not just the frequency, it is impossible to give a simple equation to compute such stalls. Fortunately, in systems with a reasonable write buffer depth (e.g., four or more words) and a memory capable of accepting writes at a rate that significantly exceeds the average write frequency in programs (e.g., by a factor of 2), the write buffer stalls will be small, and we can safely ignore them. If a system did not meet these criteria, it would not be well designed; instead, the designer should have used either a deeper write buffer or a write-back organization.

Write-back schemes also have potential additional stalls arising from the need to write a cache block back to memory when the block is replaced. We will discuss this more in Section 5.8.

In most write-through cache organizations, the read and write miss penalties are the same (the time to fetch the block from memory). If we assume that the write buffer stalls are negligible, we can combine the reads and writes by using a single miss rate and the miss penalty:

$$\text{Memory-stall clock cycles} = \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

We can also factor this as

$$\text{Memory-stall clock cycles} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

Let's consider a simple example to help us understand the impact of cache performance on processor performance.

Calculating Cache Performance

EXAMPLE

Assume the miss rate of an instruction cache is 2% and the miss rate of the data cache is 4%. If a processor has a CPI of 2 without any memory stalls, and the miss penalty is 100 cycles for all misses, determine how much faster a processor would run with a perfect cache that never missed. Assume the frequency of all loads and stores is 36%.

ANSWER

The number of memory miss cycles for instructions in terms of the Instruction count (I) is

$$\text{Instruction miss cycles} = \text{I} \times 2\% \times 100 = 2.00 \times \text{I}$$

As the frequency of all loads and stores is 36%, we can find the number of memory miss cycles for data references:

$$\text{Data miss cycles} = \text{I} \times 36\% \times 4\% \times 100 = 1.44 \times \text{I}$$

The total number of memory-stall cycles is 2.00 I + 1.44 I = 3.44 I. This is more than three cycles of memory stall per instruction. Accordingly, the total CPI including memory stalls is 2 + 3.44 = 5.44. Since there is no change in instruction count or clock rate, the ratio of the CPU execution times is

$$\frac{\text{CPU time with stalls}}{\text{CPU time with perfect cache}} = \frac{\text{I} \times \text{CPI}_{\text{stall}} \times \text{Clock cycle}}{\text{I} \times \text{CPI}_{\text{perfect}} \times \text{Clock cycle}} = \frac{\text{CPI}_{\text{stall}}}{\text{CPI}_{\text{perfect}}} = \frac{5.44}{2}$$

The performance with the perfect cache is better by $\frac{5.44}{2} = 2.72$.

What happens if the processor is made faster, but the memory system is not? The amount of time spent on memory stalls will take up an increasing fraction of the execution time; Amdahl's Law, which we examined in Chapter 1, reminds us of this fact. A few simple examples show how serious this problem can be. Suppose we speed-up the computer in the previous example by reducing its CPI from 2 to 1 without changing the clock rate, which might be done with an improved pipeline. The system with cache misses would then have a CPI of 1 + 3.44 = 4.44, and the system with the perfect cache would be

$$\frac{4.44}{1} = 4.44 \text{ times as fast.}$$

The amount of execution time spent on memory stalls would have risen from

$$\frac{3.44}{5.44} = 63\%$$

to

$$\frac{3.44}{4.44} = 77\%$$

Similarly, increasing the clock rate without changing the memory system also increases the performance lost due to cache misses.

The previous examples and equations assume that the hit time is not a factor in determining cache performance. Clearly, if the hit time increases, the total time to access a word from the memory system will increase, possibly causing an increase in the processor cycle time. Although we will see additional examples of what can raise

hit time shortly, one example is increasing the cache size. A larger cache could clearly have a bigger access time, just as, if your desk in the library was very large (say, 3 square meters), it would take longer to locate a book on the desk. An increase in hit time likely adds another stage to the pipeline, since it may take multiple cycles for a cache hit. Although it is more complex to calculate the performance impact of a deeper pipeline, at some point the increase in hit time for a larger cache could dominate the improvement in hit rate, leading to a decrease in processor performance.

To capture the fact that the time to access data for both hits and misses affects performance, designers sometime use *average memory access time* (AMAT) as a way to examine alternative cache designs. Average memory access time is the average time to access memory considering both hits and misses and the frequency of different accesses; it is equal to the following:

$$\text{AMAT} = \text{Time for a hit} + \text{Miss rate} \times \text{Miss penalty}$$

Calculating Average Memory Access Time

EXAMPLE

Find the AMAT for a processor with a 1 ns clock cycle time, a miss penalty of 20 clock cycles, a miss rate of 0.05 misses per instruction, and a cache access time (including hit detection) of 1 clock cycle. Assume that the read and write miss penalties are the same and ignore other write stalls.

ANSWER

The average memory access time per instruction is

$$\begin{aligned}\text{AMAT} &= \text{Time for a hit} + \text{Miss rate} \times \text{Miss penalty}\\ &= 1 + 0.05 \times 20\\ &= 2 \text{ clock cycles}\end{aligned}$$

or 2 ns.

The next subsection discusses alternative cache organizations that decrease miss rate but may sometimes increase hit time; additional examples appear in Section 5.16.

Reducing Cache Misses by More Flexible Placement of Blocks

So far, when we put a block in the cache, we have used a simple placement scheme: A block can go in exactly one place in the cache. As mentioned earlier, it is called *direct mapped* because there is a direct mapping from any block address in memory to a single location in the upper level of the hierarchy. However, there is actually a whole range of schemes for placing blocks. Direct mapped, where a block can be placed in exactly one location, is at one extreme.

At the other extreme is a scheme where a block can be placed in *any* location in the cache. Such a scheme is called **fully associative**, because a block in memory may be associated with any entry in the cache. To find a given block in a fully associative cache, all the entries in the cache must be searched because a block can be placed in any one. To make the search practical, it is done in parallel with a comparator associated with each cache entry. These comparators significantly increase the hardware cost, effectively making fully associative placement practical only for caches with small numbers of blocks.

fully associative cache A cache structure in which a block can be placed in any location in the cache.

The middle range of designs between direct mapped and fully associative is called **set associative**. In a set-associative cache, there are a fixed number of locations where each block can be placed. A set-associative cache with n locations for a block is called an n-way set-associative cache. An n-way set-associative cache consists of a number of sets, each of which consists of n blocks. Each block in the memory maps to a unique *set* in the cache given by the index field, and a block can be placed in *any* element of that set. Thus, a set-associative placement combines direct-mapped placement and fully associative placement: a block is directly mapped into a set, and then all the blocks in the set are searched for a match. For example, Figure 5.14 shows where block 12 may be put in a cache with eight blocks total, according to the three block placement policies.

set-associative cache A cache that has a fixed number of locations (at least two) where each block can be placed.

Remember that in a direct-mapped cache, the position of a memory block is given by

(Block number) modulo (Number of *blocks* in the cache)

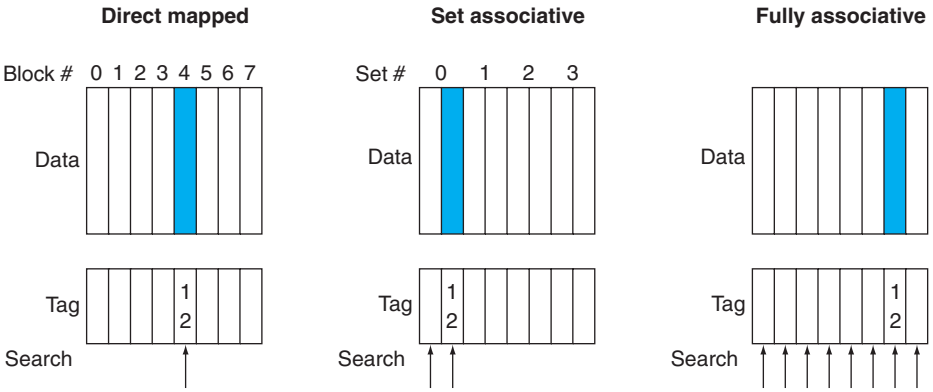


FIGURE 5.14 The location of a memory block whose address is 12 in a cache with eight blocks varies for direct-mapped, set-associative, and fully associative placement. In direct-mapped placement, there is only one cache block where memory block 12 can be found, and that block is given by (12 modulo 8) = 4. In a two-way set-associative cache, there would be four sets, and memory block 12 must be in set (12 mod 4) = 0; the memory block could be in either element of the set. In a fully associative placement, the memory block for block address 12 can appear in any of the eight cache blocks.

In a set-associative cache, the set containing a memory block is given by

(Block number) modulo (Number of *sets* in the cache)

Since the block may be placed in any element of the set, *all the tags of all the elements of the set* must be searched. In a fully associative cache, the block can go anywhere, and *all tags of all the blocks in the cache* must be searched.

We can also think of all block placement strategies as a variation on set associativity. Figure 5.15 shows the possible associativity structures for an eight-block cache. A direct-mapped cache is just a one-way set-associative cache: each cache entry holds one block and each set has one element. A fully associative cache with m entries is simply an m-way set-associative cache; it has one set with m blocks, and an entry can reside in any block within that set.

The advantage of increasing the degree of associativity is that it usually decreases the miss rate, as the next example shows. The main disadvantage, which we discuss in more detail shortly, is a potential increase in the hit time.

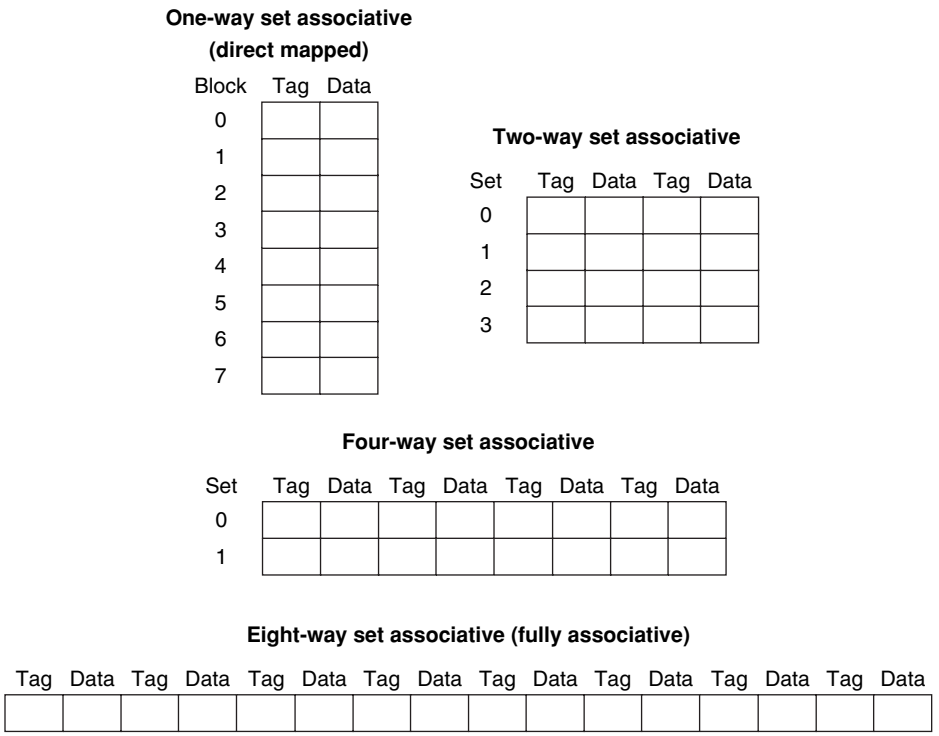


FIGURE 5.15 An eight-block cache configured as direct-mapped, two-way set associative, four-way set associative, and fully associative. The total size of the cache in blocks is equal to the number of sets times the associativity. Thus, for a fixed cache size, increasing the associativity decreases the number of sets while increasing the number of elements per set. With eight blocks, an eight-way set-associative cache is the same as a fully associative cache.

Misses and Associativity in Caches

EXAMPLE

Assume there are three small caches, each consisting of four one-word blocks. One cache is fully associative, a second is two-way set associative, and the third is direct-mapped. Find the number of misses for each cache organization given the following sequence of block addresses: 0, 8, 0, 6, and 8.

ANSWER

The direct-mapped case is easiest. First, let's determine to which cache block each block address maps:

Block address	Cache block
0	(0 modulo 4) = 0
6	(6 modulo 4) = 2
8	(8 modulo 4) = 0

Now we can fill in the cache contents after each reference, using a blank entry to mean that the block is invalid, colored text to show a new entry added to the cache for the associated reference, and plain text to show an old entry in the cache:

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		0	1	2	3
0	miss	Memory[0]			
8	miss	Memory[8]			
0	miss	Memory[0]			
6	miss	Memory[0]		Memory[6]	
8	miss	Memory[8]		Memory[6]	

The direct-mapped cache generates five misses for the five accesses.

The set-associative cache has two sets (with indices 0 and 1) with two elements per set. Let's first determine to which set each block address maps:

Block address	Cache set
0	(0 modulo 2) = 0
6	(6 modulo 2) = 0
8	(8 modulo 2) = 0

Because we have a choice of which entry in a set to replace on a miss, we need a replacement rule. Set-associative caches usually replace the least recently used block within a set; that is, the block that was used furthest in the past is replaced. (We will discuss other replacement rules in more detail shortly.)

Using this replacement rule, the contents of the set-associative cache after each reference look like this:

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Set 0	Set 0	Set 1	Set 1
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[6]		
8	miss	Memory[8]	Memory[6]		

Notice that when block 6 is referenced, it replaces block 8, since block 8 has been less recently referenced than block 0. The two-way set-associative cache has four misses, one less than the direct-mapped cache.

The fully associative cache has four cache blocks (in a single set); any memory block can be stored in any cache block. The fully associative cache has the best performance, with only three misses:

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Block 0	Block 1	Block 2	Block 3
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[8]	Memory[6]	
8	hit	Memory[0]	Memory[8]	Memory[6]	

For this series of references, three misses is the best we can do, because three unique block addresses are accessed. Notice that if we had eight blocks in the cache, there would be no replacements in the two-way set-associative cache (check this for yourself), and it would have the same number of misses as the fully associative cache. Similarly, if we had 16 blocks, all three caches would have the identical number of misses. Even this trivial example shows that cache size and associativity are not independent in determining cache performance.

How much of a reduction in the miss rate is achieved by associativity? Figure 5.16 shows the improvement for a 64 KiB data cache with a 16-word block, and associativity ranging from direct-mapped to eight-way. Going from one-way to two-way associativity decreases the miss rate by about 15%, but there is little further improvement in going to higher associativity.

Associativity	Data miss rate
1	10.3%
2	8.6%
4	8.3%
8	8.1%

FIGURE 5.16 The data cache miss rates for an organization like the Intrinsity FastMATH processor for SPEC CPU2000 benchmarks with associativity varying from one-way to eight-way. These results for 10 SPEC CPU2000 programs are from Hennessy and Patterson (2003).

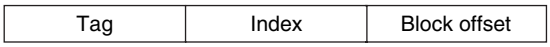


FIGURE 5.17 The three portions of an address in a set-associative or direct-mapped cache. The index is used to select the set, then the tag is used to choose the block by comparison with the blocks in the selected set. The block offset is the address of the desired data within the block.

Locating a Block in the Cache

Now, let's consider the task of finding a block in a cache that is set associative. Just as in a direct-mapped cache, each block in a set-associative cache includes an address tag that gives the block address. The tag of every cache block within the appropriate set is checked to see if it matches the block address from the processor. Figure 5.17 decomposes the address. The index value is used to select the set containing the address of interest, and the tags of all the blocks in the set must be searched. Because speed is of the essence, all the tags in the selected set are searched in parallel. As in a fully associative cache, a sequential search would make the hit time of a set-associative cache too slow.

If the total cache size is kept the same, increasing the associativity raises the number of blocks per set, which is the number of simultaneous compares needed to perform the search in parallel: each increase by a factor of 2 in associativity doubles the number of blocks per set and halves the number of sets. Accordingly, each factor-of-2 increase in associativity decreases the size of the index by 1 bit and expands the size of the tag by 1 bit. In a fully associative cache, there is effectively only one set, and all the blocks must be checked in parallel. Thus, there is no index, and the entire address, excluding the block offset, is compared against the tag of every block. In other words, we search the full cache without any indexing.

In a direct-mapped cache, only a single comparator is needed, because the entry can be in only one block, and we access the cache simply by indexing. Figure 5.18 shows that in a four-way set-associative cache, four comparators are needed, together with a 4-to-1 multiplexor to choose among the four potential members of the selected set. The cache access consists of indexing the appropriate set and then searching the tags of the set. The costs of an associative cache are the extra comparators and any delay imposed by having to do the compare and select from among the elements of the set.

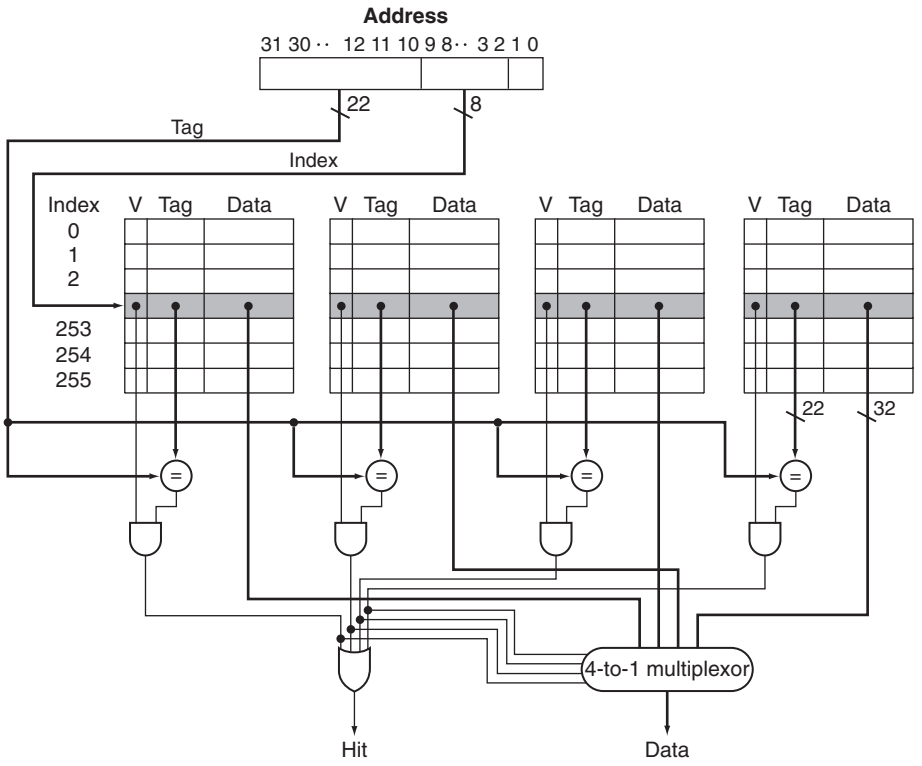


FIGURE 5.18 The implementation of a four-way set-associative cache requires four comparators and a 4-to-1 multiplexor. The comparators determine which element of the selected set (if any) matches the tag. The output of the comparators is used to select the data from one of the four blocks of the indexed set, using a multiplexor with a decoded select signal. In some implementations, the Output enable signals on the data portions of the cache RAMs can be used to select the entry in the set that drives the output. The Output enable signal comes from the comparators, causing the element that matches to drive the data outputs. This organization eliminates the need for the multiplexor.

The choice among direct-mapped, set-associative, or fully associative mapping in any memory hierarchy will depend on the cost of a miss versus the cost of implementing associativity, both in time and in extra hardware.

Elaboration: A *Content Addressable Memory* (CAM) is a circuit that combines comparison and storage in a single device. Instead of supplying an address and reading a word like a RAM, you send the data and the CAM looks to see if it has a copy and returns the index of the matching row. CAMs mean that cache designers can afford to implement much higher set associativity than if they needed to build the hardware out of SRAMs and comparators. In 2013, the greater size and power of CAM generally leads to two-way and four-way set associativity being built from standard SRAMs and comparators, with eight-way and above built using CAMs.

Choosing Which Block to Replace

When a miss occurs in a direct-mapped cache, the requested block can go in exactly one position, and the block occupying that position must be replaced. In an associative cache, we have a choice of where to place the requested block, and hence a choice of which block to replace. In a fully associative cache, all blocks are candidates for replacement. In a set-associative cache, we must choose among the blocks in the selected set.

The most commonly used scheme is **least recently used (LRU)**, which we used in the previous example. In an LRU scheme, the block replaced is the one that has been unused for the longest time. The set-associative example on page 397 uses LRU, which is why we replaced Memory(0) instead of Memory(6).

least recently used (LRU) A replacement scheme in which the block replaced is the one that has been unused for the longest time.

LRU replacement is implemented by keeping track of when each element in a set was used relative to the other elements in the set. For a two-way set-associative cache, tracking when the two elements were used can be implemented by keeping a single bit in each set and setting the bit to indicate an element whenever that element is referenced. As associativity increases, implementing LRU gets harder; in Section 5.8, we will see an alternative scheme for replacement.

Size of Tags versus Set Associativity

EXAMPLE

Increasing associativity requires more comparators and more tag bits per cache block. Assuming a cache of 4096 blocks, a four-word block size, and a 64-bit address, find the total number of sets and the total number of tag bits for caches that are direct-mapped, two-way and four-way set associative, and fully associative.

ANSWER

Since there are 16 ($= 2^4$) bytes per block, a 64-bit address yields $64 - 4 = 60$ bits to be used for index and tag. The direct-mapped cache has the same number of sets as blocks, and hence 12 bits of index, since $\log_2(4096) = 12$; hence, the total number is $(60 - 12) \times 4096 = 48 \times 4096 = 197$ K tag bits.

Each degree of associativity decreases the number of sets by a factor of 2 and thus decreases the number of bits used to index the cache by 1 and increases the number of bits in the tag by 1. Thus, for a two-way set-associative cache, there are 2048 sets, and the total number of tag bits is $(60 - 11) \times 2 \times 2048 = 98 \times 2048 = 401$ Kbits. For a four-way set-associative cache, the total number of sets is 1024, and the total number is $(60 - 10) \times 4 \times 1024 = 100 \times 1024 = 205$ K tag bits.

For a fully associative cache, there is only one set with 4096 blocks, and the tag is 60 bits, leading to $60 \times 4096 \times 1 = 246$ K tag bits.

Reducing the Miss Penalty Using Multilevel Caches

All modern computers make use of caches. To close the gap further between the fast clock rates of modern processors and the increasingly long time required to access DRAMs, most microprocessors support an additional level of caching. This second-level cache is normally on the same chip and is accessed whenever a miss occurs in the primary cache. If the second-level cache contains the desired data, the miss penalty for the first-level cache will be essentially the access time of the second-level cache, which will be much less than the access time of main memory. If neither the primary nor the secondary cache contains the data, a main memory access is required, and a larger miss penalty is incurred.

How significant is the performance improvement from the use of a secondary cache? The next example shows us.

EXAMPLE

Performance of Multilevel Caches

Suppose we have a processor with a base CPI of 1.0, assuming all references hit in the primary cache, and a clock rate of 4 GHz. Assume a main memory access time of 100 ns, including all the miss handling. Suppose the miss rate per instruction at the primary cache is 2%. How much faster will the processor be if we add a secondary cache that has a 5-ns access time for either a hit or a miss and is large enough to reduce the miss rate to main memory to 0.5%?

ANSWER

The miss penalty to main memory is

$$\frac{100\,\text{ns}}{0.25\dfrac{\text{ns}}{\text{clock cycle}}} = 400 \text{ clock cycles}$$

The effective CPI with one level of caching is given by

$$\text{Total CPI} = \text{Base CPI} + \text{Memory-stall cycles per instruction}$$

For the processor with one level of caching,

$$\text{Total CPI} = 1.0 + \text{Memory-stall cycles per instruction} = 1.0 + 2\% \times 400 = 9$$

With two levels of caching, a miss in the primary (or first-level) cache can be satisfied either by the secondary cache or by main memory. The miss penalty for an access to the second-level cache is

$$\frac{5\,\text{ns}}{0.25\dfrac{\text{ns}}{\text{clock cycle}}} = 20 \text{ clock cycles}$$

If the miss is satisfied in the secondary cache, then this is the entire miss penalty. If the miss needs to go to main memory, then the total miss penalty is the sum of the secondary cache access time and the main memory access time.

Thus, for a two-level cache, total CPI is the sum of the stall cycles from both levels of cache and the base CPI:

$$\begin{aligned}\text{Total CPI} &= 1 + \text{Primary stalls per instruction} + \text{Secondary stalls per instruction} \\ &= 1 + 2\% \times 20 + 0.5\% \times 400 = 1 + 0.4 + 2.0 = 3.4\end{aligned}$$

Thus, the processor with the secondary cache is faster by

$$\frac{9.0}{3.4} = 2.6$$

Alternatively, we could have computed the stall cycles by summing the stall cycles of those references that hit in the secondary cache ($(2\% - 0.5\%) \times 20 = 0.3$). Those references that go to main memory, which must include the cost to access the secondary cache as well as the main memory access time, are ($0.5\% \times (20 + 400) = 2.1$). The sum, $1.0 + 0.3 + 2.1$, is again 3.4.

The design considerations for a primary and secondary cache are significantly different, because the presence of the other cache changes the best choice versus a single-level cache. In particular, a two-level cache structure allows the primary cache to focus on minimizing hit time to yield a shorter clock cycle or fewer pipeline stages, while allowing the secondary cache to focus on miss rate to reduce the penalty of long memory access times.

The effect of these changes on the two caches can be seen by comparing each cache to the optimal design for a single level of cache. In comparison to a single-level cache, the primary cache of a **multilevel cache** is often smaller. Furthermore, the primary cache may use a smaller block size, to go with the smaller cache size and also to reduce the miss penalty. In comparison, the secondary cache will be much larger than in a single-level cache, since the access time of the secondary cache is less critical. With a larger total size, the secondary cache may use a larger block size than appropriate with a single-level cache. It often uses higher associativity than the primary cache given the focus of reducing miss rates.

multilevel cache
A memory hierarchy with multiple levels of caches, rather than just a cache and main memory.

Understanding Program Performance

Sorting has been exhaustively analyzed to find better algorithms: Bubble Sort, Quicksort, Radix Sort, and so on. Figure 5.19(a) shows instructions executed by item searched for Radix Sort versus Quicksort. As expected, for large arrays, Radix Sort has an algorithmic advantage over Quicksort in terms of number of operations. Figure 5.19(b) shows time per key instead of instructions executed. We see that the lines start on the same trajectory as in Figure 5.19(a), but then the Radix Sort line

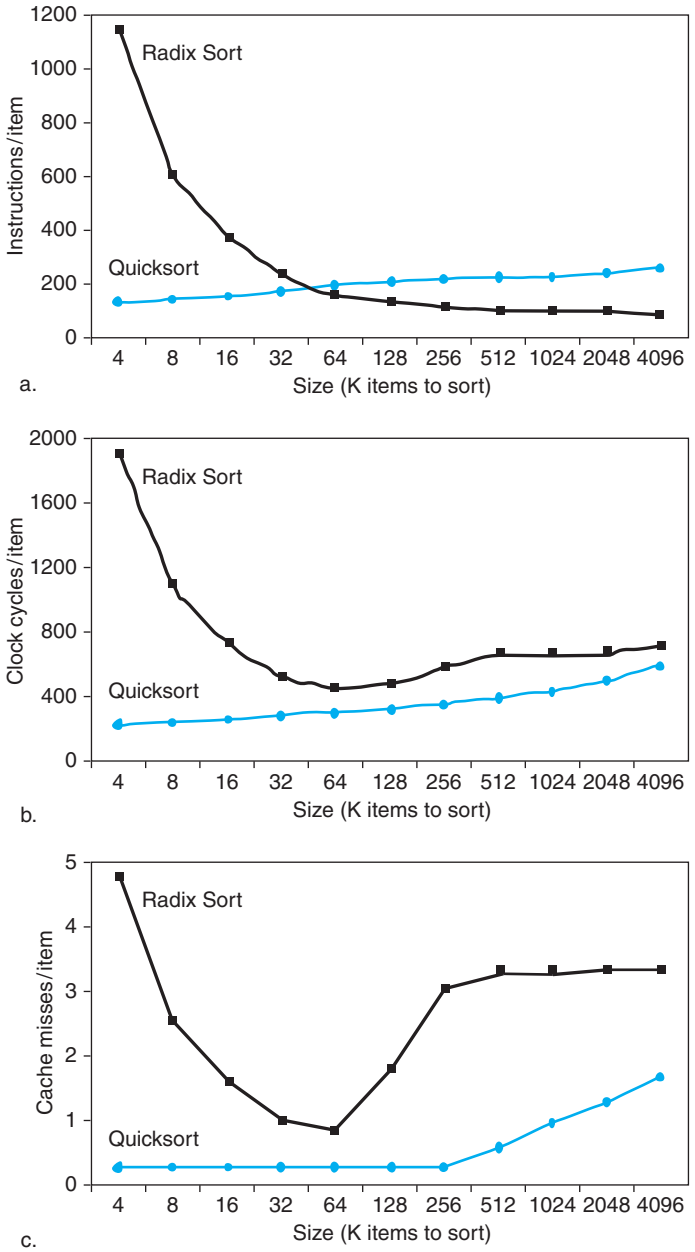


FIGURE 5.19 Comparing Quicksort and Radix Sort by (a) instructions executed per item sorted, (b) time per item sorted, and (c) cache misses per item sorted. These data are from a paper by LaMarca and Ladner [1996]. Due to such results, new versions of Radix Sort have been invented that take memory hierarchy into account, to regain its algorithmic advantages (see Section 5.15). The basic idea of cache optimizations is to use all the data in a block repeatedly before they are replaced on a miss.

diverges as the data to sort increase. What is going on? Figure 5.19(c) answers by looking at the cache misses per item sorted: Quicksort consistently has many fewer misses per item to be sorted.

Alas, standard algorithmic analysis often ignores the impact of the memory hierarchy. As faster clock rates and **Moore's Law** allow architects to squeeze all the performance out of a stream of instructions, using the memory hierarchy well is vital to high performance. As we said in the introduction, understanding the behavior of the memory hierarchy is critical to understanding the performance of programs on today's computers.



Software Optimization via Blocking

Given the importance of the memory hierarchy to program performance, not surprisingly many software optimizations were invented that can dramatically improve performance by reusing data within the cache and hence lower miss rates due to improved temporal locality.

When dealing with arrays, we can get good performance from the memory system if we store the array in memory so that accesses to the array are sequential in memory. Suppose that we are dealing with multiple arrays, however, with some arrays accessed by rows and some by columns. Storing the arrays row-by-row (called *row major order*) or column-by-column (*column major order*) does not solve the problem because both rows and columns are used in every loop iteration.

Instead of operating on entire rows or columns of an array, *blocked* algorithms operate on submatrices or *blocks*. The goal is to maximize accesses to the data loaded into the cache before the data are replaced; that is, improve temporal locality to reduce cache misses.

For example, the inner loops of DGEMM (lines 4 through 9 of Figure 3.22 in Chapter 3) are

```
for (int j = 0; j < n; ++j)
    {
     double cij = C[i+j*n]; /* cij = C[i][j] */
     for( int k = 0; k < n; k++ )
      cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
     C[i+j*n] = cij; /* C[i][j] = cij */
     }
}
```

It reads all `N`-by-`N` elements of `B`, reads the same `N` elements in what corresponds to one row of `A` repeatedly, and writes what corresponds to one row of `N` elements of `C`. (The comments make the rows and columns of the matrices easier to identify.) Figure 5.20 gives a snapshot of the accesses to the three arrays. A dark shade indicates a recent access, a light shade indicates an older access, and white means not yet accessed.

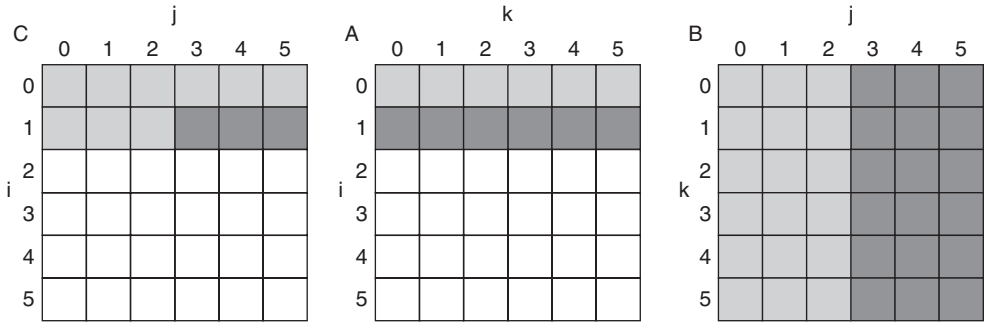


FIGURE 5.20 A snapshot of the three arrays `C`, `A`, and `B` when `N` = 6 and `i` = 1. The age of accesses to the array elements is indicated by shade: white means not yet touched, light means older accesses, and dark means newer accesses. Compared to Figure 5.22, elements of `A` and `B` are read repeatedly to calculate new elements of `C`. The variables `i`, `j`, and `k` are shown along the rows or columns used to access the arrays.

The number of capacity misses clearly depends on `N` and the size of the cache. If it can hold all three `N`-by-`N` matrices, then all is well, provided there are no cache conflicts. We purposely picked the matrix size to be 32 by 32 in DGEMM for Chapters 3 and 4 so that this would be the case. Each matrix is $32 \times 32 = 1024$ elements and each element is 8 bytes, so the three matrices occupy 24 KiB, which comfortably fit in the 32 KiB data cache of the Intel Core i7 (Sandy Bridge).

If the cache can hold one `N`-by-`N` matrix and one row of `N`, then at least the `i`th row of `A` and the array `B` may stay in the cache. Less than that and misses may occur for both `B` and `C`. In the worst case, there would be $2 \times N^3 + N^2$ memory words accessed for N^3 operations.

To ensure that the elements being accessed can fit in the cache, the original code is changed to compute on a submatrix. Hence, we essentially invoke the version of DGEMM from Figure 4.78 in Chapter 4 repeatedly on matrices of size `BLOCKSIZE` by `BLOCKSIZE`. `BLOCKSIZE` is called the *blocking factor*.

Figure 5.21 shows the blocked version of DGEMM. The function `do_block` is DGEMM from Figure 3.22 with three new parameters `si`, `sj`, and `sk` to specify the starting position of each submatrix of `A`, `B`, and `C`. The two inner loops of the `do_block` now compute in steps of size `BLOCKSIZE` rather than the full length of `B` and `C`. The gcc optimizer removes any function call overhead by "inlining" the function; that is, it inserts the code directly to avoid the conventional parameter passing and return address bookkeeping instructions.

Figure 5.22 illustrates the accesses to the three arrays using blocking. Looking only at capacity misses, the total number of memory words accessed is $2 \times N^3/\text{BLOCKSIZE} + N^2$. This total is an improvement by about a factor of `BLOCKSIZE`. Hence, blocking exploits a combination of spatial and temporal locality, since `A` benefits from spatial locality and `B` benefits from temporal locality.

```
1  #define BLOCKSIZE 32
2  void do_block (int n, int si, int sj, int sk, double *A, double
3  *B, double *C)
4  {
5     for (int i = si; i < si+BLOCKSIZE; ++i)
6        for (int j = sj; j < sj+BLOCKSIZE; ++j)
7           {
8              double cij = C[i+j*n];/* cij = C[i][j] */
9              for( int k = sk; k < sk+BLOCKSIZE; k++ )
10                cij += A[i+k*n] * B[k+j*n];/* cij+=A[i][k]*B[k][j] */
11             C[i+j*n] = cij;/* C[i][j] = cij */
12          }
13 }
14 void dgemm (int n, double* A, double* B, double* C)
15 {
16    for ( int sj = 0; sj < n; sj += BLOCKSIZE )
17       for ( int si = 0; si < n; si += BLOCKSIZE )
18          for ( int sk = 0; sk < n; sk += BLOCKSIZE )
19             do_block(n, si, sj, sk, A, B, C);
20 }
```

FIGURE 5.21 Cache blocked version of DGEMM in Figure 3.22. Assume `C` is initialized to zero. The `do_block` function is basically DGEMM from Chapter 3 with new parameters to specify the starting positions of the submatrices of `BLOCKSIZE`. The gcc optimizer can remove the function overhead instructions by inlining the `do_block` function.

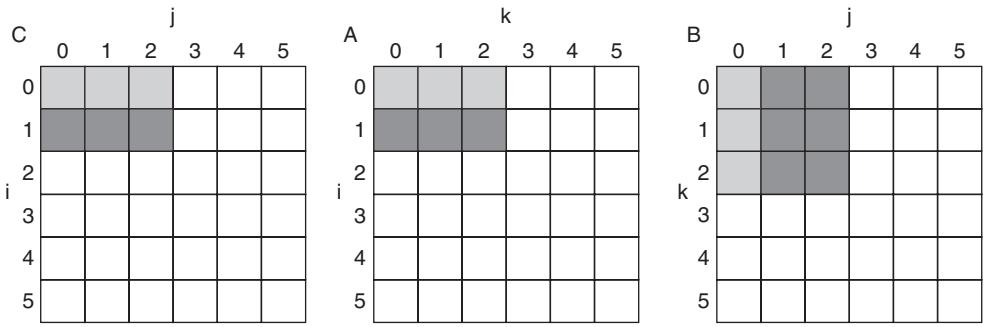


FIGURE 5.22 The age of accesses to the arrays `C`, `A`, and `B` when *BLOCKSIZE* = 3. Note that, in contrast to Figure 5.20, fewer elements are accessed.

Although we have aimed at reducing cache misses, blocking can also be used to help register allocation. By taking a small blocking size, such that the block can be held in registers, we can minimize the number of loads and stores in the program, which again improves performance.

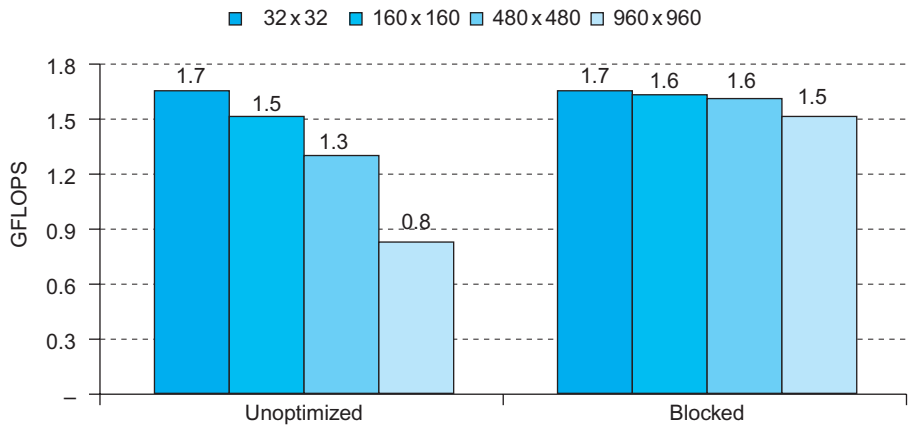


FIGURE 5.23 Performance of unoptimized DGEMM (Figure 3.22) versus cache blocked DGEMM (Figure 5.21) as the matrix dimension varies from 32 × 32 (where all three matrices fit in the cache) to 960 × 960.

Figure 5.23 shows the impact of cache blocking on the performance of the unoptimized DGEMM as we increase the matrix size beyond where all three matrices fit in the cache. The unoptimized performance is halved for the largest matrix. The cache-blocked version is less than 10% slower even at matrices that are 960 × 960, or 900 times larger than the 32 × 32 matrices in Chapters 3 and 4.

global miss rate The fraction of references that miss in all levels of a multilevel cache.

local miss rate The fraction of references to one level of a cache that miss; used in multilevel hierarchies.

Elaboration: Multilevel caches create many complications. First, there are now several different types of misses and corresponding miss rates. In the example on pages 402–403, we saw the primary cache miss rate and the **global miss rate**—the fraction of references that missed in all cache levels. There is also a miss rate for the secondary cache, which is the ratio of all misses in the secondary cache divided by the number of accesses to it. This miss rate is called the **local miss rate** of the secondary cache. Because the primary cache filters accesses, especially those with good spatial and temporal locality, the local miss rate of the secondary cache is much higher than the global miss rate. For the example on pages 402–403, we can compute the local miss rate of the secondary cache as 0.5%/2% = 25%! Luckily, the global miss rate dictates how often we must access the main memory.

Elaboration: With out-of-order processors (see Chapter 4), performance is more complex, since they execute instructions during the miss penalty. Instead of instruction miss rates and data miss rates, we use misses per instruction, and this formula:

$$\frac{\text{Memory} - \text{stall cycles}}{\text{Instruction}} = \frac{\text{Misses}}{\text{Instruction}} \times (\text{Total miss latency} - \text{Overlapped miss latency})$$

There is no general way to calculate overlapped miss latency, so evaluations of memory hierarchies for out-of-order processors inevitably require simulation of the processor and the memory hierarchy. Only by seeing the execution of the processor during each miss can we see if the processor stalls waiting for data or simply finds other work to do. A guideline is that the processor often hides the miss penalty for an L1 cache miss that hits in the L2 cache, but it rarely hides a miss to the L2 cache.

Elaboration: The performance challenge for algorithms is that the memory hierarchy varies between different implementations of the same architecture in cache size, associativity, block size, and number of caches. To cope with such variability, some recent numerical libraries parameterize their algorithms and then search the parameter space at runtime to find the best combination for a particular computer. This approach is called *autotuning*.

Check Yourself

Which of the following is generally true about a design with multiple levels of caches?

1. First-level caches are more concerned about hit time, and second-level caches are more concerned about miss rate.
2. First-level caches are more concerned about miss rate, and second-level caches are more concerned about hit time.

Summary

In this section, we focused on four topics: cache performance, using associativity to reduce miss rates, the use of multilevel cache hierarchies to reduce miss penalties, and software optimizations to improve effectiveness of caches.

The memory system has a significant effect on program execution time. The number of memory-stall cycles depends on both the miss rate and the miss penalty. The challenge, as we will see in Section 5.8, is to reduce one of these factors without significantly affecting other critical factors in the memory hierarchy.

To reduce the miss rate, we examined the use of associative placement schemes. Such schemes can reduce the miss rate of a cache by allowing more flexible placement of blocks within the cache. Fully associative schemes allow blocks to be placed anywhere, but also require that every block in the cache be searched to satisfy a request. The higher costs make large fully associative caches impractical. Set-associative caches are a practical alternative, since we need only search among the elements of a unique set that is chosen by indexing. Set-associative caches have higher miss rates but are faster to access. The amount of associativity that yields the best performance depends on both the technology and the details of the implementation.

We looked at multilevel caches as a technique to reduce the miss penalty by allowing a larger secondary cache to handle misses to the primary cache. Second-level caches have become commonplace as designers find that limited silicon and the goals of high clock rates prevent primary caches from becoming large. The secondary cache, which is often 10 or more times larger than the primary cache, handles many accesses that miss in the primary cache. In such cases, the miss penalty is that of the access time to the secondary cache (typically <10 processor

cycles) versus the access time to memory (typically > 100 processor cycles). As with associativity, the design tradeoffs between the size of the secondary cache and its access time depend on a number of aspects of the implementation.

Finally, given the importance of the memory hierarchy in performance, we looked at how to change algorithms to improve cache behavior, with blocking being an important technique when dealing with large arrays.

5.5 Dependable Memory Hierarchy



Implicit in all the prior discussion is that the memory hierarchy doesn't forget. Fast but undependable is not very attractive. As we learned in Chapter 1, the one great idea for **dependability** is redundancy. In this section we'll first go over the terms to define terms and measures associated with failure, and then show how redundancy can make nearly unforgettable memories.

Defining Failure

We start with an assumption that you have a specification of proper service. Users can then see a system alternating between two states of delivered service with respect to the service specification:

1. *Service accomplishment*, where the service is delivered as specified
2. *Service interruption*, where the delivered service is different from the specified service

Transitions from state 1 to state 2 are caused by *failures*, and transitions from state 2 to state 1 are called *restorations*. Failures can be permanent or intermittent. The latter is the more difficult case; it is harder to diagnose the problem when a system oscillates between the two states. Permanent failures are far easier to diagnose.

This definition leads to two related terms: reliability and availability.

Reliability is a measure of the continuous service accomplishment—or, equivalently, of the time to failure—from a reference point. Hence, *mean time to failure* (MTTF) is a reliability measure. A related term is *annual failure rate* (AFR), which is just the percentage of devices that would be expected to fail in a year for a given MTTF. When MTTF gets large it can be misleading, while AFR leads to better intuition.

EXAMPLE

MTTF vs. AFR of Disks

Some disks today are quoted to have a 1,000,000-hour MTTF. As 1,000,000 hours is 1,000,000/(365 × 24) = 114 years, it would seem like they practically never fail. Warehouse-scale computers that run Internet services such as Search might have 50,000 servers. Assume each server has two disks. Use AFR to calculate how many disks we would expect to fail per year.

One year is 365 × 24 = 8760 hours. A 1,000,000-hour MTTF means an AFR of 8760/1,000,000 = 0.876%. With 100,000 disks, we would expect 876 disks to fail per year, or on average more than two disk failures per day!

ANSWER

Service interruption is measured as *mean time to repair* (MTTR). *Mean time between failures* (MTBF) is simply the sum of MTTF + MTTR. Although MTBF is widely used, MTTF is often the more appropriate term. *Availability* is then a measure of service accomplishment with respect to the alternation between the two states of accomplishment and interruption. Availability is statistically quantified as

$$\text{Availability} = \frac{\text{MTTF}}{(\text{MTTF} + \text{MTTR})}$$

Note that reliability and availability are actually quantifiable measures, rather than just synonyms for dependability. Shrinking MTTR can help availability as much as increasing MTTF. For example, tools for fault detection, diagnosis, and repair can help reduce the time to repair faults and thereby improve availability.

We want availability to be very high. One shorthand is to quote the number of "nines of availability" per year. For instance, a very good Internet service today offers 4 or 5 nines of availability. Given 365 days per year, which is 365 × 24 × 60 = 526,000 minutes, then the shorthand is decoded as follows:

One nine:	90%	=> 36.5 days of repair/year
Two nines:	99%	=> 3.65 days of repair/year
Three nines:	99.9%	=> 526 minutes of repair/year
Four nines:	99.99%	=> 52.6 minutes of repair/year
Five nines:	99.999%	=> 5.26 minutes of repair/year

and so on.

To increase MTTF, you can improve the quality of the components or design systems to continue operation in the presence of components that have failed. Hence, failure needs to be defined with respect to a context, as failure of a component may not lead to a failure of the system. To make this distinction clear, the term *fault* is used to mean failure of a component. Here are three ways to improve MTTF:



PREDICTION

1. *Fault avoidance:* Preventing fault occurrence by construction.
2. *Fault tolerance:* Using redundancy to allow the service to comply with the service specification despite faults occurring.
3. *Fault forecasting:* **Predicting** the presence and creation of faults, allowing the component to be replaced *before* it fails.

The Hamming Single Error Correcting, Double Error Detecting Code (SEC/DED)

Richard Hamming invented a popular redundancy scheme for memory, for which he received the Turing Award in 1968. To invent redundant codes, it is helpful to talk about how "close" correct bit patterns can be. What we call the *Hamming distance* is just the minimum number of bits that are different between any two correct bit patterns. For example, the distance between 011011 and 001111 is two. What happens if the minimum distance between members of a code is two, and we get a one-bit error? It will turn a valid pattern in a code to an invalid one. Thus, if we can detect whether members of a code are accurate or not, we can detect single bit errors, and can say we have a single bit **error detection code**.

error detection code A code that enables the detection of an error in data, but not the precise location and, hence, correction of the error.

Hamming used a *parity code* for error detection. In a parity code, the number of 1 s in a word is counted; the word has odd parity if the number of 1 s is odd and even otherwise. When a word is written into memory, the parity bit is also written (1 for odd, 0 for even). That is, the parity of the N+1 bit word should always be even. Then, when the word is read out, the parity bit is read and checked. If the parity of the memory word and the stored parity bit do not match, an error has occurred.

EXAMPLE

Calculate the parity of a byte with the value 31_{ten} and show the pattern stored to memory. Assume the parity bit is on the right. Suppose the most significant bit was inverted in memory, and then you read it back. Did you detect the error? What happens if the two most significant bits are inverted?

ANSWER

31_{ten} is 00011111_{two}, which has five 1s. To make parity even, we need to write a 1 in the parity bit, or 000111111_{two}. If the most significant bit is inverted when we read it back, we would see 100111111_{two} which has seven 1s. Since we expect even parity and calculated odd parity, we would signal an error. If the *two* most significant bits are inverted, we would see 110111111_{two} which has eight 1s or even parity, and we would *not* signal an error.

If there are 2 bits of error, then a 1-bit parity scheme will not detect any errors, since the parity will match the data with two errors. (Actually, a 1-bit parity scheme can detect any odd number of errors; however, the probability of having three errors is much lower than the probability of having two, so, in practice, a 1-bit parity code is limited to detecting a single bit of error.)

Of course, a parity code cannot correct errors, which Hamming wanted to do as well as detect them. If we used a code that had a minimum distance of 3, then any single bit error would be closer to the correct pattern than to any other valid pattern. He came up with an easy to understand mapping of data into a distance 3 code that we call *Hamming Error Correction Code* (ECC) in his honor. We use extra

parity bits to allow the position identification of a single error. Here are the steps to calculate Hamming ECC

1. Start numbering bits from 1 on the left, contrary to the traditional numbering of the rightmost bit being 0.
2. Mark all bit positions that are powers of 2 as parity bits (positions 1, 2, 4, 8, 16, …).
3. All other bit positions are used for data bits (positions 3, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, …).
4. The position of parity bit determines sequence of data bits that it checks (Figure 5.24 shows this coverage graphically) is:
   - Bit 1 (0001_{two}) checks bits (1,3,5,7,9,11,...), which are bits where rightmost bit of address is 1 (0001_{two}, 0011_{two}, 0101_{two}, 0111_{two}, 1001_{two}, 1011_{two},…).
   - Bit 2 (0010_{two}) checks bits (2,3,6,7,10,11,14,15,…), which are the bits where the second bit to the right in the address is 1.
   - Bit 4 (0100_{two}) checks bits (4–7, 12–15, 20–23,…), which are the bits where the third bit to the right in the address is 1.
   - Bit 8 (1000_{two}) checks bits (8–15, 24–31, 40–47,...), which are the bits where the fourth bit to the right in the address is 1.

   Note that each data bit is covered by two or more parity bits.
5. Set parity bits to create even parity for each group.

Bit position		1	2	3	4	5	6	7	8	9	10	11	12
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8
Parity bit coverage	p1	X		X		X		X		X		X	
	p2		X	X			X	X			X	X	
	p4				X	X	X	X					X
	p8								X	X	X	X	X

FIGURE 5.24 Parity bits, data bits, and field coverage in a Hamming ECC code for eight data bits.

In what seems like a magic trick, you can determine whether bits are incorrect by looking at the parity bits. Using the 12 bit code in Figure 5.24, if the value of the four parity calculations (p8,p4,p2,p1) was 0000, then there was no error. However, if the pattern was, say, 1010, which is 10_{ten}, then Hamming ECC tells us that bit 10 (d6) is an error. Since the number is binary, we can correct the error just by inverting the value of bit 10.

EXAMPLE

Assume one byte data value is 10011010_{two}. First show the Hamming ECC code for that byte, and then invert bit 10 and show that the ECC code finds and corrects the single bit error.

Leaving spaces for the parity bits, the 12 bit pattern is__ 1_ 0 0 1_ 1 0 1 0.

ANSWER

Position 1 checks bits 1,3,5,7,9, and 11, which we highlight:__ **1**_ **0** 0 **1**_ **1** 0 **1** 0. To make the group even parity, we should set bit 1 to 0.
Position 2 checks bits 2,3,6,7,10,11, which is 0_ **1**_ 0 **0 1**_ 1 **0 1** 0 or odd parity, so we set position 2 to a 1.
Position 4 checks bits 4,5,6,7,12, which is 0 1 1_ **0 0 1**_ 1 0 1, so we set it to a 1.
Position 8 checks bits 8,9,10,11,12, which is 0 1 1 1 0 0 1_ **1 0 1 0**, so we set it to a 0.
The final code word is 011100101010. Inverting bit 10 changes it to 011100101**1**10.
Parity bit 1 is 0 (**0**1**1**1**0**0**1**0**1**1**1**0 is four 1s, so even parity; this group is OK).
Parity bit 2 is 1 (0**11**10**0**1011**1**0 is five 1s, so odd parity; there is an error somewhere).
Parity bit 4 is 1 (011**100**101110 is two 1s, so even parity; this group is OK).
Parity bit 8 is 1 (0111001**0111**0 is three 1s, so odd parity; there is an error somewhere).
Parity bits 2 and 8 are incorrect. As 2 + 8 = 10, bit 10 must be wrong. Hence, we can correct the error by inverting bit 10: 011100101**0**10. Voila!

Hamming did not stop at single bit error correction code. At the cost of one more bit, we can make the minimum Hamming distance in a code be 4. This means we can correct single bit errors *and detect double bit errors*. The idea is to add a parity bit that is calculated over the whole word. Let's use a 4-bit data word as an example, which would only need 7 bits for single bit error detection. Hamming parity bits H (p1 p2 p3) are computed (even parity as usual) plus the even parity over the entire word, p4:

1	2	3	4	5	6	7	8
p_1	p_2	d_1	p_3	d_2	d_3	d_4	$\mathbf{p}_4$

Then the algorithm to correct one error and detect two is just to calculate parity over the ECC groups (H) as before plus one more over the whole group (p_4). There are four cases:

1. H is even and p_4 is even, so no error occurred.
2. H is odd and p_4 is odd, so a correctable single error occurred. (p_4 should calculate odd parity if one error occurred.)
3. H is even and p_4 is odd, a single error occurred in p_4 bit, not in the rest of the word, so correct the p_4 bit.

4. H is odd and p_4 is even, a double error occurred. (p_4 should calculate even parity if two errors occurred.)

Single Error Correcting/Double Error Detecting (SEC/DED) is common in memory for servers today. Conveniently, 8-byte data blocks can get SEC/DED with just one more byte, which is why many DIMMs are 72 bits wide.

Elaboration: To calculate how many bits are needed for SEC, let p be total number of parity bits and d number of data bits in $p + d$ bit word. If p error correction bits are to point to error bit ($p + d$ cases) plus one case to indicate that no error exists, we need:

$$2^p \geq p + d + 1 \text{ bits, and thus } p \geq \log(p + d + 1).$$

For example, for 8 bits data means $d = 8$ and $2^p \geq p + 8 + 1$, so $p = 4$. Similarly, $p = 5$ for 16 bits of data, 6 for 32 bits, 7 for 64 bits, and so on.

Elaboration: In very large systems, the possibility of multiple errors as well as complete failure of a single wide memory chip becomes significant. IBM introduced *chipkill* to solve this problem, and many big systems use this technology. (Intel calls their version SDDC.) Similar in nature to the RAID approach used for disks (see **Section 5.11**), Chipkill distributes the data and ECC information, so that the complete failure of a single memory chip can be handled by supporting the reconstruction of the missing data from the remaining memory chips. Assuming a 10,000-processor cluster with 4 GiB per processor, IBM calculated the following rates of *unrecoverable* memory errors in 3 years of operation:

- Parity only—about 90,000, or one unrecoverable (or undetected) failure every 17 minutes.
- SEC/DED only—about 3500, or about one undetected or unrecoverable failure every 7.5 hours.
- Chipkill—6, or about one undetected or unrecoverable failure every 2 months.

Hence, Chipkill is a requirement for warehouse-scale computers.

Elaboration: While single or double bit errors are typical for memory systems, networks can have bursts of bit errors. One solution is called *Cyclic Redundancy Check*. For a block of k bits, a transmitter generates an n-k bit frame check sequence. It transmits n bits exactly divisible by some number. The receiver divides the frame by that number. If there is no remainder, it assumes there is no error. If there is, the receiver rejects the message, and asks the transmitter to send again. As you might guess from Chapter 3, it is easy to calculate division for some binary numbers with a shift register, which made CRC codes popular even when hardware was more precious. Going even further, Reed-Solomon codes use Galois fields to *correct* multibit transmission errors, but now data are considered coefficients of a polynomial and the code space is values of a polynomial. The Reed-Solomon calculation is considerably more complicated than binary division!

5.6 Virtual Machines

Virtual machines (VM) were first developed in the mid-1960s, and they have remained an important part of mainframe computing over the years. Although largely ignored in the single-user PC era in the 1980s and 1990s, they have recently gained popularity due to

- The increasing importance of isolation and security in modern systems
- The failures in security and reliability of standard operating systems
- The sharing of a single computer among many unrelated users, in particular for Cloud computing
- The dramatic increases in raw speed of processors over the decades, which makes the overhead of VMs more acceptable

The broadest definition of VMs includes basically all emulation methods that provide a standard software interface, such as the Java VM. In this section, we are interested in VMs that provide a complete system-level environment at the binary *instruction set architecture* (ISA) level. Although some VMs run different ISAs in the VM from the native hardware, we assume they always match the hardware. Such VMs are called (Operating) *System Virtual Machines*. IBM VM/370, VirtualBox, VMware ESX Server, and Xen are examples.

System virtual machines present the illusion that the users have an entire computer to themselves, including a copy of the operating system. A single computer runs multiple VMs and can support a number of different *operating systems* (OSes). On a conventional platform, a single OS "owns" all the hardware resources, but with a VM, multiple OSes all share the hardware resources.

The software that supports VMs is called a *virtual machine monitor* (VMM) or *hypervisor*; the VMM is the heart of virtual machine technology. The underlying hardware platform is called the *host*, and its resources are shared among the *guest* VMs. The VMM determines how to map virtual resources to physical resources: a physical resource may be time-shared, partitioned, or even emulated in software. The VMM is much smaller than a traditional OS; the isolation portion of a VMM is perhaps only 10,000 lines of code.

Although our interest here is in VMs for improving protection, VMs provide two other benefits that are commercially significant:

1. *Managing software*. VMs provide an abstraction that can run the complete software stack, even including old operating systems like DOS. A typical deployment might be some VMs running legacy OSes, many running the current stable OS release, and a few testing the next OS release.
2. *Managing hardware*. One reason for multiple servers is to have each application running with the compatible version of the operating system on separate computers, as this separation can improve dependability. VMs

allow these separate software stacks to run independently yet share hardware, thereby consolidating the number of servers. Another example is that some VMMs support migration of a running VM to a different computer, either to balance load or to evacuate from failing hardware.

Hardware/ Software Interface

Amazon Web Services (AWS) uses the virtual machines in its Cloud computing offering EC2 for five reasons:

1. It allows AWS to protect users from each other while sharing the same server.
2. It simplifies software distribution within a warehouse-scale computer. A customer installs a virtual machine image configured with the appropriate software, and AWS distributes it to all the instances a customer wants to use.
3. Customers (and AWS) can reliably "kill" a VM to control resource usage when customers complete their work.
4. Virtual machines hide the identity of the hardware on which the customer is running, which means AWS can keep using old servers *and* introduce new, more efficient servers. The customer expects performance for instances to match their ratings in "EC2 Compute Units," which AWS defines: to "provide the equivalent CPU capacity of a 1.0–1.2 GHz 2007 AMD Opteron or 2007 Intel Xeon processor." Thanks to **Moore's Law**, newer servers clearly offer more EC2 Compute Units than older ones, but AWS can keep renting old servers as long as they are economical.
5. Virtual machine monitors can control the rate that a VM uses the processor, the network, and disk space, which allows AWS to offer many price points of instances of different types running on the same underlying servers. For example, in 2012 AWS offered 14 instance types, from small standard instances at $0.08 per hour to high I/O quadruple extra-large instances at $3.10 per hour.



In general, the cost of processor virtualization depends on the workload. User-level processor-bound programs have zero virtualization overhead, because the OS is rarely invoked, so everything runs at native speeds. I/O-intensive workloads are generally also OS-intensive, executing many system calls and privileged instructions that can result in high virtualization overhead. On the other hand, if the I/O-intensive workload is also *I/O-bound*, the cost of processor virtualization can be completely hidden, since the processor is often idle waiting for I/O.

The overhead is determined by both the number of instructions that must be emulated by the VMM and by how much time each takes to emulate. Hence, when the guest VMs run the same ISA as the host, as we assume here, the goal of the architecture and the VMM is to run almost all instructions directly on the native hardware.

Requirements of a Virtual Machine Monitor

What must a VM monitor do? It presents a software interface to guest software, it must isolate the state of guests from each other, and it must protect itself from guest software (including guest OSes). The qualitative requirements are:

- Guest software should behave on a VM exactly as if it were running on the native hardware, except for performance-related behavior or limitations of fixed resources shared by multiple VMs.
- Guest software should not be able to change the allocation of real system resources directly.

To "virtualize" the processor, the VMM must control just about everything—access to privileged state, I/O, exceptions, and interrupts—even though the guest VM and OS presently running are temporarily using them.

For example, in the case of a timer interrupt, the VMM would suspend the currently running guest VM, save its state, handle the interrupt, determine which guest VM to run next, and then load its state. Guest VMs that rely on a timer interrupt are provided with a virtual timer and an emulated timer interrupt by the VMM.

To be in charge, the VMM must be at a higher privilege level than the guest VM, which generally runs in user mode; this also ensures that the execution of any privileged instruction will be handled by the VMM. The basic system requirements to support VMMs are:

- At least two processor modes, system and user.
- A privileged subset of instructions that is available only in system mode, resulting in a trap if executed in user mode; all system resources must be controllable just via these instructions.

(Lack of) Instruction Set Architecture Support for Virtual Machines

If VMs are planned for during the design of the ISA, it's relatively easy to reduce both the number of instructions that must be executed by a VMM and improve their emulation speed. An architecture that allows the VM to execute directly on the hardware earns the title *virtualizable*, and the IBM 370 and the RISC-V architectures proudly bear that label.

Alas, since VMs have been considered for PC and server applications only fairly recently, most instruction sets were created without virtualization in mind. These culprits include x86 and most RISC architectures, including ARMv7 and MIPS.

Because the VMM must ensure that the guest system only interacts with virtual resources, a conventional guest OS runs as a user mode program on top of the VMM. Then, if a guest OS attempts to access or modify information related to hardware resources via a privileged instruction—for example, reading or writing a status bit that enables interrupts—it will trap to the VMM. The VMM can then affect the appropriate changes to corresponding real resources.

Hence, if any instruction that tries to read or write such sensitive information traps when executed in user mode, the VMM can intercept it and support a virtual version of the sensitive information, as the guest OS expects.

In the absence of such support, other measures must be taken. A VMM must take special precautions to locate all problematic instructions and ensure that they behave correctly when executed by a guest OS, thereby increasing the complexity of the VMM and reducing the performance of running the VM.

Protection and Instruction Set Architecture

Protection is a joint effort of architecture and operating systems, but architects had to modify some awkward details of existing instruction set architectures when virtual memory became popular.

For example, the x86 instruction POPF loads the flag registers from the top of the stack in memory. One of the flags is the *Interrupt Enable* (IE) flag. If you run the POPF instruction in user mode, rather than trap it, it simply changes all the flags except IE. In system mode, it does change the IE. Since a guest OS runs in user mode inside a VM, this is a problem, as it expects to see a changed IE.

Historically, IBM mainframe hardware and VMM took three steps to improve the performance of virtual machines:

1. Reduce the cost of processor virtualization.
2. Reduce interrupt overhead cost due to the virtualization.
3. Reduce interrupt cost by steering interrupts to the proper VM without invoking VMM.

AMD and Intel tried to address the first point in 2006 by reducing the cost of processor virtualization. It will be interesting to see how many generations of architecture and VMM modifications it will take to address all three points, and how long before virtual machines of the 21st century for x86 will be as efficient as the IBM mainframes and VMMs of the 1970s.

Elaboration: RISC-V traps all privileged instructions when running in user mode, so it supports *classical virtualization*, wherein the guest OS runs in user mode and the VMM runs in supervisor mode.

5.7 Virtual Memory

… a system has been devised to make the core drum combination appear to the programmer as a single level store, the requisite transfers taking place automatically.

Kilburn et al., One-level storage system, 1962

In earlier sections, we saw how caches provided fast access to recently-used portions of a program's code and data. Similarly, the main memory can act as a "cache" for the secondary storage, traditionally implemented with magnetic disks. This technique

virtual memory A technique that uses main memory as a "cache" for secondary storage.

is called **virtual memory**. Historically, there were two major motivations for virtual memory: to allow efficient and safe sharing of memory among several programs, such as for the memory needed by multiple virtual machines for Cloud computing, and to remove the programming burdens of a small, limited amount of main memory. Five decades after its invention, it's the former reason that reigns today.

Of course, to allow multiple virtual machines to share the same memory, we must be able to protect the virtual machines from each other, ensuring that a program can just read and write the portions of main memory that have been assigned to it. Main memory need contain only the active portions of the many virtual machines, just as a cache contains only the active portion of one program. Thus, the principle of locality enables virtual memory as well as caches, and virtual memory allows us to share the processor efficiently as well as the main memory.

We cannot know which virtual machines will share the memory with other virtual machines when we compile them. In fact, the virtual machines sharing the memory change dynamically while they are running. Because of this dynamic interaction, we would like to compile each program into its own *address space*—a separate range of memory locations accessible only to this program. Virtual memory implements the translation of a program's address space to **physical addresses**. This translation process enforces **protection** of a program's address space from other virtual machines.

physical address An address in main memory.

protection A set of mechanisms for ensuring that multiple processes sharing the processor, memory, or I/O devices cannot interfere, intentionally or unintentionally, with one another by reading or writing each other's data. These mechanisms also isolate the operating system from a user process.

The second motivation for virtual memory is to allow a single-user program to exceed the size of primary memory. Formerly, if a program became too large for memory, it was up to the programmer to make it fit. Programmers divided programs into pieces and then identified the pieces that were mutually exclusive. These *overlays* were loaded or unloaded under user program control during execution, with the programmer ensuring that the program at no time tried to access an overlay that was not loaded and that the overlays loaded never exceeded the total size of the memory. Overlays were traditionally organized as modules, each containing both code and data. Calls between procedures in different modules would lead to overlaying of one module with another.

As you can well imagine, this responsibility was a substantial burden on programmers. Virtual memory, which was invented to relieve programmers of this difficulty, automatically manages the two levels of the memory hierarchy represented by main memory (sometimes called *physical memory* to distinguish it from virtual memory) and secondary storage.

page fault An event that occurs when an accessed page is not present in main memory.

Although the concepts at work in virtual memory and in caches are the same, their differing historical roots have led to the use of different terminology. A virtual memory block is called a *page*, and a virtual memory miss is called a **page fault**. With virtual memory, the processor produces a **virtual address**, which is translated by a combination of hardware and software to a *physical address*, which in turn can be used to access main memory. Figure 5.25 shows the virtually addressed memory with pages mapped to main memory. This process is called *address mapping* or **address translation**. Today, the two memory hierarchy levels controlled by virtual memory are usually DRAMs and flash memory in personal mobile devices and

virtual address An address that corresponds to a location in virtual space and is translated by address mapping to a physical address when memory is accessed.

DRAMs and magnetic disks in servers (see Section 5.2). If we return to our library analogy, we can think of a virtual address as the title of a book and a physical address as the location of that book in the library, such as might be given by the Library of Congress call number.

address translation Also called **address mapping**. The process by which a virtual address is mapped to an address used to access memory.

Virtual memory also simplifies loading the program for execution by providing *relocation*. Relocation maps the virtual addresses used by a program to different physical addresses before the addresses are used to access memory. This relocation allows us to load the program anywhere in main memory. Furthermore, all virtual memory systems in use today relocate the program as a set of fixed-size blocks (pages), thereby eliminating the need to find a contiguous block of memory to allocate to a program; instead, the operating system needs only to find enough pages in main memory.

In virtual memory, the address is broken into a *virtual page number* and a *page offset*. Figure 5.26 shows the translation of the virtual page number to a *physical page number*. While RISC-V has a 64-bit address, the upper 16 bits are not used, so the address to be mapped is 48 bits. This figure assumes the physical memory is 1 TiB, or 2^{40} bytes, which needs a 40-bit address. The physical page number constitutes the upper portion of the physical address, while the page offset, which is not changed, constitutes the lower portion. The number of bits in the page offset field determines the page size. The number of pages addressable with the virtual address can be different than the number of pages addressable with the physical address. Having a larger number of virtual pages than physical pages is the basis for the illusion of an essentially unbounded amount of virtual memory.

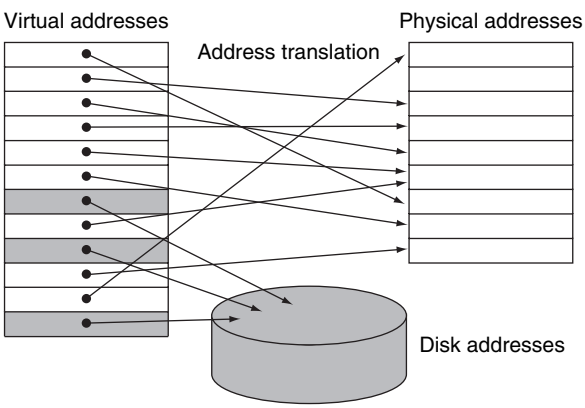


FIGURE 5.25 In virtual memory, blocks of memory (called *pages*) are mapped from one set of addresses (called *virtual addresses*) to another set (called *physical addresses*). The processor generates virtual addresses while the memory is accessed using physical addresses. Both the virtual memory and the physical memory are broken into pages, so that a virtual page is mapped to a physical page. Of course, it is also possible for a virtual page to be absent from main memory and not be mapped to a physical address; in that case, the page resides on disk. Physical pages can be shared by having two virtual addresses point to the same physical address. This capability is used to allow two different programs to share data or code.

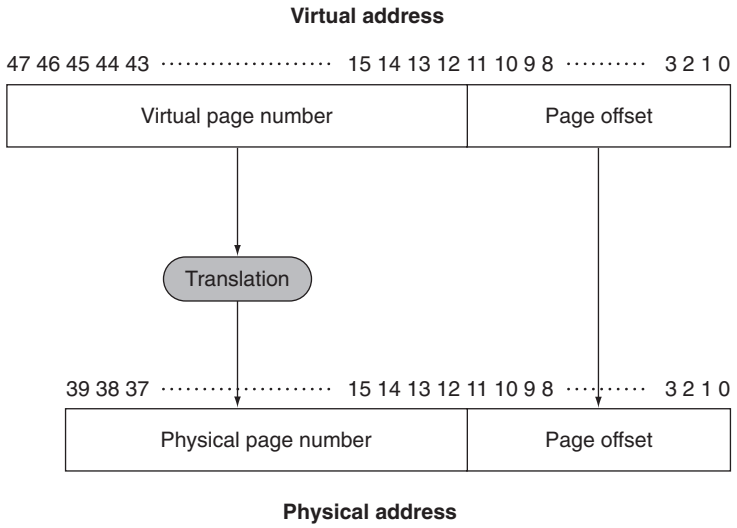


FIGURE 5.26 Mapping from a virtual to a physical address. The page size is 2^{12} = 4 KiB. The number of physical pages allowed in memory is 2^{28}, since the physical page number has 28 bits in it. Thus, main memory can have at most 1 TiB, while the virtual address space is 256 TiB. RISC-V allows physical memory to be up to 1 PiB; we chose 1 TiB because it is ample for many computers in 2016.

Many design choices in virtual memory systems are motivated by the high cost of a page fault. A page fault to disk will take millions of clock cycles to process. (The table on page 372 shows that main memory latency is about 100,000 times quicker than disk.) This enormous miss penalty, dominated by the time to get the first word for typical page sizes, leads to several key decisions in designing virtual memory systems:

- Pages should be large enough to try to amortize the high access time. Sizes from 4 KiB to 64 KiB are typical today. New desktop and server systems are being developed to support 32 KiB and 64 KiB pages, but new embedded systems are going in the other direction, to 1 KiB pages.
- Organizations that reduce the page fault rate are attractive. The primary technique used here is to allow fully associative placement of pages in memory.
- Page faults can be handled in software because the overhead will be small compared to the disk access time. In addition, software can afford to use clever algorithms for choosing how to place pages because even little reductions in the miss rate will pay for the cost of such algorithms.
- Write-through will not work for virtual memory, since writes take too long. Instead, virtual memory systems use write-back.

The next few subsections address these factors in virtual memory design.

Elaboration: We present the motivation for virtual memory as many virtual machines sharing the same memory, but virtual memory was originally invented so that many programs could share a computer as part of a timesharing system. Since many readers today have no experience with time-sharing systems, we use virtual machines to motivate this section.

Elaboration: RISC-V supports a variety of virtual memory configurations. In addition to the 48-bit virtual address scheme, which is a good fit for large servers in 2016, the architecture can support 39-bit and 57-bit virtual address spaces. All of these configurations use a page size of 4 Kibibytes.

Elaboration: For servers and even PCs, 32-bit address processors are problematic. Although we normally think of virtual addresses as much larger than physical addresses, the opposite can occur when the processor address size is small relative to the state of the memory technology. No single program or virtual machine can benefit, but a collection of programs or virtual machines running at the same time can benefit from not having to be swapped out of main memory or by running on parallel processors.

Elaboration: The discussion of virtual memory in this book focuses on paging, which uses fixed-size blocks. There is also a variable-size block scheme called **segmentation**. In segmentation, an address consists of two parts: a segment number and a segment offset. The segment number is mapped to a physical address, and the offset is *added* to find the actual physical address. Because the segment can vary in size, a bounds check is also needed to make sure that the offset is within the segment. The major use of segmentation is to support more powerful methods of protection and sharing in an address space. Most operating system textbooks contain extensive discussions of segmentation compared to paging and of the use of segmentation to share the address space logically. The major disadvantage of segmentation is that it splits the address space into logically separate pieces that must be manipulated as a two-part address: the segment number and the offset. Paging, in contrast, makes the boundary between page number and offset invisible to programmers and compilers.

segmentation A variable-size address mapping scheme in which an address consists of two parts: a segment number, which is mapped to a physical address, and a segment offset.

Segments have also been used as a method to extend the address space without changing the word size of the computer. Such attempts have been unsuccessful because of the awkwardness and performance penalties inherent in a two-part address, of which programmers and compilers must be aware.

Many architectures divide the address space into large fixed-size blocks that simplify protection between the operating system and user programs and increase the efficiency of implementing paging. Although these divisions are often called “segments,” this mechanism is much simpler than variable block size segmentation and is not visible to user programs; we discuss it in more detail shortly.

Placing a Page and Finding It Again

Because of the incredibly high penalty for a page fault, designers reduce page fault frequency by optimizing page placement. If we allow a virtual page to be mapped to any physical page, the operating system can then choose to replace any page it wants when a page fault occurs. For example, the operating system can use a sophisticated algorithm and complex data structures that track page usage to try to choose a page that will not be needed for a long time. The ability to use a clever and flexible replacement scheme reduces the page fault rate and simplifies the use of fully associative placement of pages.

As mentioned in Section 5.4, the difficulty in using fully associative placement is in locating an entry, since it can be anywhere in the upper level of the hierarchy. A full search is impractical. In virtual memory systems, we locate pages by using a table that indexes the main memory; this structure is called a **page table**, and it resides in main memory. A page table is indexed by the page number from the virtual address to discover the corresponding physical page number. Each program has its own page table, which maps the virtual address space of that program to main memory. In our library analogy, the page table corresponds to a mapping between book titles and library locations. Just as the card catalog may contain entries for books in another library on campus rather than the local branch library, we will see that the page table may contain entries for pages not present in memory. To indicate the location of the page table in memory, the hardware includes a register that points to the start of the page table; we call this the *page table register*. Assume for now that the page table is in a fixed and contiguous area of memory.

page table The table containing the virtual to physical address translations in a virtual memory system. The table, which is stored in memory, is typically indexed by the virtual page number; each entry in the table contains the physical page number for that virtual page if the page is currently in memory.

Hardware/ Software Interface

The page table, together with the program counter and the registers, specifies the *state* of a virtual machine. If we want to allow another virtual machine to use the processor, we must save this state. Later, after restoring this state, the virtual machine can continue execution. We often refer to this state as a *process*. The process is considered *active* when it is in possession of the processor; otherwise, it is considered *inactive*. The operating system can make a process active by loading the process's state, including the program counter, which will initiate execution at the value of the saved program counter.

The process's address space, and hence all the data it can access in memory, is defined by its page table, which resides in memory. Rather than save the entire page table, the operating system simply loads the page table register to point to the page table of the process it wants to make active. Each process has its own page table, since different processes use the same virtual addresses. The operating system is responsible for allocating the physical memory and updating the page tables, so that the virtual address spaces of distinct processes do not collide. As we will see shortly, the use of separate page tables also provides protection of one process from another.

Figure 5.27 uses the page table register, the virtual address, and the indicated page table to show how the hardware can form a physical address. A valid bit is used in each page table entry, just as we did in a cache. If the bit is off, the page is not present in main memory and a page fault occurs. If the bit is on, the page is in memory and the entry contains the physical page number.

Because the page table contains a mapping for every possible virtual page, no tags are required. In cache terminology, the index that is used to access the page table consists of the full block address, which in this case is the virtual page number.

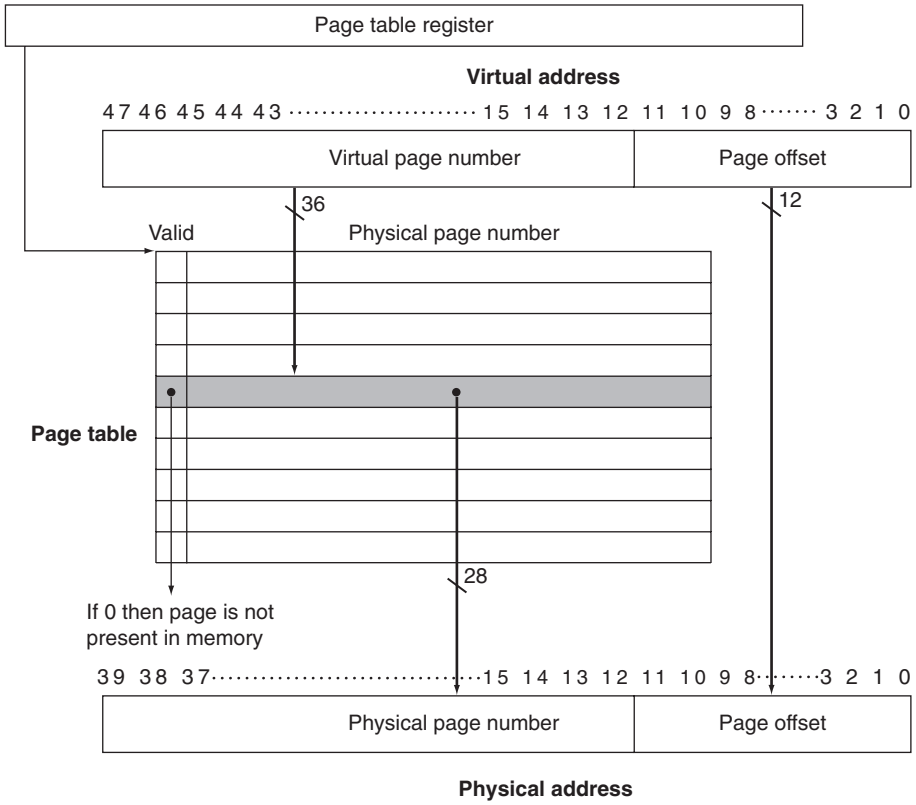


FIGURE 5.27 The page table is indexed with the virtual page number to obtain the corresponding portion of the physical address. We assume a 48-bit address. The page table pointer gives the starting address of the page table. In this figure, the page size is 2^{12} bytes, or 4 KiB. The virtual address space is 2^{48} bytes, or 256 TiB, and the physical address space is 2^{40} bytes, which allows main memory of up to 1 TiB. If RISC-V used a single page table as shown in this figure, the number of entries in the page table would be 2^{36}, or about 64 billion entries. (We'll see what RISC-V does to reduce the number of entries shortly.) The valid bit for each entry indicates whether the mapping is legal. If it is off, then the page is not present in memory. Although the page table entry shown here need only be 29 bits wide, it would typically be rounded up to a power of 2 bits for ease of indexing. The page table entries in RISC-V are 64 bits. The extra bits would be used to store additional information that needs to be kept on a per-page basis, such as protection.

Page Faults

If the valid bit for a virtual page is off, a page fault occurs. The operating system must be given control. This transfer is done with the exception mechanism, which we saw in Chapter 4 and will discuss again later in this section. Once the operating system gets control, it must find the page in the next level of the hierarchy (usually flash memory or magnetic disk) and decide where to place the requested page in the main memory.

The virtual address alone does not immediately tell us where the page is in secondary memory. Returning to our library analogy, we cannot find the location of a library book on the shelves just by knowing its title. Instead, we go to the catalog and look up the book, obtaining an address for the location on the shelves, such as the Library of Congress call number. Likewise, in a virtual memory system, we must keep track of the location in secondary memory of each page in virtual address space.

swap space The space on the disk reserved for the full virtual memory space of a process.

Because we do not know ahead of time when a page in memory will be replaced, the operating system usually creates the space on flash memory or disk for all the pages of a process when it creates the process. This space is called the **swap space**. At that time, it also creates a data structure to record where each virtual page is stored on disk. This data structure may be part of the page table or may be an auxiliary data structure indexed in the same way as the page table. Figure 5.28 shows the organization when a single table holds either the physical page number or the secondary memory address.

The operating system also creates a data structure that tracks which processes and which virtual addresses use each physical page. When a page fault occurs, if all the pages in main memory are in use, the operating system must choose a page to replace. Because we want to minimize the number of page faults, most operating systems try to choose a page that they hypothesize will not be needed soon. Using the past to predict the future, operating systems follow the *least recently used* (LRU) replacement scheme, which we mentioned in Section 5.4. The operating system searches for the least recently used page, assuming that a page that has not been used in a long time is less likely to be needed than a more recently accessed page. The replaced pages are written to swap space in secondary memory. In case you are wondering, the operating system is just another process, and these tables controlling memory are in memory; the details of this seeming contradiction will be explained shortly.

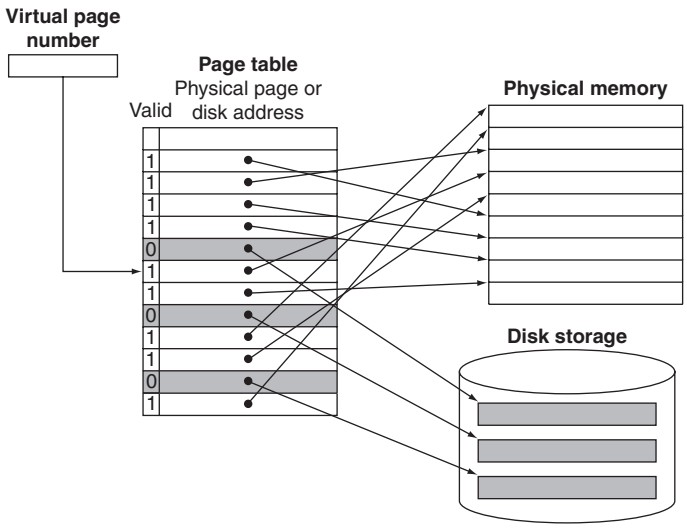


FIGURE 5.28 The page table maps each page in virtual memory to either a page in main memory or a page stored on disk, which is the next level in the hierarchy. The virtual page number is used to index the page table. If the valid bit is on, the page table supplies the physical page number (i.e., the starting address of the page in memory) corresponding to the virtual page. If the valid bit is off, the page currently resides only on disk, at a specified disk address. In many systems, the table of physical page addresses and disk page addresses, while logically one table, is stored in two separate data structures. Dual tables are justified in part because we must keep the disk addresses of all the pages, even if they are currently in main memory. Remember that the pages in main memory and the pages on disk are the same size.

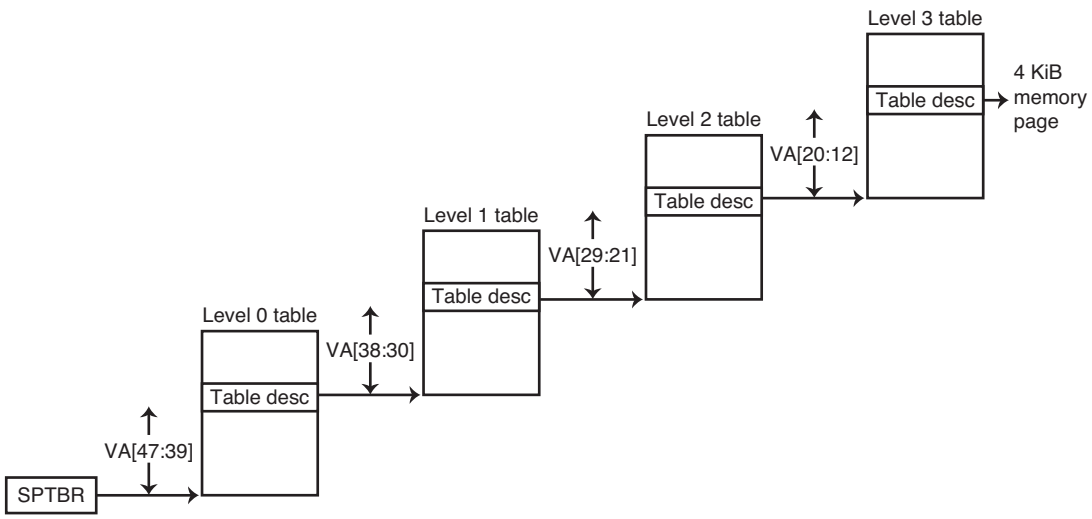


FIGURE 5.29 RISC-V uses four levels of tables to translate a 48-bit virtual address into a 40-bit physical address. Rather than needing 64 billion page table entries for the single page table in Figure 5.27, this hierarchical approach needs just a tiny fraction. Each step of the translation uses 9 bits of the virtual address to find the next level table, until the upper 36 bits of the virtual address are mapped to the physical address of the desired 4 KiB page. Each RISC-V page table entry is 8 bytes, so the 512 entries of a table fill a single 4 KiB page. The *Supervisor Page Table Base Register* (SPTBR) gives the starting address of the first page table.

Hardware/ Software Interface

reference bit Also called **use bit** or **access bit**. A field that is set whenever a page is accessed and that is used to implement LRU or other replacement schemes.

Implementing a completely accurate LRU scheme is too expensive, since it requires updating a data structure on *every* memory reference. Thus, most operating systems approximate LRU by keeping track of which pages have and which pages have not been recently used. To help the operating system estimate the LRU pages, RISC-V computers provide a **reference bit**, sometimes called a **use bit** or **access bit**, which is set whenever a page is accessed. The operating system periodically clears the reference bits and later records them so it can determine which pages were touched during a particular time period. With this usage information, the operating system can select a page that is among the least recently referenced (detected by having its reference bit off). If this bit is not provided by the hardware, the operating system must find another way to estimate which pages have been accessed.

Virtual Memory for Large Virtual Addresses

The caption in Figure 5.27 points out that with a single level page table for a 48-bit address with 4 KiB pages, we need 64 billion table entries. As each page table entry is 8 bytes for RISC-V, it would require 0.5 TiB just to map the virtual addresses to physical addresses! Moreover, there could be hundreds of processes running, each with its own page table. That much memory for translation would be unaffordable even for the largest systems.

A range of techniques is used to reduce the amount of storage required for the page table. The five techniques below aim at reducing the total maximum storage required as well as minimizing the main memory dedicated to page tables:

1. The simplest technique is to keep a limit register that restricts the size of the page table for a given process. If the virtual page number becomes larger than the contents of the limit register, entries must be added to the page table. This technique allows the page table to grow as a process consumes more space. Thus, the page table will only be large if the process is using many pages of virtual address space. This technique requires that the address space expand in just one direction.

2. Allowing growth in only one direction is not sufficient, since most languages require two areas whose size is expandable: one area holds the stack, and the other area holds the heap. Because of this duality, it is convenient to divide the page table and let it grow from the highest address down, as well as from the lowest address up. This means that there will be two separate page tables and two separate limits. The use of two page tables breaks the address space into two segments. The high-order bit of an address usually determines which segment and thus which page table to use for that address. Since the high-order address bit specifies the segment, each segment can be

as large as one-half of the address space. A limit register for each segment specifies the current size of the segment, which grows in units of pages. Unlike the type of segmentation discussed in the second elaboration on page 423, this form of segmentation is invisible to the application program, although not to the operating system. The major disadvantage of this scheme is that it does not work well when the address space is used in a sparse fashion rather than as a contiguous set of virtual addresses.

3. Another approach to reducing the page table size is to apply a hashing function to the virtual address so that the page table need be only the size of the number of *physical* pages in main memory. Such a structure is called an *inverted page table*. Of course, the lookup process is slightly more complex with an inverted page table, because we can no longer just index the page table.

4. To reduce the actual main memory tied up in page tables, most modern systems also allow the page tables to be paged. Although this sounds tricky, it works by using the same basic ideas of virtual memory and simply allowing the page tables to reside in the virtual address space. In addition, there are some small but critical problems, such as a never-ending series of page faults, which must be avoided. How these problems are overcome is both very detailed and typically highly processor-specific. In brief, these problems are avoided by placing all the page tables in the address space of the operating system and placing at least some of the page tables for the operating system in a portion of main memory that is physically addressed and is always present and thus never in secondary memory.

5. Multiple levels of page tables can also be used to reduce the total amount of page table storage, and this is the solution that RISC-V uses to reduce the memory footprint of address translation. Figure 5.29 above shows the four levels of address translation to go from a 48-bit virtual address to a 40-bit physical address of a 4 KiB page. Address translation happens by first looking in the level 0 table, using the highest-order bits of the address. If the address in this table is valid, the next set of high-order bits is used to index the page table indicated by the segment table entry, and so on. Thus, the level 0 table maps the virtual address to a 512 GB (2^{39} bytes) region. The level 1 table in turn maps the virtual address to a 1 GB (2^{30}) region. The next level maps this down to a 2 MB (2^{21}) region. The final table maps the virtual address to the 4 KiB (2^{12}) memory page. This scheme allows the address space to be used in a sparse fashion (multiple noncontiguous segments can be active) without having to allocate the entire page table. Such schemes are particularly useful with very large address spaces and in software systems that require noncontiguous allocation. The primary disadvantage of this multi-level mapping is the more complex process for address translation.

What about Writes?

The difference between the access time to the cache and main memory is tens to hundreds of cycles, and write-through schemes can be used, although we need a write buffer to hide the latency of the write from the processor. In a virtual memory system, writes to the next level of the hierarchy (disk) can take millions of processor clock cycles; therefore, building a write buffer to allow the system to write-through to disk would be completely impractical. Instead, virtual memory systems must use write-back, performing the individual writes into the page in memory, and copying the page back to secondary memory when it is replaced in the main memory.

Hardware/ Software Interface

A write-back scheme has another major advantage in a virtual memory system. Because the disk transfer time is small compared with its access time, copying back an entire page is much more efficient than writing individual words back to the disk. A write-back operation, although faster than transferring separate words, is still costly. Thus, we would like to know whether a page *needs* to be copied back when we choose to replace it. To track whether a page has been written since it was read into the memory, a *dirty bit* is added to the page table. The dirty bit is set when any word in a page is written. If the operating system chooses to replace the page, the dirty bit indicates whether the page needs to be written out before its location in memory can be given to another page. Hence, a modified page is often called a *dirty* page.

Making Address Translation Fast: the TLB

Since the page tables are stored in main memory, every memory access by a program can take at least twice as long: one memory access to obtain the physical address and a second access to get the data. The key to improving access performance is to rely on locality of reference to the page table. When a translation for a virtual page number is used, it will probably be needed again soon, because the references to the words on that page have both temporal and spatial locality.

Accordingly, modern processors include a special cache that keeps track of recently used translations. This special address translation cache is traditionally referred to as a **translation-lookaside buffer (TLB)**, although it would be more accurate to call it a translation cache. The TLB corresponds to that little piece of paper we typically use to record the location of a set of books we look up in the card catalog; rather than continually searching the entire catalog, we record the location of several books and use the scrap of paper as a cache of Library of Congress call numbers.

translation-lookaside buffer (TLB) A cache that keeps track of recently used address mappings to try to avoid an access to the page table.

Figure 5.30 shows that each tag entry in the TLB holds a portion of the virtual page number, and each data entry of the TLB holds a physical page number. Because we

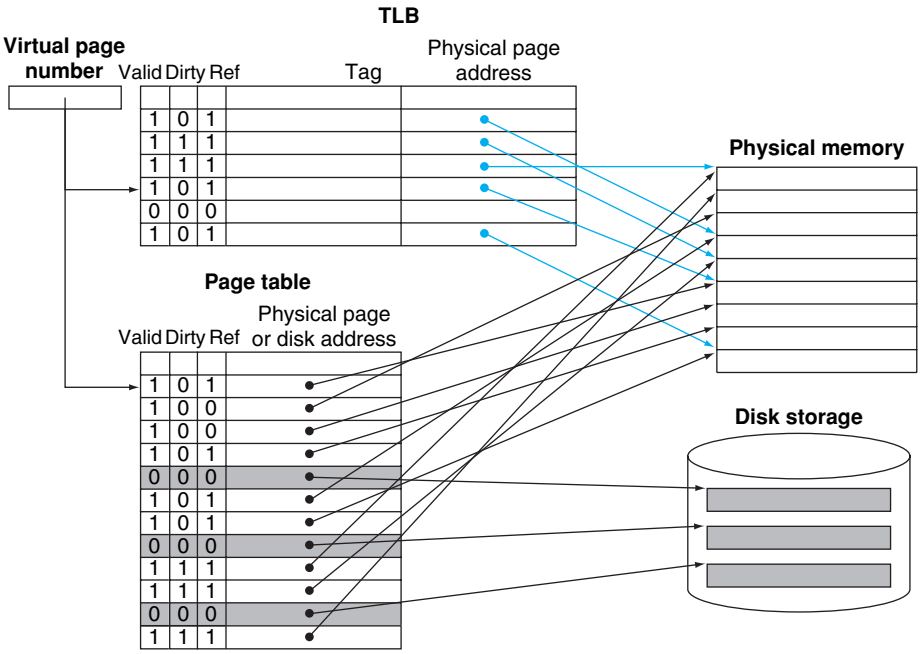


FIGURE 5.30 The TLB acts as a cache of the page table for the entries that map to physical pages only. The TLB contains a subset of the virtual-to-physical page mappings that are in the page table. The TLB mappings are shown in color. Because the TLB is a cache, it must have a tag field. If there is no matching entry in the TLB for a page, the page table must be examined. The page table either supplies a physical page number for the page (which can then be used to build a TLB entry) or indicates that the page resides on disk, in which case a page fault occurs. Since the page table has an entry for every virtual page, no tag field is needed; in other words, unlike a TLB, a page table is *not* a cache.

access the TLB instead of the page table on every reference, the TLB will need to include other status bits, such as the dirty and the reference bits. Although Figure 5.30 shows a single page table, TLBs work fine with multi-level page tables as well. The TLB simply loads the physical address and protection tags from the last level page table.

On every reference, we look up the virtual page number in the TLB. If we get a hit, the physical page number is used to form the address, and the corresponding reference bit is turned on. If the processor is performing a write, the dirty bit is also turned on. If a miss in the TLB occurs, we must determine whether it is a page fault or merely a TLB miss. If the page exists in memory, then the TLB miss indicates only that the translation is missing. In such cases, the processor can handle the TLB miss by loading the translation from the (last-level) page table into the TLB and then trying the reference again. If the page is not present in memory, then the TLB miss indicates a true page fault. In this case, the processor invokes the operating system using an exception. Because the TLB has many fewer entries than the number of pages in main memory, TLB misses will be much more frequent than true page faults.

TLB misses can be handled either in hardware or in software. In practice, with care there can be little performance difference between the two approaches, because the basic operations are the same in either case.

After a TLB miss occurs and the missing translation has been retrieved from the page table, we will need to select a TLB entry to replace. Because the reference and dirty bits are contained in the TLB entry, we need to copy these bits back to the page table entry when we replace an entry. These bits are the only portion of the TLB entry that can be changed. Using write-back—that is, copying these entries back at miss time rather than when they are written—is very efficient, since we expect the TLB miss rate to be small. Some systems use other techniques to approximate the reference and dirty bits, eliminating the need to write into the TLB except to load a new table entry on a miss.

Some typical values for a TLB might be

- TLB size: 16–512 entries
- Block size: 1–2 page table entries (typically 4–8 bytes each)
- Hit time: 0.5–1 clock cycle
- Miss penalty: 10–100 clock cycles
- Miss rate: 0.01%–1%

Designers have used a wide variety of associativities in TLBs. Some systems use small, fully associative TLBs because a fully associative mapping has a lower miss rate; furthermore, since the TLB is small, the cost of a fully associative mapping is not too high. Other systems use large TLBs, often with small associativity. With a fully associative mapping, choosing the entry to replace becomes tricky since implementing a hardware LRU scheme is too expensive. Furthermore, since TLB misses are much more frequent than page faults and thus must be handled more cheaply, we cannot afford an expensive software algorithm, as we can for page faults. As a result, many systems provide some support for randomly choosing an entry to replace. We'll examine replacement schemes in a little more detail in Section 5.8.

The Intrinsity FastMATH TLB

To see these ideas in a real processor, let's take a closer look at the TLB of the Intrinsity FastMATH. The memory system uses 4 KiB pages and just a 32-bit address space; thus, the virtual page number is 20 bits long. The physical address is the same size as the virtual address. The TLB contains 16 entries, it is fully associative, and it is shared between the instruction and data references. Each entry is 64 bits wide and contains a 20-bit tag (which is the virtual page number for that TLB entry), the corresponding physical page number (also 20 bits), a valid bit, a dirty bit, and other bookkeeping bits. Like most MIPS systems, it uses software to handle TLB misses.

Figure 5.31 shows the TLB and one of the caches, while Figure 5.32 shows the steps in processing a read or write request. When a TLB miss occurs, the hardware saves the page number of the reference in a special register and generates an

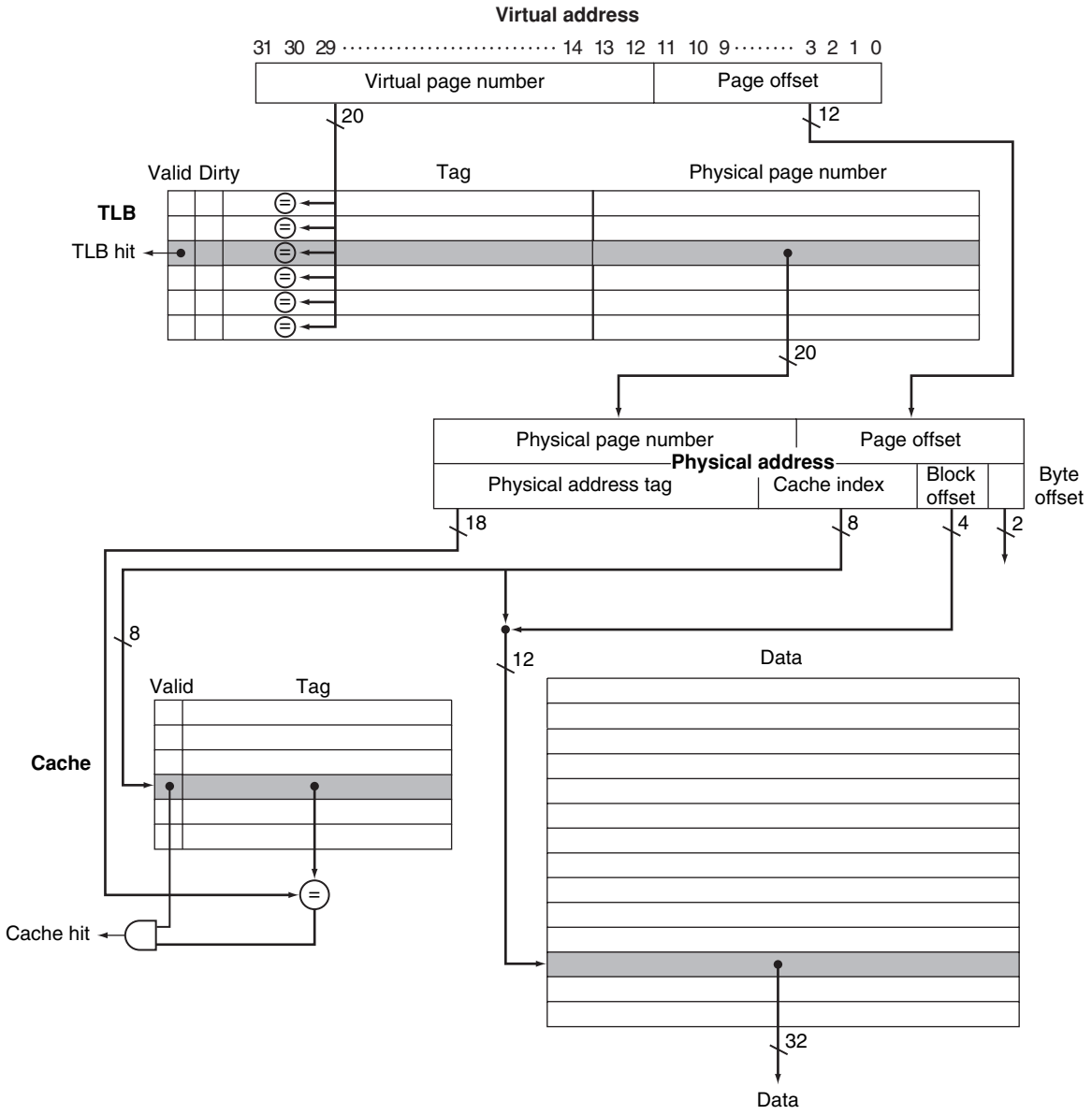


FIGURE 5.31 The TLB and cache implement the process of going from a virtual address to a data item in the Intrinsity FastMATH. This figure shows the organization of the TLB and the data cache, assuming a 4 KiB page size. Note that the address size for this computer is just 32 bits. This diagram focuses on a read; Figure 5.32 describes how to handle writes. Note that unlike Figure 5.12, the tag and data RAMs are split. By addressing the long but narrow data RAM with the cache index concatenated with the block offset, we select the desired word in the block without a 16:1 multiplexor. While the cache is direct mapped, the TLB is fully associative. Implementing a fully associative TLB requires that every TLB tag be compared against the virtual page number, since the entry of interest can be anywhere in the TLB. (See content addressable memories in the *Elaboration* on page 400.) If the valid bit of the matching entry is on, the access is a TLB hit, and bits from the physical page number together with bits from the page offset form the index that is used to access the cache.

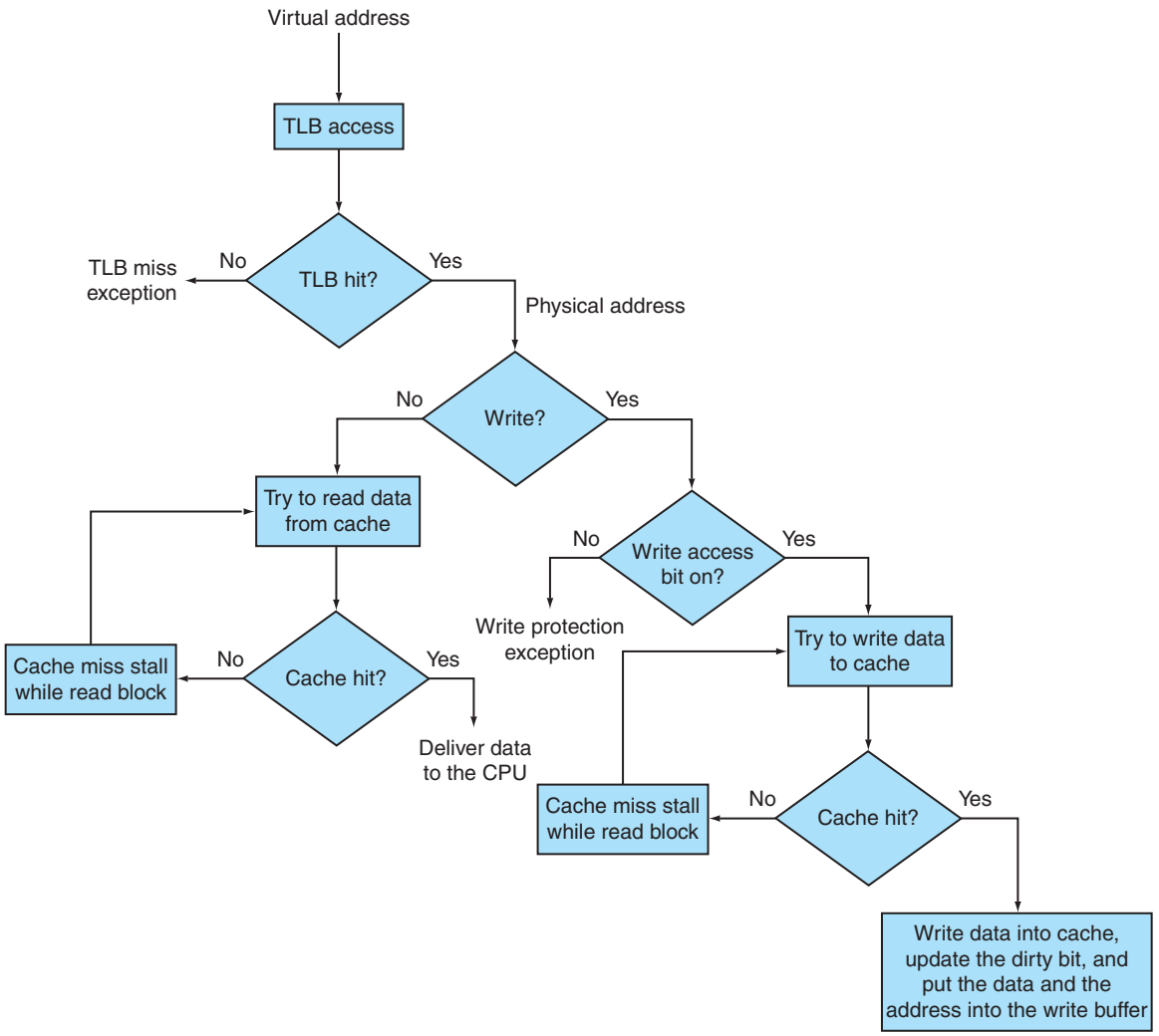


FIGURE 5.32 Processing a read or a write-through in the Intrinsity FastMATH TLB and cache. If the TLB generates a hit, the cache can be accessed with the resulting physical address. For a read, the cache generates a hit or miss and supplies the data or causes a stall while the data are brought from memory. If the operation is a write, a portion of the cache entry is overwritten for a hit and the data are sent to the write buffer if we assume write-through. A write miss is just like a read miss except that the block is modified after it is read from memory. Write-back requires writes to set a dirty bit for the cache block, and a write buffer is loaded with the whole block only on a read miss or write miss if the block to be replaced is dirty. Notice that a TLB hit and a cache hit are independent events, but a cache hit can only occur after a TLB hit occurs, which means that the data must be present in memory. The relationship between TLB misses and cache misses is examined further in the following example and the exercises at the end of this chapter. Note that the address size for this computer is just 32 bits.

exception. The exception invokes the operating system, which handles the miss in software. To find the physical address for the missing page, a TLB miss indexes the page table using the page number of the virtual address and the page table register, which indicates the starting address of the active process page table. Using a special set of system instructions that can update the TLB, the operating system places the physical address from the page table into the TLB. A TLB miss takes about 13 clock cycles, assuming the code and the page table entry are in the instruction cache and data cache, respectively. A true page fault occurs if the page table entry does not have a valid physical address. The hardware maintains an index that indicates the recommended entry to replace; it is chosen randomly.

There is an extra complication for write requests: namely, the write access bit in the TLB must be checked. This bit prevents the program from writing into pages for which it has only read access. If the program attempts a write and the write access bit is off, an exception is generated. The write access bit forms part of the protection mechanism, which we will discuss shortly.

Integrating Virtual Memory, TLBs, and Caches

Our virtual memory and cache systems work together as a hierarchy, so that data cannot be in the cache unless it is present in main memory. The operating system helps maintain this hierarchy by flushing the contents of any page from the cache when it decides to migrate that page to secondary memory. At the same time, the OS modifies the page tables and TLB, so that an attempt to access any data on the migrated page will generate a page fault.

Under the best of circumstances, a virtual address is translated by the TLB and sent to the cache where the appropriate data are found, retrieved, and sent back to the processor. In the worst case, a reference can miss in all three components of the memory hierarchy: the TLB, the page table, and the cache. The following example illustrates these interactions in more detail.

Overall Operation of a Memory Hierarchy

EXAMPLE

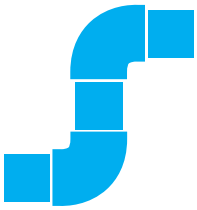
In a memory hierarchy like that of Figure 5.31, which includes a TLB and a cache organized as shown, a memory reference can encounter three different types of misses: a TLB miss, a page fault, and a cache miss. Consider all the combinations of these three events with one or more occurring (seven possibilities). For each possibility, state whether this event can actually occur and under what circumstances.

ANSWER

Figure 5.33 shows all combinations and whether each is possible in practice.

TLB	Page table	Cache	Possible? If so, under what circumstance?
Hit	Hit	Miss	Possible, although the page table is never really checked if TLB hits.
Miss	Hit	Hit	TLB misses, but entry found in page table; after retry, data is found in cache.
Miss	Hit	Miss	TLB misses, but entry found in page table; after retry, data misses in cache.
Miss	Miss	Miss	TLB misses and is followed by a page fault; after retry, data must miss in cache.
Hit	Miss	Miss	Impossible: cannot have a translation in TLB if page is not present in memory.
Hit	Miss	Hit	Impossible: cannot have a translation in TLB if page is not present in memory.
Miss	Miss	Hit	Impossible: data cannot be allowed in cache if the page is not in memory.

FIGURE 5.33 The possible combinations of events in the TLB, virtual memory system, and cache. Three of these combinations are impossible, and one is possible (TLB hit, page table hit, cache miss) but never detected.



PIPELINING

Elaboration: Figure 5.33 assumes that all memory addresses are translated to physical addresses before the cache is accessed. In this organization, the cache is *physically indexed* and *physically tagged* (both the cache index and tag are physical, rather than virtual, addresses). In such a system, the amount of time to access memory, assuming a cache hit, must accommodate both a TLB access and a cache access; of course, these accesses can be **pipelined**.

Alternatively, the processor can index the cache with an address that is completely or partially virtual. This is called a **virtually addressed cache**, and it uses tags that are virtual addresses; hence, such a cache is *virtually indexed* and *virtually tagged*. In such caches, the address translation hardware (TLB) is unused during the normal cache access, since the cache is accessed with a virtual address that has not been translated to a physical address. This takes the TLB out of the critical path, reducing cache latency. When a cache miss occurs, however, the processor needs to translate the address to a physical address so that it can fetch the cache block from main memory.

virtually addressed cache A cache that is accessed with a virtual address rather than a physical address.

When the cache is accessed with a virtual address and pages are shared between processes (which may access them with different virtual addresses), there is the possibility of **aliasing**. Aliasing occurs when the same object has two names—in this case, two virtual addresses for the same page. This ambiguity creates a problem, because a word on such a page may be cached in two different locations, each corresponding to distinct virtual addresses. This ambiguity would allow one program to write the data without the other program being aware that the data had changed. Completely virtually addressed caches either introduce design limitations on the cache and TLB to reduce aliases or require the operating system, and possibly the user, to take steps to ensure that aliases do not occur.

aliasing A situation in which two addresses access the same object; it can occur in virtual memory when there are two virtual addresses for the same physical page.

A common compromise between these two design points is caches that are virtually indexed—sometimes using just the page-offset portion of the address, which is really a physical address since it is not translated—but use physical tags. These designs, which are *virtually indexed but physically tagged*, attempt to achieve the performance advantages of virtually indexed caches with the architecturally simpler advantages of a **physically addressed cache**. For example, there is no alias problem in this case. Figure 5.31 assumed a 4 KiB page size, but it's really 16 KiB, so the Intrinsity FastMATH can use this trick. To pull it off, there must be careful coordination between the minimum page size, the cache size, and associativity. RISC-V requires caches to *behave as*

physically addressed cache A cache that is addressed by a physical address.

though physically tagged and indexed, but it does not mandate this implementation. For example, virtually indexed, physically tagged data caches could use additional logic to ensure that software cannot tell the difference.

Implementing Protection with Virtual Memory

Perhaps the most important function of virtual memory today is to allow sharing of a single main memory by multiple processes, while providing memory protection among these processes and the operating system. The protection mechanism must ensure that although multiple processes are sharing the same main memory, one renegade process cannot write into the address space of another user process or into the operating system either intentionally or unintentionally. The write access bit in the TLB can protect a page from being written. Without this level of protection, computer viruses would be even more widespread.

Hardware/Software Interface

To enable the operating system to implement protection in the virtual memory system, the hardware must provide at least the three basic capabilities summarized below. Note that the first two are the same requirements as needed for virtual machines (Section 5.6).

1. Support at least two modes that indicate whether the running process is a user process or an operating system process, variously called a **supervisor** process, a **kernel** process, or an *executive* process.
2. Provide a portion of the processor state that a user process can read but not write. This state includes the user/supervisor mode bit, which dictates whether the processor is in user or supervisor mode, the page table pointer, and the TLB. To write these elements, the operating system uses special instructions that are only available in supervisor mode.
3. Provide mechanisms whereby the processor can go from user mode to supervisor mode and vice versa. The first direction is typically accomplished by a **system call** exception, implemented as a special instruction (`ecall` in the RISC-V instruction set) that transfers control to a dedicated location in supervisor code space. As with any other exception, the program counter from the point of the system call is saved in the *supervisor exception program counter* (SEPC), and the processor is placed in supervisor mode. To return to user mode from the exception, use the *supervisor exception return* (`sret`) instruction, which resets to user mode and jumps to the address in SEPC.

supervisor mode Also called **kernel mode**. A mode indicating that a running process is an operating system process.

system call A special instruction that transfers control from user mode to a dedicated location in supervisor code space, invoking the exception mechanism in the process.

By using these mechanisms and storing the page tables in the operating system's address space, the operating system can change the page tables while preventing a user process from changing them, ensuring that a user process can access only the storage provided to it by the operating system.

We also want to prevent a process from reading the data of another process. For example, we wouldn't want a student program to read the teacher's grades while they were in the processor's memory. Once we begin sharing main memory, we must provide the ability for a process to protect its data from both reading and writing by another process; otherwise, sharing the main memory will be a mixed blessing!

Remember that each process has its own virtual address space. Thus, if the operating system keeps the page tables organized so that the independent virtual pages map to disjoint physical pages, one process will not be able to access another's data. Of course, this also requires that a user process be unable to change the page table mapping. The operating system can assure safety if it prevents the user process from modifying its own page tables. However, the operating system must be able to modify the page tables. Placing the page tables in the protected address space of the operating system satisfies both requirements.

When processes want to share information in a limited way, the operating system must assist them, since accessing the information of another process requires changing the page table of the accessing process. The write access bit can be used to restrict the sharing to just read sharing, and, like the rest of the page table, this bit can be changed only by the operating system. To allow another process, say, P1, to read a page owned by process P2, P2 would ask the operating system to create a page table entry for a virtual page in P1's address space that points to the same physical page that P2 wants to share. The operating system could use the write protection bit to prevent P1 from writing the data, if that was P2's wish. Any bits that determine the access rights for a page must be included in both the page table and the TLB, because the page table is accessed only on a TLB *miss*.

context switch
A changing of the internal state of the processor to allow a different process to use the processor that includes saving the state needed to return to the currently executing process.

Elaboration: When the operating system decides to change from running process P1 to running process P2 (called a **context switch** or *process switch*), it must ensure that P2 cannot get access to the page tables of P1 because that would compromise protection. If there is no TLB, it suffices to change the page table register to point to P2's page table (rather than to P1's); with a TLB, we must clear the TLB entries that belong to P1—both to protect the data of P1 and to force the TLB to load the entries for P2. If the process switch rate were high, this could be quite inefficient. For example, P2 might load only a few TLB entries before the operating system switched back to P1. Unfortunately, P1 would then find that all its TLB entries were gone and would have to pay TLB misses to reload them. This problem arises because the virtual addresses used by P1 and P2 are the same, and we must clear out the TLB to avoid confusing these addresses.

A common alternative is to extend the virtual address space by adding a *process identifier* or *task identifier*. The Intrinsity FastMATH has an 8-bit *address space ID* (ASID) field for this purpose. This small field identifies the currently running process; it is kept in a register loaded by the operating system when it switches processes. RISC-V also offers ASID to reduce TLB flushes on context switches. The process identifier is concatenated to the tag portion of the TLB, so that a TLB hit occurs only if both the page number *and* the process identifier match. This combination eliminates the need to clear the TLB, except on rare occasions.

Similar problems can occur for a cache, since on a process switch, the cache will contain data from the running process. These problems arise in different ways for physically addressed and virtually addressed caches, and a variety of solutions, such as process identifiers, are used to ensure that a process gets its own data.

Handling TLB Misses and Page Faults

Although the translation of virtual to physical addresses with a TLB is straightforward when we get a TLB hit, as we saw earlier, handling TLB misses and page faults is more complex. A TLB miss occurs when no entry in the TLB matches a virtual address. Recall that a TLB miss can indicate one of two possibilities:

1. The page is present in memory, and we need only create the missing TLB entry.
2. The page is not present in memory, and we need to transfer control to the operating system to deal with a page fault.

Handling a TLB miss or a page fault requires using the exception mechanism to interrupt the active process, transferring control to the operating system, and later resuming execution of the interrupted process. A page fault will be recognized sometime during the clock cycle used to access memory. To restart the instruction after the page fault is handled, the program counter of the instruction that caused the page fault must be saved. The *supervisor exception program counter* (SEPC) register is used to hold this value.

In addition, a TLB miss or page fault exception must be asserted by the end of the same clock cycle that the memory access occurs, so that the next clock cycle will begin exception processing rather than continue normal instruction execution. If the page fault was not recognized in this clock cycle, a load instruction could overwrite a register, and this could be disastrous when we try to restart the instruction. For example, consider the instruction `lb x10, 0(x10)`: the computer must be able to prevent the write pipeline stage from occurring; otherwise, it could not properly restart the instruction, since the contents of `x10` would have been destroyed. A similar complication arises on stores. We must prevent the write into memory from actually completing when there is a page fault; this is usually done by deasserting the write control line to the memory.

Hardware/Software Interface

Between the time we begin executing the exception handler in the operating system and the time that the operating system has saved all the state of the process, the operating system is particularly vulnerable. For instance, if another exception occurred when we were processing the first exception in the operating system, the control unit would overwrite the exception link register, making it impossible to return to the instruction that caused the page fault! We can avoid this disaster by providing the ability to **disable** and **enable exceptions**. When an exception first occurs, the processor sets a bit that disables all other exceptions; this could happen at the same time the processor sets the supervisor mode bit. The operating system will then save just enough state to allow it to recover if another exception occurs—namely, the *supervisor exception rogram counter* (SEPC) and the *supervisor exception cause* (SCAUSE) registers, which as we saw in Chapter 4 records the reason for the exception. SEPC and SCAUSE in RISC-V are two of the special control registers that help with exceptions, TLB misses, and page faults. The operating system can then re-enable exceptions. These steps make sure that exceptions will not cause the processor to lose any state and thereby be unable to restart execution of the interrupting instruction.

exception enable Also called interrupt enable. A signal or action that controls whether the process responds to an exception or not; necessary for preventing the occurrence of exceptions during intervals before the processor has safely saved the state needed to restart.

Once the operating system knows the virtual address that caused the page fault, it must complete three steps:

1. Look up the page table entry using the virtual address and find the location of the referenced page in secondary memory.
2. Choose a physical page to replace; if the chosen page is dirty, it must be written out to secondary memory before we can bring a new virtual page into this physical page.
3. Start a read to bring the referenced page from secondary memory into the chosen physical page.

Of course, this last step will take millions of processor clock cycles for disks (so will the second if the replaced page is dirty); accordingly, the operating system will usually select another process to execute in the processor until the disk access completes. Because the operating system has saved the state of the process, it can freely give control of the processor to another process.

When the read of the page from secondary memory is complete, the operating system can restore the state of the process that originally caused the page fault and execute the instruction that returns from the exception. This instruction will reset the processor from kernel to user mode, as well as restore the program counter. The user process then re-executes the instruction that faulted, accesses the requested page successfully, and continues execution.

Page fault exceptions for data accesses are difficult to implement properly in a processor because of a combination of three characteristics:

1. They occur in the middle of instructions, unlike instruction page faults.
2. The instruction cannot be completed before handling the exception.
3. After handling the exception, the instruction must be restarted as if nothing had occurred.

restartable instruction An instruction that can resume execution after an exception is resolved without the exception's affecting the result of the instruction.

Making instructions **restartable**, so that the exception can be handled and the instruction later continued, is relatively easy in an architecture like the RISC-V. Because each instruction writes only one data item and this write occurs at the end of the instruction cycle, we can simply prevent the instruction from completing (by not writing) and restart the instruction at the beginning.

Elaboration: For processors with more complex instructions that can touch many memory locations and write many data items, making instructions restartable is much harder. Processing one instruction may generate a number of page faults in the middle of the instruction. For example, x86 processors have block move instructions that touch thousands of data words. In such processors, instructions often cannot be restarted from the beginning, as we do for RISC-V instructions. Instead, the instruction must be interrupted and later continued midstream in its execution. Resuming an instruction in the middle of its execution usually requires saving some special state, processing the exception, and restoring that special state. Making this work properly requires careful and detailed coordination between the exception-handling code in the operating system and the hardware.

Elaboration: Rather than pay an extra level of indirection on every memory access, the Virtual Memory Monitor (Section 5.6) maintains a *shadow page table* that maps directly from the guest virtual address space to the physical address space of the hardware. By detecting all modifications to the guest's page table, the VMM can ensure the shadow page table entries being used by the hardware for translations correspond to those of the guest OS environment, with the exception of the correct physical pages substituted for the real pages in the guest tables. Hence, the VMM must trap any attempt by the guest OS to change its page table or to access the page table pointer. This is commonly done by write protecting the guest page tables and trapping any access to the page table pointer by a guest OS. As noted above, the latter happens naturally if accessing the page table pointer is a privileged operation.

Elaboration: The final portion of the architecture to virtualize is I/O. This is by far the most difficult part of system virtualization because of the increasing number of I/O devices attached to the computer *and* the expanding diversity of I/O device types. Another difficulty is the sharing of a real device among multiple VMs, and yet another comes from supporting the myriad of device drivers that are required, especially if different guest OSes are supported on the same VM system. The VM illusion can be maintained by giving each VM generic versions of each type of I/O device driver, and then leaving it to the VMM to handle real I/O.

Elaboration: In addition to virtualizing the instruction set for a virtual machine, another challenge is virtualization of virtual memory, as each guest OS in every virtual machine manages its own set of page tables. To make this work, the VMM separates the notions of *real* and *physical memory* (which are often treated synonymously), and makes real memory a separate, intermediate level between virtual memory and physical memory. (Some use the terms *virtual memory, physical memory*, and *machine memory* to name the same three levels.) The guest OS maps virtual memory to real memory via its page tables, and the VMM page tables map the guest's real memory to physical memory. The virtual memory architecture is typically specified via page tables, as in IBM VM/370, the x86, and RISC-V.

Summary

Virtual memory is the name for the level of memory hierarchy that manages caching between the main memory and secondary memory. Virtual memory allows a single program to expand its address space beyond the limits of main memory. More importantly, virtual memory supports sharing of the main memory among multiple, simultaneously active processes, in a protected manner.

Managing the memory hierarchy between main memory and disk is challenging because of the high cost of page faults. Several techniques are used to reduce the miss rate:

1. Pages are made large to take advantage of spatial locality and to reduce the miss rate.
2. The mapping between virtual addresses and physical addresses, which is implemented with a page table, is made fully associative so that a virtual page can be placed anywhere in main memory.
3. The operating system uses techniques, such as LRU and a reference bit, to choose which pages to replace.

Writes to secondary memory are expensive, so virtual memory uses a write-back scheme and also tracks whether a page is unchanged (using a dirty bit) to avoid writing clean pages.

The virtual memory mechanism provides address translation from a virtual address used by the program to the physical address space used for accessing memory. This address translation allows protected sharing of the main memory and provides several additional benefits, such as simplifying memory allocation. Ensuring that processes are protected from each other requires that only the operating system can change the address translations, which is implemented by preventing user programs from altering the page tables. Controlled sharing of pages between processes can be implemented with the help of the operating system and access bits in the page table that indicate whether the user program has read or write access to a page.

If a processor had to access a page table resident in memory to translate every access, virtual memory would be too expensive, as caches would be pointless! Instead, a TLB acts as a cache for translations from the page table. Addresses are then translated from virtual to physical using the translations in the TLB.

Caches, virtual memory, and TLBs all rely on a common set of principles and policies. The next section discusses this common framework.

Understanding Program Performance

Although virtual memory was invented to enable a small memory to act as a large one, the performance difference between secondary memory and main memory means that if a program routinely accesses more virtual memory than it has physical memory, it will run very slowly. Such a program would be continuously swapping pages between main memory and secondary memory, called *thrashing*. Thrashing is a disaster if it occurs, but it is rare. If your program thrashes, the easiest solution is to run it on a computer with more memory or buy more memory for your computer. A more complex choice is to re-examine your algorithm and data structures to see if you can change the locality and thereby reduce the number of pages that your program uses simultaneously. This set of popular pages is informally called the *working set*.

A more common performance problem is TLB misses. Since a TLB might handle only 32–64 page entries at a time, a program could easily see a high TLB miss rate, as the processor may access less than a quarter mebibyte directly: 64×4 KiB $= 0.25$ MiB. For example, TLB misses are often a challenge for Radix Sort. To try to alleviate this problem, most computer architectures now offer support for larger page sizes. For instance, in addition to the minimum 4 KiB page, RISC-V hardware supports 2 MiB and 1 GiB pages. Hence, if a program uses large page sizes, it can access more memory directly without TLB misses.

The practical challenge is getting the operating system to allow programs to select these larger page sizes. Once again, the more complex solution to reducing TLB misses is to re-examine the algorithm and data structures to reduce the working set of pages; given the importance of memory accesses to performance and the frequency of TLB misses, some programs with large working sets have been redesigned with that goal.

Elaboration: RISC-V supports the larger page sizes via the multi-level page table of Figure 5.29. In addition to pointing at the next level page table in levels 1 and 2, it allows a *superpage translation* to map the virtual address to a 1 GiB physical address (if the block translation is in level 1) or a 2 MiB physical address (if the block translation is in level 2).

5.8 A Common Framework for Memory Hierarchy

By now, you've recognized that the different types of memory hierarchies have a great deal in common. Although many of the aspects of memory hierarchies differ quantitatively, many of the policies and features that determine how a hierarchy functions are similar qualitatively. Figure 5.34 shows how some of the quantitative characteristics of memory hierarchies can differ. In the rest of this section, we will discuss the common operational alternatives for memory hierarchies, and how these determine their behavior. We will examine these policies in a series of four questions that apply between any two levels of a memory hierarchy, although for simplicity, we will primarily use terminology for caches.

Question 1: Where Can a Block Be Placed?

We have seen that block placement in the upper level of the hierarchy can use a range of schemes, from direct mapped to set associative to fully associative. As mentioned above, this entire range of schemes can be thought of as variations on a set-associative scheme where the number of sets and the number of blocks per set varies:

Scheme name	Number of sets	Blocks per set
Direct mapped	Number of blocks in cache	1
Set associative	$\frac{\text{Number of blocks in the cache}}{\text{Associativity}}$	Associativity (typically 2–16)
Fully associative	1	Number of blocks in the cache

Feature	Typical values for L1 caches	Typical values for L2 caches	Typical values for paged memory	Typical values for a TLB
Total size in blocks	250–2000	2500–25,000	16,000–250,000	40–1024
Total size in kilobytes	16–64	125–2000	1,000,000–1,000,000,000	0.25–16
Block size in bytes	16–64	64–128	4000–64,000	4–32
Miss penalty in clocks	10–25	100–1000	10,000,000–100,000,000	10–1000
Miss rates (global for L2)	2%–5%	0.1%–2%	0.00001%–0.0001%	0.01%–2%

FIGURE 5.34 The key quantitative design parameters that characterize the major elements of memory hierarchy in a computer. These are typical values for these levels as of 2012. Although the range of values is wide, this is partially because many of the values that have shifted over time are related; for example, as caches become larger to overcome larger miss penalties, block sizes also grow. While not shown, server microprocessors today also have L3 caches, which can be 2 to 8 MiB and contain many more blocks than L2 caches. L3 caches lower the L2 miss penalty to 30 to 40 clock cycles.

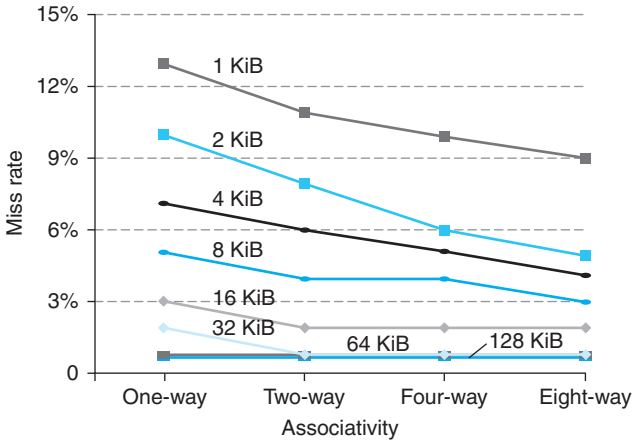


FIGURE 5.35 The data cache miss rates for each of eight cache sizes improve as the associativity increases. While the benefit of going from one-way (direct mapped) to two-way set associative is significant, the benefits of further associativity are smaller (e.g., 1–10% improvement going from two-way to four-way versus 20–30% improvement going from one-way to two-way). There is even less improvement in going from four-way to eight-way set associative, which, in turn, comes very close to the miss rates of a fully associative cache. Smaller caches obtain a significantly larger absolute benefit from associativity because the base miss rate of a small cache is larger. Figure 5.16 explains how these data were collected.

The advantage of increasing the degree of associativity is that it usually decreases the miss rate. The improvement in miss rate comes from reducing misses that compete for the same location. We will examine these in more detail shortly. First, let's look at how much improvement is gained. Figure 5.35 shows the miss rates for several cache sizes as associativity varies from direct mapped to eight-way set associative. The largest gains are obtained in going from direct mapped to two-way set associative, which yields between a 20% and 30% reduction in the miss rate. As cache sizes grow, the relative improvement from associativity increases only slightly; since the overall miss rate of a larger cache is lower, the opportunity for improving the miss rate decreases and the absolute improvement in the miss rate from associativity shrinks significantly. The potential disadvantages of associativity, as we mentioned earlier, are increased cost and slower access time.

Question 2: How Is a Block Found?

The choice of how we locate a block depends on the block placement scheme, since that dictates the number of possible locations. We can summarize the schemes as follows:

Associativity	Location method	Comparisons required
Direct mapped	Index	1
Set associative	Index the set, search among elements	Degree of associativity
Full	Search all cache entries	Size of the cache
	Separate lookup table	0

The choice among direct-mapped, set-associative, or fully associative mapping in any memory hierarchy will depend on the cost of a miss versus the cost of implementing associativity, both in time and in extra hardware. Including the L2 cache on the chip enables much higher associativity, because the hit times are not as critical and the designer does not have to rely on standard SRAM chips as the building blocks. Fully associative caches are prohibitive except for small sizes, where the cost of the comparators is not overwhelming and where the absolute miss rate improvements are greatest.

In virtual memory systems, a separate mapping table—the page table—is kept to index the memory. In addition to the storage needed for the table, using an index table requires an extra memory access. The choice of full associativity for page placement and the extra table is motivated by these facts:

1. Full associativity is beneficial, since misses are very expensive.
2. Full associativity allows software to use sophisticated replacement schemes that are designed to reduce the miss rate.
3. The full map can be easily indexed with no extra hardware and no searching required.

Therefore, virtual memory systems almost always use fully associative placement.

Set-associative placement is often used for caches and TLBs, where the access combines indexing and the search of a small set. A few systems have used direct-mapped caches because of their advantage in access time and simplicity. The advantage in access time occurs because finding the requested block does not depend on a comparison. Such design choices depend on many details of the implementation, such as whether the cache is on-chip, the technology used for implementing the cache, and the critical role of cache access time in determining the processor cycle time.

Question 3: Which Block Should Be Replaced on a Cache Miss?

When a miss occurs in an associative cache, we must decide which block to replace. In a fully associative cache, all blocks are candidates for replacement. If the cache is set associative, we must choose among the blocks in the set. Of course, replacement is easy in a direct-mapped cache because there is only one candidate.

There are the two primary strategies for replacement in set-associative or fully associative caches:

- *Random:* Candidate blocks are randomly selected, possibly using some hardware assistance.
- *Least recently used* (LRU): The block replaced is the one that has been unused for the longest time.

In practice, LRU is too costly to implement for hierarchies with more than a small degree of associativity (two to four, typically), since tracking the usage information is expensive. Even for four-way set associativity, LRU is often approximated—for

example, by keeping track of which pair of blocks is LRU (which requires 1 bit), and then tracking which block in each pair is LRU (which requires 1 bit per pair).

For larger associativity, either LRU is approximated or random replacement is used. In caches, the replacement algorithm is in hardware, which means that the scheme should be easy to implement. Random replacement is simple to build in hardware, and for a two-way set-associative cache, random replacement has a miss rate about 1.1 times higher than LRU replacement. As the caches become larger, the miss rate for both replacement strategies falls, and the absolute difference becomes small. In fact, random replacement can sometimes be better than the simple LRU approximations that are easily implemented in hardware.

In virtual memory, some form of LRU is always approximated, since even a tiny reduction in the miss rate can be important when the cost of a miss is enormous. Reference bits or equivalent functionality are often provided to make it easier for the operating system to track a set of less recently used pages. Because misses are so expensive and relatively infrequent, approximating this information primarily in software is acceptable.

Question 4: What Happens on a Write?

A key characteristic of any memory hierarchy is how it deals with writes. We have already seen the two basic options:

- *Write-through:* The information is written to both the block in the cache and the block in the lower level of the memory hierarchy (main memory for a cache). The caches in Section 5.3 used this scheme.
- *Write-back:* The information is written just to the block in the cache. The modified block is written to the lower level of the hierarchy only when it is replaced. Virtual memory systems always use write-back, for the reasons discussed in Section 5.7.

Both write-back and write-through have their advantages. The key advantages of write-back are the following:

- Individual words can be written by the processor at the rate that the cache, rather than the memory, can accept them.
- Multiple writes within a block require only one write to the lower level in the hierarchy.
- When blocks are written back, the system can make effective use of a high-bandwidth transfer, since the entire block is written.

Write-through has these advantages:

- Misses are simpler and cheaper because they never require a block to be written back to the lower level.
- Write-through is easier to implement than write-back, although to be realistic, a write-through cache will still need to use a write buffer.

Caches, TLBs, and virtual memory may initially look very different, but they rely on the same two principles of locality, and they can be understood by their answers to four questions:

Question 1:	Where can a block be placed?
Answer:	One place (direct mapped), a few places (set associative), or any place (fully associative).
Question 2:	How is a block found?
Answer:	There are four methods: indexing (as in a direct-mapped cache), limited search (as in a set-associative cache), full search (as in a fully associative cache), and a separate lookup table (as in a page table).
Question 3:	What block is replaced on a miss?
Answer:	Typically, either the least recently used or a random block.
Question 4:	How are writes handled?
Answer:	Each level in the hierarchy can use either write-through or write-back.

The BIG Picture

In virtual memory systems, only a write-back policy is practical because of the long latency of a write to the lower level of the hierarchy. The rate at which writes are generated by a processor generally exceeds the rate at which the memory system can process them, even allowing for physically and logically wider memories and burst modes for DRAM. Consequently, today lowest-level caches typically use write-back.

The Three Cs: An Intuitive Model for Understanding the Behavior of Memory Hierarchies

In this subsection, we look at a model that provides insight into the sources of misses in a memory hierarchy and how the misses will be affected by changes in the hierarchy. We will explain the ideas in terms of caches, although the ideas carry over directly to any other level in the hierarchy. In this model, all misses are classified into one of three categories (the **three Cs**):

- **Compulsory misses**: These are cache misses caused by the first access to a block that has never been in the cache. These are also called **cold-start misses**.
- **Capacity misses**: These are cache misses caused when the cache cannot contain all the blocks needed during execution of a program. Capacity misses occur when blocks are replaced and then later retrieved.
- **Conflict misses**: These are cache misses that occur in set-associative or direct-mapped caches when multiple blocks compete for the same set. Conflict misses are those misses in a direct-mapped or set-associative cache that are eliminated in a fully associative cache of the same size. These cache misses are also called **collision misses**.

three Cs model A cache model in which all cache misses are classified into one of three categories: compulsory misses, capacity misses, and conflict misses.

compulsory miss Also called **cold-start miss**. A cache miss caused by the first access to a block that has never been in the cache.

capacity miss A cache miss that occurs because the cache, even with full associativity, cannot contain all the blocks needed to satisfy the request.

conflict miss Also called **collision miss**. A cache miss that occurs in a set-associative or direct-mapped cache when multiple blocks compete for the same set and that are eliminated in a fully associative cache of the same size.

Figure 5.36 shows how the miss rate divides into the three sources. These sources of misses can be directly attacked by changing some aspect of the cache design. Since conflict misses arise straight from contention for the same cache block, increasing associativity reduces conflict misses. Associativity, however, may slow access time, leading to lower overall performance.

Capacity misses can easily be reduced by enlarging the cache; indeed, second-level caches have been growing steadily bigger for many years. Of course, when we make the cache larger, we must also be careful about increasing the access time, which could lead to lower overall performance. Thus, first-level caches have been growing slowly, if at all.

Because compulsory misses are generated by the first reference to a block, the primary way for the cache system to reduce the number of compulsory misses is to increase the block size. This will reduce the number of references required to touch each block of the program once, because the program will consist of fewer cache blocks. As mentioned above, increasing the block size too much can have a negative effect on performance because of the increase in the miss penalty.

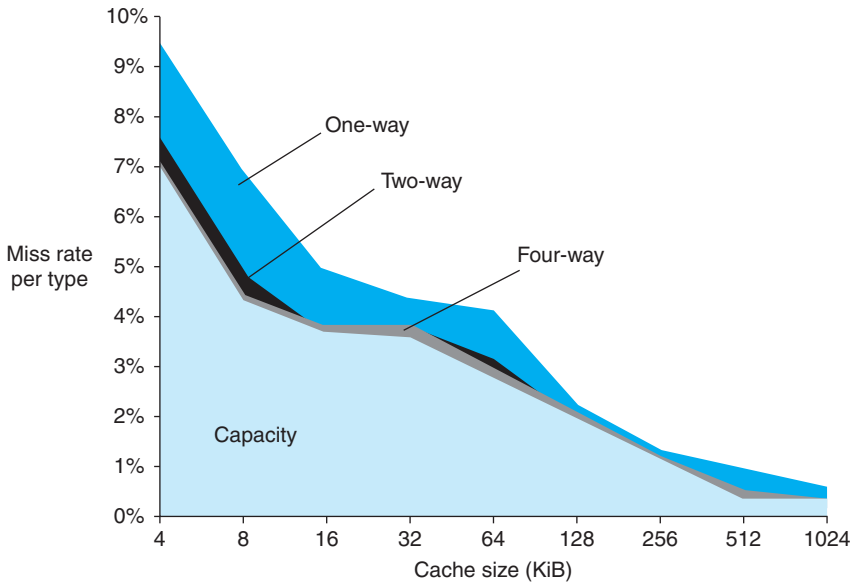


FIGURE 5.36 The miss rate can be broken into three sources of misses. This graph shows the total miss rate and its components for a range of cache sizes. These data are for the SPEC CPU2000 integer and floating-point benchmarks and are from the same source as the data in Figure 5.35. The compulsory miss component is 0.006% and cannot be seen in this graph. The next component is the capacity miss rate, which depends on cache size. The conflict portion, which depends both on associativity and on cache size, is shown for a range of associativities from one-way to eight-way. In each case, the labeled section corresponds to the increase in the miss rate that occurs when the associativity is changed from the next higher degree to the labeled degree of associativity. For example, the section labeled *two-way* indicates the additional misses arising when the cache has associativity of two rather than four. Thus, the difference in the miss rate incurred by a direct-mapped cache versus a fully associative cache of the same size is given by the sum of the sections marked *four-way, two-way*, and *one-way*. The difference between eight-way and four-way is so small that it is difficult to see on this graph.

Design change	Effect on miss rate	Possible negative performance effect
Increases cache size	Decreases capacity misses	May increase access time
Increases associativity	Decreases miss rate due to conflict misses	May increase access time
Increases block size	Decreases miss rate for a wide range of block sizes due to spatial locality	Increases miss penalty. Very large block could increase miss rate

FIGURE 5.37 Memory hierarchy design challenges.

The BIG Picture

The challenge in designing memory hierarchies is that every change that potentially improves the miss rate can also negatively affect overall performance, as Figure 5.37 summarizes. This combination of positive and negative effects is what makes the design of a memory hierarchy interesting.

The decomposition of misses into the three Cs is a useful qualitative model. In real cache designs, many of the design choices interact, and changing one cache characteristic will often affect several components of the miss rate. Despite such shortcomings, this model is a useful way to gain insight into the performance of cache designs.

Check Yourself

Which of the following statements (if any) is generally true?

1. There is no way to reduce compulsory misses.
2. Fully associative caches have no conflict misses.
3. In reducing misses, associativity is more important than capacity.

5.9 Using a Finite-State Machine to Control a Simple Cache

We can now build control for a cache, just as we implemented control for the single-cycle and pipelined datapaths in Chapter 4. This section starts with a definition of a simple cache and then a description of *finite-state machines* (FSMs). It finishes with the FSM of a controller for this simple cache. **Section 5.12** goes into more depth, showing the cache and controller in a new hardware description language.

A Simple Cache

We're going to design a controller for a straightforward cache. Here are the key characteristics of the cache:

- Direct-mapped cache
- Write-back using write allocate
- Block size is four words (16 bytes or 128 bits)
- Cache size is 16 KiB, so it holds 1024 blocks
- 32-bit addresses
- The cache includes a valid bit and dirty bit per block

From Section 5.3, we can now calculate the fields of an address for the cache:

- Cache index is 10 bits
- Block offset is 4 bits
- Tag size is 32− (10+ 4) or 18 bits

The signals between the processor to the cache are

- 1-bit Read or Write signal
- 1-bit Valid signal, saying whether there is a cache operation or not
- 32-bit address
- 32-bit data from processor to cache
- 32-bit data from cache to processor
- 1-bit Ready signal, saying the cache operation is complete

The interface between the memory and the cache has the same fields as between the processor and the cache, except that the data fields are now 128 bits wide. The extra memory width is generally found in microprocessors today, which deal with either 32-bit or 64-bit words in the processor while the DRAM controller is often 128 bits. Making the cache block match the width of the DRAM simplified the design. Here are the signals:

- 1-bit Read or Write signal
- 1-bit Valid signal, saying whether there is a memory operation or not
- 32-bit address
- 128-bit data from cache to memory
- 128-bit data from memory to cache
- 1-bit Ready signal, saying the memory operation is complete

Note that the interface to memory is not a fixed number of cycles. We assume a memory controller that will notify the cache via the Ready signal when the memory read or write is finished.

Before describing the cache controller, we need to review finite-state machines, which allow us to control an operation that can take multiple clock cycles.

Finite-State Machines

To design the control unit for the single-cycle datapath, we used truth tables that specified the setting of the control signals based on the instruction class. For a cache, the control is more complex because the operation can be a series of steps. The control for a cache must specify both the signals to be set in any step and the next step in the sequence.

The most common multistep control method is based on **finite-state machines**, which are usually represented graphically. A finite-state machine consists of a set of states and directions on how to change states. The directions are defined by a **next-state function**, which maps the current state and the inputs to a new state. When we use a finite-state machine for control, each state also specifies a set of outputs that are asserted when the machine is in that state. The implementation of a finite-state machine usually assumes that all outputs that are not explicitly asserted are deasserted. Similarly, the correct operation of the datapath depends on the fact that a signal that is not explicitly asserted is deasserted, rather than acting as a don't care.

finite-state machine
A sequential logic function consisting of a set of inputs and outputs, a next-state function that maps the current state and the inputs to a new state, and an output function that maps the current state and possibly the inputs to a set of asserted outputs.

next-state function
A combinational function that, given the inputs and the current state, determines the next state of a finite-state machine.

Multiplexor controls are slightly different, since they select one of the inputs, whether they are 0 or 1. Thus, in the finite-state machine, we always specify the setting of all the multiplexor controls that we care about. When we implement the finite-state machine with logic, setting a control to 0 may be the default and therefore may not require any gates. A simple example of a finite-state machine appears in Appendix A, and if you are unfamiliar with the concept of a finite-state machine, you may want to examine Appendix A before proceeding.

A finite-state machine can be implemented with a temporary register that holds the current state and a block of combinational logic that determines both the data-path signals to be asserted and the next state. Figure 5.38 shows how such an implementation might look. **Appendix C** describes in detail how the finite-state machine is implemented using this structure. In Section A.3, the combinational control logic for a finite-state machine is implemented both with either a ROM (*read-only memory*) or a PLA (*programmable logic array*). (Also see Appendix A for a description of these logic elements.)

Elaboration: Note that this simple design is called a *blocking* cache, in that the processor must wait until the cache has finished the request. **Section 5.12** describes the alternative, which is called a *nonblocking* cache.

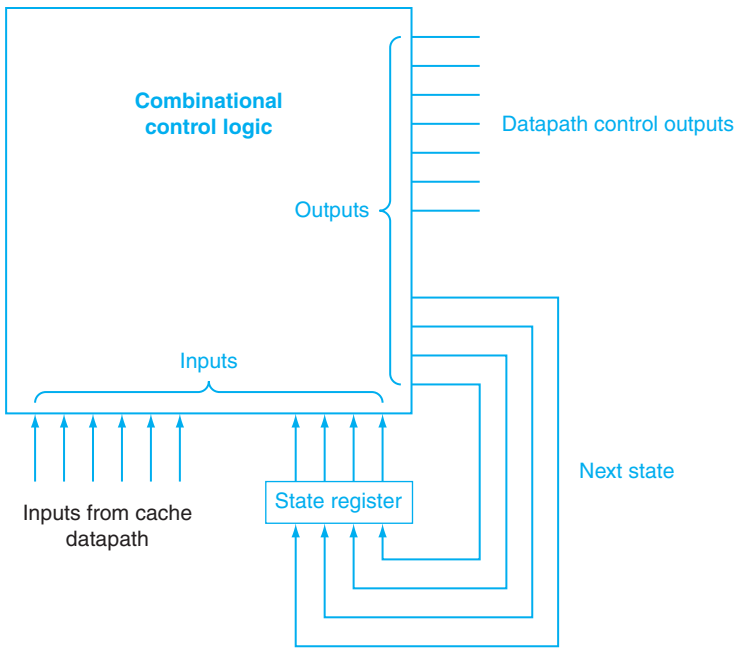


FIGURE 5.38 Finite-state machine controllers are typically implemented using a block of combinational logic and a register to hold the current state. The outputs of the combinational logic are the next-state number and the control signals to be asserted for the current state. The inputs to the combinational logic are the current state and any inputs used to determine the next state. Notice that in the finite-state machine used in this chapter, the outputs depend only on the current state, not on the inputs. We use color to indicate that these are control lines and logic versus data lines and logic. The *Elaboration* below explains this in more detail.

Elaboration: The style of finite-state machine in this book is called a Moore machine, after Edward Moore. Its identifying characteristic is that the output depends only on the current state. For a Moore machine, the box labeled combinational control logic can be split into two pieces. One piece has the control output and only the state input, while the other has just the next-state output.

An alternative style of machine is a Mealy machine, named after George Mealy. The Mealy machine allows both the input and the current state to be used to determine the output. Moore machines have potential implementation advantages in speed and size of the control unit. The speed advantages arise because the control outputs, which are needed early in the clock cycle, do not depend on the inputs, but only on the current state. In Appendix A, when the implementation of this finite-state machine is taken down to logic gates, the size advantage can be clearly seen. The potential disadvantage of a Moore machine is that it may require additional states. For example, in situations where there is a one-state difference between two sequences of states, the Mealy machine may unify the states by making the outputs depend on the inputs.

FSM for a Simple Cache Controller

Figure 5.39 shows the four states of our simple cache controller:

- *Idle:* This state waits for a valid read or write request from the processor, which moves the FSM to the Compare Tag state.
- *Compare Tag:* As the name suggests, this state tests to see if the requested read or write is a hit or a miss. The index portion of the address selects the tag to be compared. If the data in the cache block referred to by the index portion of the address are valid, and the tag portion of the address matches the tag, then it is a hit. Either the data are read from the selected word if it is a load or written to the selected word if it is a store. The Cache Ready signal is then set. If it is a write, the dirty bit is set to 1. Note that a write hit also sets the valid bit and the tag field; while it seems unnecessary, it is included because the tag is a single memory, so to change the dirty bit we likewise need to change the valid and tag fields. If it is a hit and the block is valid, the FSM returns to the idle state. A miss first updates the cache tag and then goes either to the Write-Back state, if the block at this location has dirty bit value of 1, or to the Allocate state if it is 0.

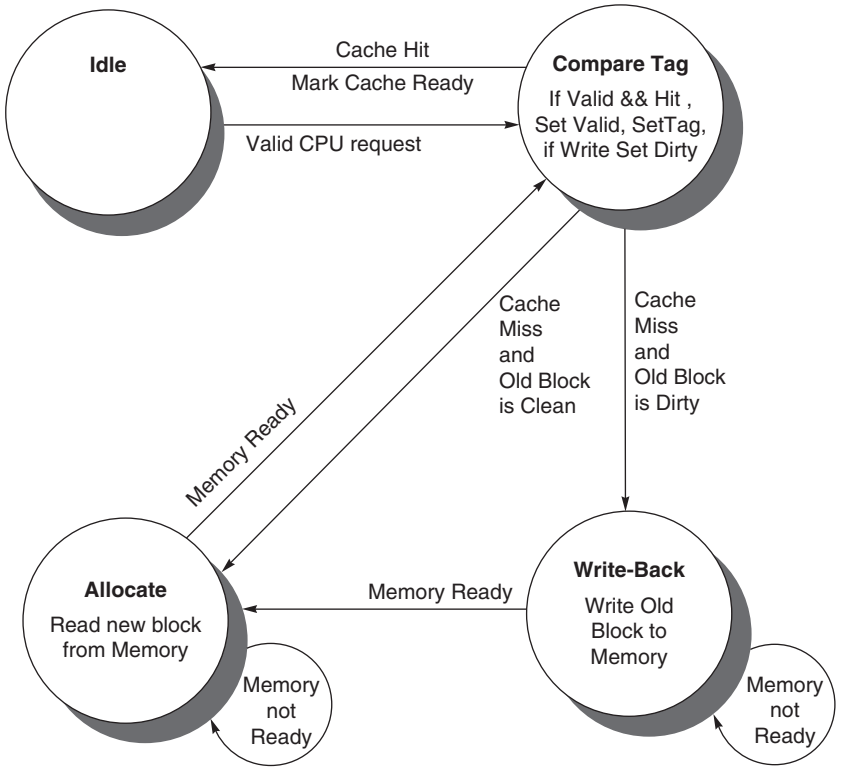


FIGURE 5.39 Four states of the simple controller.

- *Write-Back:* This state writes the 128-bit block to memory using the address composed from the tag and cache index. We remain in this state waiting for the Ready signal from memory. When the memory write is complete, the FSM goes to the Allocate state.
- *Allocate:* The new block is fetched from memory. We remain in this state waiting for the Ready signal from memory. When the memory read is complete, the FSM goes to the Compare Tag state. Although we could have gone to a new state to complete the operation instead of reusing the Compare Tag state, there is a good deal of overlap, including the update of the appropriate word in the block if the access was a write.

This simple model could easily be extended with more states to try to improve performance. For example, the Compare Tag state does both the compare and the read or write of the cache data in a single clock cycle. Often the compare and cache access are done in separate states to try to improve the clock cycle time. Another optimization would be to add a write buffer so that we could save the dirty block and then read the new block first so that the processor doesn't have to wait for two memory accesses on a dirty miss. The cache would next write the dirty block from the write buffer while the processor is operating on the requested data.

Section 5.12 goes into more detail about the FSM, showing the full controller in a hardware description language and a block diagram of this simple cache.

5.10 Parallelism and Memory Hierarchy: Cache Coherence

Given that a multicore multiprocessor means multiple processors on a single chip, these processors very likely share a common physical address space. Caching shared data introduces a new problem, because the view of memory held by two different processors is through their individual caches, which, without any additional precautions, could end up seeing two distinct values. Figure 5.40 illustrates the problem and shows how two different processors can have two different values for the same location. This difficulty is generally referred to as the *cache coherence problem*.

Informally, we could say that a memory system is coherent if any read of a data item returns the most recently written value of that data item. This definition, although intuitively appealing, is vague and simplistic; the reality is much more complex. This simple definition contains two different aspects of memory system behavior, both of which are critical to writing correct shared memory programs. The first aspect, called *coherence*, defines *what values* can be returned by a read. The second aspect, called *consistency*, determines *when* a written value will be returned by a read.

Let's look at coherence first. A memory system is coherent if

1. A read by a processor P to a location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P. Thus, in Figure 5.40, if CPU A were to read X after time step 3, it should see the value 1.
2. A read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses. Thus, in Figure 5.40, we need a mechanism so that the value 0 in the cache of CPU B is replaced by the value 1 after CPU A stores 1 into memory at address X in time step 3.
3. Writes to the same location are *serialized*; that is, two writes to the same location by any two processors are seen in the same order by all processors. For example, if CPU B stores 2 into memory at address X after time step 3, processors can never read the value at location X as 2 and then later read it as 1.

The first property simply preserves program order—we certainly expect this property to be true in uniprocessors, for instance. The second property defines the notion of what it means to have a coherent view of memory: if a processor could continuously read an old data value, we would clearly say that memory was incoherent.

The need for *write serialization* is more subtle, but equally important. Suppose we did not serialize writes, and processor P1 writes location X followed by P2 writing location X. Serializing the writes ensures that every processor will see the write done by P2 at some point. If we did not serialize the writes, it might be the

Time step	Event	Cache contents for CPU A	Cache contents for CPU B	Memory contents for location X
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A stores 1 into X	1	0	1

FIGURE 5.40 The cache coherence problem for a single memory location (X), read and written by two processors (A and B). We initially assume that neither cache contains the variable and that X has the value 0. We also assume a write-through cache; a write-back cache adds some additional but similar complications. After the value of X has been written by A, A's cache and the memory both contain the new value, but B's cache does not, and if B reads the value of X, it will receive 0!

case that some processor could see the write of P2 first and then see the write of P1, maintaining the value written by P1 indefinitely. The simplest way to avoid such difficulties is to ensure that all writes to the same location are seen in the identical order, which we call *write serialization*.

Basic Schemes for Enforcing Coherence

In a cache coherent multiprocessor, the caches provide both *migration* and *replication* of shared data items:

- *Migration:* A data item can be moved to a local cache and used there in a transparent fashion. Migration reduces both the latency to access a shared data item that is allocated remotely and the bandwidth demand on the shared memory.
- *Replication:* When shared data are being simultaneously read, the caches make a copy of the data item in the local cache. Replication reduces both latency of access and contention for a read shared data item.

Supporting migration and replication is critical to performance in accessing shared data, so many multiprocessors introduce a hardware protocol to maintain coherent caches. The protocols to maintain coherence for multiple processors are called *cache coherence protocols*. Key to implementing a cache coherence protocol is tracking the state of any sharing of a data block.

The most popular cache coherence protocol is *snooping*. Every cache that has a copy of the data from a block of physical memory also has a copy of the sharing status of the block, but no centralized state is kept. The caches are all accessible via some broadcast medium (a bus or network), and all cache controllers monitor or *snoop* on the medium to determine whether or not they have a copy of a block that is requested on a bus or switch access.

In the following section we explain snooping-based cache coherence as implemented with a shared bus, but any communication medium that broadcasts cache misses to all processors can be used to implement a snooping-based coherence scheme. This broadcasting to all caches makes snooping protocols simple to implement but also limits their scalability.

Snooping Protocols

One method of enforcing coherence is to ensure that a processor has exclusive access to a data item before it writes that item. This style of protocol is called a *write invalidate protocol* because it invalidates copies in other caches on a write. Exclusive access ensures that no other readable or writable copies of an item exist when the write occurs: all other cached copies of the item are invalidated.

Figure 5.41 shows an example of an invalidation protocol for a snooping bus with write-back caches in action. To see how this protocol ensures coherence, consider a write followed by a read by another processor: since the write requires

Processor activity	Bus activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of memory location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Invalidation for X	1		0
CPU B reads X	Cache miss for X	1	1	1

FIGURE 5.41 An example of an invalidation protocol working on a snooping bus for a single cache block (X) with write-back caches. We assume that neither cache initially holds X and that the value of X in memory is 0. The CPU and memory contents show the value after the processor and bus activity have both completed. A blank indicates no activity or no copy cached. When the second miss by B occurs, CPU A responds with the value canceling the response from memory. In addition, both the contents of B's cache and the memory contents of X are updated. This update of memory, which occurs when a block becomes shared, simplifies the protocol, but it is possible to track the ownership and force the write-back only if the block is replaced. This requires the introduction of an additional state called "owner," which indicates that a block may be shared, but the owning processor is responsible for updating any other processors and memory when it changes the block or replaces it.

exclusive access, any copy held by the reading processor must be invalidated (hence the protocol name). Thus, when the read occurs, it misses in the cache, and the cache is forced to fetch a new copy of the data. For a write, we require that the writing processor have exclusive access, preventing any other processor from being able to write simultaneously. If two processors do attempt to write the same data at the same time, one of them wins the race, causing the other processor's copy to be invalidated. For the other processor to complete its write, it must obtain a new copy of the data, which must now contain the updated value. Therefore, this protocol also enforces write serialization.

Hardware/ Software Interface

One insight is that block size plays an important role in cache coherency. For example, take the case of snooping on a cache with a block size of eight words, with a single word alternatively written and read by two processors. Most protocols exchange full blocks between processors, thereby increasing coherency bandwidth demands.

Large blocks can also cause what is called **false sharing**: when two unrelated shared variables are located in the same cache block, the whole block is exchanged between processors even though the processors are accessing different variables. Programmers and compilers should lay out data carefully to avoid false sharing.

false sharing When two unrelated shared variables are located in the same cache block and the full block is exchanged between processors even though the processors are accessing different variables.

Elaboration: Although the three properties on page 455 are sufficient to ensure coherence, the question of when a written value is seen is also important. To see why, observe that we cannot require that a read of X in Figure 5.40 instantaneously sees the value written for X by some other processor. If, for example, a write of X on one processor precedes a read of X on another processor very shortly beforehand, it may be impossible to ensure that the read returns the value of the data written, since the written data may not even have left the processor at that point. The issue of exactly *when* a written value must be seen by a reader is defined by a *memory consistency model*.

We make the following two assumptions. First, a write does not complete (and allow the next write to occur) until all processors have seen the effect of that write. Second, the processor does not change the order of any write with respect to any other memory access. These two conditions mean that if a processor writes location X followed by location Y, any processor that sees the new value of Y must also see the new value of X. These restrictions allow the processor to reorder reads, but force the processor to finish a write in program order.

Elaboration: Since input can change memory behind the caches, and since output could need the latest value in a write back cache, there is also a cache coherency problem for I/O with the caches of a single processor as well as just between caches of multiple processors. The cache coherence problem for multiprocessors and I/O (see Chapter 6), although similar in origin, has different characteristics that affect the appropriate solution. Unlike I/O, where multiple data copies are a rare event—one to be avoided whenever possible—a program running on multiple processors will normally have copies of the same data in several caches.

Elaboration: In addition to the snooping cache coherence protocol where the status of shared blocks is distributed, a *directory-based* cache coherence protocol keeps the sharing status of a block of physical memory in just one location, called the *directory*. Directory-based coherence has slightly higher implementation overhead than snooping, but it can reduce traffic between caches and thus scale to larger processor counts.



Parallelism and Memory Hierarchy: Redundant Arrays of Inexpensive Disks



This online 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, however, was due more to the considerably greater dependability offered by including a modest number of redundant disks. The section explains the differences in performance, cost, and **dependability** between the RAID levels.



Parallelism and Memory Hierarchy: Redundant Arrays of Inexpensive Disks

Amdahl's law in Chapter 1 reminds us that neglecting I/O in this parallel revolution is foolhardy. A simple example demonstrates this.

Impact of I/O on System Performance

EXAMPLE

Suppose we have a benchmark that executes in 100 seconds of elapsed time, of which 90 seconds is CPU time, and the rest is I/O time. Suppose the number of processors doubles every 2 years, but the processors remain at the same speed, and I/O time doesn't improve. How much faster will our program run at the end of 6 years?

ANSWER

We know that

$$\begin{aligned} \text{Elapsed time} &= \text{CPU time} + \text{I/O time} \\ 100 &= 90 + \text{I/O time} \\ \text{I/O time} &= 10\,\text{seconds} \end{aligned}$$

The new CPU times and the resulting elapsed times are computed in the following table.

After *n* years	CPU time	I/O time	Elapsed time	% I/O time
0 years	90 seconds	10 seconds	100 seconds	10%
2 years	$\frac{90}{2} = 45$ seconds	10 seconds	55 seconds	18%
4 years	$\frac{45}{2} = 23$ seconds	10 seconds	33 seconds	31%
6 years	$\frac{23}{2} = 11$ seconds	10 seconds	21 seconds	47%

The improvement in CPU performance after 6 years is

$$\frac{90}{11} = 8$$

However, the improvement in elapsed time is only

$$\frac{100}{21} = 4.7$$

and the I/O time has increased from 10% to 47% of the elapsed time.

Hence, the parallel revolution needs to come to I/O as well as to computation, or the effort spent in parallelizing could be squandered whenever programs do I/O, which they all must do.

Accelerating I/O performance was the original motivation of disk arrays. In the late 1980s, the high-performance storage of choice was large, expensive disks. The argument was that by replacing a few big disks with many small disks, performance would improve because there would be more read heads. This shift is a good match for multiple processors as well, since many read/write heads mean the storage system could support many more independent accesses as well as large transfers spread across many disks. That is, you could get both high I/Os per second and high data transfer rates. In addition to higher performance, there could be advantages in cost, power, and floor space, since smaller disks are generally more efficient per gigabyte than larger disks.

The flaw in the argument was that disk arrays could make reliability much worse. These smaller, inexpensive drives had lower MTTF ratings than the large drives, but more importantly, by replacing a single drive with, say, 50 small drives, the failure rate would go up by at least a factor of 50.



DEPENDABILITY

redundant arrays of inexpensive disks (RAID) An organization of disks that uses an array of small and inexpensive disks so as to increase both performance and reliability.

The solution was to add redundancy so that the system could cope with disk failures without losing information. By having many little disks, the cost of extra redundancy to improve dependability is small, relative to the solutions for a few large disks. Thus, **dependability** was more affordable if you constructed a redundant array of inexpensive disks. This observation led to its name: **redundant arrays of inexpensive disks**, abbreviated **RAID**.

In retrospect, although its invention was motivated by performance, dependability was the key reason for the widespread popularity of RAID. The parallel revolution has resurfaced the original performance side of the argument for RAID. The rest of this section surveys the options for dependability and their impacts on cost and performance.

How much redundancy do you need? Do you need extra information to find the faults? Does it matter how you organize the data and the additional check information on these disks? The paper that coined the term gave an evolutionary answer to these questions, starting with the simplest but most expensive solution. Figure e5.11.1 shows the evolution and example cost in the number of extra check disks. To keep track of the evolution, the authors numbered the stages of RAID, and they are still used today.

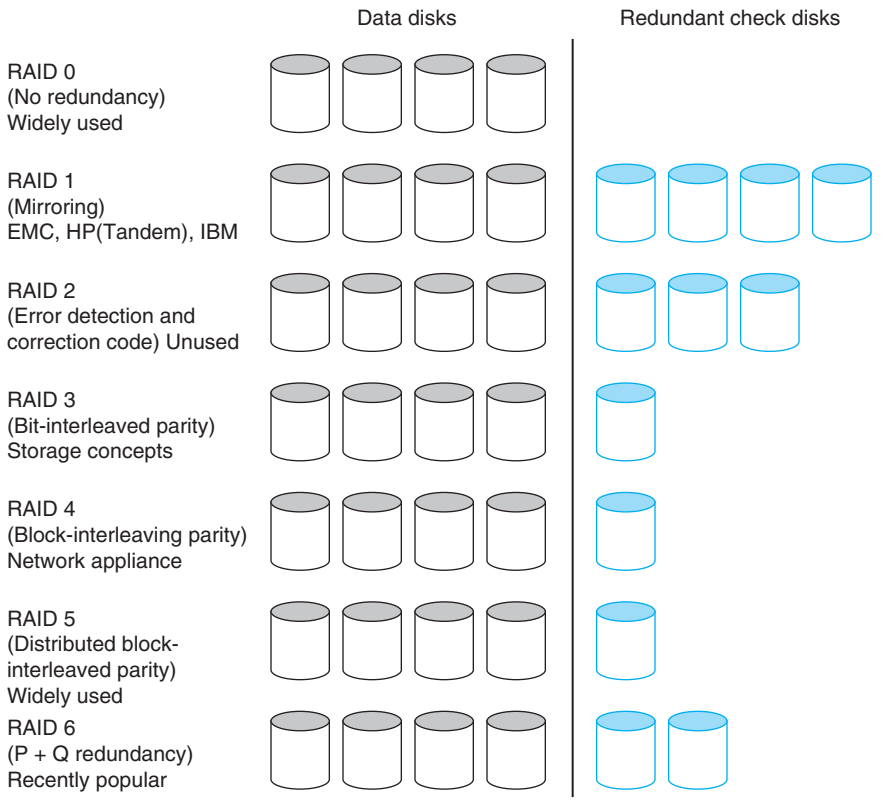


FIGURE e5.11.1 RAID for an example of four data disks showing extra check disks per RAID level and companies that use each level. Figures e5.11.2 and e5.11.3 explain the difference between RAID 3, RAID 4, and RAID 5.

No Redundancy (RAID 0)

striping Allocation of logically sequential blocks to separate disks to allow higher performance than a single disk can deliver.

Simply spreading data over multiple disks, called **striping**, automatically forces accesses to several disks. Striping across a set of disks makes the collection appear to software as a single large disk, which simplifies storage management. It also improves performance for large accesses, since many disks can operate at once. Video-editing systems, for example, frequently stripe their data and may not worry about dependability as much as, say, databases.

RAID 0 is something of a misnomer, as there is no redundancy. However, RAID levels are often left to the operator to set when creating a storage system, and RAID 0 is often listed as one of the options. Hence, the term RAID 0 has become widely used.

Mirroring (RAID 1)

mirroring Writing identical data to multiple disks to increase data availability.

This traditional scheme for tolerating disk failure, called **mirroring** or *shadowing*, uses twice as many disks as does RAID 0. Whenever data are written to one disk, that data are also written to a redundant disk, so that there are always two copies of the information. If a disk fails, the system just goes to the "mirror" and reads its contents to get the desired information. Mirroring is the most expensive RAID solution, since it requires the most disks.

Error Detecting and Correcting Code (RAID 2)

RAID 2 borrows an error detection and correction scheme most often used for memories (see Section 5.5). Since RAID 2 has fallen into disuse, we'll not describe it here.

Bit-Interleaved Parity (RAID 3)

protection group The group of data disks or blocks that share a common check disk or block.

The cost of higher availability can be reduced to $1/n$, where n is the number of disks in a **protection group**. Rather than have a complete copy of the original data for each disk, we need only add enough redundant information to restore the lost information on a failure. Reads or writes go to all disks in the group, with one extra disk to hold the check information in case there is a failure. RAID 3 is popular in applications with large data sets, such as multimedia and some scientific codes.

Parity is one such scheme. Readers unfamiliar with parity can think of the redundant disk as having the sum of all the data in the other disks. When a disk fails, then you subtract all the data in the good disks from the parity disk; the remaining information must be the missing information. Parity is simply the sum modulo two.

Unlike RAID 1, many disks must be read to determine the missing data. The assumption behind this technique is that taking longer to recover from failure but spending less on redundant storage is a good tradeoff.

Block-Interleaved Parity (RAID 4)

RAID 4 uses the same ratio of data disks and check disks as RAID 3, but they access data differently. The parity is stored as blocks and associated with a set of data blocks.

In RAID 3, every access went to all disks. However, some applications prefer smaller accesses, allowing independent accesses to occur in parallel. That is the purpose of the RAID levels 4 to 7. Since error detection information in each sector is checked on reads to see if the data are correct, such "small reads" to each disk can occur independently as long as the minimum access is one sector. In the RAID context, a small access goes to just one disk in a protection group while a large access goes to all the disks in a protection group.

Writes are another matter. It would seem that each small write would demand that all other disks be accessed to read the rest of the information needed to recalculate the new parity, as in the left in Figure e5.11.2. A "small write" would

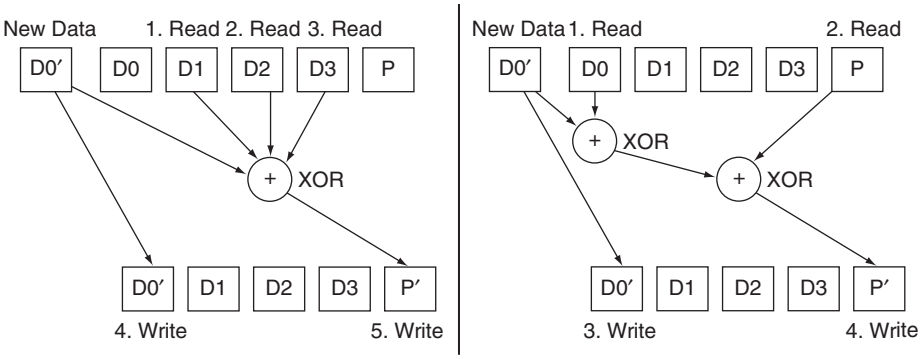


FIGURE e5.11.2 Small write update on RAID 4. This optimization for small writes reduces the number of disk accesses as well as the number of disks occupied. This figure assumes we have four blocks of data and one block of parity. The naive RAID 4 parity calculation in the left of the figure reads blocks D1, D2, and D3 before adding block D0? to calculate the new parity P? (In case you were wondering, the new data D0? comes directly from the CPU, so disks are not involved in reading it.) The RAID 4 shortcut on the right reads the old value D0 and compares it to the new value D0? to see which bits will change. You next read the old parity P and then change the corresponding bits to form P? The logical function exclusive OR does exactly what we want. This example replaces three disk reads (D1, D2, D3) and two disk writes (D0?, P?) involving all the disks for two disk reads (D0, P) and two disk writes (D0?, P?), which involve just two disks. Enlarging the size of the parity group increases the savings of the shortcut. RAID 5 uses the same shortcut.

require reading the old data and old parity, adding the new information, and then writing the new parity to the parity disk and the new data to the data disk.

The key insight to reduce this overhead is that parity is simply a sum of information; by watching which bits change when we write the new information, we need only change the corresponding bits on the parity disk. The right of Figure e5.11.2 shows the shortcut. We must read the old data from the disk being written, compare old data to the new data to see which bits change, read the old parity, change the corresponding bits, and then write the new data and new parity. Thus, the small write involves four disk accesses to two disks instead of accessing all disks. This organization is RAID 4.

Distributed Block-Interleaved Parity (RAID 5)

RAID 4 efficiently supports a mixture of large reads, large writes, and small reads, plus it allows small writes. One drawback to the system is that the parity disk must be updated on every write, so the parity disk is the bottleneck for back-to-back writes.

To fix the parity-write bottleneck, the parity information can be spread throughout all the disks so that there is no single bottleneck for writes. The distributed parity organization is RAID 5.

Figure e5.11.3 shows how data are distributed in RAID 4 versus RAID 5. As the organization on the right shows, in RAID 5 the parity associated with each row of data blocks is no longer restricted to a single disk. This organization allows multiple writes to occur simultaneously as long as the parity blocks are not located on the same disk. For example, a write to block 8 on the right must also access its parity

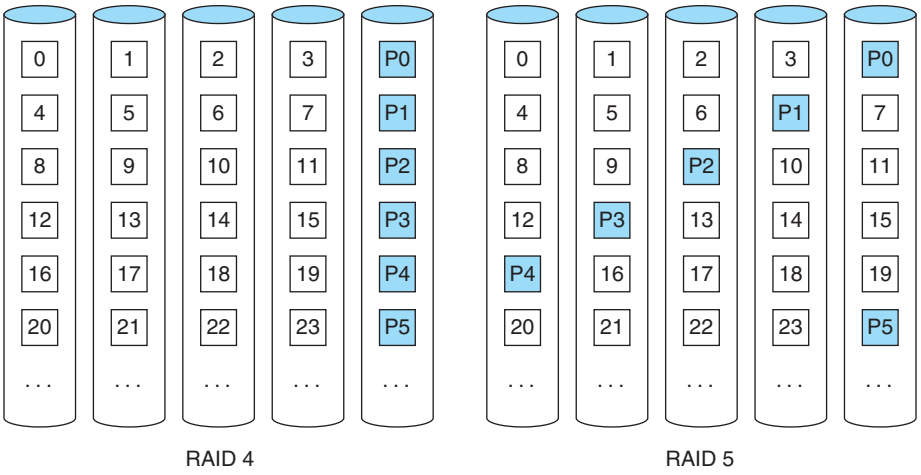


FIGURE e5.11.3 Block-interleaved parity (RAID 4) versus distributed block-interleaved parity (RAID 5). By distributing parity blocks to all disks, some small writes can be performed in parallel.

block P2, thereby occupying the first and third disks. A second write to block 5 on the right, implying an update to its parity block P1, accesses the second and fourth disks and thus could occur concurrently with the write to block 8. Those same writes to the organization on the left result in changes to blocks P1 and P2, both on the fifth disk, which is a bottleneck.

P + Q Redundancy (RAID 6)

Parity-based schemes protect against a single self-identifying failure. When a single failure correction is not sufficient, parity can be generalized to have a second calculation over the data and another check disk of information. This second check block allows recovery from a second failure. Thus, the storage overhead is twice that of RAID 5. The small write shortcut of Figure e5.11.2 works as well, except now there are six disk accesses instead of four to update both P and Q information.

RAID Summary

RAID 1 and RAID 5 are widely used in servers; one estimate is that 80% of disks in servers are found in a RAID organization.

One weakness of the RAID systems is repair. First, to avoid making the data unavailable during repair, the array must be designed to allow the failed disks to be replaced without having to turn off the system. RAIDs have enough redundancy to allow continuous operation, but **hot-swapping** disks place demands on the physical and electrical design of the array and the disk interfaces. Second, another failure could occur during repair, so the repair time affects the chances of losing data: the longer the repair time, the greater the chances of another failure that will

hot-swapping Replacing a hardware component while the system is running.

lose data. Rather than having to wait for the operator to bring in a good disk, some systems include **standby spares** so that the data can be reconstructed instantly upon discovery of the failure. The operator can then replace the failed disks in a more leisurely fashion. Note that a human operator ultimately determines which disks to remove. Operators are only human, so they occasionally remove the good disk instead of the broken disk, leading to an unrecoverable disk failure.

standby spares Reserve hardware resources that can immediately take the place of a failed component.

In addition to designing the RAID system for repair, there are questions about how disk technology changes over time. Although disk manufacturers quote very high MTTF for their products, those numbers are under nominal conditions. If a particular disk array has been subject to temperature cycles due to, say, the failure of the air-conditioning system, or to shaking due to a poor rack design, construction, or installation, the failure rates can be three to six times higher (see the fallacy on page 470). The calculation of RAID reliability assumes independence between disk failures, but disk failures could be correlated, because such damage due to the environment would likely happen to all the disks in the array. Another concern is that since disk bandwidth is growing more slowly than disk capacity, the time to repair a disk in a RAID system is increasing, which in turn enhances the chances of a second failure. For example, a 3-TB disk could take almost 9 hours to read sequentially, assuming no interference. Given that the damaged RAID is likely to continue to serve data, reconstruction could be stretched considerably. Besides increasing that time, another concern is that reading much more data during reconstruction means increasing the chance of an uncorrectable read media failure, which would result in data loss. Other arguments for concern about simultaneous multiple failures are the increasing number of disks in arrays and the use of higher-capacity disks.

Hence, these trends have led to a growing interest in protecting against more than one failure, and so RAID 6 is increasingly being offered as an option and being used in the field.

Check Yourself

Which of the following are true about RAID levels 1, 3, 4, 5, and 6?

1. RAID systems rely on redundancy to achieve high availability.
2. RAID 1 (mirroring) has the highest check disk overhead.
3. For small writes, RAID 3 (bit-interleaved parity) has the worst throughput.
4. For large writes, RAID 3, 4, and 5 have the same throughput.

Elaboration One issue is how mirroring interacts with striping. Suppose you had, say, four disks' worth of data to store and eight physical disks to use. Would you create four pairs of disks—each organized as RAID 1—and then stripe data across the four RAID 1 pairs? Alternatively, would you create two sets of four disks—each organized as RAID 0—and then mirror writes to both RAID 0 sets? The RAID terminology has evolved to call the former RAID 1 + 0 or RAID 10 ("striped mirrors") and the latter RAID 0 + 1 or RAID 01 ("mirrored stripes").



Advanced Material: Implementing Cache Controllers

This online section shows how to implement control for a cache, just as we implemented control for the single-cycle and pipelined datapaths in Chapter 4. This section starts with a description of finite-state machines and the implementation of a cache controller for a simple data cache, including a description of the cache controller in a hardware description language. It then goes into details of an example cache coherence protocol and the difficulties in implementing such a protocol.

5.13 Real Stuff: The ARM Cortex-A53 and Intel Core i7 Memory Hierarchies

In this section, we will look at the memory hierarchy of the same two microprocessors described in Chapter 4: the ARM Cortex-A53 and Intel Core i7. This section is in part based on Section 2.6 of *Computer Architecture: A Quantitative Approach*, 5th edition.

Figure 5.42 summarizes the address sizes and TLBs of the two processors. Note that the Cortex-A53 has two 10-entry fully associative micro-TLBs backed by a shared 512-entry four-way set associative main TLB with a 48-bit virtual address space and a 40-bit physical address space. The Core i7 has three TLBs with a 48-bit virtual address and a 44-bit physical address. Although the 64-bit registers of these processors could hold a larger virtual address, there was no software need for such a large space, and 48-bit virtual addresses shrinks both the page table memory footprint and the TLB hardware.

Figure 5.43 shows their caches. The Cortex-A53 has between one and four processors or cores while the Core i7 is fixed at four. Cortex-A53 has a 16 to 64 KiB, two-way L1 instruction cache (per core) and the Core i7 has a 32 KiB, four-way set associative, L1 instruction cache (per core). Both use 64 byte blocks. The Cortex-A53 increases the associativity to four-way for the data cache, other variables remain the same. Similarly, the Core i7 keeps everything the same except the associativity, which it increases to eight-way. The Core i7 provides a 256 KiB, eight-way set associative unified L2 cache (per core) with 64 byte blocks. In contrast, the Cortex-A53 provides a L2 cache that is shared between one and four cores. This cache is 16-way set associative with 64 byte blocks and between 128 KiB and 2 MiB in size. As the Core i7 is used for servers, it also offers an L3 cache shared by all the cores on the chip. Its size varies depending on the number of cores. With four cores, as in this case, the size is 8 MiB.



Advanced Material: Implementing Cache Controllers

The section starts with the SystemVerilog of the cache controller from Section 5.9 in eight figures. It then goes into details of an example cache coherency protocol and the difficulties in implementing such a protocol.

SystemVerilog of a Simple Cache Controller

The hardware description language we are using in this section is SystemVerilog. The biggest change from prior versions of Verilog is that it borrows structures from C to make the code easier to read. Figures e5.12.1 through e5.12.8 show the SystemVerilog description of the cache controller.

```
package cache_def;
  // data structures for cache tag & data

  parameter int TAGMSB = 31;     //tag msb
  parameter int TAGLSB = 14;     //tag lsb

  //data structure for cache tag
  typedef struct packed {
    bit    valid;                //valid bit
    bit    dirty;                //dirty bit
    bit [TAGMSB:TAGLSB]tag;      //tag bits
  }cache_tag_type;

  //data structure for cache memory request
  typedef struct {
    bit [9:0]index;              //10-bit index
    bit    we;                   //write enable
  }cache_req_type;

  //128-bit cache line data
  typedef bit [127:0]cache_data_type;
```

FIGURE e5.12.1 Type declarations in SystemVerilog for the cache tags and data. The tag field is 18 bits wide and the index field is 10 bits wide, while a 2-bit field (bits 3–2) is used to index the block and select the word from the block. The rest of the type declaration is found in the following figure.

```
// data structures for CPU<->Cache controller interface

// CPU request (CPU->cache controller)
typedef struct {
  bit [31:0]addr;            //32-bit request addr
  bit [31:0]data;            //32-bit request data (used when write)
  bit rw;                    //request type : 0 = read, 1 = write
  bit valid;                 //request is valid
}cpu_req_type;

// Cache result (cache controller->cpu)
typedef struct {
  bit [31:0]data;            //32-bit data
  bit ready;                 //result is ready
}cpu_result_type;

//-----------------------------------------------------------------------
// data structures for cache controller<->memory interface

// memory request (cache controller->memory)
typedef struct {
  bit [31:0]addr;            //request byte addr
  bit [127:0]data;           //128-bit request data (used when write)
  bit rw;                    //request type : 0 = read, 1 = write
  bit valid;                 //request is valid
}mem_req_type;

// memory controller response (memory -> cache controller)
typedef struct {
  cache_data_type data;      //128-bit read back data
  bit    ready;              //data is ready
}mem_data_type;

endpackage
```

FIGURE e5.12.2 Type declarations in SystemVerilog for the CPU-cache and cache-memory interfaces. These are nearly identical except that the data are 32 bits wide between the CPU and cache and are 128 bits wide between the cache and memory.

Figures e5.12.1 and e5.12.2 declare the structures that are used in the definition of the cache in the following figures. For example, the cache tag structure (`cache_tag_type`) contains a valid bit (`valid`), a dirty bit (`dirty`), and an 18-bit tag field (`[TAGMSB:TAGLSB] tag`). Figure e5.12.3 shows the block diagram of the cache using the names from the Verilog description.

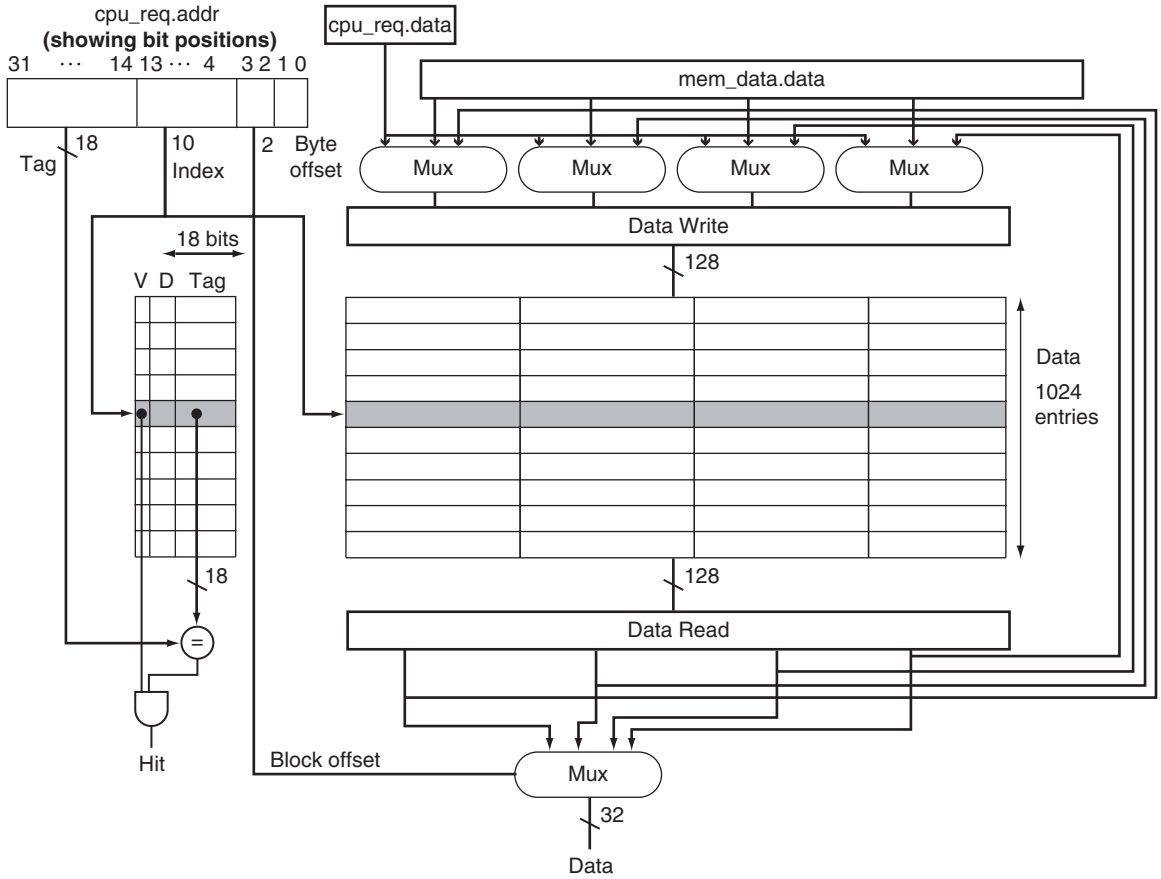


FIGURE e5.12.3 Block diagram of the simple cache using the Verilog names. Not shown are the write enables for the cache tag memory and for the cache data memory, or the control signals for multiplexors that supply data for the Data Write variable. Rather than have separate write enables on every word of the cache data block, the Verilog reads the old value of the block into Data Write and then updates the word in that variable on a write. It then writes the whole 128-bit block.

Figure e5.12.4 defines modules for the cache data (`dm_cache_data`) and cache tag (`dm_cache_tag`). These memories can be read at any time, but writes only occur on the positive clock edge (`posedge(clk)`) and only if write enable is a 1 (`data_req.we` or `tag_req.we`).

Figure e5.12.5 defines the inputs, outputs, and states of the FSM. The inputs are the requests from the CPU (`cpu_req`) and responses from memory (`mem_data`), and the outputs are responses to the CPU (`cpu_res`) and requests to memory (`mem_req`). The figure also declares the internal variables needed by the FSM. For example, the current state and next state registers of the FSM are `rstate` and `vstate`, respectively.

Figure e5.12.6 lists the default values of the control signals, including the word to be read or written from a block, setting the cache write enables to 0, and so on. These values are set every clock cycle, so the write enable for a portion of the cache—for example, `tag_req.we`—would be set to 1 for one clock cycle in the figures below and then would be reset to 0 according to the Verilog in this figure.

The last two figures show the FSM as a large case statement (`case(rstate)`), with the four states split across the two figures. Figure e5.12.7 starts with the Idle state (`idle`), which simply goes to the Compare Tag state (`compare_tag`) if the CPU makes a valid request. It then describes most of the Compare Tag state. The Compare Tag state checks to see if the tags match and the entry is valid. If so, then it first sets the Cache Ready signal (`v_cpu_res.ready`). If the request is a write, it sets the tag field, the valid bit, and the dirty bit. The next state is Idle. If it is a miss, then the state prepares to change the tag entry and valid and dirty bits. If the block to be replaced is clean or invalid, the next state is Allocate.

Figure e5.12.8 continues the Compare Tag state. If the block to be replaced is dirty, then the next state is Write-Back. The figure shows the Allocate state (`allocate`) next, which simply reads the new block. It keeps looping until the memory is ready; when it is, it goes to the Compare Tag state. This is followed in the figure by the Write-Back state (`write_back`). As the figure shows, the Write-Back state merely writes the dirty block to memory, once again looping until memory is ready. When memory is ready, indicating the write is complete, we go to the Allocate state.

The code at the end sets the current state from the next state or resets the FSM to the Idle state on the next clock edge, depending on a reset signal (`rst`).

The online material includes a Test Case module that will be useful to check the code in these figures. This SystemVerilog could be used to create a cache and cache controller in an FPGA.

```
/*cache: data memory, single port, 1024 blocks*/
module dm_cache_data(input  bit clk,
     input  cache_req_type  data_req,//data request/command, e.g. RW, valid
     input  cache_data_type data_write, //write port (128-bit line)
     output cache_data_type data_read); //read port
  timeunit 1ns; timeprecision 1ps;

  cache_data_typedata_mem[0:1023];

  initial begin
    for (int i=0; i<1024; i++)
         data_mem[i] = '0;
  end

  assign data_read = data_mem[data_req.index];

  always_ff @(posedge(clk)) begin
    if (data_req.we)
      data_mem[data_req.index] <= data_write;
  end
endmodule

/*cache: tag memory, single port, 1024 blocks*/
module dm_cache_tag(input  bit clk, //write clock
     input  cache_req_type tag_req, //tag request/command, e.g. RW, valid
     input  cache_tag_type tag_write,//write port
     output cache_tag_type tag_read);//read port
  timeunit 1ns; timeprecision 1ps;

  cache_tag_typetag_mem[0:1023];

  initial begin
     for (int i=0; i<1024; i++)
     tag_mem[i] = '0;
  end

  assign tag_read = tag_mem[tag_req.index];

  always_ff @(posedge(clk)) begin
    if (tag_req.we)
      tag_mem[tag_req.index] <= tag_write;
  end

endmodule
```

FIGURE e5.12.4 Cache data and tag modules in SystemVerilog. These are nearly identical except that the data are 32 bits wide between the CPU and cache and are 128 bits wide between the cache and memory. Both only write on positive clock edges if the write enable is set.

```
/*cache finite state machine*/

module dm_cache_fsm(input  bit clk, input  bit rst,
        input  cpu_req_type   cpu_req,     //CPU request input (CPU->cache)
        input  mem_data_type  mem_data,    //memory response (memory->cache)
        output mem_req_type   mem_req,     //memory request (cache->memory)
        output cpu_result_type cpu_res     //cache result (cache->CPU)
     );

  timeunit 1ns;
  timeprecision 1ps;

  /*write clock*/
  typedef enum {idle, compare_tag, allocate, write_back} cache_state_type;

  /*FSM state register*/
  cache_state_typevstate, rstate;

  /*interface signals to tag memory*/
  cache_tag_typetag_read;                  //tag read result
  cache_tag_typetag_write;                 //tag write data
  cache_req_typetag_req;                   //tag request

  /*interface signals to cache data memory*/
  cache_data_typedata_read;                //cache line read data
  cache_data_typedata_write;               //cache line write data
  cache_req_typedata_req;                  //data req


  /*temporary variable for cache controller result*/
  cpu_result_typev_cpu_res;

  /*temporary variable for memory controller request*/
  mem_req_typev_mem_req;

  assign mem_req = v_mem_req;              //connect to output ports
  assign cpu_res = v_cpu_res;
```

FIGURE e5.12.5 FSM in SystemVerilog, part I. These modules instantiate the memories according to the type definitions in the previous figure.

```
always_comb begin

    /*-------------------------default values for all signals------------*/
    /*no state change by default*/
    vstate = rstate;
    v_cpu_res = '{0, 0}; tag_write = '{0, 0, 0};

    /*read tag by default*/
    tag_req.we = '0;
    /*direct map index for tag*/
    tag_req.index = cpu_req.addr[13:4];

    /*read current cache line by default*/
    data_req.we = '0;
    /*direct map index for cache data*/
    data_req.index = cpu_req.addr[13:4];

    /*modify correct word (32-bit) based on address*/
    data_write = data_read;
    case(cpu_req.addr[3:2])
    2'b00:data_write[31:0] = cpu_req.data;
    2'b01:data_write[63:32] = cpu_req.data;
    2'b10:data_write[95:64] = cpu_req.data;
    2'b11:data_write[127:96] = cpu_req.data;
    endcase

    /*read out correct word(32-bit) from cache (to CPU)*/
    case(cpu_req.addr[3:2])
    2'b00:v_cpu_res.data = data_read[31:0];
    2'b01:v_cpu_res.data = data_read[63:32];
    2'b10:v_cpu_res.data = data_read[95:64];
    2'b11:v_cpu_res.data = data_read[127:96];
    endcase

    /*memory request address (sampled from CPU request)*/
    v_mem_req.addr = cpu_req.addr;
    /*memory request data (used in write)*/
    v_mem_req.data = data_read;
    v_mem_req.rw = '0;
```

FIGURE e5.12.6 FSM in SystemVerilog, part II. This section describes the default value of all signals. The following figures will set these values for one clock cycle, and this Verilog will reset it to these values for the following clock cycle.

```
//-----------------------------------Cache FSM-------------------------
case(rstate)
/*idle state*/
idle : begin
    /*If there is a CPU request, then compare cache tag*/
    if (cpu_req.valid)
       vstate = compare_tag;
       end
/*compare_tag state*/
compare_tag : begin
          /*cache hit (tag match and cache entry is valid)*/
    if (cpu_req.addr[TAGMSB:TAGLSB] == tag_read.tag && tag_read.valid) begin
          v_cpu_res.ready = '1;

       /*write hit*/
       if (cpu_req.rw) begin
          /*read/modify cache line*/
          tag_req.we = '1; data_req.we = '1;

          /*no change in tag*/
          tag_write.tag = tag_read.tag;
          tag_write.valid = '1;
          /*cache line is dirty*/
          tag_write.dirty = '1;
       end

          /*xaction is finished*/
          vstate = idle;
    end
    /*cache miss*/
    else begin
      /*generate new tag*/
      tag_req.we = '1;
      tag_write.valid = '1;
      /*new tag*/
      tag_write.tag = cpu_req.addr[TAGMSB:TAGLSB];
      /*cache line is dirty if write*/
      tag_write.dirty = cpu_req.rw;

      /*generate memory request on miss*/
      v_mem_req.valid = '1;
      /*compulsory miss or miss with clean block*/
      if (tag_read.valid == 1'b0 || tag_read.dirty == 1'b0)
         /*wait till a new block is allocated*/
         vstate = allocate;
```

FIGURE e5.12.7 FSM in SystemVerilog, part III. Actual FSM states via case statement in this figure and the next. This figure has the Idle state and most of the Compare Tag state.

```
      else begin
      /*miss with dirty line*/
         /*write back address*/
         v_mem_req.addr = {tag_read.tag, cpu_req.addr[TAGLSB-1:0]};
         v_mem_req.rw = '1;
         /*wait till write is completed*/
         vstate = write_back;
      end
     end
 end
 /*wait for allocating a new cache line*/
 allocate: begin
     /*memory controller has responded*/
     if (mem_data.ready) begin
        /*re-compare tag for write miss (need modify correct word)*/
        vstate = compare_tag;
        data_write = mem_data.data;
        /*update cache line data*/
        data_req.we = '1;
     end
      end
 /*wait for writing back dirty cache line*/
 write_back : begin
     /*write back is completed*/
     if (mem_data.ready) begin
        /*issue new memory request (allocating a new line)*/
        v_mem_req.valid = '1;
        v_mem_req.rw = '0;

        vstate = allocate;
     end
   end
 endcase
end

always_ff @(posedge(clk)) begin
 if (rst)
   rstate <= idle;       //reset to idle state
 else
   rstate <= vstate;
end
/*connect cache tag/data memory*/
dm_cache_tag  ctag(.*);
dm_cache_data cdata(.*);
endmodule
```

FIGURE e5.12.8 FSM in SystemVerilog, part IV. Actual FSM states via the case statement in the prior figure and this one. This figure has the last part of the Compare Tag state, plus Allocate and Write-Back states.

Basic Coherent Cache Implementation Techniques

The key to implementing an invalidate protocol is the use of the bus, or another broadcast medium, to perform invalidates. To invalidate, the processor simply acquires bus access and broadcasts the address to be invalidated on the bus. All processors continuously snoop on the bus, watching the addresses. The processors check whether the address on the bus is in their cache. If so, the corresponding data in the cache are invalidated.

When a write to a block that is shared occurs, the writing processor must acquire bus access to broadcast its invalidation. If two processors try to write shared blocks at the same time, their attempts to broadcast an invalidate operation will be serialized when they arbitrate for the bus. The first processor to obtain bus access will cause any other copies of the block it is writing to be invalidated. If the processors were attempting to write the same block, the serialization enforced by the bus also serializes their writes. One implication of this scheme is that a write to a shared data item cannot actually complete until it obtains bus access. All coherence schemes require some method of serializing accesses to the same cache block, by serializing access either to the communication medium or another shared structure.

In addition to invalidating outstanding copies of a cache block that is being written into, we also need to locate a data item when a cache miss occurs. In a write-through cache, it is easy to find the recent value of a data item, since all written data are unfailingly sent to the memory, from which the most-recent value of a data item can always be fetched. In a design with adequate memory bandwidth to support the write traffic from the processors, using write-through simplifies the implementation of cache coherence.

For a write-back cache, finding the most-recent data value is more difficult, since the most recent value of a data item can be in a cache rather than in memory. Happily, write-back caches can use the same snooping scheme both for cache misses and for writes: each processor snoops all addresses placed on the bus. If a processor finds that it has a dirty copy of the requested cache block, it provides that cache block in response to the read request and causes the memory access to be aborted. The increased complexity comes from having to retrieve the cache block from a processor's cache, which can often take longer than retrieving it from the shared memory if the processors are in separate chips. Since write-back caches generate lower requirements for memory bandwidth, they can support larger numbers of faster processors and have been the approach chosen in most multiprocessors, despite the additional complexity of maintaining coherence. Therefore, we will examine the implementation of coherence with write-back caches.

The normal cache tags can be used to implement the process of snooping, and the valid bit for each block makes invalidation easy to implement. Read misses, whether generated by an invalidation or by some other event, are also straightforward, since they simply rely on the snooping capability. For writes, we'd like to know whether any other copies of the block are cached, because if there are

no other cached copies, the write need not be placed on the bus in a write-back cache. Not sending the write reduces both the time taken by the write and the required bandwidth.

To track whether or not a cache block is shared, we can add an extra state bit associated with each cache block, just as we have a valid bit and a dirty bit. By adding a bit indicating whether the block is shared, we can decide whether a write must generate an invalidate. When a write to a block in the shared state occurs, the cache generates an invalidation on the bus and marks the block as *exclusive*. No further invalidations will be sent by that processor for that block. The processor with the sole copy of a cache block is normally called the *owner* of the cache block.

When an invalidation is sent, the state of the owner's cache block is changed from shared to unshared (or exclusive). If another processor later requests this cache block, the state must be made shared again. Since our snooping cache also sees any misses, it knows when the exclusive cache block has been requested by another processor, and the state should be made shared.

Every bus transaction must check the cache-address tags, which could potentially interfere with processor cache accesses. One way to reduce this interference is to duplicate the tags. The interference can also be reduced in a multilevel cache by directing the snoop requests to the L2 cache, which the processor uses only when it has a miss in the L1 cache. For this scheme to work, every entry in the L1 cache must be present in the L2 cache, a property called the *inclusion property*. If the snoop gets a hit in the L2 cache, then it must arbitrate for the L1 cache to update the state and possibly retrieve the data, which usually requires a stall of the processor. Sometimes it may even be useful to duplicate the tags of the secondary cache to further decrease contention between the processor and the snooping activity.

An Example Cache Coherency Protocol

A snooping coherence protocol is usually implemented by incorporating a finite-state controller in each node. This controller responds to requests from the processor and from the bus (or other broadcast medium), changing the state of the selected cache block, as well as using the bus to access data or to invalidate it. Logically, you can think of a separate controller being associated with each block; that is, snooping operations or cache requests for different blocks can proceed independently. In actual implementations, a single controller allows multiple operations to distinct blocks to proceed in interleaved fashion (that is, one operation may be initiated before another is completed, even though only one cache access or one bus access is allowed at a time). Also, remember that although we refer to a bus in the following description, any interconnection network that supports a broadcast to all the coherence controllers and their associated caches can be used to implement snooping.

The simple protocol we consider has three states: invalid, shared, and modified. The shared state indicates that the block is potentially shared, while the modified state indicates that the block has been updated in the cache; note that the

modified state *implies* that the block is exclusive. Figure e5.12.9 shows the requests generated by the processor-cache module in a node (in the first nine rows of the table) as well as those coming from the bus (in the last five rows of the table). This protocol is for a write-back cache, but it can be easily changed to work for a write-through cache by reinterpreting the modified state as an exclusive state and updating the cache on writes in the normal fashion for a write-through cache. The most common extension of this basic protocol is the addition of an exclusive state, which describes a block that is unmodified but held in only one cache; the caption of Figure e5.12.9 describes this state and its addition in more detail.

When an invalidate or a write miss is placed on the bus, any processors with copies of the cache block invalidate it. For a write-through cache, the data for a write miss can always be retrieved from the memory. For a write miss in a writeback cache, if the block is exclusive in just one cache, that cache also writes back the block; otherwise, the data can be read from memory.

Figure e5.12.10 shows a finite-state transition diagram for a single cache block using a write invalidation protocol and a write-back cache. For simplicity, the three states of the protocol are duplicated to represent transitions based on processor requests (on the left, which corresponds to the top half of the table in Figure e5.12.9), contrary to transitions based on bus requests (on the right, which corresponds to the last five rows of the table in Figure e5.12.9). Boldface type is used to distinguish the bus actions, in contrast to the conditions on which a state transition depends. The state in each node represents the state of the selected cache block specified by the processor or bus request.

All of the states in this cache protocol would be needed in a uniprocessor cache, where they would correspond to the invalid, valid (and clean), and dirty states. Most of the state changes indicated by arcs in the left half of Figure e5.12.10 would be needed in a write-back uniprocessor cache, with the exception being the invalidate on a write hit to a shared block. The state changes represented by the arcs in the right half of Figure e5.12.10 are needed only for coherence and would not appear at all in an uniprocessor cache controller.

As mentioned earlier, there is only one finite-state machine per cache, with stimuli coming either from the attached processor or from the bus. Figure e5.12.11 shows how the state transitions in the right half of Figure e5.12.10 are combined with those in the left half of the figure to form a single state diagram for each cache block.

To understand why this protocol works, observe that any valid cache block is either in the shared state in one or more caches or in the exclusive state in exactly one cache. Any transition to the exclusive state (which is required for a processor to write to the block) requires an invalidate or write miss to be placed on the bus, causing all caches to make the block invalid. In addition, if some other cache had the block in the exclusive state, that cache generates a write back, which supplies the block containing the desired address. Finally, if a read miss occurs on the bus to a block in the exclusive state, the cache with the exclusive copy changes its state to shared.

Request	Source	State of addressed cache block	Type of cache action	Function and explanation
Read hit	processor	shared or modified	normal hit	Read data in cache.
Read miss	processor	invalid	normal miss	Place read miss on bus.
Read miss	processor	shared	replacement	Address conflict miss: place read miss on bus.
Read miss	processor	modified	replacement	Address conflict miss: write-back block, then place read miss on bus.
Write hit	processor	modified	normal hit	Write data in cache.
Write hit	processor	shared	coherence	Place invalidate on bus. These operations are often called upgrade or *ownership* misses, since they do not fetch the data but only change the state.
Write miss	processor	invalid	normal miss	Place write miss on bus.
Write miss	processor	shared	replacement	Address conflict miss: place write miss on bus.
Write miss	processor	modified	replacement	Address conflict miss: write-back block, then place write miss on bus.
Read miss	bus	shared	no action	Allow memory to service read miss.
Read miss	bus	modified	coherence	Attempt to share data: place cache block on bus and change state to shared.
Invalidate	bus	shared	coherence	Attempt to write shared block; invalidate the block.
Write miss	bus	shared	coherence	Attempt to write block that is shared; invalidate the cache block.
Write miss	bus	modified	coherence	Attempt to write block that is exclusive elsewhere: write-back the cache block and make its state invalid.

FIGURE e5.12.9 The cache coherence mechanism receives requests from both the processor and the bus and responds to these based on the type of request, whether it hits or misses in the cache, and the state of the cache block specified in the request. The fourth column describes the type of cache action as normal hit or miss (the same as a uniprocessor cache would see), replacement (a uniprocessor cache replacement miss), or coherence (required to maintain cache coherence); a normal or replacement action may cause a coherence action depending on the state of the block in other caches. For read misses, write misses, or invalidates snooped from the bus, an action is required only if the read or write addresses match a block in the cache and the block is valid. Some protocols also introduce a state to designate when a block is exclusively in one cache but has not yet been written. This state can arise if a write access is broken into two pieces: getting the block exclusively in one cache and then subsequently updating it; in such a protocol this "exclusive unmodified state" is transient, ending as soon as the write is completed. Other protocols use and maintain an exclusive state for an unmodified block. In a snooping protocol, this state can be entered when a processor reads a block that is not resident in any other cache. Because all subsequent accesses are snooped, it is possible to maintain the accuracy of this state. In particular, if another processor issues a read miss, the state is changed from exclusive to shared. The advantage of adding this state is that a subsequent write to a block in the exclusive state by the same processor need not acquire bus access or generate an invalidate, since the block is known to be exclusively in this cache; the processor merely changes the state to modified. This state is easily added by using the bit that encodes the coherent state as an exclusive state and using the dirty bit to indicate that a block is modified. The popular MESI protocol, which is named for the four states it includes (modified, exclusive, shared, and invalid), uses this structure. The MOESI protocol introduces another extension: the "owned" state.

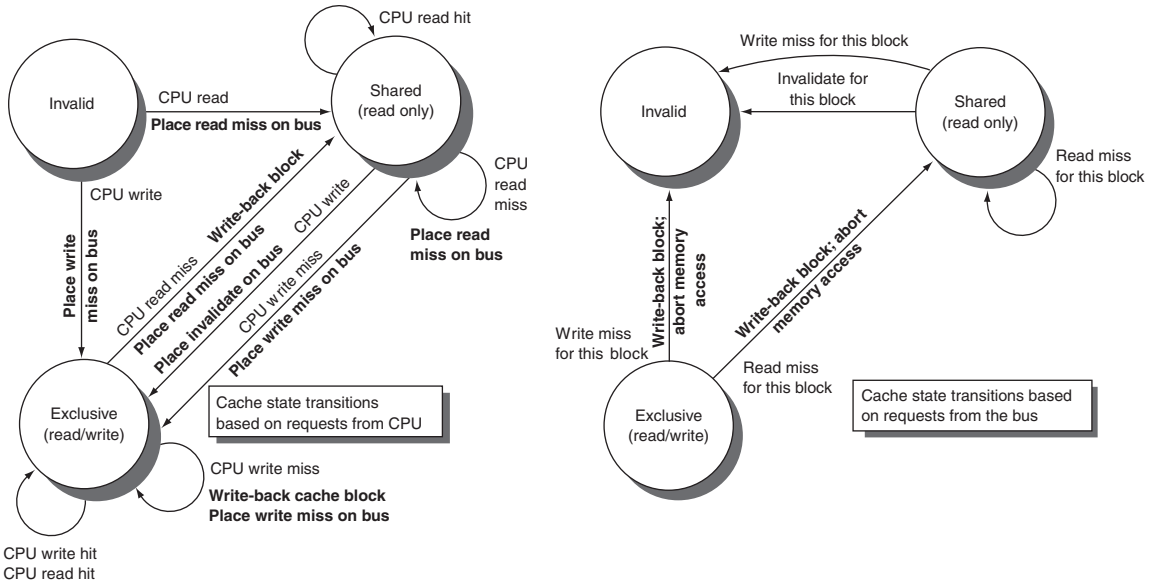


FIGURE e5.12.10 A write-invalidate, cache-coherence protocol for a write-back cache, showing the states and state transitions for each block in the cache. The cache states are shown in circles, with any access permitted by the processor without a state transition shown in parentheses under the name of the state. The stimulus causing a state change is shown on the transition arcs in regular type, and any bus actions generated as part of the state transition are shown on the transition arc in bold. The stimulus actions apply to a block in the cache, not to a specific address in the cache. Hence, a read miss to a block in the shared state is a miss for that cache block but for a different address. The left side of the diagram shows state transitions based on actions of the processor associated with this cache; the right side shows transitions based on operations on the bus. A read miss in the exclusive or shared state and a write miss in the exclusive state occur when the address requested by the processor does not match the address in the cache block. Such a miss is a standard cache replacement miss. An attempt to write a block in the shared state generates an invalidate. Whenever a bus transaction occurs, all caches that contain the cache block specified in the bus transaction take the action dictated by the right half of the diagram. The protocol assumes that memory provides data on a read miss for a block that is clean in all caches. In actual implementations, these two sets of state diagrams are combined. In practice, there are many subtle variations on invalidate protocols, including the introduction of the exclusive unmodified state, as to whether a processor or memory provides data on a miss.

The actions in gray in Figure e5.12.11, which handle read and write misses on the bus, are essentially the snooping component of the protocol. One other property that is preserved in this protocol, and in most other protocols, is that any memory block in the shared state is always up to date in the memory, which simplifies the implementation.

Although our simple cache protocol is correct, it omits a number of complications that make the implementation much trickier. The most important of these is that the protocol assumes that operations are *atomic*—that is, an operation can be done in such a way that no intervening operation can occur. For example, the protocol described assumes that write misses can be detected, acquire the bus, and receive a response as a single atomic action. In reality, this is not true. Similarly, if we used

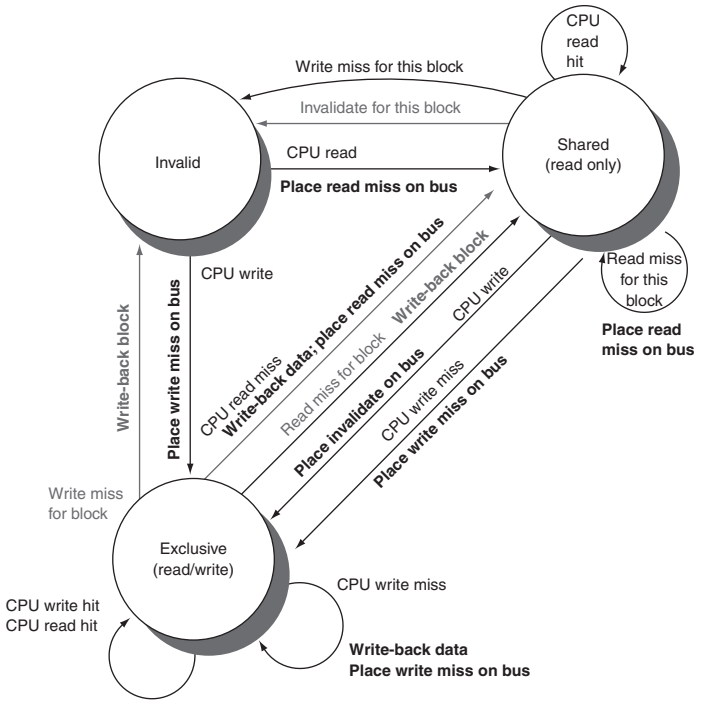


FIGURE e5.12.11 Cache coherence state diagram with the state transitions induced by the local processor shown in black and by the bus activities shown in gray. As in Figure e5.12.10, the activities on a transition are shown in bold.

a switch, as all recent multiprocessors do, then even read misses would also not be atomic.

Nonatomic actions introduce the possibility that the protocol can *deadlock*, meaning that it reaches a state where it cannot continue. On the next page, we will discuss how these protocols are implemented without a bus.

Constructing small-scale (two to four processors) multiprocessors has become very easy. For example, the Intel Nehalem and AMD Opteron processors are designed for use in cache-coherent multiprocessors and have an external interface that supports snooping and allows two to four processors to be directly connected. They also have larger on-chip caches to reduce bus utilization. In the case of the Opteron processors, the support for interconnecting multiple processors is integrated onto the processor chip, as are the memory interfaces. In the case of the Intel design, a two-processor system can be built with only a few additional external chips to interface with the memory system and I/O. Although these designs cannot be easily scaled to larger processor counts, they offer an extremely cost-effective solution for two to four processors.

The devil is in the details.

Classic proverb.

Implementing Snoopy Cache Coherence

As we said earlier, the major complication in actually implementing the snooping coherence protocol we have described is that write and upgrade misses are not atomic in any recent multiprocessor. The steps of detecting a write or upgrade miss; communicating with the other processors and memory; getting the most-recent value for a write miss and ensuring that any invalidates are processed; and updating the cache cannot be done as if they took a single cycle.

In a simple single-bus system, these steps can be made effectively atomic by arbitrating for the bus first (before changing the cache state) and not releasing the bus until all actions are complete. How can the processor know when all the invalidates are complete? In most bus-based multiprocessors, a single line is used to signal when all necessary invalidates have been received and are being processed. Following that signal, the processor that generated the miss can release the bus, knowing that any required actions will be completed before any activity related to the next miss. By holding the bus exclusively during these steps, the processor effectively makes the individual steps atomic.

In a system without a bus, we must find some other method of making the steps in a miss atomic. In particular, we must ensure that two processors that attempt to write the same block at the same time, a situation which is called a *race*, are strictly ordered: one write is processed before the next is begun. It does not matter which of two writes in a race wins the race, just that there be only a single winner whose coherence actions are completed first. In a snoopy system, ensuring that a race has only one winner is accomplished by using broadcast for all misses, as well as some basic properties of the interconnection network. These properties, together with the ability to restart the miss handling of the loser in a race, are the keys to implementing snoopy cache coherence without a bus.

Characteristic	ARM Cortex-A53	Intel Core i7
Virtual address	48 bits	48 bits
Physical address	40 bits	44 bits
Page size	Variable: 4, 16, 64 KiB, 1, 2 MiB, 1 GiB	Variable: 4 KiB, 2/4 MiB
TLB organization	1 TLB for instructions and 1 TLB for data per core Both micro TLBs are fully associative, with 10 entries, round robin replacement 64-entry, four-way set-associative TLBs TLB misses handled in hardware	1 TLB for instructions and 1 TLB for data per core Both L1 TLBs are four-way set associative, LRU replacement L1 I-TLB has 128 entries for small pages, seven per thread for large pages L1 D-TLB has 64 entries for small pages, 32 for large pages The L2 TLB is four-way set associative, LRU replacement The L2 TLB has 512 entries TLB misses handled in hardware

FIGURE 5.42 Address translation and TLB hardware for the ARM Cortex-A53 and Intel Core i7 920. Both processors provide support for large pages, which are used for things like the operating system or mapping a frame buffer. The large-page scheme avoids using a large number of entries to map a single object that is always present.



PARALLELISM

A significant challenge facing cache designers is to support processors like the Cortex-A53 and the Core i7 that can execute more than one memory instruction per clock cycle. A popular technique is to break the cache into banks and allow multiple, independent, **parallel** accesses, provided the accesses are to different banks. The technique is similar to interleaved DRAM banks (see Section 5.2).

The Cortex-A53 and the Core i7 have additional optimizations that allow them to reduce the miss penalty. The first of these is the return of the requested word first on a miss. They also continue to execute instructions that access the data cache during a cache miss. Designers who are attempting to hide the cache miss latency commonly use this technique, called a **nonblocking cache**. They implement two flavors of nonblocking. *Hit under miss* allows additional cache hits during a miss, while *miss under miss* allows multiple outstanding cache misses. The aim of the first of these two is hiding some miss latency with other work, while the aim of the second is overlapping the latency of two different misses.

nonblocking cache
A cache that allows the processor to make references to the cache while the cache is handling an earlier miss.

Overlapping a large fraction of miss times for multiple outstanding misses requires a high-bandwidth memory system capable of handling multiple misses in parallel. In a personal mobile device, the memory system below it can often pipeline, merge, reorder, or prioritize requests appropriately. Large servers and multiprocessors typically have memory systems capable of handling several outstanding misses in parallel.

Characteristic	ARM Cortex-A53	Intel Core i7
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	Configurable 16 to 64 KiB each for instructions/data	32 KiB each for instructions/data per core
L1 cache associativity	Two-way (I), four-way (D) set associative	Four-way (I), eight-way (D) set associative
L1 replacement	Random	Approximated LRU
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, variable allocation policies (default is Write-allocate)	Write-back, No-write-allocate
L1 hit time (load-use)	Two clock cycles	Four clock cycles, pipelined
L2 cache organization	Unified (instruction and data)	Unified (instruction and data) per core
L2 cache size	128 KiB to 2 MiB	256 KiB (0.25 MiB)
L2 cache associativity	16-way set associative	8-way set associative
L2 replacement	Approximated LRU	Approximated LRU
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L2 hit time	12 clock cycles	10 clock cycles
L3 cache organization	–	Unified (instruction and data)
L3 cache size	–	8 MiB, shared
L3 cache associativity	–	16-way set associative
L3 replacement	–	Approximated LRU
L3 block size	–	64 bytes
L3 write policy	–	Write-back, Write-allocate
L3 hit time	–	35 clock cycles

FIGURE 5.43 Caches in the ARM Cortex-A53 and Intel Core i7 920.

The Cortex-A53 and the Core i7 have prefetch mechanisms for data accesses. They look at a pattern of data misses and uses this information to try to predict the next address to start fetching the data before the miss occurs. Such techniques generally work best when accessing arrays in loops.

The sophisticated memory hierarchies of these chips and the large fraction of the dies dedicated to caches and TLBs show the significant design effort expended to try to close the gap between processor cycle times and memory latency.

Performance of the Cortex-A53 and Core i7 Memory Hierarchies

The memory hierarchy of the Cortex-A53 was measured using a 32 KiB two-way set associative L1 instruction cache, a 32 KiB four-way set associative L1 data cache, and a 1 MiB 16-way set associative L2 cache running the integer SPEC2006 benchmarks.

The Cortex-A53 instruction cache miss rates for these benchmarks are very small. Figure 5.44 shows the data cache results for the Cortex-A53, which have significant L1 and L2 miss rates. The L1 data cache miss rates go from 0.5% to 37.3%, with a mean of 6.4% and a median of 2.4%. The (global) L2 cache miss rates vary from 0.1% to 9.0%, with a mean of 1.3% and a median of 0.3%. The L1 miss penalty for a 1 GHz Cortex-A53 is 12 clock cycles, while the L2 miss penalty is 124 clock cycles. Using these miss penalties, Figure 5.45 shows the average miss penalty per data access. When these low miss rates are multiplied by their high miss penalties, you can see that they can represent a significant fraction of the CPI for 5 of the 12 SPEC2006 programs.

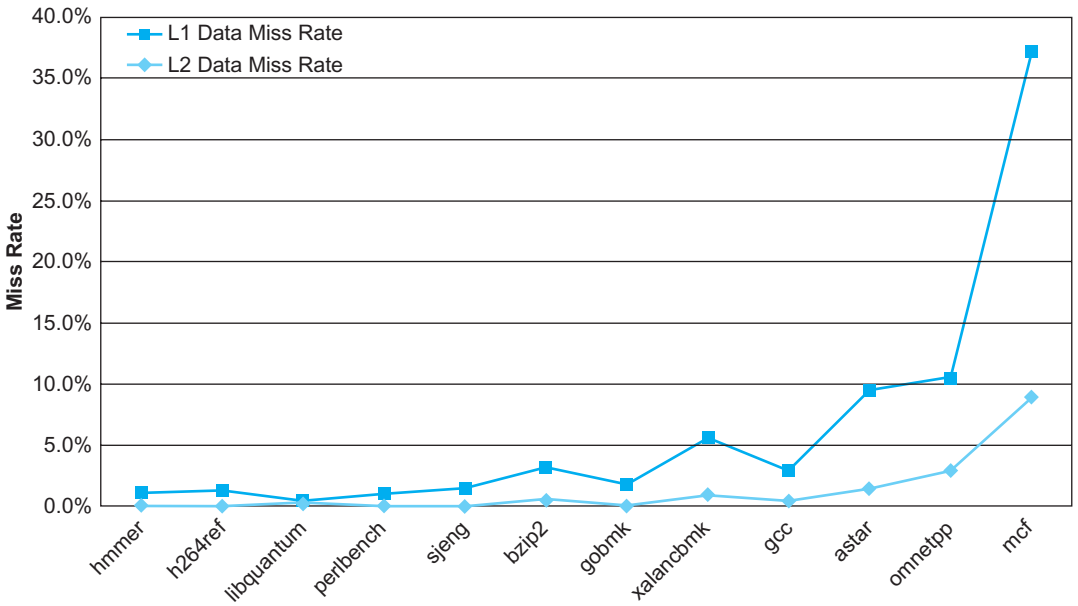


FIGURE 5.44 Data cache miss rates for ARM Cortex-A53 when running SPEC2006int. Applications with larger memory footprints tend to have higher miss rates in both L1 and L2. Note that the L2 rate is the global miss rate; that is, counting all references, including those that hit in L1. (See the *Elaboration* in Section 5.4.) mcf is known as a cache buster. Note that this figure is for the same systems and benchmarks as Figure 4.74 in Chapter 4.

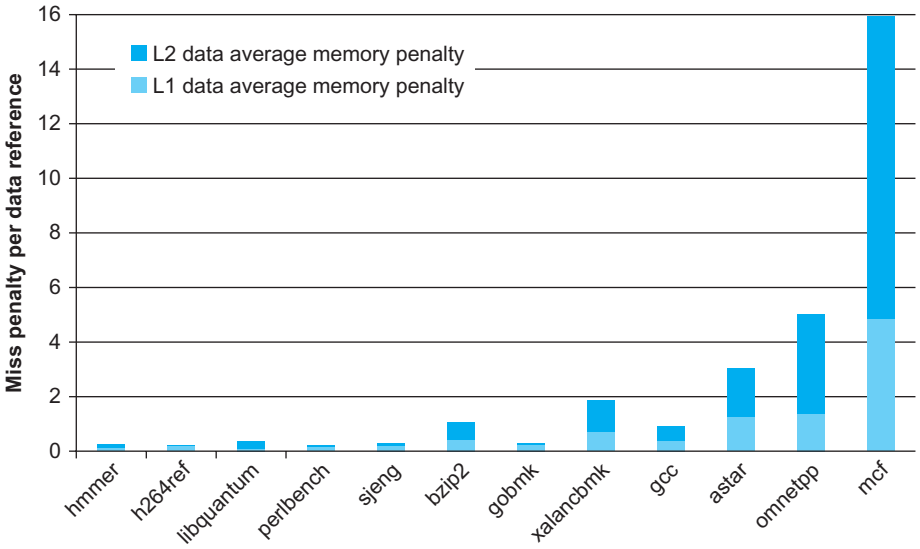


FIGURE 5.45 The average memory access penalty in clock cycles per data memory reference coming from L1 and L2 is shown for the ARM processor when running SPEC2006int. Although the miss rates for L1 are significantly higher, the L2 miss penalty, which is more than five times higher, means that the L2 misses can contribute significantly.

Figure 5.46 shows the miss rates for the caches of the Core i7 using the SPEC2006 benchmarks. The L1 instruction cache miss rate varies from 0.1% to 1.8%, averaging just over 0.4%. This rate is in keeping with other studies of instruction cache behavior for the SPECCPU2006 benchmarks, which show low instruction cache miss rates. With L1 data cache miss rates running 5% to 10%, and sometimes higher, the importance of the L2 and L3 caches should be obvious. Since the cost for a miss to memory is over 100 cycles, and the average data miss rate in L2 is 4%, L3 is obviously critical. Assuming about half the instructions is loads or stores, without L3 the L2 cache misses could add two cycles per instruction to the CPI! In comparison, the average L3 data miss rate of 1% is still significant but four times lower than the L2 miss rate and six times less than the L1 miss rate.

Elaboration: Because speculation may sometimes be wrong (see Chapter 4), there are references to the L1 data cache that do not correspond to loads or stores that eventually complete execution. The data in Figure 5.44 are measured against all data requests, including some that are cancelled. The miss rate when measured against only completed data accesses is 1.6 times higher (an average of 9.5% versus 5.9% for L1 Dcache misses).

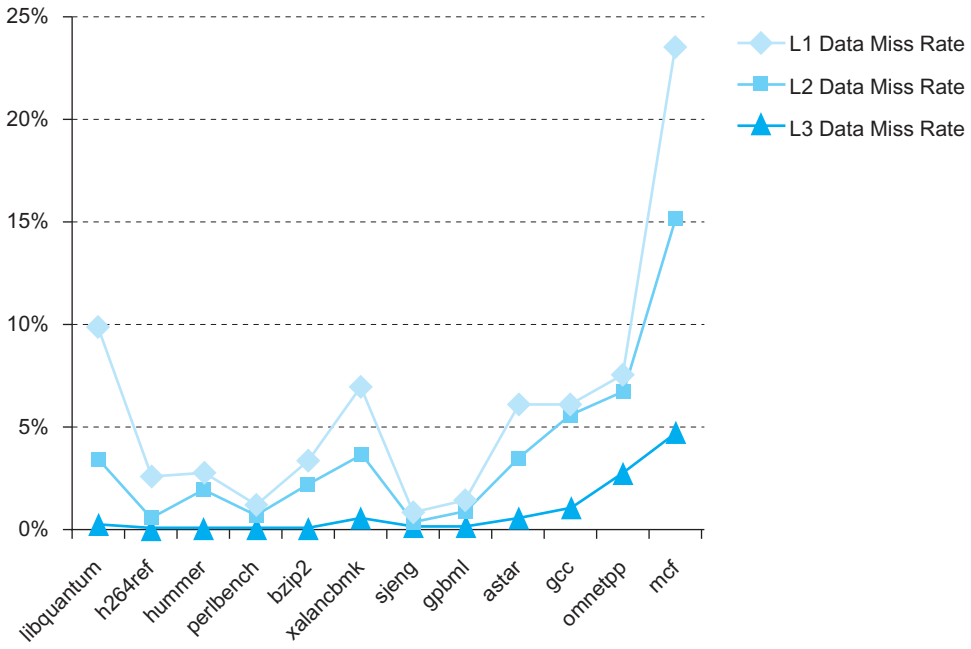


FIGURE 5.46 The L1, L2, and L3 data cache miss rates for the Intel Core i7 920 running the full integer SPECCPU2006 benchmarks.

5.14 Real Stuff: The Rest of the RISC-V System and Special Instructions

Figure 5.48 lists the 13 remaining RISC-V instructions in the special purpose and systems category.

The fence instructions provide synchronization barriers for instructions (`fence.i`), data (`fence`), and address translations (`sfence.vma`). The first, `fence.i`, informs the processor that software has modified instruction memory, so that it can guarantee that instruction fetch will reflect the updated instructions. The second, `fence`, affects data memory access ordering for multiprocessing and I/O. The third, `sfence.vma`, informs the processor that software has modified the page tables, so that it can guarantee that address translations will reflect the updates.

The six *control and status register* (CSR) access instructions move data between general-purpose registers and CSRs. The `csrrwi` instruction (CSR read/write immediate) copies a CSR to an integer register, then overwrites the CSR with an immediate. `csrrsi` (CSR read/set immediate) copies a CSR to an integer register, and overwrites the CSR with the bitwise OR of the CSR and an immediate. `csrrci`

(CSR read/clear) is like `csrrsi`, but clears bits instead of setting them. The `csrrw`, `csrrs`, and `csrrc` instructions use a register operand instead of an immediate, but otherwise do the same thing.

Two instructions' only purpose is to generate exceptions: `ecall` generates an environment call exception to invoke the OS, and `ebreak` generates a breakpoint exception to invoke the debugger. The supervisor exception-return instruction (`sret`), naturally enough, allows the program to return from an exception handler.

Finally, the wait-for-interrupt instruction, `wfi`, informs the processor that it may enter an idle state until an interrupt occurs.

5.15 Going Faster: Cache Blocking and Matrix Multiply

Our next step in the continuing saga of improving performance of DGEMM by tailoring it to the underlying hardware is to add cache blocking to the subword parallelism and instruction level parallelism optimizations of Chapters 3 and 4. Figure 5.47 shows the blocked version of DGEMM from Figure 4.78. The changes are the same as was made earlier in going from unoptimized DGEMM in Figure 3.22 to blocked DGEMM in Figure 5.21 above. This time we take the unrolled version of DGEMM from Chapter 4 and invoke it many times on the submatrices of `A`, `B`, and `C`. Indeed, lines 28–34 and lines 7–8 in Figure 5.47 mirror lines 14–20 and lines 5–6 in Figure 5.21, except for incrementing the for loop in line 7 by the amount unrolled.

Unlike the earlier chapters, we do not show the resulting x86 code because the inner loop code is nearly identical to Figure 4.79, as the blocking does not affect the computation, just the order that it accesses data in memory. What does change is the bookkeeping integer instructions to implement the loops. It expands from 14 instructions before the inner loop and eight after the loop for Figure 4.78 to 40 and 28 instructions respectively for the bookkeeping code generated for Figure 5.47. Nevertheless, the extra instructions executed pale in comparison to the performance improvement of reducing cache misses. Figure 5.49 compares unoptimized to optimized for subword parallelism, instruction level parallelism, and caches. Blocking improves performance over unrolled AVX code by factors of 2 to 2.5 for the larger matrices. When we compare unoptimized code to the code with all three optimizations, the performance improvement is factors of 8 to 15, with the largest increase for the largest matrix.

Elaboration: As mentioned in the *Elaboration* in Section 3.9, these results are with Turbo mode turned off. As in Chapters 3 and 4, when we turn it on, we improve all the results by the temporary increase in the clock rate of 3.3/2.6 = 1.27. Turbo mode works particularly well in this case because it is using only a single core of an eight-core chip. However, if we want to run fast we should use all cores, which we'll see in Chapter 6.

```
 1 #include <x86intrin.h>
 2 #define UNROLL (4)
 3 #define BLOCKSIZE 32
 4 void do_block (int n, int si, int sj, int sk,
 5                 double *A, double *B, double *C)
 6 {
 7   for ( int i = si; i < si+BLOCKSIZE; i+=UNROLL*4 )
 8     for ( int j = sj; j < sj+BLOCKSIZE; j++ ) {
 9       __m256d c[4];
10       for ( int x = 0; x < UNROLL; x++ )
11         c[x] = _mm256_load_pd(C+i+x*4+j*n);
12      /* c[x] = C[i][j] */
13       for( int k = sk; k < sk+BLOCKSIZE; k++ )
14       {
15         __m256d b = _mm256_broadcast_sd(B+k+j*n);
16      /* b = B[k][j] */
17         for (int x = 0; x < UNROLL; x++)
18           c[x] = _mm256_add_pd(c[x], /* c[x]+=A[i][k]*b */
19                  _mm256_mul_pd(_mm256_load_pd(A+n*k+x*4+i), b));
20       }
21
22
23       for ( int x = 0; x < UNROLL; x++ )
24         _mm256_store_pd(C+i+x*4+j*n, c[x]);
           /* C[i][j] = c[x] */
25     }
26 }
27
28 void dgemm (int n, double* A, double* B, double* C)
29 {
30   for ( int sj = 0; sj < n; sj += BLOCKSIZE )
31     for ( int si = 0; si < n; si += BLOCKSIZE )
32       for ( int sk = 0; sk < n; sk += BLOCKSIZE )
33         do_block(n, si, sj, sk, A, B, C);
34 }
```

FIGURE 5.47 Optimized C version of DGEMM from Figure 4.78 using cache blocking. These changes are the same ones found in Figure 5.21. The assembly language produced by the compiler for the `do_block` function is nearly identical to Figure 4.79. Once again, there is no overhead to call the `do_block` because the compiler inlines the function call.

Type	Mnemonic	Name
Mem. Ordering	FENCE.I	Instruction Fence
	FENCE	Fence
	SFENCE.VMA	Address Translation Fence
CSR Access	CSRRWI	CSR Read/Write Immediate
	CSRRSI	CSR Read/Set Immediate
	CSRRCI	CSR Read/Clear Immediate
	CSRRW	CSR Read/Write
	CSRRS	CSR Read/Set
	CSRRC	CSR Read/Clear
System	ECALL	Environment Call
	EBREAK	Environment Breakpoint
	SRET	Supervisor Exception Return
	WFI	Wait for Interrupt

FIGURE 5.48 The list of assembly language instructions for the systems and special operations in the full RISC-V instruction set.

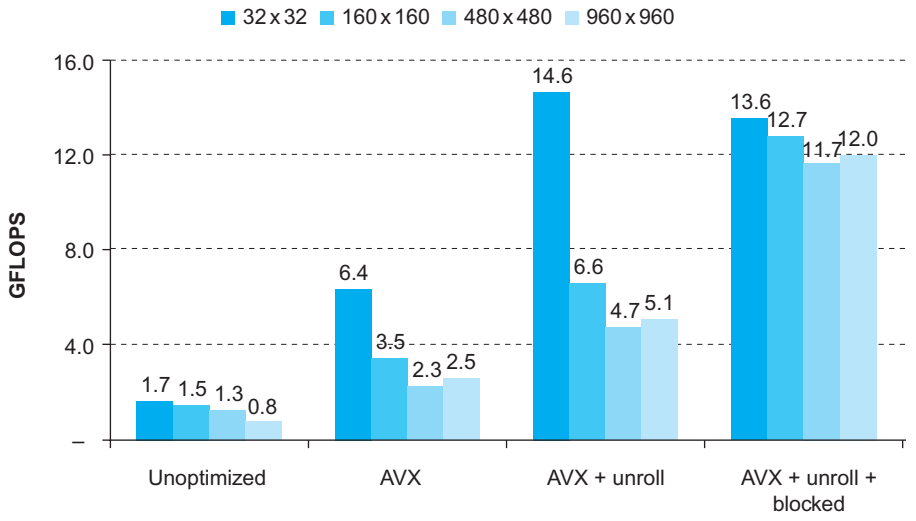


FIGURE 5.49 Performance of four versions of DGEMM from matrix dimensions 32 × 32 to 960 × 960. The fully optimized code for the largest matrix is almost 15 times as fast the unoptimized version in Figure 3.22 in Chapter 3.

5.16 Fallacies and Pitfalls

As one of the most naturally quantitative aspects of computer architecture, the memory hierarchy would seem to be less vulnerable to fallacies and pitfalls. Not only have there been many fallacies propagated and pitfalls encountered, but some have led to major negative outcomes. We start with a pitfall that often traps students in exercises and exams.

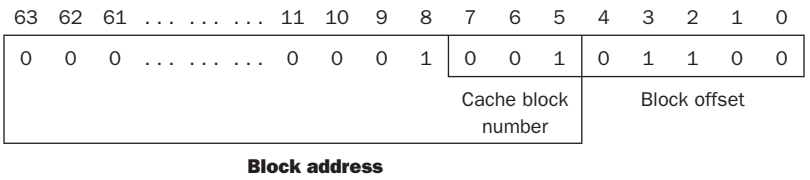
Pitfall: Ignoring memory system behavior when writing programs or when generating code in a compiler.

This could be rewritten as a fallacy: "Programmers can ignore memory hierarchies in writing code." The evaluation of sort in Figure 5.19 and of cache blocking in Section 5.14 demonstrate that programmers can easily double performance if they factor the behavior of the memory system into the design of their algorithms.

Pitfall: Forgetting to account for byte addressing or the cache block size in simulating a cache.

When simulating a cache (by hand or by computer), we need to make sure we account for the effect of byte addressing and multiword blocks in determining into which cache block a given address maps. For example, if we have a 32-byte direct-mapped cache with a block size of 4 bytes, the byte address 36 maps into block 1 of the cache, since byte address 36 is block address 9 and (9 modulo 8) = 1. On the other hand, if address 36 is a word address, then it maps into block (36 mod 8) = 4. Make sure the problem clearly states the base of the address.

In like fashion, we must account for the block size. Suppose we have a cache with 256 bytes and a block size of 32 bytes. Into which block does the byte address 300 fall? If we break the address 300 into fields, we can see the answer:



Byte address 300 is block address

$$\left\lfloor \frac{300}{32} \right\rfloor = 9$$

The number of blocks in the cache is

$$\left\lfloor \frac{256}{32} \right\rfloor = 8$$

Block number 9 falls into cache block number (9 modulo 8) = 1.

This mistake catches many people, including the authors (in earlier drafts) and instructors who forget whether they intended the addresses to be in doublewords, words, bytes, or block numbers. Remember this pitfall when you tackle the exercises.

Pitfall: Having less set associativity for a shared cache than the number of cores or threads sharing that cache.



Without extra care, a **parallel** program running on 2^n processors or threads can easily allocate data structures to addresses that would map to the same set of a shared L2 cache. If the cache is at least 2^n-way associative, then these accidental conflicts are hidden by the hardware from the program. If not, programmers could face apparently mysterious performance bugs—actually due to L2 conflict misses—when migrating from, say, a 16-core design to 32-core design if both use 16-way associative L2 caches.

Pitfall: Using average memory access time to evaluate the memory hierarchy of an out-of-order processor.

If a processor stalls during a cache miss, then you can separately calculate the memory-stall time and the processor execution time, and hence evaluate the memory hierarchy independently using average memory access time (see page 391).

If the processor continues to execute instructions, and may even sustain more cache misses during a cache miss, then the only accurate assessment of the memory hierarchy is to simulate the out-of-order processor along with the memory hierarchy.

Pitfall: Extending an address space by adding segments on top of an unsegmented address space.

During the 1970s, many programs grew so large that not all the code and data could be addressed with just a 16-bit address. Computers were then revised to offer 32-bit addresses, either through an unsegmented 32-bit address space (also called a *flat address space*) or by adding 16 bits of segment to the existing 16-bit address. From a marketing point of view, adding segments that were programmer-visible and that forced the programmer and compiler to decompose programs into segments could solve the addressing problem. Unfortunately, there is trouble any time a programming language wants an address that is larger than one segment, such as indices for large arrays, unrestricted pointers, or reference parameters. Moreover, adding segments can turn every address into two words—one for the segment number and one for the segment offset—causing problems in the use of addresses in registers.

Fallacy: Disk failure rates in the field match their specifications.

Two recent studies evaluated large collections of disks to check the relationship between results in the field compared to specifications. One study was of almost 100,000 disks that had quoted MTTF of 1,000,000 to 1,500,000 hours, or AFR of 0.6% to 0.8%. They found AFRs of 2% to 4% to be common, often three to five times higher than the specified rates [Schroeder and Gibson, 2007]. A second study of more than 100,000 disks at Google, which had a quoted AFR of about 1.5%, saw

failure rates of 1.7% for drives in their first year rise to 8.6% for drives in their third year, or about five to six times the declared rate [Pinheiro, Weber, and Barroso, 2007].

Fallacy: Operating systems are the best place to schedule disk accesses.

As mentioned in Section 5.2, higher-level disk interfaces offer logical block addresses to the host operating system. Given this high-level abstraction, the best an OS can do to try to help performance is to sort the logical block addresses into increasing order. However, since the disk knows the actual mapping of the logical addresses onto the physical geometry of sectors, tracks, and surfaces, it can reduce the rotational and seek latencies by rescheduling.

For example, suppose the workload is four reads [Anderson, 2003]:

Operation	Starting LBA	Length
Read	724	8
Read	100	16
Read	9987	1
Read	26	128

The host might reorder the four reads into logical block order:

Operation	Starting LBA	Length
Read	26	128
Read	100	16
Read	724	8
Read	9987	1

Depending on the relative location of the data on the disk, reordering could make it worse, as Figure 5.50 shows. The disk-scheduled reads complete in three-quarters of a disk revolution, but the OS-scheduled reads take three revolutions.

Pitfall: Implementing a virtual machine monitor on an instruction set architecture that wasn't designed to be virtualizable.

Many architects in the 1970s and 1980s weren't careful to make sure that all instructions reading or writing information related to hardware resource information were privileged. This *laissez-faire* attitude causes problems for VMMs for all of these architectures, including the x86, which we use here as an example.

Figure 5.51 describes the 18 instructions that cause problems for virtualization [Robin and Irvine, 2000]. The two broad classes are instructions that

- Read control registers in user mode that reveals that the guest operating system is running in a virtual machine (such as POPF, mentioned earlier)
- Check protection as required by the segmented architecture but assume that the operating system is running at the highest privilege level

To simplify implementations of VMMs on the x86, both AMD and Intel have proposed extensions to the architecture via a new mode. Intel's VT-x provides

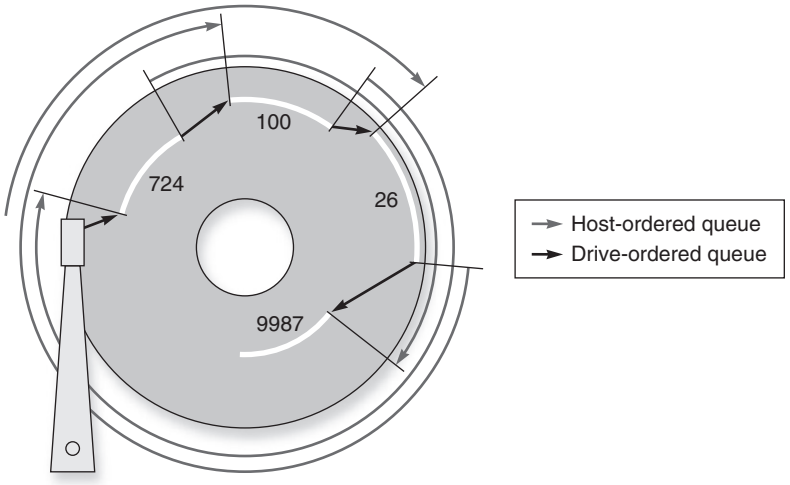


FIGURE 5.50 Example showing OS versus disk schedule accesses, labeled host-ordered versus drive-ordered. The former takes three revolutions to complete the four reads, while the latter completes them in just three-fourths of a revolution. *From Anderson [2003].*

Problem category	Problem x86 instructions
Access sensitive registers without trapping when running in user mode	Store global descriptor table register (SGDT) Store local descriptor table register (SLDT) Store interrupt descriptor table register (SIDT) Store machine status word (SMSW) Push flags (PUSHF, PUSHFD) Pop flags (POPF, POPFD)
When accessing virtual memory mechanisms in user mode, instructions fail the x86 protection checks	Load access rights from segment descriptor (LAR) Load segment limit from segment descriptor (LSL) Verify if segment descriptor is readable (VERR) Verify if segment descriptor is writable (VERW) Pop to segment register (POP CS, POP SS, . . .) Push segment register (PUSH CS, PUSH SS, . . .) Far call to different privilege level (CALL) Far return to different privilege level (RET) Far jump to different privilege level (JMP) Software interrupt (INT) Store segment selector register (STR) Move to/from segment registers (MOVE)

FIGURE 5.51 Summary of 18 x86 instructions that cause problems for virtualization [Robin and Irvine, 2000]. The first five instructions in the top group allow a program in user mode to read a control register, such as descriptor table registers, without causing a trap. The pop flags instruction modifies a control register with sensitive information but fails silently when in user mode. The protection checking of the segmented architecture of the x86 is the downfall of the bottom group, as each of these instructions checks the privilege level implicitly as part of instruction execution when reading a control register. The checking assumes that the OS must be at the highest privilege level, which is not the case for guest VMs. Only the Move to segment register tries to modify control state, and protection checking foils it as well.

a new execution mode for running VMs, an architected definition of the VM state, instructions to swap VMs rapidly, and a large set of parameters to select the circumstances where a VMM must be invoked. Altogether, VT-x adds 11 new instructions for the x86. AMD's Pacifica makes similar proposals.

An alternative to modifying the hardware is to make small changes to the operating system to avoid using the troublesome pieces of the architecture. This technique is called *paravirtualization*, and the open source Xen VMM is a good example. The Xen VMM provides a guest OS with a virtual machine abstraction that uses only the easy-to-virtualize parts of the physical x86 hardware on which the VMM runs.

5.17 Concluding Remarks

The difficulty of building a memory system to keep pace with faster processors is underscored by the fact that the raw material for main memory, DRAMs, is essentially the same in the fastest computers as it is in the slowest and cheapest.



HIERARCHY

It is the principle of locality that gives us a chance to overcome the long latency of memory access—and the soundness of this strategy is demonstrated at all levels of the **memory hierarchy**. Although these levels of the hierarchy look quite different in quantitative terms, they follow similar strategies in their operation and exploit the same properties of locality.

Multilevel caches make it possible to use more cache optimizations more easily for two reasons. First, the design parameters of a lower-level cache are different from a first-level cache. For example, because a lower-level cache will be much larger, it is possible to use bigger block sizes. Second, a lower-level cache is not constantly being used by the processor, as a first-level cache is. This allows us to consider having the lower-level cache do something when it is idle that may be useful in preventing future misses.



PREDICTION

Another trend is to seek software help. Efficiently managing the memory hierarchy using a variety of program transformations and hardware facilities is a major focus of compiler enhancements. Two different ideas are being explored. One idea is to reorganize the program to enhance its spatial and temporal locality. This approach focuses on loop-oriented programs that use sizable arrays as the major data structure; large linear algebra problems are a typical example, such as DGEMM. By restructuring the loops that access the arrays, substantially improved locality—and, therefore, cache performance—can be obtained.

prefetching A technique in which data blocks needed in the future are brought into the cache early by using special instructions that specify the address of the block.

Another approach is **prefetching**. In prefetching, a block of data is brought into the cache before it is actually referenced. Many microprocessors use hardware prefetching to try to *predict* accesses that may be difficult for software to notice.

A third approach is special cache-aware instructions that optimize memory transfer. For example, the microprocessors in Section 6.10 in Chapter 6 use an optimization that does not fetch the contents of a block from memory on a write miss because the program is going to write the full block. This optimization significantly reduces memory traffic for one kernel.

As we will see in Chapter 6, memory systems are a central design issue for parallel processors. The growing significance of the memory hierarchy in determining system performance means that this important area will continue to be a focus for both designers and researchers for some years to come.



Historical Perspective and Further Reading

This section, which appears online, gives an overview of memory technologies, from mercury delay lines to DRAM, the invention of the memory hierarchy, protection mechanisms, and virtual machines, and concludes with a brief history of operating systems, including CTSS, MULTICS, UNIX, BSD UNIX, MS-DOS, Windows, and Linux.

5.19 Exercises

Assume memory is byte addressable and words are 64 bits, unless specified otherwise.

5.1 In this exercise we look at memory locality properties of matrix computation. The following code is written in C, where elements within the same row are stored contiguously. Assume each word is a 64-bit integer.

```
for (I=0; I<8; I++)
  for (J=0; J<8000; J++)
    A[I][J]=B[I][0]+A[J][I];
```

5.1.1 [5] <§5.1> How many 64-bit integers can be stored in a 16-byte cache block?

5.1.2 [5] <§5.1> Which variable references exhibit temporal locality?

5.1.3 [5] <§5.1> Which variable references exhibit spatial locality?

Locality is affected by both the reference order and data layout. The same computation can also be written below in Matlab, which differs from C in that it stores matrix elements within the same column contiguously in memory.

```
for I=1:8
  for J=1:8000
    A(I,J)=B(I,0)+A(J,I);
  end
end
```



Historical Perspective and Further Reading

…the one single development that put computers on their feet was the invention of a reliable form of memory, namely, the core memory.… Its cost was reasonable, it was reliable and, because it was reliable, it could in due course be made large.

Maurice Wilkes, *Memoirs of a Computer Pioneer*, 1985

This history section gives an overview of memory technologies, from mercury delay lines to DRAM, the invention of the memory hierarchy, and protection mechanisms, and concludes with a brief history of operating systems, including CTSS, MULTICS, UNIX, BSD UNIX, MS-DOS, Windows, and Linux.

The developments of most of the concepts in this chapter have been driven by revolutionary advances in the technology we use for memory. Before we discuss how memory hierarchies were evolved, let's take a brief tour of the development of memory technology.

The ENIAC had only a small number of registers (about 20) for its storage and implemented these with the same basic vacuum tube technology that it used for building logic circuitry. However, the vacuum tube technology was far too expensive to be used to build a larger memory capacity. Eckert came up with the idea of developing a new technology based on mercury delay lines. In this technology, electrical signals were converted into vibrations that were sent down a tube of mercury, reaching the other end, where they were read out and recirculated. One mercury delay line could store about 0.5 Kbits. Although these bits were accessed serially, the mercury delay line was about a hundred times more cost-effective than vacuum tube memory. The first known working mercury delay lines were developed at Cambridge for the EDSAC. Figure e5.17.1 shows the mercury delay lines of the EDSAC, which had 32 tanks and 512 36-bit words.

Despite the tremendous advance offered by the mercury delay lines, they were terribly unreliable and still rather expensive. The breakthrough came with the invention of core memory by J. Forrester at MIT as part of the Whirlwind project in the early 1950s (see Figure e5.17.2). Core memory uses a ferrite core, which can be magnetized, and once magnetized, it acts as a store (just as a magnetic recording tape stores information). A set of wires running through the center of the core, which had a dimension of 0.1–1.0 millimeters, makes it possible to read the value stored on any ferrite core. The Whirlwind eventually included a core memory with 2048 16-bit words, or 32 Kbits. Core memory was a tremendous advance: it was cheaper, faster, considerably more reliable, and had higher density. Core memory was so much better than the alternatives that it became the dominant memory technology only a few years after its invention and remained so for nearly 20 years.

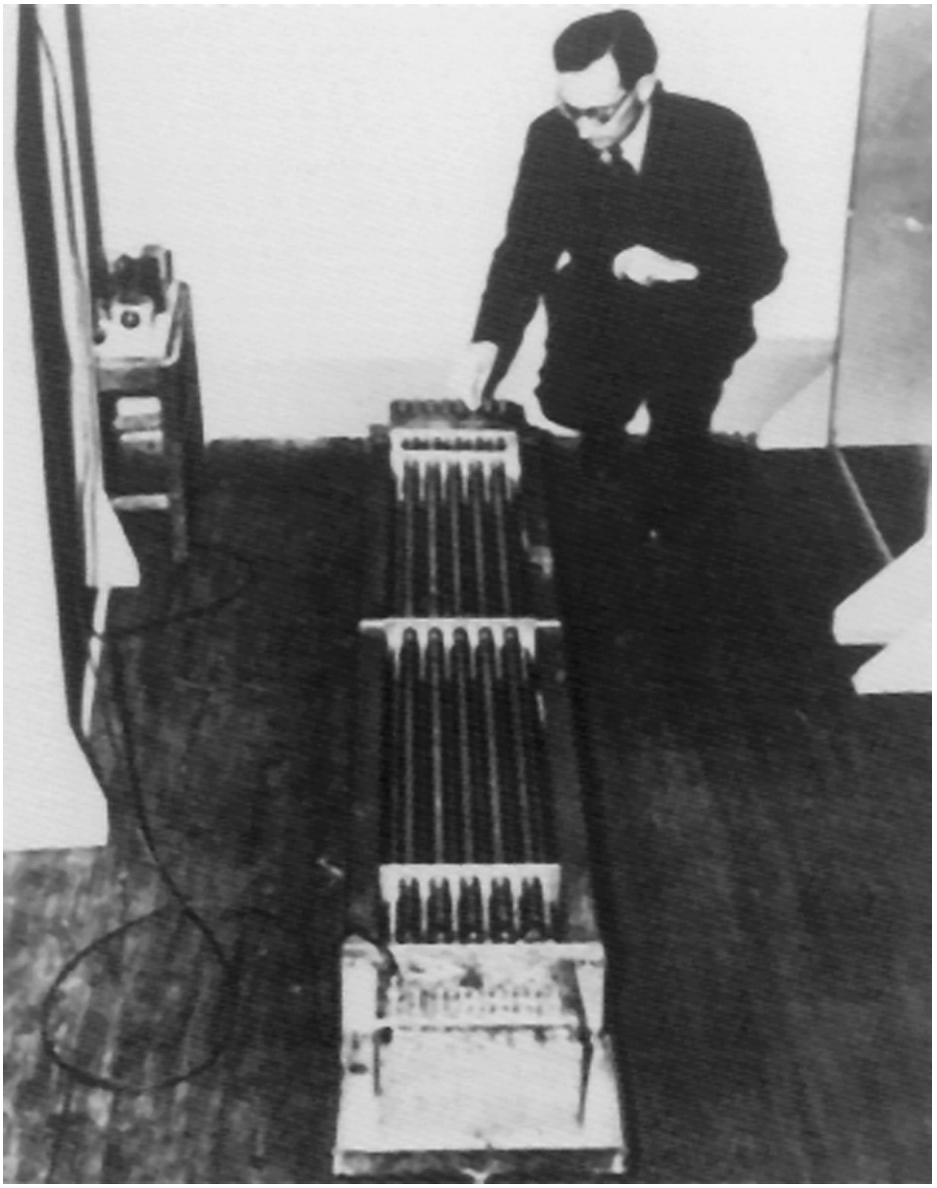


FIGURE e5.17.1 The mercury delay lines in the EDSAC. This technology made it possible to build the first stored-program computer. The young engineer in this photograph is none other than Maurice Wilkes, the lead architect of the EDSAC.

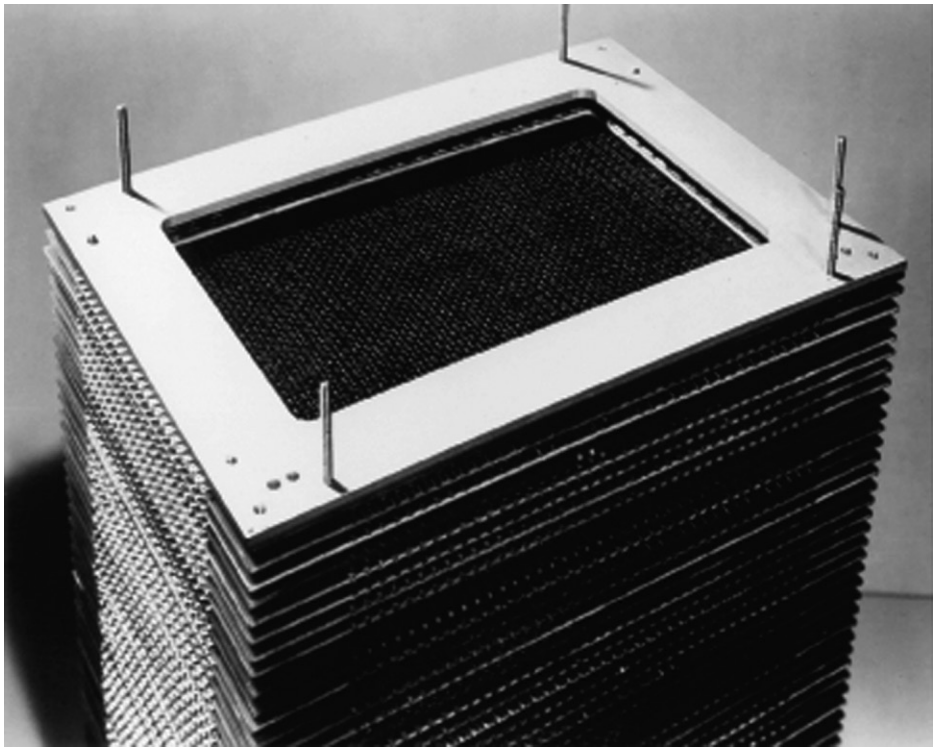


FIGURE e5.17.2 A core memory plane from the Whirlwind containing 256 cores arranged in a 16 × 16 array. Core memory was invented for the Whirlwind, which was used for air defense problems, and is now on display at the Smithsonian. (Incidentally, Ken Olsen, the founder of Digital and its president for 20 years, built the computer that tested these core memories; it was his first computer.)

The technology that replaced core memory was the same one that we now use both for logic and for memory: the integrated circuit. While registers were built out of transistorized memory in the 1960s, and IBM computers used transistorized memory for microcode store and caches in 1970, building main memory out of transistors remained prohibitively expensive until the development of the integrated circuit. With the integrated circuit, it became possible to build a DRAM (dynamic random access memory—see Appendix A for a description). The first DRAMs were built at Intel in 1970, and the computers using DRAM memories (as a high-speed option to core) came shortly thereafter; they used 1 Kbit DRAMs. In fact, computer folklore says that Intel developed the microprocessor partly to help

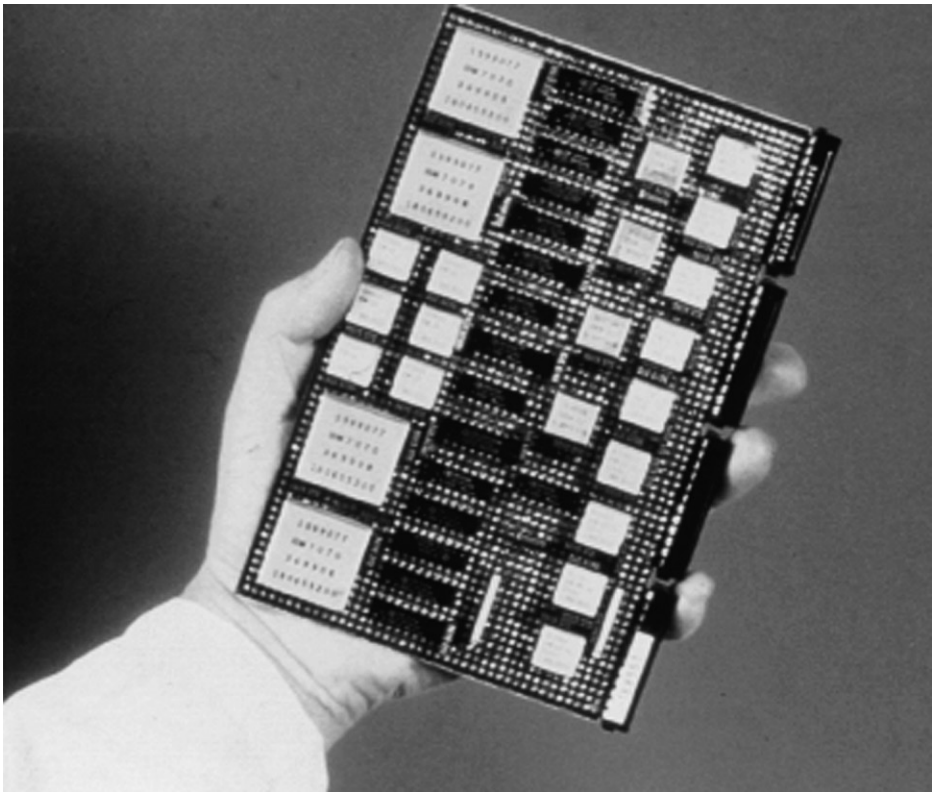


FIGURE e5.17.3 An early DRAM board. This board uses 18 Kbit chips.

sell more DRAM. Figure e5.17.3 shows an early DRAM board. By the late 1970s, core memory had become a historical curiosity. Just as core memory technology had allowed a tremendous expansion in memory size, DRAM technology allowed a comparable expansion. In the 1990s, many personal computers had as much memory as the largest computers using core memory ever had.

Nowadays, DRAMs are typically packaged with multiple chips on a little board called a DIMM (dual inline memory module). The SIMM (single inline memory module) shown in Figure e5.17.4 contains a total of 1 MB and sold for about $5 in 1997. As of 2004, DIMMs were available with up to 1024 MB and sold for about $100. While DRAMs will remain the dominant memory technology for some time to come, innovations in the packaging of DRAMs to provide both higher bandwidth and greater density are ongoing.

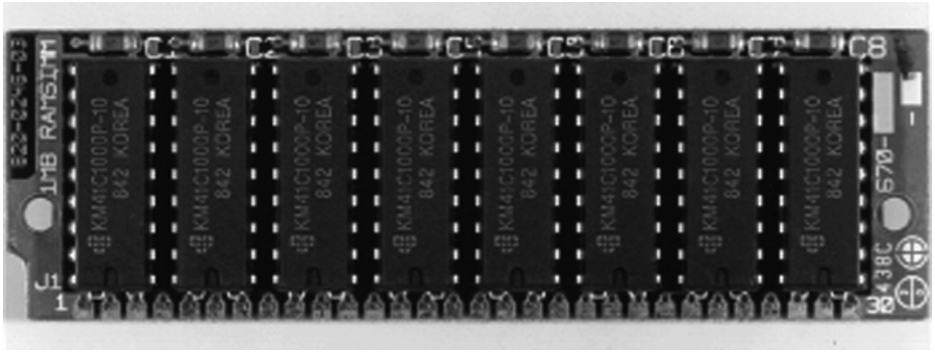


FIGURE e5.17.4 A 1 MB SIMM, built in 1986, using 1 Mbit chips. This SIMM sold for about $5/MB in 1997. As of 2006, most main memory is packed in DIMMs similar to this, though using much higher-density memory chips (1 Gbit).

The Development of Memory Hierarchies



Although the pioneers of computing foresaw the need for a **memory hierarchy** and coined the term, the automatic management of two levels was first proposed by Kilburn and his colleagues and demonstrated at the University of Manchester with the Atlas computer, which implemented virtual memory. This was the year *before* the IBM 360 was announced. IBM planned to include virtual memory with the next generation (System/370), but the OS/360 operating system wasn't up to the challenge in 1970. Virtual memory was announced for the 370 family in 1972, and it was for this computer that the term *translation-lookaside buffer* was coined. All but some embedded computers use virtual memory today.

The problems of inadequate address space have plagued designers repeatedly. The architects of the PDP-11 identified a small address space as the only architectural mistake from which it is difficult to recover. When the PDP-11 was designed, core memory densities were increasing at a very slow rate, and the competition from 100 other minicomputer companies meant that DEC might not have a cost-competitive product if every address had to go through the 16-bit datapath twice—hence, the decision to add just 4 more address bits than the predecessor of the PDP-11, to 16 from 12. The architects of the IBM 360 were aware of the importance of address size and planned for the architecture to extend to 32 bits of address. Only 24 bits were used in the IBM 360, however, because the low-end 360 models would have been even slower with the larger addresses. Unfortunately, the expansion effort was greatly complicated by programmers who stored extra information in the upper 8

"unused" address bits. The wider address lasted until 2000, when IBM expanded the architecture to 64 bits in the z-series.

Running out of address space has often been the cause of death for an architecture, while other architectures have managed to make the transition to a larger address space. For example, the PDP-11, a 16-bit computer, was replaced by the 32-bit VAX. The 80386 extended the 80286 architecture from a segmented 24-bit address space to a flat 32-bit address space in 1985. In the 1990s, several RISC instruction sets made the transition from 32-bit addressing to 64-bit addressing by providing a compatible extension of their instruction sets. MIPS was the first to do so. A decade later, Intel and HP announced the IA-64 in large part to provide a 64-bit address successor to the 32-bit Intel IA-32 and HP Precision architectures. The evolutionary AMD64 won that battle versus the revolutionary IA-64, and all but a few thousand of the 64-bit address computers from Intel are based on the x86.

Many of the early ideas in memory hierarchies originated in England. Just a few years after the Atlas paper, Wilkes [1965] published the first paper describing the concept of a cache, calling it a "slave":

> *The use is discussed of a fast core memory of, say, 32,000 words as slave to a slower core memory of, say, one million words in such a way that in practical cases the effective access time is nearer that of the fast memory than that of the slow memory.*

This two-page paper describes a direct-mapped cache. Although this was the first publication on caches, the first implementation was probably a direct-mapped instruction cache built at the University of Cambridge by Scarrott and described at the 1965 IFIP Congress. It was based on tunnel diode memory, the fastest form of memory available at the time.

Subsequent to that publication, IBM started a project that led to the first commercial computer with a cache, the IBM 360/85. Gibson at IBM recognized that memory-accessing behavior would have a significant impact on performance. He described how to measure program behavior and cache behavior and showed that the miss rate varies between programs. Using a sample of 20 programs (each with 3 million references—an incredible number for that time), Gibson analyzed the effectiveness of caches using average memory access time as the metric. Conti, Gibson, and Pitowsky described the resulting performance of the 360/85 in the first paper to use the term *cache* in 1968. Since this early work, it has become clear that caches are one of the most important ideas not only in computer architecture but in software systems as well. The idea of caching has found applications in operating systems, networking systems, databases, and compilers, to name a few. There are

thousands of papers on the topic of caching, and it continues to be a popular area of research.

One of the first papers on nonblocking caches was by Kroft in 1981, who may have coined the term. He later explained that he was the first to design a computer with a cache at Control Data Corporation, and when using old concepts for new mechanisms, he hit upon the idea of allowing his two-ported cache to continue to service other accesses on a miss.

Multilevel caches were the inevitable resolution to the lack of improvement in main memory latency and the higher clock rates of microprocessors. Only those in the field for a while are surprised by the size of some second- or third-level caches, as they are larger than main memories of past machines. The other surprise is that the number of levels is continually increasing, even on a single-chip microprocessor.

Disk Storage

In 1956, IBM developed the first disk storage system with both moving heads and multiple disk surfaces in San Jose, helping to seed the birth of the magnetic storage industry in the southern end of Silicon Valley. Reynold B. Johnson led the development of the IBM 305 RAMAC (Random Access Method of Accounting and Control). It could store 5 million characters (5 MB) of data on 50 disks, each 24 inches in diameter. The RAMAC is shown in Figures e5.17.5 and e5.17.6. Although the disk pioneers would be amazed at the size, cost, and capacity of modern disks, the basic mechanical design is the same as the RAMAC.

Moving-head disks quickly became the dominant high-speed magnetic storage, though their high cost meant that magnetic tape continued to be used extensively until the 1970s. The next key milestone for hard disks was the removable hard disk drive developed by IBM in 1962; this made it possible to share the expensive drive electronics and helped disks overtake tapes as the preferred storage medium. Figure e5.17.7 shows a removable disk drive and the multiplatter disk used in the drive. IBM also invented the floppy disk drive in 1970, originally to hold microcode for the IBM 370 series. Floppy disks became popular with the PC about 10 years later.

The sealed Winchester disk, which was developed by IBM in 1973, completely dominates disk technology today. Winchester disks benefited from two related properties. First, reductions in the cost of the disk electronics made it unnecessary to share the electronics and thus made nonremovable disks economical. Since the disk was fixed and could be in a sealed enclosure, both the environmental and control problems were greatly reduced, allowing significant gains in density. The first disk that IBM shipped had two spindles, each with a 30 MB disk; the moniker "30-30" for the disk led to the name Winchester. Winchester disks grew rapidly in popularity in the 1980s, completely replacing removable disks by the middle of that decade.

The historic role of IBM in the disk industry came to an end in 2002, when IBM sold its disk storage division to Hitachi. IBM continues to make storage subsystems, but it purchases its disk drives from others.

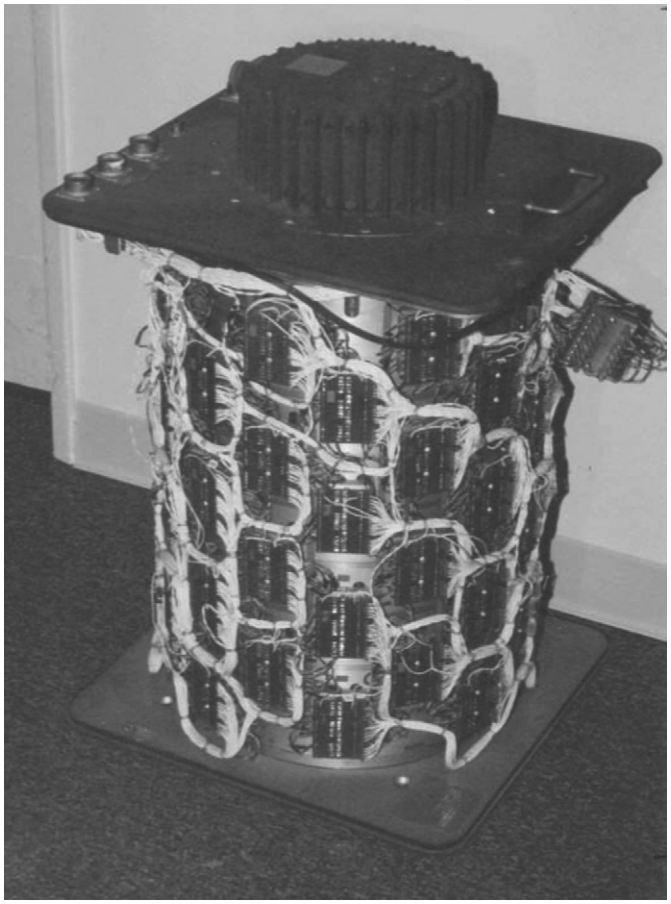


FIGURE e5.17.5 A magnetic drum made by Digital Development Corporation in the 1960s and used on a CDC machine. The electronics supporting the read/write heads can be seen on the outside of the drum.

A Very Brief History of Flash Memory

Flash memory was invented by researchers at Toshiba in the 1980s. They invented both the NOR-based Flash memory in 1984 and the denser NAND-based Flash memory in 1989. The first use was in digital cameras, starting with the CompactFlash form factor for NOR Flash memory and the SmartMedia form factor for NAND Flash memory. Today, all digital cameras, cell phones, music players, and tablets rely on Flash memory, and an increasing fraction of laptops use flash memory instead of disk.



FIGURE e5.17.6 The RAMAC disk drive from IBM, made in 1956, was the first disk drive with a moving head and the first with multiple platters. The IBM storage technology Web site has a discussion of IBM's major contributions to storage technology.

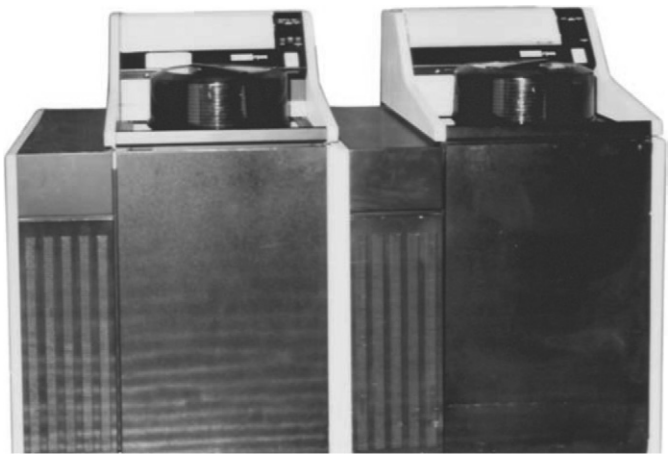


FIGURE e5.17.7 This is a DEC disk drive and the removable pack. These disks became popular starting in the mid-1960s and dominated disk technology until Winchester drives in the late 1970s. This drive was made in the mid-1970s; each disk pack in this drive could hold 80 MB.

A Brief History of Databases

Although there had been data stores of punch cards and later magnetic tapes, the emergence of the magnetic disk led to modern databases.

In 1961, Charles Bachman at General Electric created a pioneering database management system called Integrated Data Store (IDS) to take advantage of the new magnetic disks. In 1971, Bachman and others published standards on how to manage databases using Cobol programs, named the Codasyl approach after

the standards committee on which they served. Many companies offered Codasyl-compatible databases, but not IBM. IBM had introduced IMS in 1968, which was derived from IBM's work on the NASA Apollo project. Both Codasyl databases and IMS are classified as navigational databases because programs had to navigate through the data.

Ted Codd, a researcher at IBM, thought the navigational approach was wrong-headed. He recalled that people didn't write programs when dealing with the old punch card databases. Instead, they set up data flows through series of punch card machines that would perform simple functions like copy or sort. Once the card machines were set up, you just pushed all the cards through to get your results. In his view, users should only declare the type of data they were looking for and leave it up to computers to process it. In 1970, he published a new way to organize and access data called the relational model. It was based on set theory; data were independent of the implementation and users described what they were looking for in a declarative, nonprocedural language.

This paper led to considerable controversy within IBM, because it already had a database product. Codd even arranged a public debate between him and Bachman, which led to internal criticism at IBM that Codd was undermining IMS. The good news was that the debate led researchers at IBM and U.C. Berkeley to try to demonstrate the viability of relational databases by building System R and Ingres.

System R in 1974–79 demonstrated its feasibility and, perhaps more importantly, created the Structured Query Language (SQL) that is still widely used today. However, these results were not sufficient to convince IBM, and some of the researchers left IBM to build relational databases for other companies.

Mike Stonebraker and Gene Wong were interested in geographic data systems, and in 1973 they decided to pursue relational databases. Rather than build on IBM mainframes, the Ingres project was built on DEC minicomputers and Unix. Ingres was important because it led to a company that tried to commercialize the ideas, because 1000 copies of its source code were openly distributed, and because it trained a generation of database developers and researchers. The code and people led to many other companies, including Sybase. Larry Ellison started Oracle by first reading the papers from the System R and Ingres groups and then by hiring people who worked on those projects. Microsoft later purchased a copy of Sybase sources that became the foundation of its SQL Server product.

Relational databases matured in the 1980s, with IBM developing its own relational databases, including DB2. The 1990s saw both the development of object-oriented databases, to address the impedance mismatch between databases and programming, and the evolution of parallel databases for analytic processing and data mining.

ACM showered awards on this community. The ACM Turing Award went to Charles Bachman in 1973 for his contributions via IDS and the Codasyl group. Codd won it in 1980 for the relational model. In 1988, the developers of System R (Donald Chamberlin, Jim Gray, Raymond Lorie, Gianfranco Putzolu, Patricia Selinger, and Irving Traiger) shared the ACM Systems Software Award with the developers of Ingres (Gerald Held, Michael Stonebraker, and Eugene Wong). Jim

Gray won the Turing Award in 1998 for his contributions to transaction processing and databases. Stonebraker won it in 2014 for contributions to the concepts and practices underlying modern database systems. Finally, the first two ACM SIGMOD Innovations Awards went to Stonebraker and Gray, and the 2002 and 2003 editions went to Selinger and Chamberlin.

RAID

The small-form-factor hard disks for PCs in the mid-1980s led a group at Berkeley to propose redundant arrays of inexpensive disks (RAID). This group had worked on the reduced instruction set computer effort and so expected much faster processors to become available. Their two questions were: What could be done with the small disks that accompanied their PCs? What could be done in the area of I/O to keep up with much faster processors? They argued to replace one large mainframe drive with 50 small drives, as you could get much greater performance with that many independent arms. The many small drives even offered savings in power consumption and floor space.

The downside of many disks was much lower MTTF. Hence, on their own they reasoned out the advantages of redundant disks and rotating parity to address how to get greater performance with many small drives yet have reliability as high as that of a single mainframe disk.

The problem they experienced when explaining their ideas was that some researchers had heard of disk arrays with some form of redundancy, and they didn't understand the Berkeley proposal. Hence, the first RAID paper [Patterson, Gibson, and Katz 1987] is not only a case for arrays of small-form-factor disk drives, but also something of a tutorial and classification of existing work on disk arrays. Mirroring (RAID 1) had long been used in fault-tolerant computers such as those sold by Tandem. Thinking Machines had arrays with 32 data disks and seven check disks using ECC for correction (RAID 2) in 1987, and Honeywell Bull had a RAID 2 product even earlier. Also, disk arrays with a single parity disk had been used in scientific computers in the same time frame (RAID 3). Their paper then described a single parity disk with support for sector accesses (RAID 4) and rotated parity (RAID 5). Chen et al. [1994] survey the original RAID ideas, commercial products, and other developments.

Unknown to the Berkeley group, engineers at IBM working on the AS/400 computer also came up with rotated parity to give greater reliability for a collection of large disks. IBM filed a patent on RAID 5 shortly before the Berkeley group submitted their paper. Patents for RAID 1, RAID 2, and RAID 3 from several companies predate the IBM RAID 5 patent, which has led to plenty of courtroom action.

EMC had been a supplier of DRAM boards for IBM computers, but around 1988 new policies from IBM made it nearly impossible for EMC to continue to sell IBM memory boards. The Berkeley paper crossed the desks of EMC executives, and so they decided to go after the market dominated by IBM disk storage products. As the paper advocated, their model was to use many small drives to compete with mainframe drives, and EMC announced a RAID product in 1990. It relied on

mirroring (RAID 1) for reliability; RAID 5 products came much later for EMC. Over the next year, Micropolis offered a RAID 3 product; Compaq offered a RAID 4 product; and Data General, IBM, and NCR offered RAID 5 products.

The RAID ideas soon spread to the rest of the workstation and server industry. An article explaining RAID in *Byte* magazine led to RAID products being offered on desktop PCs, which was something of a surprise to the Berkeley group. They had focused on performance with good availability, but higher availability was attractive to the PC market.

Another surprise was the cost of the disk arrays. With redundant power supplies and fans, the ability to "hot-swap" a disk drive, the RAID hardware controller itself, the redundant disks, and so on, the first disk arrays cost many times the cost of the disks. Perhaps as a result, the "inexpensive" in RAID morphed into "independent." Many marketing departments and technical writers today know of RAID only as "redundant arrays of independent disks."

In 2004, more than 80% of the nondesktop drive sales were found in RAIDs. In recognition of their role, in 1999 Garth Gibson, Randy Katz, and David Patterson received the IEEE Reynold B. Johnson Information Storage Award "for the development of Redundant Arrays of Inexpensive Disks (RAID)."

Protection Mechanisms

Architectural support for protection has varied greatly over the past 20 years. In early computers, before virtual memory, protection was very simple at best. In the 1960s, more sophisticated mechanisms that supported different protection levels (called *rings*) were invented. In the late 1970s and early 1980s, very elaborate mechanisms for protection were devised and later built; these mechanisms supported a variety of powerful protection schemes that allowed controlled instances of sharing, in such a way that a process could share data while controlling exactly what was done to the data. The most powerful method, called *capabilities*, created a data object that described the access rights to some portion of memory. These capabilities could then be passed to other processes, thus granting access to the object described by the capability. Supporting this sophisticated protection mechanism was both complex and costly, because creation, copying, and manipulation of capabilities required a combination of operating system and hardware support. Recent computers all support a simpler protection scheme based on virtual memory, similar to that discussed in Section 5.7. Given current concerns about computer security due to the costs of worms and viruses, perhaps we will see a renaissance in protection research, potentially renewing interest in 20-year-old publications.

As mentioned in the text, system virtual machines were pioneered at IBM as part of its investigation into virtual memory. IBM's first computer with virtual memory was the IBM 360/67, introduced in 1967. IBM researchers wrote the program CP-67, which created the illusion of several independent 360 computers. They then wrote an interactive, single-user operating system called CMS that ran on these virtual machines. CP-67 led to the product VM/370, and today IBM sells z/VM for its mainframe computers.

A Brief History of Modern Operating Systems

MIT developed the first timesharing system, CTSS (Compatible Time-Sharing System), in 1961. John McCarthy is generally given credit for the idea of timesharing, but Fernando Corbato was the systems person who realized the concept in the form of the CTSS. CTSS allowed three people to share a machine, and its response time of minutes or seconds was a dramatic improvement over the batch processing system it replaced. Moreover, it demonstrated the value of interactive computing.

Flush with the success of their first system, this group launched into their second system, MULTICS (Multiplexed Information and Computing Service). They included many innovations, such as strong protection, controlled sharing, and dynamic libraries. However, it suffered from the "second system effect." Fred Brooks, Jr. described the second system effect in his classic book about lessons learned from developing an operating system for the IBM mainframe, *The Mythical Man Month:*

> *When one is designing the successor to a relatively small, elegant, and successful system, there is a tendency to become grandiose in one's success and design an elephantine feature-laden monstrosity.*

MULTICS took sharing to a logical extreme to discover the issues, including that it was too extreme. MIT, General Electric, and later Bell Labs all tried to build an economical and useful system. Despite a great deal of time and money, they failed.

UC Berkeley was building its own timesharing system, Cal TSS. ("Cal" is a nickname for University of California.) The people leading that project included Peter Deutsch, Butler Lampson, Chuck Thacker, and Ken Thompson. They added paging virtual memory hardware to an SDS 920 and wrote an operating system for it. SDS sold this computer as the SDS-930, and it was the first commercially available timesharing system to have operational hardware and software. Thompson graduated and joined Bell Labs. The others founded Berkeley Computer Corporation (BCC), with the goal of selling time-sharing hardware and software. We'll pick up BCC later in the story, but for now let's follow Thompson.

At Bell Labs in 1971, Thompson led the development of a simple timesharing system that had some of the good ideas of MULTICS but left out many of the complex features. To demonstrate the contrast, it was first called UNICS. As they were joined by others at Bell Labs who had been burned from the MULTICS experience, it was renamed UNIX, with the *x* coming from Phoenix, the legendary bird that rose from the ashes.

Their result was the most elegant operating system ever built. Forced to live in the 16-bit address space of the DEC minicomputers, it had an amazing amount of functionality per line of code. Major contributions were pipes, a uniform file system, a uniform process model, and the shell user interface that allowed users to connect programs together using pipes and files.

Dennis Ritchie joined the UNIX team in 1973 from MIT, where he had experience in MULTICS, which was written in a high-level language. Like prior operating systems, UNIX had been written in assembly language. Ritchie designed a language for system implementation called C, and it was used to make UNIX portable.

Between 1971 and 1976, Bell released six editions of the UNIX timesharing system. Thompson took a sabbatical at his alma mater and brought UNIX with him. Berkeley and many other universities began to use UNIX on the popular PDP-11 minicomputer.

When DEC announced the VAX, a 32-bit virtual address successor to the PDP-11, the question arose as to what operating system should be run. UNIX became the first operating system to be migrated to a different computer when it was ported to the VAX.

Students at Berkeley had one of the first VAXes, and they were soon adding features to UNIX for the VAX, such as paging and a very efficient implementation of the TCP/IP protocol. The Berkeley implementation of TCP/IP was notable not just because it was fast. It was essentially the *only* implementation of TCP/IP for years, since early implementations in most other operating systems consisted of copying the Berkeley code verbatim, with minimal changes to integrate into the local system.

The Advanced Research Project Agency (ARPA), which funded computer science research, asked a Stanford professor, Forrest Basket, to recommend which system the academic community should use: the DEC operating system VMS, led by David Cutler, or the Berkeley version of UNIX, led by a graduate student named Bill Joy. He recommended the latter, and Berkeley UNIX soon became the academic standard bearer.

The Berkeley Software Distribution (BSD) of UNIX, first released in 1978, was essentially one of the first open source movements. The sources were shipped with the tapes, and systems developers around the world learned their craft by studying the UNIX code.

BSD was also the first split of UNIX, because AT&T Bell Labs continued to develop UNIX on its own. This eventually led to a forest of UNIXes, as each company compiled the UNIX source code for their architecture. Bill Joy graduated from Berkeley and helped found Sun Microsystems, so naturally Sun OS was based on BSD UNIX. Among the many UNIX flavors were Santa Cruz Operation UNIX, HP-UX, and IBM's AIX. AT&T and Sun attempted to unify UNIX by striking a deal whereby AT&T and Sun would combine forces and jointly develop AT&T UNIX. This led to an adverse reaction from HP, IBM, and others, because they did not want a competitor supplying their code, so they created the Open Source Foundation as a competing organization.

In addition to the UNIX variants from companies, public domain versions also proliferated. The BSD team at Berkeley rewrote substantial portions of UNIX so that they could distribute it without needing a license from AT&T. This eventually led to a lawsuit, which Berkeley won. BSD UNIX soon split into FreeBSD, NetBSD, and OpenBSD, provided by competing camps of developers. Apple's current operating system, OS X, is based on Free BSD.

Let's go back to Berkeley Computer Corporation. Alas, this effort was not commercially viable. About the same time as BCC was getting in trouble, Xerox hired Robert Taylor to build the computer science division of the new Xerox Palo Alto

Research Center (PARC) in 1970. He had just returned from a tour of duty at ARPA, where he had funded the Berkeley research. He recruited Deutsch, Lampson, and Thacker from BCC to form the core of PARC's team: 11 of the initial 20 employees were from BCC, and they decided to build small computers for individuals rather than large computers for groups. This first personal computer, called the Alto, was built from the same technology as minicomputers, but it had a keyboard, mouse, graphical display, and windows. It popularized windows and led to many inventions, including client-server computing, the Ethernet, and print servers. It directly inspired the Macintosh, which was the successor to the popular Apple II.

IBM had long been interested in selling to the home, so the success of the Apple II led IBM to start a competing project. In contrast to its tradition, for this project IBM designed everything from components outside of the company. They selected the new 16-bit microprocessor from Intel, the 8086. (To lower costs, they started with the version with the 8-bit bus, called the 8088.) They visited Microsoft to see if this small company would be willing to sell their popular Basic interpreter and asked for recommendations for an operating system. Gates volunteered that Microsoft could deliver both an interpreter and an operating system, as long as they were paid a royalty fee of between $10 and $50 for each copy rather than a flat fee. IBM agreed, provided Microsoft could meet their deadlines. Microsoft didn't have an operating system, nor the time and resources to build one, but Gates knew that a Seattle company had developed an operating system for the Intel 8086. Microsoft purchased QDOS (Quick and Dirty Operating System) for $15,000, made a small change and relabeled it MS-DOS. MS-DOS was a simple operating system without any modern features—no protection, no processes, and no virtual memory—in part because they believed it wasn't necessary for a personal computer.

Announced in 1980, the IBM PC became a tremendous success for IBM and the companies it relied upon. Microsoft sold 500,000 copies of MS-DOS by 1983, and the $10 million income allowed Microsoft to start new software projects.

After seeing a version of the Macintosh under development, Microsoft hired some people from PARC to lead its reply. The Macintosh was announced in 1984, and Windows was available on PCs the following year. It was originally an application that ran on top of DOS, but was later integrated with DOS and renamed Windows 2.0. Microsoft hired Cutler from DEC to lead the development of Windows NT, a new operating system. NT was a modern operating system with protection, processors, and so on and has much in common with DEC's VMS. Today's PC operating systems are more sophisticated than any of the timesharing systems of 20 years ago, yet they still suffer from the need to maintain compatibility with the crippled first PC operating systems such as MS-DOS.

The popularity of the PC led to a desire for a UNIX that ran on it. Many tried to develop one, but the most successful was written from scratch in 1991 by Linus Torvalds. In addition to making the source code available, like BSD, he allowed everyone to make changes and submit them for inclusion in his next release. Linux popularized open source development as we know it today, with such software getting hundreds of volunteers to test releases and add new features.

Many people in this story won awards for their roles in the development of modern operating systems. McCarthy received an ACM Turing Award in 1971 in part for his contributions to timesharing. In 1983, Thompson and Ritchie received one for UNIX. The announcement said that "the genius of the UNIX system is its framework, which enables programmers to stand on the work of others." In 1990, Corbato received the Turing Award for his contributions to CTSS and MULTICS. Two years later, Lampson won it in part for his work on personal computing and operating systems.

Further Reading

Brooks, F. P. [1975]. *The mythical man-month*. Reading: Addison-Wesley.

The classic book that explains the challenge of software engineering using IBM OS development as the example.

Cantin, J. F. and M. D. Hill [2001]. "Cache performance for selected SPEC CPU2000 benchmarks", *SIGARCH Computer Architecture News* 29:4 þ (September), 13–18.

A reference paper of cache miss rates for many cache sizes for the SPEC2000 benchmarks.

Chen, P. M., E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson [1994]. "RAID: High-performance, reliable secondary storage", *ACM Computing Surveys* 26:2 (June) 145–88.

A tutorial covering disk arrays and the advantages of such an organization.

Conti, C., D. H. Gibson, and S. H. Pitowsky [1968]. "Structural aspects of the System/360 Model 85, part I: General organization", *IBM Systems J.* 7:1, 2–14.

A classic paper that describes the first commercial computer to use a cache and its resulting performance.

Hennessy, J. and D. Patterson [2003]. Chapter 5 in *Computer Architecture: A Quantitative Approach*, third edition, Morgan Kaufmann Publishers, San Francisco.

For more in-depth coverage of a variety of topics including protection, cache performance of out-of-order processors, virtually addressed caches, multilevel caches, compiler optimizations, additional latency tolerance mechanisms, and cache coherency.

Kilburn, T., D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner [1962]. "One-level storage system", *IRE Transactions on Electronic Computers* EC-11(April), 223–335. Also appears in D. P. Siewiorek, C. G. Bell, and A. Newell [1982], *Computer Structures: Principles and Examples*, McGraw-Hill, New York, 135–48.

This classic paper is the first proposal for virtual memory.

LaMarca, A. and R. E. Ladner [1996]. "The influence of caches on the performance of heaps", *ACM J. of Experimental Algorithmics*, Vol. 1.

This paper shows the difference between complexity analysis of an algorithm, instruction count performance, and memory hierarchy for four sorting algorithms.

McCalpin, J.D. [1995]. "*STREAM: Sustainable Memory Bandwidth in High Performance Computers*", https://www.cs.virginia.edu/stream/.

A widely used microbenchmark that measures the performance of the memory system behind the caches.

Patterson, D., G. Gibson, and R. Katz [1988]. "*A case for redundant arrays of inexpensive disks (RAID)*", SIGMOD Conference, 109–116.

A classic paper that advocates arrays of smaller disks and introduces RAID levels.

Przybylski, S. A. [1990]. *Cache and Memory Hierarchy Design: A Performance-Directed Approach*, Morgan Kaufmann Publishers, San Francisco.

A thorough exploration of multilevel memory hierarchies and their performance.

Ritchie, D. [1984]. "The evolution of the UNIX time-sharing system", *AT&T Bell Laboratories Technical Journal* 1984, 1577–1593.

The history of UNIX from one of its inventors.

Ritchie, D. M. and K. Thompson [1978]. "The UNIX time-sharing system", *Bell System Technical Journal* (August), 1991–2019.

A paper describing the most elegant operating system ever invented.

Silberschatz, A., P. Galvin, and G. Grange [2003]. *Operating System Concepts*, sixth edition, Addison-Wesley, Reading, MA.

An operating systems textbook with a thorough discussion of virtual memory, processes and process management, and protection issues.

Smith, A. J. [1982]. "Cache memories," *Computing Surveys* 14:3 (September), 473–530.

The classic survey paper on caches. This paper defined the terminology for the field and has served as a reference for many computer designers.

Smith, D. K. and R. C. Alexander [1988]. *Fumbling the Future: How Xerox Invented, Then Ignored, the First Personal Computer*, Morrow, New York.

A popular book that explains the role of Xerox PARC in laying the foundation for today's computing, but which Xerox did not substantially benefit from.

Tanenbaum, A. [2001]. *Modern Operating Systems*, second edition, Upper Saddle River, Prentice Hall, NJ.

An operating system textbook with a good discussion of virtual memory.

Wilkes, M. [1965]. "Slave memories and dynamic storage allocation", *IEEE Trans. Electronic Computers* EC-14:2 (April), 270–71.

The first classic paper on caches.

5.1.4 [5] <§5.1> Which variable references exhibit temporal locality?

5.1.5 [5] <§5.1> Which variable references exhibit spatial locality?

5.1.6 [15] <§5.1> How many 16-byte cache blocks are needed to store all 64-bit matrix elements being referenced using Matlab's matrix storage? How many using C's matrix storage? (Assume each row contains more than one element.)

5.2 Caches are important to providing a high-performance memory hierarchy to processors. Below is a list of 64-bit memory address references, given as word addresses.

```
0x03, 0xb4, 0x2b, 0x02, 0xbf, 0x58, 0xbe, 0x0e, 0xb5,
0x2c, 0xba, 0xfd
```

5.2.1 [10] <§5.3> For each of these references, identify the binary word address, the tag, and the index given a direct-mapped cache with 16 one-word blocks. Also list whether each reference is a hit or a miss, assuming the cache is initially empty.

5.2.2 [10] <§5.3> For each of these references, identify the binary word address, the tag, the index, and the offset given a direct-mapped cache with two-word blocks and a total size of eight blocks. Also list if each reference is a hit or a miss, assuming the cache is initially empty.

5.2.3 [20] <§§5.3, 5.4> You are asked to optimize a cache design for the given references. There are three direct-mapped cache designs possible, all with a total of eight words of data:

- C1 has 1-word blocks,
- C2 has 2-word blocks, and
- C3 has 4-word blocks.

5.3 By convention, a cache is named according to the amount of data it contains (i.e., a 4 KiB cache can hold 4 KiB of data); however, caches also require SRAM to store metadata such as tags and valid bits. For this exercise, you will examine how a cache's configuration affects the total amount of SRAM needed to implement it as well as the performance of the cache. For all parts, assume that the caches are byte addressable, and that addresses and words are 64 bits.

5.3.1 [10] <§5.3> Calculate the total number of bits required to implement a 32 KiB cache with two-word blocks.

5.3.2 [10] <§5.3> Calculate the total number of bits required to implement a 64 KiB cache with 16-word blocks. How much bigger is this cache than the 32 KiB cache described in Exercise 5.3.1? (Notice that, by changing the block size, we doubled the amount of data without doubling the total size of the cache.)

5.3.3 [5] <§5.3> Explain why this 64 KiB cache, despite its larger data size, might provide slower performance than the first cache.

5.3.4 [10] <§§5.3, 5.4> Generate a series of read requests that have a lower miss rate on a 32 KiB two-way set associative cache than on the cache described in Exercise 5.3.1.

5.4 [15] <§5.3> Section 5.3 shows the typical method to index a direct-mapped cache, specifically (Block address) modulo (Number of blocks in the cache). Assuming a 64-bit address and 1024 blocks in the cache, consider a different indexing function, specifically (Block address[63:54] XOR Block address[53:44]). Is it possible to use this to index a direct-mapped cache? If so, explain why and discuss any changes that might need to be made to the cache. If it is not possible, explain why.

5.5 For a direct-mapped cache design with a 64-bit address, the following bits of the address are used to access the cache.

Tag	Index	Offset
63–10	9–5	4–0

5.5.1 [5] <§5.3> What is the cache block size (in words)?

5.5.2 [5] <§5.3> How many blocks does the cache have?

5.5.3 [5] <§5.3> What is the ratio between total bits required for such a cache implementation over the data storage bits?

Beginning from power on, the following byte-addressed cache references are recorded.

Address											
Hex	00	04	10	84	E8	A0	400	1E	8C	C1C	B4	884
Dec	0	4	16	132	232	160	1024	30	140	3100	180	2180

5.5.4 [20] <§5.3> For each reference, list (1) its tag, index, and offset, (2) whether it is a hit or a miss, and (3) which bytes were replaced (if any).

5.5.5 [5] <§5.3> What is the hit ratio?

5.5.6 [5] <§5.3> List the final state of the cache, with each valid entry represented as a record of <index, tag, data>. For example,

```
<0, 3, Mem[0xC00]-Mem[0xC1F]>
```

5.6 Recall that we have two write policies and two write allocation policies, and their combinations can be implemented either in L1 or L2 cache. Assume the following choices for L1 and L2 caches:

L1	L2
Write through, non-write allocate	Write back, write allocate

5.6.1 [5] <§§5.3, 5.8> Buffers are employed between different levels of memory hierarchy to reduce access latency. For this given configuration, list the possible buffers needed between L1 and L2 caches, as well as L2 cache and memory.

5.6.2 [20] <§§5.3, 5.8> Describe the procedure of handling an L1 write-miss, considering the components involved and the possibility of replacing a dirty block.

5.6.3 [20] <§§5.3, 5.8> For a multilevel exclusive cache configuration (a block can only reside in one of the L1 and L2 caches), describe the procedures of handling an L1 write-miss and an L1 read-miss, considering the components involved and the possibility of replacing a dirty block.

5.7 Consider the following program and cache behaviors.

Data Reads per 1000 Instructions	Data Writes per 1000 Instructions	Instruction Cache Miss Rate	Data Cache Miss Rate	Block Size (bytes)
250	100	0.30%	2%	64

5.7.1 [10] <§§5.3, 5.8> Suppose a CPU with a write-through, write-allocate cache achieves a CPI of 2. What are the read and write bandwidths (measured by bytes per cycle) between RAM and the cache? (Assume each miss generates a request for one block.)

5.7.2 [10] <§§5.3, 5.8> For a write-back, write-allocate cache, assuming 30% of replaced data cache blocks are dirty, what are the read and write bandwidths needed for a CPI of 2?

5.8 Media applications that play audio or video files are part of a class of workloads called "streaming" workloads (i.e., they bring in large amounts of data but do not reuse much of it). Consider a video streaming workload that accesses a 512 KiB working set sequentially with the following word address stream:

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ...
```

5.8.1 [10] <§§5.4, 5.8> Assume a 64 KiB direct-mapped cache with a 32-byte block. What is the miss rate for the address stream above? How is this miss rate sensitive to the size of the cache or the working set? How would you categorize the misses this workload is experiencing, based on the 3C model?

5.8.2 [5] <§§5.1, 5.8> Re-compute the miss rate when the cache block size is 16 bytes, 64 bytes, and 128 bytes. What kind of locality is this workload exploiting?

5.8.3 [10] <§5.13> "Prefetching" is a technique that leverages predictable address patterns to speculatively bring in additional cache blocks when a particular cache block is accessed. One example of prefetching is a stream buffer that prefetches sequentially adjacent cache blocks into a separate buffer when a particular cache block is brought in. If the data are found in the prefetch buffer, it is considered

as a hit, moved into the cache, and the next cache block is prefetched. Assume a two-entry stream buffer; and, assume that the cache latency is such that a cache block can be loaded before the computation on the previous cache block is completed. What is the miss rate for the address stream above?

5.9 Cache block size (B) can affect both miss rate and miss latency. Assuming a machine with a base CPI of 1, and an average of 1.35 references (both instruction and data) per instruction, find the block size that minimizes the total miss latency given the following miss rates for various block sizes.

8: 4%	16: 3%	32: 2%	64: 1.5%	128: 1%

5.9.1 [10] <§5.3> What is the optimal block size for a miss latency of 20 × B cycles?

5.9.2 [10] <§5.3> What is the optimal block size for a miss latency of 24 + B cycles?

5.9.3 [10] <§5.3> For constant miss latency, what is the optimal block size?

5.10 In this exercise, we will look at the different ways capacity affects overall performance. In general, cache access time is proportional to capacity. Assume that main memory accesses take 70 ns and that 36% of all instructions access data memory. The following table shows data for L1 caches attached to each of two processors, P1 and P2.

	L1 Size	L1 Miss Rate	L1 Hit Time
P1	2 KiB	8.0%	0.66 ns
P2	4 KiB	6.0%	0.90 ns

5.10.1 [5] <§5.4> Assuming that the L1 hit time determines the cycle times for P1 and P2, what are their respective clock rates?

5.10.2 [10] <§5.4> What is the Average Memory Access Time for P1 and P2 (in cycles)?

5.10.3 [5] <§5.4> Assuming a base CPI of 1.0 without any memory stalls, what is the total CPI for P1 and P2? Which processor is faster? (When we say a "base CPI of 1.0", we mean that instructions complete in one cycle, unless either the instruction access or the data access causes a cache miss.)

For the next three problems, we will consider the addition of an L2 cache to P1 (to presumably make up for its limited L1 cache capacity). Use the L1 cache capacities and hit times from the previous table when solving these problems. The L2 miss rate indicated is its local miss rate.

L2 Size	L2 Miss Rate	L2 Hit Time
1 MiB	95%	5.62 ns

5.10.4 [10] <§5.4> What is the AMAT for P1 with the addition of an L2 cache? Is the AMAT better or worse with the L2 cache?

5.10.5 [5] <§5.4> Assuming a base CPI of 1.0 without any memory stalls, what is the total CPI for P1 with the addition of an L2 cache?

5.10.6 [10] <§5.4> What would the L2 miss rate need to be in order for P1 with an L2 cache to be faster than P1 without an L2 cache?

5.10.7 [15] <§5.4> What would the L2 miss rate need to be in order for P1 with an L2 cache to be faster than P2 without an L2 cache?

5.11 This exercise examines the effect of different cache designs, specifically comparing associative caches to the direct-mapped caches from Section 5.4. For these exercises, refer to the sequence of word address shown below.

```
0x03, 0xb4, 0x2b, 0x02, 0xbe, 0x58, 0xbf, 0x0e, 0x1f,
0xb5, 0xbf, 0xba, 0x2e, 0xce
```

5.11.1 [10] <§5.4> Sketch the organization of a three-way set associative cache with two-word blocks and a total size of 48 words. Your sketch should have a style similar to Figure 5.18, but clearly show the width of the tag and data fields.

5.11.2 [10] <§5.4> Trace the behavior of the cache from Exercise 5.11.1. Assume a true LRU replacement policy. For each reference, identify

- the binary word address,
- the tag,
- the index,
- the offset
- whether the reference is a hit or a miss, and
- which tags are in each way of the cache after the reference has been handled.

5.11.3 [5] <§5.4> Sketch the organization of a fully associative cache with one-word blocks and a total size of eight words. Your sketch should have a style similar to Figure 5.18, but clearly show the width of the tag and data fields.

5.11.4 [10] <§5.4> Trace the behavior of the cache from Exercise 5.11.3. Assume a true LRU replacement policy. For each reference, identify

- the binary word address,
- the tag,
- the index,
- the offset,

- whether the reference is a hit or a miss, and
- the contents of the cache after each reference has been handled.

5.11.5 [5] <§5.4> Sketch the organization of a fully associative cache with two-word blocks and a total size of eight words. Your sketch should have a style similar to Figure 5.18, but clearly show the width of the tag and data fields.

5.11.6 [10] <§5.4> Trace the behavior of the cache from Exercise 5.11.5. Assume an LRU replacement policy. For each reference, identify

- the binary word address,
- the tag,
- the index,
- the offset,
- whether the reference is a hit or a miss, and
- the contents of the cache after each reference has been handled.

5.11.7 [10] <§5.4> Repeat Exercise 5.11.6 using MRU (*most recently used*) replacement.

5.11.8 [15] <§5.4> Repeat Exercise 5.11.6 using the optimal replacement policy (i.e., the one that gives the lowest miss rate).

5.12 Multilevel caching is an important technique to overcome the limited amount of space that a first-level cache can provide while still maintaining its speed. Consider a processor with the following parameters:

Base CPI, No Memory Stalls	Processor Speed	Main Memory Access Time	First-Level Cache Miss Rate per Instruction**	Second-Level Cache, Direct-Mapped Speed	Miss Rate with Second-Level Cache, Direct-Mapped	Second-Level Cache, Eight-Way Set Associative Speed	Miss Rate with Second-Level Cache, Eight-Way Set Associative
1.5	2 GHz	100 ns	7%	12 cycles	3.5%	28 cycles	1.5%

**First Level Cache miss rate is per instruction. Assume the total number of L1 cache misses (instruction and data combined) is equal to 7% of the number of instructions.

5.12.1 [10] <§5.4> Calculate the CPI for the processor in the table using: 1) only a first-level cache, 2) a second-level direct-mapped cache, and 3) a second-level eight-way set associative cache. How do these numbers change if main memory access time doubles? (Give each change as both an absolute CPI and a percent change.) Notice the extent to which an L2 cache can hide the effects of a slow memory.

5.12.2 [10] <§5.4> It is possible to have an even greater cache hierarchy than two levels? Given the processor above with a second-level, direct-mapped cache, a designer wants to add a third-level cache that takes 50 cycles to access and will have a 13% miss rate. Would this provide better performance? In general, what are the advantages and disadvantages of adding a third-level cache?

5.12.3 [20] <§5.4> In older processors, such as the Intel Pentium or Alpha 21264, the second level of cache was external (located on a different chip) from the main processor and the first-level cache. While this allowed for large second-level caches, the latency to access the cache was much higher, and the bandwidth was typically lower because the second-level cache ran at a lower frequency. Assume a 512 KiB off-chip second-level cache has a miss rate of 4%. If each additional 512 KiB of cache lowered miss rates by 0.7%, and the cache had a total access time of 50 cycles, how big would the cache have to be to match the performance of the second-level direct-mapped cache listed above?

5.13 *Mean time between failures* (MTBF), *mean time to replacement* (MTTR), and *mean time to failure* (MTTF) are useful metrics for evaluating the reliability and availability of a storage resource. Explore these concepts by answering the questions about a device with the following metrics:

MTTF	MTTR
3 Years	1 Day

5.13.1 [5] <§5.5> Calculate the MTBF for such a device.

5.13.2 [5] <§5.5> Calculate the availability for such a device.

5.13.3 [5] <§5.5> What happens to availability as the MTTR approaches 0? Is this a realistic situation?

5.13.4 [5] <§5.5> What happens to availability as the MTTR gets very high, i.e., a device is difficult to repair? Does this imply the device has low availability?

5.14 This exercise examines the *single error correcting, double error detecting* (SEC/DED) Hamming code.

5.14.1 [5] <§5.5> What is the minimum number of parity bits required to protect a 128-bit word using the SEC/DED code?

5.14.2 [5] <§5.5> Section 5.5 states that modern server memory modules (DIMMs) employ SEC/DED ECC to protect each 64 bits with 8 parity bits. Compute the cost/performance ratio of this code to the code from Exercise 5.14.1. In this case, cost is the relative number of parity bits needed while performance is the relative number of errors that can be corrected. Which is better?

5.14.3 [5] <§5.5> Consider a SEC code that protects 8 bit words with 4 parity bits. If we read the value 0x375, is there an error? If so, correct the error.

5.15 For a high-performance system such as a B-tree index for a database, the page size is determined mainly by the data size and disk performance. Assume that, on average, a B-tree index page is 70% full with fix-sized entries. The utility of a page is its B-tree depth, calculated as $\log_2$(entries). The following table shows that for 16-byte entries, and a 10-year-old disk with a 10 ms latency and 10 MB/s transfer rate, the optimal page size is 16 K.

Page Size (KiB)	Page Utility or B-Tree Depth (Number of Disk Accesses Saved)	Index Page Access Cost (ms)	Utility/Cost
2	6.49 (or $\log_2(2048/16 \times 0.7)$)	10.2	0.64
4	7.49	10.4	0.72
8	8.49	10.8	0.79
16	9.49	11.6	0.82
32	10.49	13.2	0.79
64	11.49	16.4	0.70
128	12.49	22.8	0.55
256	13.49	35.6	0.38

5.15.1 [10] <§5.7> What is the best page size if entries now become 128 bytes?

5.15.2 [10] <§5.7> Based on Exercise 5.15.1, what is the best page size if pages are half full?

5.15.3 [20] <§5.7> Based on Exercise 5.15.2, what is the best page size if using a modern disk with a 3 ms latency and 100 MB/s transfer rate? Explain why future servers are likely to have larger pages.

Keeping "frequently used" (or "hot") pages in DRAM can save disk accesses, but how do we determine the exact meaning of "frequently used" for a given system? Data engineers use the cost ratio between DRAM and disk access to quantify the reuse time threshold for hot pages. The cost of a disk access is $Disk/accesses_per_sec, while the cost to keep a page in DRAM is $DRAM_MiB/page_size. The typical DRAM and disk costs and typical database page sizes at several time points are listed below:

Year	DRAM Cost ($/MiB)	Page Size (KiB)	Disk Cost ($/disk)	Disk Access Rate (access/sec)
1987	5000	1	15,000	15
1997	15	8	2000	64
2007	0.05	64	80	83

5.15.4 [20] <§5.7> What other factors can be changed to keep using the same page size (thus avoiding software rewrite)? Discuss their likeliness with current technology and cost trends.

5.16 As described in Section 5.7, virtual memory uses a page table to track the mapping of virtual addresses to physical addresses. This exercise shows how this

table must be updated as addresses are accessed. The following data constitute a stream of virtual byte addresses as seen on a system. Assume 4 KiB pages, a four-entry fully associative TLB, and true LRU replacement. If pages must be brought in from disk, increment the next largest page number.

Decimal	4669	2227	13916	34587	48870	12608	49225
hex	0x123d	0x08b3	0x365c	0x871b	0xbee6	0x3140	0xc049

TLB

Valid	Tag	Physical Page Number	Time Since Last Access
1	0xb	12	4
1	0x7	4	1
1	0x3	6	3
0	0x4	9	7

Page table

Index	Valid	Physical Page or in Disk
0	1	5
1	0	Disk
2	0	Disk
3	1	6
4	1	9
5	1	11
6	0	Disk
7	1	4
8	0	Disk
9	0	Disk
a	1	3
b	1	12

5.16.1 [10] <§5.7> For each access shown above, list

- whether the access is a hit or miss in the TLB,
- whether the access is a hit or miss in the page table,
- whether the access is a page fault,
- the updated state of the TLB.

5.16.2 [15] <§5.7> Repeat Exercise 5.16.1, but this time use 16 KiB pages instead of 4 KiB pages. What would be some of the advantages of having a larger page size? What are some of the disadvantages?

5.16.3 [15] <§5.7> Repeat Exercise 5.16.1, but this time use 4 KiB pages and a two-way set associative TLB.

5.16.4 [15] <§5.7> Repeat Exercise 5.16.1, but this time use 4 KiB pages and a direct mapped TLB.

5.16.5 [10] <§§5.4, 5.7> Discuss why a CPU must have a TLB for high performance. How would virtual memory accesses be handled if there were no TLB?

5.17 There are several parameters that affect the overall size of the page table. Listed below are key page table parameters.

Virtual Address Size	Page Size	Page Table Entry Size
32 bits	8 KiB	4 bytes

5.17.1 [5] <§5.7> Given the parameters shown above, calculate the maximum possible page table size for a system running five processes.

5.17.2 [10] <§5.7> Given the parameters shown above, calculate the total page table size for a system running five applications that each utilize half of the virtual memory available, given a two-level page table approach with up to 256 entries at the 1st level. Assume each entry of the main page table is 6 bytes. Calculate the minimum and maximum amount of memory required for this page table.

5.17.3 [10] <§5.7> A cache designer wants to increase the size of a 4 KiB virtually indexed, physically tagged cache. Given the page size shown above, is it possible to make a 16 KiB direct-mapped cache, assuming two 64-bit words per block? How would the designer increase the data size of the cache?

5.18 In this exercise, we will examine space/time optimizations for page tables. The following list provides parameters of a virtual memory system.

Virtual Address (bits)	Physical DRAM Installed	Page Size	PTE Size (byte)
43	16 GiB	4 KiB	4

5.18.1 [10] <§5.7> For a single-level page table, how many *page table entries* (PTEs) are needed? How much physical memory is needed for storing the page table?

5.18.2 [10] <§5.7> Using a multi-level page table can reduce the physical memory consumption of page tables by only keeping active PTEs in physical memory. How many levels of page tables will be needed if the segment tables (the upper-level page tables) are allowed to be of unlimited size? How many memory references are needed for address translation if missing in TLB?

5.18.3 [10] <§5.7> Suppose the segments are limited to the 4 KiB page size (so that they can be paged). Is 4 bytes large enough for all page table entries (including those in the segment tables?

5.18.4 [10] <§5.7> How many levels of page tables are needed if the segments are limited to the 4 KiB page size?

5.18.5 [15] <§5.7> An inverted page table can be used to further optimize space and time. How many PTEs are needed to store the page table? Assuming a hash table implementation, what are the common case and worst case numbers of memory references needed for servicing a TLB miss?

5.19 The following table shows the contents of a four-entry TLB.

Entry-ID	Valid	VA Page	Modified	Protection	PA Page
1	1	140	1	RW	30
2	0	40	0	RX	34
3	1	200	1	RO	32
4	1	280	0	RW	31

5.19.1 [5] <§5.7> Under what scenarios would entry 3's valid bit be set to zero?

5.19.2 [5] <§5.7> What happens when an instruction writes to VA page 30? When would a software managed TLB be faster than a hardware managed TLB?

5.19.3 [5] <§5.7> What happens when an instruction writes to VA page 200?

5.20 In this exercise, we will examine how replacement policies affect miss rate. Assume a two-way set associative cache with four one-word blocks. Consider the following word address sequence: 0, 1, 2, 3, 4, 2, 3, 4, 5, 6, 7, 0, 1, 2, 3, 4, 5, 6, 7, 0.

Consider the following address sequence: 0, 2, 4, 8, 10, 12, 14, 16, 0

5.20.1 [5] <§§5.4, 5.8> Assuming an LRU replacement policy, which accesses are hits?

5.20.2 [5] <§§5.4, 5.8> Assuming an MRU (*most recently used*) replacement policy, which accesses are hits?

5.20.3 [5] <§§5.4, 5.8> Simulate a random replacement policy by flipping a coin. For example, "heads" means to evict the first block in a set and "tails" means to evict the second block in a set. How many hits does this address sequence exhibit?

5.20.4 [10] <§§5.4, 5.8> Describe an optimal replacement policy for this sequence. Which accesses are hits using this policy?

5.20.5 [10] <§§5.4, 5.8> Describe why it is difficult to implement a cache replacement policy that is optimal for all address sequences.

5.20.6 [10] <§§5.4, 5.8> Assume you could make a decision upon each memory reference whether or not you want the requested address to be cached. What effect could this have on miss rate?

5.21 One of the biggest impediments to widespread use of virtual machines is the performance overhead incurred by running a virtual machine. Listed below are various performance parameters and application behavior.

Base CPI	Privileged O/S accesses per 10,000 instructions	Overhead to trap to the guest O/S	Overhead to trap to VMM	I/O access per 10,000 instructions	I/O access time (includes time to trap to guest O/S)
1.5	120	15 cycles	175 cycles	30	1100 cycles

5.21.1 [10] <§5.6> Calculate the CPI for the system listed above assuming that there are no accesses to I/O. What is the CPI if the VMM overhead doubles? If it is cut in half? If a virtual machine software company wishes to limit the performance degradation to 10%, what is the longest possible penalty to trap to the VMM?

5.21.2 [15] <§5.6> I/O accesses often have a large effect on overall system performance. Calculate the CPI of a machine using the performance characteristics above, assuming a non-virtualized system. Calculate the CPI again, this time using a virtualized system. How do these CPIs change if the system has half the I/O accesses?

5.22 [15] <§§5.6, 5.7> Compare and contrast the ideas of virtual memory and virtual machines. How do the goals of each compare? What are the pros and cons of each? List a few cases where virtual memory is desired, and a few cases where virtual machines are desired.

5.23 [10] <§5.6> Section 5.6 discusses virtualization under the assumption that the virtualized system is running the same ISA as the underlying hardware. However, one possible use of virtualization is to emulate non-native ISAs. An example of this is QEMU, which emulates a variety of ISAs such as MIPS, SPARC, and PowerPC. What are some of the difficulties involved in this kind of virtualization? Is it possible for an emulated system to run faster than on its native ISA?

5.24 In this exercise, we will explore the control unit for a cache controller for a processor with a write buffer. Use the finite state machine found in Figure 5.39 as a starting point for designing your own finite state machines. Assume that the cache controller is for the simple direct-mapped cache described on page 453 (Figure 5.39 in Section 5.9), but you will add a write buffer with a capacity of one block.

Recall that the purpose of a write buffer is to serve as temporary storage so that the processor doesn't have to wait for two memory accesses on a dirty miss. Rather than writing back the dirty block before reading the new block, it buffers the dirty block and immediately begins reading the new block. The dirty block can then be written to main memory while the processor is working.

5.24.1 [10] <§§5.8, 5.9> What should happen if the processor issues a request that *hits* in the cache while a block is being written back to main memory from the write buffer?

5.24.2 [10] <§§5.8, 5.9> What should happen if the processor issues a request that *misses* in the cache while a block is being written back to main memory from the write buffer?

5.24.3 [30] <§§5.8, 5.9> Design a finite state machine to enable the use of a write buffer.

5.25 Cache coherence concerns the views of multiple processors on a given cache block. The following data show two processors and their read/write operations on two different words of a cache block X (initially X[0] = X[1] = 0).

P1	P2
`X[0] ++; X[1] = 3;`	`X[0] = 5; X[1] +=2;`

5.25.1 [15] <§5.10> List the possible values of the given cache block for a correct cache coherence protocol implementation. List at least one more possible value of the block if the protocol doesn't ensure cache coherency.

5.25.2 [15] <§5.10> For a snooping protocol, list a valid operation sequence on each processor/cache to finish the above read/write operations.

5.25.3 [10] <§5.10> What are the best-case and worst-case numbers of cache misses needed to execute the listed read/write instructions?

Memory consistency concerns the views of multiple data items. The following data show two processors and their read/write operations on different cache blocks (A and B initially 0).

P1	P2
`A = 1; B = 2; A+=2; B++;`	`C = B; D = A;`

5.25.4 [15] <§5.10> List the possible values of C and D for all implementations that ensure both consistency assumptions on page 455.

5.25.5 [15] <§5.10> List at least one more possible pair of values for C and D if such assumptions are not maintained.

5.25.6 [15] <§§5.3, 5.10> For various combinations of write policies and write allocation policies, which combinations make the protocol implementation simpler?

5.26 Chip multiprocessors (CMPs) have multiple cores and their caches on a single chip. CMP on-chip L2 cache design has interesting trade-offs. The following

table shows the miss rates and hit latencies for two benchmarks with private vs. shared L2 cache designs. Assume the L1 cache has a 3% miss rate and a 1-cycle access time.

	Private	Shared
Benchmark A miss rate	10%	4%
Benchmark B miss rate	2%	1%

Assume the following hit latencies:

Private Cache	Shared Cache	Memory
5	20	180

5.26.1 [15] <§5.13> Which cache design is better for each of these benchmarks? Use data to support your conclusion.

5.26.2 [15] <§5.13> Off-chip bandwidth becomes the bottleneck as the number of CMP cores increases. How does this bottleneck affect private and shared cache systems differently? Choose the best design if the latency of the first off-chip link doubles.

5.26.3 [10] <§5.13> Discuss the pros and cons of shared vs. private L2 caches for both single-threaded, multi-threaded, and multiprogrammed workloads, and reconsider them if having on-chip L3 caches.

5.26.4 [10] <§5.13> Would a non-blocking L2 cache produce more improvement on a CMP with a shared L2 cache or a private L2 cache? Why?

5.26.5 [10] <§5.13> Assume new generations of processors double the number of cores every 18 months. To maintain the same level of per-core performance, how much more off-chip memory bandwidth is needed for a processor released in three years?

5.26.6 [15] <§5.13> Consider the entire memory hierarchy. What kinds of optimizations can improve the number of concurrent misses?

5.27 In this exercise we show the definition of a web server log and examine code optimizations to improve log processing speed. The data structure for the log is defined as follows:

```
struct entry {
 int srcIP; // remote IP address
 char URL[128]; // request URL (e.g., "GET index.html")
 long long refTime; // reference time
 int status; // connection status
 char browser[64]; // client browser name
} log [NUM_ENTRIES];
```

Assume the following processing function for the log:

```
topK_sourceIP (int hour);
```

This function determines the most frequently observed source IPs during the given hour.

5.27.1 [5] <§5.15> Which fields in a log entry will be accessed for the given log processing function? Assuming 64-byte cache blocks and no prefetching, how many cache misses per entry does the given function incur on average?

5.27.2 [5] <§5.15> How can you reorganize the data structure to improve cache utilization and access locality?

5.27.3 [10] <§5.15> Give an example of another log processing function that would prefer a different data structure layout. If both functions are important, how would you rewrite the program to improve the overall performance? Supplement the discussion with code snippet and data.

5.28 For the problems below, use data from "Cache Performance for SPEC CPU2000 Benchmarks" (http://www.cs.wisc.edu/multifacet/misc/spec2000cache-data/) for the pairs of benchmarks shown in the following table.

a.	`Mesa/gcc`
b.	`mcf/swim`

5.28.1 [10] <§5.15> For 64 KiB data caches with varying set associativities, what are the miss rates broken down by miss types (cold, capacity, and conflict misses) for each benchmark?

5.28.2 [10] <§5.15> Select the set associativity to be used by a 64 KiB L1 data cache shared by both benchmarks. If the L1 cache has to be directly mapped, select the set associativity for the 1 MiB L2 cache.

5.28.3 [20] <§5.15> Give an example in the miss rate table where higher set associativity actually increases miss rate. Construct a cache configuration and reference stream to demonstrate this.

5.29 To support multiple virtual machines, two levels of memory virtualization are needed. Each virtual machine still controls the mapping of *virtual address* (VA) to *physical address* (PA), while the hypervisor maps the *physical address* (PA) of each virtual machine to the actual *machine address* (MA). To accelerate such mappings, a software approach called "shadow paging" duplicates each virtual machine's page tables in the hypervisor, and intercepts VA to PA mapping changes to keep both copies consistent. To remove the complexity of shadow page tables, a

hardware approach called *nested page table* (NPT) explicitly supports two classes of page tables (VA⇒PA and PA⇒MA) and can walk such tables purely in hardware.

Consider the following sequence of operations: (1) Create process; (2) TLB miss; (3) page fault; (4) context switch;

5.29.1 [10] <§§5.6, 5.7> What would happen for the given operation sequence for shadow page table and nested page table, respectively?

5.29.2 [10] <§§5.6, 5.7> Assuming an x86-based four-level page table in both guest and nested page table, how many memory references are needed to service a TLB miss for native vs. nested page table?

5.29.3 [15] <§§5.6, 5.7> Among TLB miss rate, TLB miss latency, page fault rate, and page fault handler latency, which metrics are more important for shadow page table? Which are important for nested page table?

Assume the following parameters for a shadow paging system.

TLB Misses per 1000 Instructions	NPT TLB Miss Latency	Page Faults per 1000 Instructions	Shadowing Page Fault Overhead
0.2	200 cycles	0.001	30,000 cycles

5.29.4 [10] <§5.6> For a benchmark with native execution CPI of 1, what are the CPI numbers if using shadow page tables vs. NPT (assuming only page table virtualization overhead)?

5.29.5 [10] <§5.6> What techniques can be used to reduce page table shadowing induced overhead?

5.29.6 [10] <§5.6> What techniques can be used to reduce NPT induced overhead?

Answers to Check Yourself

§5.1, page 369: 1 and 4. (3 is false because the cost of the memory hierarchy varies per computer, but in 2016 the highest cost is usually the DRAM.)
§5.3, page 390: 1 and 4: A lower miss penalty can enable smaller blocks, since you don't have that much latency to amortize, yet higher memory bandwidth usually leads to larger blocks, since the miss penalty is only slightly larger.
§5.4, page 409: 1.
§5.8, page 449: 2. (Both large block sizes and prefetching may reduce compulsory misses, so 1 is false.)

6

Parallel Processors from Client to Cloud

"I swing big, with everything I've got. I hit big or I miss big. I like to live as big as I can."

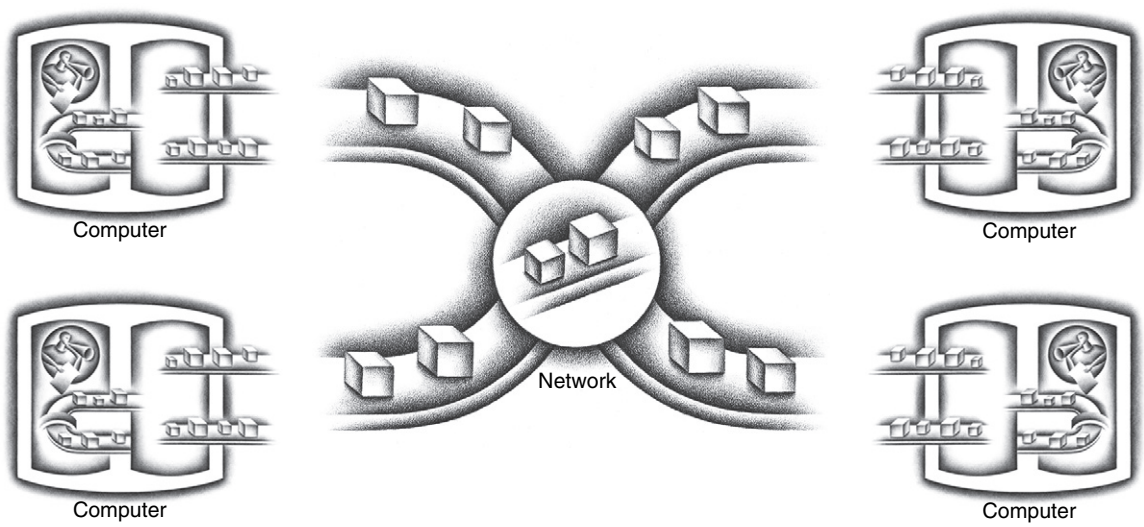
Babe Ruth
American baseball player

6.1 **Introduction** 492
6.2 **The Difficulty of Creating Parallel Processing Programs** 494
6.3 **SISD, MIMD, SIMD, SPMD, and Vector** 499
6.4 **Hardware Multithreading** 506
6.5 **Multicore and Other Shared Memory Multiprocessors** 509
6.6 **Introduction to Graphics Processing Units** 514

Computer Organization and Design. DOI: http://dx.doi.org/10.1016/B978-0-12-812275-4.00006-3
© 2018 Elsevier Inc. All rights reserved.

6.7 **Clusters, Warehouse Scale Computers, and Other Message-Passing Multiprocessors** 521
6.8 **Introduction to Multiprocessor Network Topologies** 526
6.9 **Communicating to the Outside World: Cluster Networking** 529
6.10 **Multiprocessor Benchmarks and Performance Models** 530
6.11 **Real Stuff: Benchmarking and Rooflines of the Intel Core i7 960 and the NVIDIA Tesla GPU** 540
6.12 **Going Faster: Multiple Processors and Matrix Multiply** 545
6.13 **Fallacies and Pitfalls** 548
6.14 **Concluding Remarks** 550
6.15 **Historical Perspective and Further Reading** 553
6.16 **Exercises** 553

Multiprocessor or Cluster Organization



6.1 Introduction

Over the Mountains Of the Moon, Down the Valley of the Shadow, Ride, boldly ride the shade replied—If you seek for El Dorado!

Edgar Allan Poe, "El Dorado," stanza 4, 1849

Computer architects have long sought the "The City of Gold" (El Dorado) of computer design: to create powerful computers simply by connecting many existing smaller ones. This golden vision is the fountainhead of **multiprocessors**. Ideally, customers order as many processors as they can afford and receive a commensurate amount of performance. Thus, multiprocessor software must be designed to work with a variable number of processors. As mentioned in Chapter 1, energy has become the overriding issue for both microprocessors and datacenters. Replacing large inefficient processors with many smaller, efficient processors can deliver better performance per joule both in the large and in the small, if software can efficiently use them. Therefore, improved energy efficiency joins scalable performance in the case for multiprocessors.

multiprocessor A computer system with at least two processors. This computer is in contrast to a uniprocessor, which has one, and is increasingly hard to find today.

Since multiprocessor software should scale, some designs support operation in the presence of broken hardware; that is, if a single processor fails in a multiprocessor with n processors, these systems would continue to provide service with $n - 1$ processors. Hence, multiprocessors can also improve availability (see Chapter 5).

High performance can mean greater throughput for independent tasks, called **task-level parallelism** or **process-level parallelism**. These tasks are independent single-threaded applications, and they are an important and popular use of multiple processors. This approach contrasts with running a single job on multiple processors. We use the term **parallel processing program** to refer to a single program that runs on multiple processors simultaneously.



PARALLELISM

task-level parallelism or **process-level parallelism** Utilizing multiple processors by running independent programs simultaneously.

parallel processing program A single program that runs on multiple processors simultaneously.

There have long been scientific problems that have needed much faster computers, and this class of problems has been used to justify many novel parallel computers over the decades. Some of these problems can be handled simply today, using a **cluster** composed of microprocessors housed in many independent servers (see Section 6.7). In addition, clusters can serve equally demanding applications outside the sciences, such as search engines, Web servers, email servers, and databases.

cluster A set of computers connected over a local area network that function as a single large multiprocessor.

As described in Chapter 1, multiprocessors have been shoved into the spotlight because the energy problem means that future increases in performance will primarily come from explicit hardware parallelism rather than much higher clock rates or vastly improved CPI. As we said in Chapter 1, they are called

multicore microprocessors instead of multiprocessor microprocessors, presumably to avoid redundancy in naming. Hence, processors are often called *cores* in a multicore chip. The number of cores is expected to increase with **Moore's Law**. These multicores are almost always **Shared Memory Processors (SMPs)**, as they usually share a single physical address space. We'll see SMPs more in Section 6.5.

multicore microprocessor A microprocessor containing multiple processors ("cores") in a single integrated circuit. Virtually all microprocessors today in desktops and servers are multicore.

shared memory multiprocessor (SMP) A parallel processor with a single physical address space.

The state of technology today means that programmers who care about performance must become parallel programmers, for sequential code now means slow code.

The tall challenge facing the industry is to create hardware and software that will make it easy to write correct parallel processing programs that will execute efficiently in performance and energy as the number of cores per chip scales.

This abrupt shift in microprocessor design caught many off guard, so there is a great deal of confusion about the terminology and what it means. Figure 6.1 tries to clarify the terms serial, parallel, sequential, and concurrent. The columns of this figure represent the software, which is either inherently sequential or concurrent. The rows of the figure represent the hardware, which is either serial or parallel. For example, the programmers of compilers think of them as sequential programs: the steps include parsing, code generation, optimization, and so on. In contrast, the programmers of operating systems normally think of them as concurrent programs: cooperating processes handling I/O events due to independent jobs running on a computer.



The point of these two axes of Figure 6.1 is that concurrent software can run on serial hardware, such as operating systems for the Intel Pentium 4 uniprocessor, or on parallel hardware, such as an OS on the more recent Intel Core i7. The same is true for sequential software. For example, the MATLAB programmer writes a matrix multiply thinking about it sequentially, but it could run serially on the Pentium 4 or in parallel on the Intel Core i7.

You might guess that the only challenge of the parallel revolution is figuring out how to make naturally sequential software have high performance on parallel hardware, but it is also to make concurrent programs have high performance on multiprocessors as the number of processors increases. With this distinction made, in the rest of this chapter we will use *parallel processing program* or *parallel software* to mean either sequential or concurrent software running on parallel hardware. The next section of this chapter describes why it is hard to create efficient parallel processing programs.

		Software	
		Sequential	**Concurrent**
Hardware	Serial	Matrix Multiply written in MatLab running on an Intel Pentium 4	Windows Vista Operating System running on an Intel Pentium 4
	Parallel	Matrix Multiply written in MATLAB running on an Intel Core i7	Windows Vista Operating System running on an Intel Core i7

FIGURE 6.1 Hardware/software categorization and examples of application perspective on concurrency versus hardware perspective on parallelism.

Before proceeding further down the path to parallelism, don't forget our initial incursions from the earlier chapters:

- Chapter 2, Section 2.11: Parallelism and Instructions: Synchronization
- Chapter 3, Section 3.6: Parallelism and Computer Arithmetic: Subword Parallelism
- Chapter 4, Section 4.10: Parallelism via Instructions
- Chapter 5, Section 5.10: Parallelism and Memory Hierarchy: Cache Coherence

Check Yourself

True or false: To benefit from a multiprocessor, an application must be concurrent.

6.2 The Difficulty of Creating Parallel Processing Programs

The difficulty with parallelism is not the hardware; it is that too few important application programs have been rewritten to complete tasks sooner on multiprocessors. It is difficult to write software that uses multiple processors to complete one task faster, and the problem gets worse as the number of processors increases.

Why has this been so? Why have parallel processing programs been so much harder to develop than sequential programs?

The first reason is that you *must* get better performance or better energy efficiency from a parallel processing program on a multiprocessor; otherwise, you would just use a sequential program on a uniprocessor, as sequential programming is simpler. In fact, uniprocessor design techniques, such as superscalar and out-of-order execution, take advantage of instruction-level parallelism (see Chapter 4), normally without the involvement of the programmer. Such innovations reduced the demand for rewriting programs for multiprocessors, since programmers could do nothing and yet their sequential programs would run faster on new computers.

Why is it difficult to write parallel processing programs that are fast, especially as the number of processors increases? In Chapter 1, we used the analogy of eight reporters trying to write a single story in hopes of doing the work eight times faster. To succeed, the task must be broken into eight equal-sized pieces, because otherwise some reporters would be idle while waiting for the ones with larger pieces to finish. Another speed-up obstacle could be that the reporters would spend too much time communicating with each other instead of writing their pieces of the story. For both this analogy and parallel programming, the challenges include scheduling, partitioning the work into parallel pieces, balancing the load evenly between the workers, time to synchronize, and overhead for communication between the

parties. The challenge is stiffer with the more reporters for a newspaper story and with the more processors for parallel programming.

Our discussion in Chapter 1 reveals another obstacle, namely Amdahl's Law. It reminds us that even small parts of a program must be parallelized if the program is to make good use of many cores.

Speed-up Challenge

EXAMPLE

Suppose you want to achieve a speed-up of 90 times faster with 100 processors. What percentage of the original computation can be sequential?

ANSWER

Amdahl's Law (Chapter 1) says

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

We can reformulate Amdahl's Law in terms of speed-up versus the initial execution time:

$$\text{Speed-up} = \frac{\text{Execution time before}}{(\text{Execution time before} - \text{Execution time affected}) + \dfrac{\text{Execution time affected}}{\text{Amount of improvement}}}$$

This formula is usually rewritten assuming that the execution time before is 1 for some unit of time, and the execution time affected by improvement is considered the fraction of the original execution time:

$$\text{Speed-up} = \frac{1}{(1 - \text{Fraction time affected}) + \dfrac{\text{Fraction time affected}}{\text{Amount of improvement}}}$$

Substituting 90 for speed-up and 100 for the amount of improvement into the formula above:

$$90 = \frac{1}{(1 - \text{Fraction time affected}) + \dfrac{\text{Fraction time affected}}{100}}$$

Then simplifying the formula and solving for fraction time affected:

$$\begin{aligned} 90 \times (1 - 0.99 \times \text{Fraction time affected}) &= 1 \\ 90 - (90 \times 0.99 \times \text{Fraction time affected}) &= 1 \\ 90 - 1 &= 90 \times 0.99 \times \text{Fraction time affected} \\ \text{Fraction time affected} &= 89/89.1 = 0.999 \end{aligned}$$

Thus, to achieve a speed-up of 90 from 100 processors, the sequential percentage can only be 0.1%.

However, there are applications with plenty of parallelism, as we shall see next.

EXAMPLE

Speed-up Challenge: Bigger Problem

Suppose you want to perform two sums: one is a sum of 10 scalar variables, and one is a matrix sum of a pair of two-dimensional arrays, with dimensions 10 by 10. For now let's assume only the matrix sum is parallelizable; we'll see soon how to parallelize scalar sums. What speed-up do you get with 10 versus 40 processors? Next, calculate the speed-ups assuming the matrices grow to 20 by 20.

ANSWER

If we assume performance is a function of the time for an addition, t, then there are 10 additions that do not benefit from parallel processors and 100 additions that do. If the time for a single processor is 110 t, the execution time for 10 processors is

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

$$\text{Execution time after improvement} = \frac{100t}{10} + 10t = 20t$$

so the speed-up with 10 processors is $110t/20t = 5.5$. The execution time for 40 processors is

$$\text{Execution time after improvement} = \frac{100t}{40} + 10t = 12.5t$$

so the speed-up with 40 processors is $110t/12.5t = 8.8$. Thus, for this problem size, we get about 55% of the potential speed-up with 10 processors, but only 22% with 40.

Look what happens when we increase the matrix. The sequential program now takes $10t + 400t = 410t$. The execution time for 10 processors is

$$\text{Execution time after improvement} = \frac{400t}{10} + 10t = 50t$$

so the speed-up with 10 processors is $410t/50t = 8.2$. The execution time for 40 processors is

$$\text{Execution time after improvement} = \frac{400t}{40} + 10t = 20t$$

so the speed-up with 40 processors is $410t/20t = 20.5$. Thus, for this larger problem size, we get 82% of the potential speed-up with 10 processors and 51% with 40.

These examples show that getting good speed-up on a multiprocessor while keeping the problem size fixed is harder than getting good speed-up by increasing the size of the problem. This insight allows us to introduce two terms that describe ways to scale up.

Strong scaling means measuring speed-up while keeping the problem size fixed. **Weak scaling** means that the problem size grows proportionally to the increase in the number of processors. Let's assume that the size of the problem, M, is the working set in main memory, and we have P processors. Then the memory per processor for strong scaling is approximately M/P, and for weak scaling, it is about M.

strong scaling Speed-up achieved on a multiprocessor without increasing the size of the problem.

weak scaling Speed-up achieved on a multiprocessor while increasing the size of the problem proportionally to the increase in the number of processors.

Note that the **memory hierarchy** can interfere with the conventional wisdom about weak scaling being easier than strong scaling. For example, if the weakly scaled dataset no longer fits in the last level cache of a multicore microprocessor, the resulting performance could be much worse than by using strong scaling.

Depending on the application, you can argue for either scaling approach. For example, the TPC-C debit-credit database benchmark requires that you scale up the number of customer accounts in proportion to the higher transactions per minute. The argument is that it's nonsensical to think that a given customer base is suddenly going to start using ATMs 100 times a day just because the bank gets a faster computer. Instead, if you're going to demonstrate a system that can perform 100 times the numbers of transactions per minute, you should run the experiment with 100 times as many customers. Bigger problems often need more data, which is an argument for weak scaling.

This final example shows the importance of load balancing.



EXAMPLE

Speed-up Challenge: Balancing Load

To achieve the speed-up of 20.5 on the previous larger problem with 40 processors, we assumed the load was perfectly balanced. That is, each of the 40

processors had 2.5% of the work to do. Instead, show the impact on speed-up if one processor's load is higher than all the rest. Calculate at twice the load (5%) and five times the load (12.5%) for that hardest working processor. How well utilized are the rest of the processors?

ANSWER

If one processor has 5% of the parallel load, then it must do 5% × 400 or 20 additions, and the other 39 will share the remaining 380. Since they are operating simultaneously, we can just calculate the execution time as a maximum

$$\text{Execution time after improvement} = \text{Max}\left(\frac{380t}{39}, \frac{20t}{1}\right) + 10t = 30t$$

The speed-up drops from 20.5 to $410t/30t = 14$. The remaining 39 processors are utilized less than half the time: while waiting $20t$ for the hardest working processor to finish, they only compute for $380t/39 = 9.7t$.

If one processor has 12.5% of the load, it must perform 50 additions. The formula is:

$$\text{Execution time after improvement} = \text{Max}\left(\frac{350t}{39}, \frac{50t}{1}\right) + 10t = 60t$$

The speed-up drops even further to $410t/60t = 7$. The rest of the processors are utilized less than 20% of the time ($9t/50t$). This example demonstrates the importance of balancing load, for just a single processor with twice the load of the others cuts speed-up by a third, and five times the load on just one processor reduces speed-up by almost a factor of three.

Now that we better understand the goals and challenges of parallel processing, we give an overview of the rest of the chapter. Section 6.3 describes a much older classification scheme than in Figure 6.1. In addition, it describes two styles of instruction set architectures that support running of sequential applications on parallel hardware, namely *SIMD* and *vector*. Section 6.4 then describes *multithreading*, a term often confused with multiprocessing, in part because it relies upon similar concurrency in programs. Section 6.5 describes the first the two alternatives of a fundamental parallel hardware characteristic, which is whether or not all the processors in the systems rely upon a single physical address space. As mentioned above, the two popular versions of these alternatives are called *shared memory multiprocessors* (SMPs) and *clusters*, and this section covers the former. Section 6.6 describes a relatively new style of computer from the graphics hardware community, called a *graphics-processing unit* (GPU) that also assumes a single physical address. (🌐 **Appendix B** describes GPUs in even more detail.) Section 6.7 describes clusters, a popular example of a computer with multiple physical address spaces. Section 6.8 shows typical topologies used to connect many processors together, either server nodes in a cluster or cores in a microprocessor. 🌐 **Section 6.9** describes the hardware and software for communicating between

nodes in a cluster using Ethernet. It shows how to optimize its performance using custom software and hardware. We next discuss the difficulty of finding parallel benchmarks in Section 6.10. This section also includes a simple, yet insightful performance model that helps in the design of applications as well as architectures. We use this model as well as parallel benchmarks in Section 6.11 to compare a multicore computer to a GPU. Section 6.12 divulges the final and largest step in our journey of accelerating matrix multiply. For matrices that don't fit in the cache, parallel processing uses 16 cores to improve performance by a factor of 14. We close with fallacies and pitfalls and our conclusions for parallelism.

In the next section, we introduce acronyms that you probably have already seen to identify different types of parallel computers.

Check Yourself

True or false: Strong scaling is not bound by Amdahl's Law.

6.3 SISD, MIMD, SIMD, SPMD, and Vector

One categorization of parallel hardware proposed in the 1960s is still used today. It was based on the number of instruction streams and the number of data streams. Figure 6.2 shows the categories. Thus, a conventional uniprocessor has a single instruction stream and single data stream, and a conventional multiprocessor has multiple instruction streams and multiple data streams. These two categories are abbreviated **SISD** and **MIMD**, respectively.

SISD or Single Instruction stream, Single Data stream. A uniprocessor.

MIMD or Multiple Instruction streams, Multiple Data streams. A multiprocessor.

While it is possible to write separate programs that run on different processors on a MIMD computer and yet work together for a grander, coordinated goal, programmers normally write a single program that runs on all processors of a **MIMD** computer, relying on conditional statements when different processors should execute distinct sections of code. This style is called **Single Program Multiple Data (SPMD)**, but it is just the normal way to program a MIMD computer.

SPMD Single Program, Multiple Data streams. The conventional MIMD programming model, where a single program runs across all processors.

The closest we can come to multiple instruction streams and single data stream **(MISD)** processor might be a "stream processor" that would perform a series of computations on a single data stream in a pipelined fashion: parse the input from the network, decrypt the data, decompress it, search for match, and so on. The inverse of MISD is much more popular. **SIMD** computers operate on vectors of

SIMD or Single Instruction stream, Multiple Data streams. The same instruction is applied to many data streams, as in a vector processor.

		Data Streams	
		Single	Multiple
Instruction Streams	Single	SISD: Intel Pentium 4	SIMD: SSE instructions of x86
	Multiple	MISD: No examples today	MIMD: Intel Core i7

FIGURE 6.2 Hardware categorization and examples based on number of instruction streams and data streams: SISD, SIMD, MISD, and MIMD.

data. For example, a single SIMD instruction might add 64 numbers by sending 64 data streams to 64 ALUs to form 64 sums within a single clock cycle. The subword parallel instructions that we saw in Sections 3.6 and 3.7 are another example of SIMD; indeed, the middle letter of Intel's SSE acronym stands for SIMD.

The virtues of SIMD are that all the parallel execution units are synchronized, and they all respond to a single instruction that emanates from a single *program counter* (PC). From a programmer's perspective, this is close to the already familiar SISD. Although every unit will be executing the same instruction, each execution unit has its own address registers, and so each unit can have different data addresses. Thus, in terms of Figure 6.1, a sequential application might be compiled to run on serial hardware organized as a SISD or in parallel hardware that was organized as a SIMD.

The original motivation behind SIMD was to amortize the cost of the control unit over dozens of execution units. Another advantage is the reduced instruction bandwidth and space—SIMD needs only one copy of the code that is being simultaneously executed, while message-passing MIMDs may need a copy in every processor and shared memory MIMD will need multiple instruction caches.

data-level parallelism Parallelism achieved by performing the same operation on independent data.

SIMD works best when dealing with arrays in `for` loops. Hence, for parallelism to work in SIMD, there must be a great deal of identically structured data, which is called **data-level parallelism**. SIMD is at its weakest in `case` or `switch` statements, where each execution unit must perform a different operation on its data, depending on what data it has. Execution units with the wrong data must be disabled so that units with proper data may continue. If there are n cases, in these situations, SIMD processors essentially run at 1/nth of peak performance.

The so-called array processors that inspired the SIMD category have faded into history (see **Section 6.15** online), but two current interpretations of SIMD remain active today.

SIMD in x86: Multimedia Extensions

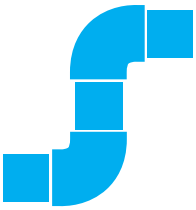


As described in Chapter 3, subword parallelism for narrow integer data was the original inspiration of the *Multimedia Extension* (MMX) instructions of the x86 in 1996. As **Moore's Law** continued, more instructions were added, leading first to *Streaming SIMD Extensions* (SSE) and now *Advanced Vector Extensions* (AVX). AVX supports the simultaneous execution of four 64-bit floating-point numbers. The width of the operation and the registers is encoded in the opcode of these multimedia instructions. As the data width of the registers and operations grew, the number of opcodes for multimedia instructions exploded, and now there are hundreds of SSE and AVX instructions (see Chapter 3).

Vector

An older and, as we shall see, more elegant interpretation of SIMD is called a *vector architecture*, which has been closely identified with computers designed by Seymour Cray starting in the 1970s. It is also a great match to problems with lots of data-level parallelism. Rather than having 64 ALUs perform 64 additions simultaneously, like the old array processors, the vector architectures pipelined the ALU to get good performance at lower cost. The basic philosophy of vector architecture is to collect

data elements from memory, put them in order into a large set of registers, operate on them sequentially in registers using **pipelined execution units**, and then write the results back to memory. A key feature of vector architectures is therefore a set of vector registers. Thus, a vector architecture might have 32 vector registers, each with 64 64-bit elements.



PIPELINING

Comparing Vector to Conventional Code

EXAMPLE

Suppose we extend the RISC-V instruction set architecture with vector instructions and vector registers. Vector operations use the same names as RISC-V operations, but with the suffix "V" appended. For example, `fadd.d.v` adds two double-precision vectors. Let's also add 32 vector registers, v0—v31, each with sixty-four 64-bit elements. The vector instructions take as their input either a pair of vector (V) registers (`fadd.d.v`) or a vector register and a scalar register (`fadd.d.vs`). In the latter case, the value in the scalar register is used as the input for all operations—the operation `fadd.d.vs` will add the contents of a scalar register to each element in a vector register. The names `fld.v` and `fsd.v` denote vector load and vector store, and they load or store an entire vector of double-precision data. One operand is the vector register to be loaded or stored; the other operand, which is a RISC-V general-purpose register, is the starting address of the vector in memory.

Given this short description, show the conventional RISC-V code versus the vector RISC-V code for

$$Y = a \times X + Y$$

where X and Y are vectors of 64 double precision floating-point numbers, initially resident in memory, and a is a scalar double precision variable. (This example is the so-called DAXPY loop that forms the inner loop of the Linpack benchmark; DAXPY stands for double precision $a \times X$ plus Y.) Assume that the starting addresses of X and Y are in `x19` and `x20`, respectively.

ANSWER

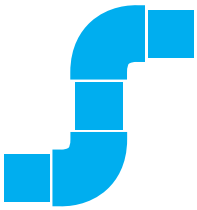
Here is the conventional RISC-V code for DAXPY:

```
       fld     f0, a(x3)      // load scalar a
       addi    x5, x19, 512   // end of array X
loop:  fld     f1, 0(x19)     // load x[i]
       fmul.d  f1, f1, f0     // a * x[i]
       fld     f2, 0(x20)     // load y[i]
       fadd.d  f2, f2, f1     // a * x[i] + y[i]
       fsd     f2, 0(x20)     // store y[i]
       addi    x19, x19, 8    // increment index to x
       addi    x20, x20, 8    // increment index to y
       bltu    x19, x5, loop  // repeat if not done
```

Here is the hypothetical RISC-V vector code for DAXPY:

```
fld        f0, a(x3)        // load scalar a
fld.v      v0, 0(x19)       // load vector x
fmul.d.vs  v0, v0, f0       // vector-scalar multiply
fld.v      v1, 0(x20)       // load vector y
fadd.d.v   v1, v1, v0       // vector-vector add
fsd.v      v1, 0(x20)       // store vector y
```

There are some interesting comparisons between the two code segments in this example. The most dramatic is that the vector processor greatly reduces the dynamic instruction bandwidth, executing only six instructions versus over 500 for the basline RISC-V architecture. This reduction occurs both because the vector operations work on 64 elements at a time and because the overhead instructions that constitute nearly half the loop on RISC-V are not present in the vector code. As you might expect, this reduction in instructions fetched and executed saves energy.



Another important difference is the frequency of **pipeline** hazards (Chapter 4). In the straightforward RISC-V code, every `fadd.d` must wait for the `fmul.d`, every `fsd` must wait for the `fadd.d`, and every `fadd.d` *and* `fmul.d` must wait on `fld`. On the vector processor, each vector instruction will only stall for the first element in each vector, and then subsequent elements will flow smoothly down the pipeline. Thus, pipeline stalls are required only once per vector *operation*, rather than once per vector *element*. In this example, the pipeline stall frequency on RISC-V will be about 64 times higher than it is on the vector version of RISC-V. The pipeline stalls can be eliminated on RISC-V by unrolling the loop (see Chapter 4). However, the large difference in instruction bandwidth cannot be reduced.

Since the vector elements are independent, they can be operated on in parallel, much like subword parallelism for the Intel x86 AVX instructions. All modern vector computers have vector functional units with multiple parallel pipelines (called *vector lanes*; see Figures 6.2 and 6.3) that can produce two or more results per clock cycle.

Elaboration: The loop in the example above exactly matched the vector length. When loops are shorter, vector architectures use a register that reduces the length of vector operations. When loops are larger, we add bookkeeping code to iterate full-length vector operations and to handle the leftovers. This latter process is called *strip mining*.

Vector versus Scalar

Vector instructions have several important properties compared to conventional instruction set architectures, which are called *scalar architectures* in this context:

- A single vector instruction specifies a great deal of work—it is equivalent to executing an entire loop. The instruction fetch and decode bandwidth needed is dramatically reduced.
- By using a vector instruction, the compiler or programmer indicates that the computation of each result in the vector is independent of the computation of other results in the same vector, so hardware does not have to check for data hazards within a vector instruction.

- Vector architectures and compilers have a reputation for making it much easier than when using MIMD multiprocessors to write efficient applications when they contain data-level parallelism.
- Hardware need only check for data hazards between two vector instructions once per vector operand, not once for every element within the vectors. Reduced checking can save energy as well as time.
- Vector instructions that access memory have a known access pattern. If the vector's elements are all adjacent, then fetching the vector from a set of heavily interleaved memory banks works very well. Thus, the cost of the latency to main memory is seen only once for the entire vector, rather than once for each word of the vector.
- Because a complete loop is replaced by a vector instruction whose behavior is predetermined, control hazards that would normally arise from the loop branch are nonexistent.
- The savings in instruction bandwidth and hazard checking plus the efficient use of memory bandwidth give vector architectures advantages in power and energy versus scalar architectures.

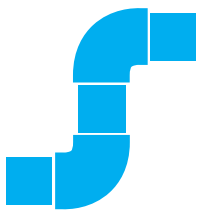
For these reasons, vector operations can be made faster than a sequence of scalar operations on the same number of data items, and designers are motivated to include vector units if the application domain can often use them.

Vector versus Multimedia Extensions

Like multimedia extensions found in the x86 AVX instructions, a vector instruction specifies multiple operations. However, multimedia extensions typically denote a few operations while vector specifies dozens of operations. Unlike multimedia extensions, the number of elements in a vector operation is not in the opcode but in a separate register. This distinction means different versions of the vector architecture can be implemented with a different number of elements just by changing the contents of that register and hence retain binary compatibility. In contrast, a new large set of opcodes is added each time the "vector" length changes in the multimedia extension architecture of the x86: MMX, SSE, SSE2, AVX, AVX2, ….

Also, unlike multimedia extensions, the data transfers need not be contiguous. Vectors support both strided accesses, where the hardware loads every *n*th data element in memory, and indexed accesses, where hardware finds the addresses of the items to be loaded into a vector register. Indexed accesses are also called *gather-scatter*, in that indexed loads gather elements from main memory into contiguous vector elements, and indexed stores scatter vector elements across main memory.

Like multimedia extensions, vector architectures easily capture the flexibility in data widths, so it is easy to make a vector operation work on 32 64-bit data elements or 64 32-bit data elements or 128 16-bit data elements or 256 8-bit data elements. The parallel semantics of a vector instruction allows an implementation to execute these operations using a deeply **pipelined** functional unit, an array of parallel functional units, or a combination of parallel and pipelined functional units. Figure 6.3 illustrates how to improve vector performance by using parallel pipelines to execute a vector add instruction.



PIPELINING

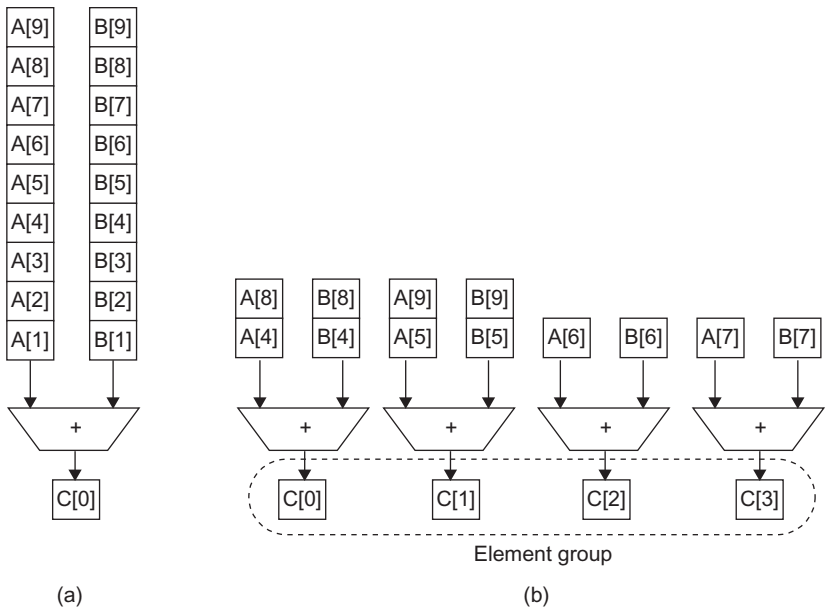


FIGURE 6.3 Using multiple functional units to improve the performance of a single vector add instruction, C = A + B. The vector processor (a) on the left has a single add pipeline and can complete one addition per cycle. The vector processor (b) on the right has four add pipelines or lanes and can complete four additions per cycle. The elements within a single vector add instruction are interleaved across the four lanes.

vector lane One or more vector functional units and a portion of the vector register file. Inspired by lanes on highways that increase traffic speed, multiple lanes execute vector operations simultaneously.

Vector arithmetic instructions usually only allow element N of one vector register to take part in operations with element N from other vector registers. This dramatically simplifies the construction of a highly parallel vector unit, which can be structured as multiple parallel **vector lanes**. As with a traffic highway, we can increase the peak throughput of a vector unit by adding more lanes. Figure 6.4 shows the structure of a four-lane vector unit. Thus, going to four lanes from one lane reduces the number of clocks per vector instruction by roughly a factor of four. For multiple lanes to be advantageous, both the applications and the architecture must support long vectors. Otherwise, they will execute so quickly that you'll run out of instructions, requiring instruction level **parallel** techniques like those in Chapter 4 to supply enough vector instructions.



Generally, vector architectures are a very efficient way to execute data parallel processing programs; they are better matches to compiler technology than multimedia extensions; and they are easier to evolve over time than the multimedia extensions to the x86 architecture.

Given these classic categories, we next see how to exploit parallel streams of instructions to improve the performance of a *single* processor, which we will reuse with multiple processors.

Check Yourself

True or false: As exemplified in the x86, multimedia extensions can be thought of as a vector architecture with short vectors that support only contiguous vector data transfers.

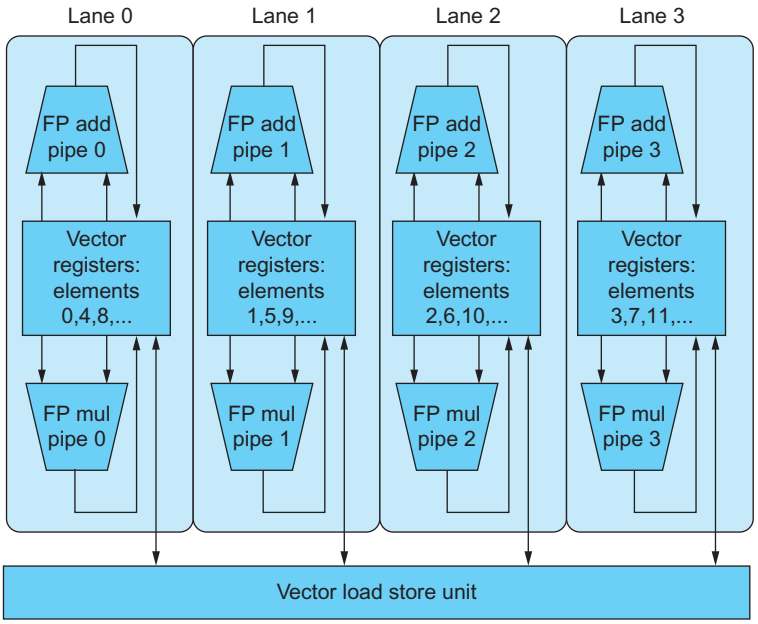


FIGURE 6.4 Structure of a vector unit containing four lanes. The vector-register storage is divided across the lanes, with each lane holding every fourth element of each vector register. The figure shows three vector functional units: an FP add, an FP multiply, and a load-store unit. Each of the vector arithmetic units contains four execution pipelines, one per lane, which acts in concert to complete a single vector instruction. Note how each section of the vector-register file only needs to provide enough read and write ports (see Chapter 4) for functional units local to its lane.

Elaboration: Given the advantages of vector, why aren't they more popular outside high-performance computing? There were concerns about the larger state for vector registers increasing context switch time and the difficulty of handling page faults in vector loads and stores, and SIMD instructions achieved some of the benefits of vector instructions. In addition, as long as advances in instruction-level parallelism could deliver on the performance promise of **Moore's Law**, there was little reason to take the chance of changing architecture styles.



Elaboration: Another advantage of vector and multimedia extensions is that it is relatively easy to extend a scalar instruction set architecture with these instructions to improve performance of data parallel operations.

Elaboration: The Haswell-generation x86 processors from Intel support AVX2, which has a gather operation but not a scatter operation.

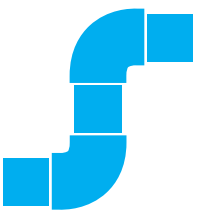
hardware multithreading Increasing utilization of a processor by switching to another thread when one thread is stalled.

thread A thread includes the program counter, the register state, and the stack. It is a lightweight process; whereas threads commonly share a single address space, processes don't.

process A process includes one or more threads, the address space, and the operating system state. Hence, a process switch usually invokes the operating system, but not a thread switch.

fine-grained multithreading A version of hardware multithreading that implies switching between threads after every instruction.

coarse-grained multithreading A version of hardware multithreading that implies switching between threads only after significant events, such as a last-level cache miss.



PIPELINING

6.4 Hardware Multithreading

A related concept to MIMD, especially from the programmer's perspective, is **hardware multithreading**. While MIMD relies on multiple **processes** or **threads** to try to keep many processors busy, hardware multithreading allows multiple threads to share the functional units of a *single* processor in an overlapping fashion to try to utilize the hardware resources efficiently. To permit this sharing, the processor must duplicate the independent state of each thread. For example, each thread would have a separate copy of the register file and the program counter. The memory itself can be shared through the virtual memory mechanisms, which already support multi-programming. In addition, the hardware must support the ability to change to a different thread relatively quickly. In particular, a thread switch should be much more efficient than a process switch, which typically requires hundreds to thousands of processor cycles while a thread switch can be instantaneous.

There are two main approaches to hardware multithreading. **Fine-grained multithreading** switches between threads on each instruction, resulting in interleaved execution of multiple threads. This interleaving is often done in a round-robin fashion, skipping any threads that are stalled at that clock cycle. To make fine-grained multithreading practical, the processor must be able to switch threads on every clock cycle. One advantage of fine-grained multithreading is that it can hide the throughput losses that arise from both short and long stalls, since instructions from other threads can be executed when one thread stalls. The primary disadvantage of fine-grained multithreading is that it slows down the execution of the individual threads, since a thread that is ready to execute without stalls will be delayed by instructions from other threads.

Coarse-grained multithreading was invented as an alternative to fine-grained multithreading. Coarse-grained multithreading switches threads only on expensive stalls, such as last-level cache misses. This change relieves the need to have thread switching be extremely fast and is much less likely to slow down the execution of an individual thread, since instructions from other threads will only be issued when a thread encounters a costly stall. Coarse-grained multithreading suffers, however, from a major drawback: it is limited in its ability to overcome throughput losses, especially from shorter stalls. This limitation arises from the **pipeline** start-up costs of coarse-grained multithreading. Because a processor with coarse-grained multithreading issues instructions from a single thread, when a stall occurs, the pipeline must be emptied or frozen. The new thread that begins executing after the stall must fill the pipeline before instructions are able to complete. Due to this start-up overhead, coarse-grained multithreading is much more useful for reducing the penalty of high-cost stalls, where pipeline refill is negligible compared to the stall time.

Simultaneous multithreading (SMT) is a variation on hardware multithreading that uses the resources of a multiple-issue, dynamically scheduled **pipelined** processor to exploit thread-level parallelism at the same time it exploits instruction-level parallelism (see Chapter 4). The key insight that motivates SMT is that multiple-issue processors often have more functional unit parallelism available than most single threads can effectively use. Furthermore, with register renaming and dynamic scheduling (see Chapter 4), multiple instructions from independent threads can be issued without regard to the dependences among them; the resolution of the dependences can be handled by the dynamic scheduling capability.

Since SMT relies on the existing dynamic mechanisms, it does not switch resources every cycle. Instead, SMT is *always* executing instructions from multiple threads, leaving it up to the hardware to associate instruction slots and renamed registers with their proper threads.

Figure 6.5 conceptually illustrates the differences in a processor's ability to exploit superscalar resources for the following processor configurations. The top portion shows

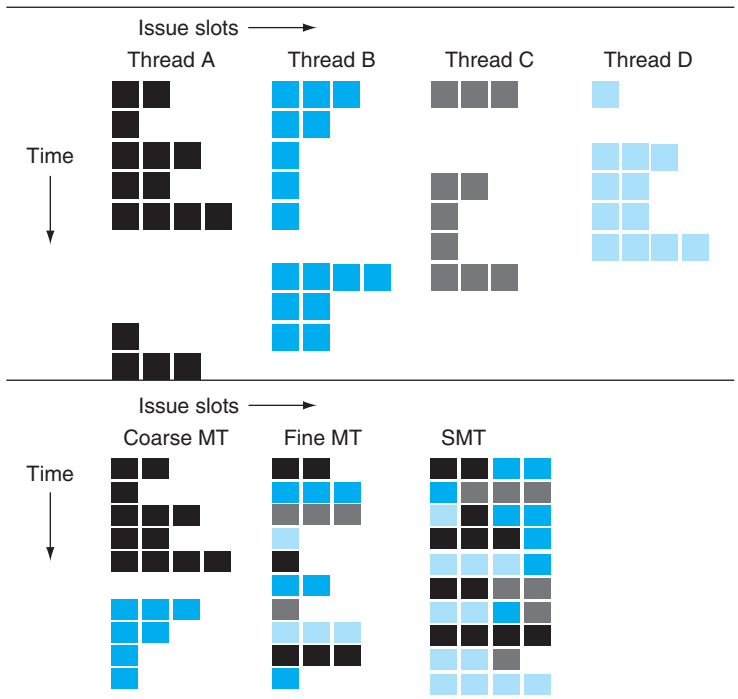
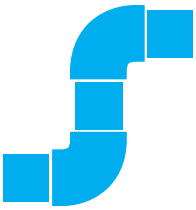


FIGURE 6.5 How four threads use the issue slots of a superscalar processor in different approaches. The four threads at the top show how each would execute running alone on a standard superscalar processor without multithreading support. The three examples at the bottom show how they would execute running together in three multithreading options. The horizontal dimension represents the instruction issue capability in each clock cycle. The vertical dimension represents a sequence of clock cycles. An empty (white) box indicates that the corresponding issue slot is unused in that clock cycle. The shades of gray and color correspond to four different threads in the multithreading processors. The additional pipeline start-up effects for coarse multithreading, which are not illustrated in this figure, would lead to further loss in throughput for coarse multithreading.



PIPELINING

simultaneous multithreading (SMT) A version of multithreading that lowers the cost of multithreading by utilizing the resources needed for multiple issue, dynamically scheduled microarchitecture.

how four threads would execute independently on a superscalar with no multithreading support. The bottom portion shows how the four threads could be combined to execute on the processor more efficiently using three multithreading options:

- A superscalar with coarse-grained multithreading
- A superscalar with fine-grained multithreading
- A superscalar with simultaneous multithreading



In the superscalar without hardware multithreading support, the use of issue slots is limited by a lack of **instruction-level parallelism**. In addition, a major stall, such as an instruction cache miss, can leave the entire processor idle.

In the coarse-grained multithreaded superscalar, the long stalls are partially hidden by switching to another thread that uses the resources of the processor. Although this reduces the number of completely idle clock cycles, the pipeline start-up overhead still leads to idle cycles, and limitations to ILP mean all issue slots will not be used. In the fine-grained case, the interleaving of threads mostly eliminates idle clock cycles. Because only a single thread issues instructions in a given clock cycle, however, limitations in instruction-level parallelism still lead to idle slots within some clock cycles.

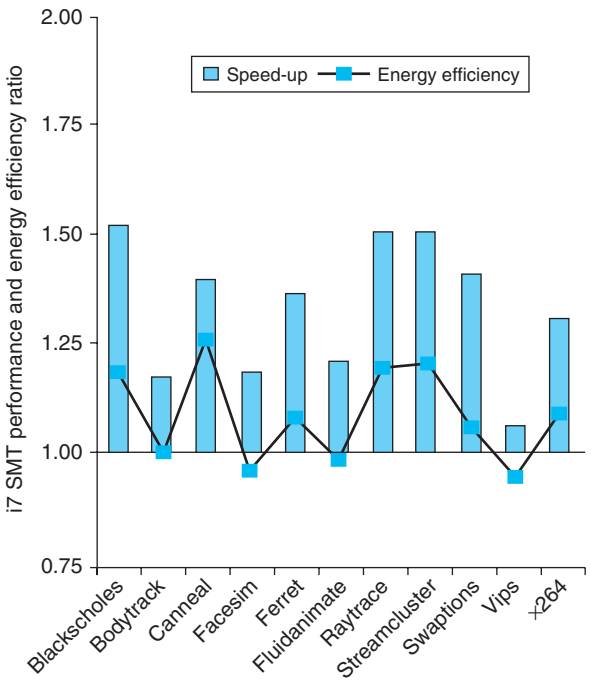


FIGURE 6.6 The speed-up from using multithreading on one core on an i7 processor averages 1.31 for the PARSEC benchmarks (see Section 6.9) and the energy efficiency improvement is 1.07. These data were collected and analyzed by Esmaeilzadeh et al. [2011].

In the SMT case, thread-level parallelism and instruction-level parallelism are both exploited, with multiple threads using the issue slots in a single clock cycle. Ideally, the issue slot usage is limited by imbalances in the resource needs and resource availability over multiple threads. In practice, other factors can restrict how many slots are used. Although Figure 6.5 greatly simplifies the real operation of these processors, it does illustrate the potential performance advantages of multithreading in general and SMT in particular.

Figure 6.6 plots the performance and energy benefits of multithreading on a single processor of the Intel Core i7 960, which has hardware support for two threads. The average speed-up is 1.31, which is not bad given the modest extra resources for hardware multithreading. The average improvement in energy efficiency is 1.07, which is excellent. In general, you'd be happy with a performance speed-up being energy neutral.

Now that we have seen how multiple threads can utilize the resources of a single processor more effectively, we next show how to use them to exploit multiple processors.

Check Yourself

1. True or false: Both multithreading and multicore rely on parallelism to get more efficiency from a chip.
2. True or false: *Simultaneous multithreading* (SMT) uses threads to improve resource utilization of a dynamically scheduled, out-of-order processor.

6.5 Multicore and Other Shared Memory Multiprocessors



While hardware multithreading improved the efficiency of processors at modest cost, the big challenge of the last decade has been to deliver on the performance potential of **Moore's Law** by efficiently programming the increasing number of processors per chip.

Given the difficulty of rewriting old programs to run well on parallel hardware, a natural question is: what can computer designers do to simplify the task? One answer was to provide a single physical address space that all processors can share, so that programs need not concern themselves with where their data are, merely that programs may be executed in parallel. In this approach, all variables of a program can be made available at any time to any processor. The alternative is to have a separate address space per processor that requires that sharing must be explicit; we'll describe this option in the Section 6.7. When the physical address space is common then the hardware typically provides cache coherence to give a consistent view of the shared memory (see Section 5.8).

As mentioned above, a *shared memory multiprocessor* (SMP) is one that offers the programmer a *single physical address space* across all processors—which is

uniform memory access (UMA) A multiprocessor in which latency to any word in main memory is about the same no matter which processor requests the access.

nonuniform memory access (NUMA) A type of single address space multiprocessor in which some memory accesses are much faster than others depending on which processor asks for which word.

synchronization The process of coordinating the behavior of two or more processes, which may be running on different processors.

lock A synchronization device that allows access to data to only one processor at a time.

nearly always the case for multicore chips—although a more accurate term would have been shared-*address* multiprocessor. Processors communicate through shared variables in memory, with all processors capable of accessing any memory location via loads and stores. Figure 6.7 shows the classic organization of an SMP. Note that such systems can still run independent jobs in their own virtual address spaces, even if they all share a physical address space.

Single address space multiprocessors come in two styles. In the first style, the latency to a word in memory does not depend on which processor asks for it. Such machines are called **uniform memory access (UMA)** multiprocessors. In the second style, some memory accesses are much faster than others, depending on which processor asks for which word, typically because main memory is divided and attached to different microprocessors or to different memory controllers on the same chip. Such machines are called **nonuniform memory access (NUMA)** multiprocessors. As you might expect, the programming challenges are harder for a NUMA multiprocessor than for a UMA multiprocessor, but NUMA machines can scale to larger sizes, and NUMAs can have lower latency to nearby memory.

As processors operating in parallel will normally share data, they also need to coordinate when operating on shared data; otherwise, one processor could start working on data before another is finished with it. This coordination is called **synchronization**, which we saw in Chapter 2. When sharing is supported with a single address space, there must be a separate mechanism for synchronization. One approach uses a **lock** for a shared variable. Only one processor at a time can acquire the lock, and other processors interested in shared data must wait until the original processor unlocks the variable. Section 2.11 of Chapter 2 describes the instructions for locking in the RISC-V instruction set.

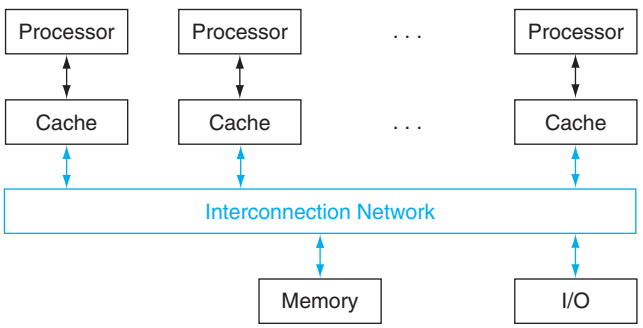


FIGURE 6.7 Classic organization of a shared memory multiprocessor.

A Simple Parallel Processing Program for a Shared Address Space

EXAMPLE

Suppose we want to sum 64,000 numbers on a shared memory multiprocessor computer with uniform memory access time. Let's assume we have 64 processors.

ANSWER

The first step is to ensure a balanced load per processor, so we split the set of numbers into subsets of the same size. We do not allocate the subsets to a different memory space, since there is a single memory space for this machine; we just give different starting addresses to each processor. Pn is the number that identifies the processor, between 0 and 63. All processors start the program by running a loop that sums their subset of numbers:

```
sum[Pn] = 0;
for (i = 1000*Pn; i < 1000*(Pn+1); i += 1)
  sum[Pn] += A[i]; /*sum the assigned areas*/
```

(Note the C code `i += 1` is just a shorter way to say `i = i + 1`.)

The next step is to add these 64 partial sums. This step is called a **reduction**, where we divide to conquer. Half of the processors add pairs of partial sums, and then a quarter add pairs of the new partial sums, and so on until we have the single, final sum. Figure 6.8 illustrates the hierarchical nature of this reduction.

reduction A function that processes a data structure and returns a single value.

In this example, the two processors must synchronize before the "consumer" processor tries to read the result from the memory location written by the "producer" processor; otherwise, the consumer may read the old value of

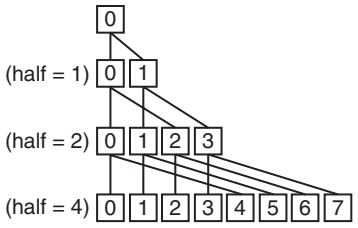


FIGURE 6.8 The last four levels of a reduction that sums results from each processor, from bottom to top. For all processors whose number i is less than half, add the sum produced by processor number (i + half) to its sum.

the data. We want each processor to have its own version of the loop counter variable `i`, so we must indicate that it is a "private" variable. Here is the code (`half` is private also):

```
half = 64; /*64 processors in multiprocessor*/
do
    synch(); /*wait for partial sum completion*/
    if (half%2 != 0 && Pn == 0)
        sum[0] += sum[half-1];
        /*Conditional sum needed when half is
        odd; Processor0 gets missing element */
        half = half/2; /*dividing line on who sums */
        if (Pn < half) sum[Pn] += sum[Pn+half];
while (half > 1); /*exit with final sum in Sum[0] */
```

Hardware/ Software Interface

OpenMP An API for shared memory multiprocessing in C, C++, or Fortran that runs on UNIX and Microsoft platforms. It includes compiler directives, a library, and runtime directives.

Given the long-term interest in parallel programming, there have been hundreds of attempts to build parallel programming systems. A limited but popular example is **OpenMP**. It is just an *Application Programmer Interface* (API) along with a set of compiler directives, environment variables, and runtime library routines that can extend standard programming languages. It offers a portable, scalable, and simple programming model for shared memory multiprocessors. Its primary goal is to parallelize loops and perform reductions.

Most C compilers already have support for OpenMP. The command to use the OpenMP API with the UNIX C compiler is just:

```
cc -fopenmp foo.c
```

OpenMP extends C using *pragmas*, which are just commands to the C macro preprocessor like `#define` and `#include`. To set the number of processors we want to use to be 64, as we wanted in the example above, we just use the command

```
#define P 64 /* define a constant that we'll use a few times */
#pragma omp parallel num_threads(P)
```

That is, the runtime libraries should use 64 parallel threads.

To turn the sequential for loop into a parallel for loop that divides the work equally between all the threads that we told it to use, we just write (assuming `sum` is initialized to 0)

```
#pragma omp parallel for
for (Pn = 0; Pn < P; Pn += 1)
    for (i = 0; 1000*Pn; i < 1000*(Pn+1); i += 1)
      sum[Pn] += A[i]; /*sum the assigned areas*/
```

To perform the reduction, we can use another command that tells OpenMP what the reduction operator is and what variable you need to use to place the result of the reduction.

```
#pragma omp parallel for reduction(+ : FinalSum)
for (i = 0; i < P; i += 1)
      FinalSum += sum[i]; /* Reduce to a single number */
```

Note that it is now up to the OpenMP library to find efficient code to sum 64 numbers efficiently using 64 processors.

While OpenMP makes it easy to write simple parallel code, it is not very helpful with debugging, so many programmers use more sophisticated parallel programming systems than OpenMP, just as many programmers today use more productive languages than C.

Given this tour of classic MIMD hardware and software, our next path is a more exotic tour of a type of MIMD architecture with a different heritage and thus a very different perspective on the parallel programming challenge.

Check Yourself

True or false: Shared memory multiprocessors cannot take advantage of task-level parallelism.

Elaboration: Some writers repurposed the acronym SMP to mean *symmetric multiprocessor*, to indicate that the latency from processor to memory was about the same for all processors. This shift was done to contrast them from large-scale NUMA multiprocessors, as both classes used a single address space. As clusters proved much more popular than large-scale NUMA multiprocessors, in this book we restore SMP to its original meaning, and use it to contrast against those that use multiple address spaces, such as clusters.

Elaboration: An alternative to sharing the physical address space would be to have separate physical address spaces but share a common virtual address space, leaving it up to the operating system to handle communication. This approach has been tried, but it has too high an overhead to offer a practical shared memory abstraction to the performance-oriented programmer.

6.6 Introduction to Graphics Processing Units



The original justification for adding SIMD instructions to existing architectures was that many microprocessors were connected to graphics displays in PCs and workstations, so an increasing fraction of processing time was used for graphics. As **Moore's Law** increased the number of transistors available to microprocessors, it therefore made sense to improve graphics processing.

A major driving force for improving graphics processing was the computer game industry, both on PCs and in dedicated game consoles such as the Sony PlayStation. The rapidly growing game market encouraged many companies to make increasing investments in developing faster graphics hardware, and this positive feedback loop led graphics processing to improve at a quicker rate than general-purpose processing in mainstream microprocessors.

Given that the graphics and game community had different goals than the microprocessor development community, it evolved its own style of processing and terminology. As the graphics processors increased in power, they earned the name *Graphics Processing Units* or *GPUs* to distinguish themselves from CPUs.

For a few hundred dollars, anyone can buy a GPU today with hundreds of parallel floating-point units, which makes high-performance computing more accessible. The interest in GPU computing blossomed when this potential was combined with a programming language that made GPUs easier to program. Hence, many programmers of scientific and multimedia applications today are pondering whether to use GPUs or CPUs.

(This section concentrates on using GPUs for computing. To see how GPU computing combines with the traditional role of graphics acceleration, see **Appendix B**.)

Here are some of the key characteristics as to how GPUs vary from CPUs:

- GPUs are accelerators that supplement a CPU, so they do not need to be able to perform all the tasks of a CPU. This role allows them to dedicate all their resources to graphics. It's fine for GPUs to perform some tasks poorly or not at all, given that in a system with both a CPU and a GPU, the CPU can do them if needed.
- The GPU problem sizes are typically hundreds of megabytes to gigabytes, but not hundreds of gigabytes to terabytes.

These differences led to different styles of architecture:

- Perhaps the biggest difference is that GPUs do not rely on multilevel caches to overcome the long latency to memory, as do CPUs. Instead, GPUs rely on hardware multithreading (Section 6.4) to hide the latency to memory. That is, between the time of a memory request and the time that data arrive, the GPU executes hundreds or thousands of threads that are independent of that request.

- The GPU memory is thus oriented toward bandwidth rather than latency. There are even special graphics DRAM chips for GPUs that are wider and have higher bandwidth than DRAM chips for CPUs. In addition, GPU memories have traditionally had smaller main memories than conventional microprocessors. In 2013, GPUs typically have 4 to 6 GiB or less, while CPUs have 32 to 256 GiB. Finally, keep in mind that for general-purpose computation, you must include the time to transfer the data between CPU memory and GPU memory, since the GPU is a coprocessor.
- Given the reliance on many threads to deliver good memory bandwidth, GPUs can accommodate many parallel processors (MIMD) as well as many threads. Hence, each GPU processor is more highly multithreaded than a typical CPU, plus they have more processors.

Hardware/ Software Interface

Although GPUs were designed for a narrower set of applications, some programmers wondered if they could specify their applications in a form that would let them tap the high potential performance of GPUs. After tiring of trying to specify their problems using the graphics APIs and languages, they developed C-inspired programming languages to allow them to write programs directly for the GPUs. An example is NVIDIA's CUDA (*Compute Unified Device Architecture*), which enables the programmer to write C programs to execute on GPUs, albeit with some restrictions. **Appendix B** gives examples of CUDA code. (OpenCL is a multi-company initiative to develop a portable programming language that provides many of the benefits of CUDA.)

NVIDIA decided that the unifying theme of all these forms of parallelism is the *CUDA Thread*. Using this lowest level of parallelism as the programming primitive, the compiler and the hardware can gang thousands of CUDA threads together to utilize the various styles of parallelism within a GPU: multithreading, MIMD, SIMD, and instruction-level parallelism. These threads are blocked together and executed in groups of 32 at a time. A multithreaded processor inside a GPU executes these blocks of threads, and a GPU consists of 8 to 32 of these multithreaded processors.

An Introduction to the NVIDIA GPU Architecture

We use NVIDIA systems as our example as they are representative of GPU architectures. Specifically, we follow the terminology of the CUDA parallel programming language and use the Fermi architecture as the example.

Like vector architectures, GPUs work well only with data-level parallel problems. Both styles have gather-scatter data transfers, and GPU processors have even more

registers than do vector processors. Unlike most vector architectures, GPUs also rely on hardware multithreading within a single multithreaded SIMD processor to hide memory latency (see Section 6.4).

A multithreaded SIMD processor is similar to a vector processor, but the former has many parallel functional units instead of just a few that are deeply pipelined, as does the latter.

As mentioned above, a GPU contains a collection of multithreaded SIMD processors; that is, a GPU is a MIMD composed of multithreaded SIMD processors. For example, NVIDIA has four implementations of the Fermi architecture at different price points with 7, 11, 14, or 15 multithreaded SIMD processors. To provide transparent scalability across models of GPUs with differing number of multithreaded SIMD processors, the Thread Block Scheduler hardware assigns blocks of threads to multithreaded SIMD processors. Figure 6.9 shows a simplified block diagram of a multithreaded SIMD processor.

Dropping down one more level of detail, the machine object that the hardware creates, manages, schedules, and executes is a *thread of SIMD instructions*, which we will also call a *SIMD thread*. It is a traditional thread, but it contains exclusively SIMD instructions. These SIMD threads have their own program counters, and they run on a multithreaded SIMD processor. The *SIMD Thread Scheduler* includes a controller that lets it know which threads of SIMD instructions are ready to run, and then it sends them off to a dispatch unit to be run on the multithreaded

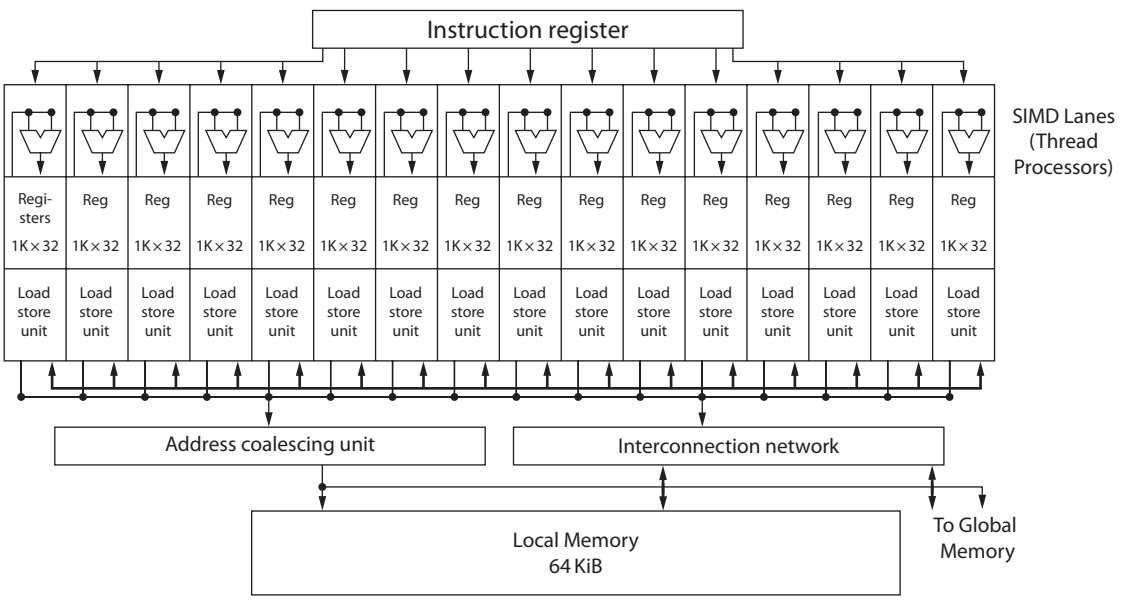


FIGURE 6.9 Simplified block diagram of the datapath of a multithreaded SIMD Processor. It has 16 SIMD lanes. The SIMD Thread Scheduler has many independent SIMD threads that it chooses from to run on this processor.

SIMD processor. It is identical to a hardware thread scheduler in a traditional multithreaded processor (see Section 6.4), except that it is scheduling threads of SIMD instructions. Thus, GPU hardware has two levels of hardware schedulers:

1. The *Thread Block Scheduler* that assigns blocks of threads to multithreaded SIMD processors, and
2. The SIMD Thread Scheduler *within* a SIMD processor, which schedules when SIMD threads should run.

The SIMD instructions of these threads are 32 wide, so each thread of SIMD instructions would compute 32 of the elements of the computation. Since the thread consists of SIMD instructions, the SIMD processor must have parallel functional units to perform the operation. We call them *SIMD Lanes*, and they are quite similar to the Vector Lanes in Section 6.3.

Elaboration: The number of lanes per SIMD processor varies across GPU generations. With Fermi, each 32-wide thread of SIMD instructions is mapped to 16 SIMD lanes, so each SIMD instruction in a thread of SIMD instructions takes two clock cycles to complete. Each thread of SIMD instructions is executed in lock step. Staying with the analogy of a SIMD processor as a vector processor, you could say that it has 16 lanes, and the vector length would be 32. This wide but shallow nature is why we use the term SIMD processor instead of vector processor, as it is more intuitive.

Since by definition the threads of SIMD instructions are independent, the SIMD Thread Scheduler can pick whatever thread of SIMD instructions is ready, and need not stick with the next SIMD instruction in the sequence within a single thread. Thus, using the terminology of Section 6.4, it uses fine-grained multithreading.

To hold these memory elements, a Fermi SIMD processor has an impressive 32,768 32-bit registers. Just like a vector processor, these registers are divided logically across the vector lanes or, in this case, SIMD lanes. Each SIMD thread is limited to no more than 64 registers, so you might think of a SIMD thread as having up to 64 vector registers, with each vector register having 32 elements and each element being 32 bits wide.

Since Fermi has 16 SIMD lanes, each contains 2048 registers. Each CUDA thread gets one element of each of the vector registers. Note that a CUDA thread is just a vertical cut of a thread of SIMD instructions, corresponding to one element executed by one SIMD lane. Beware that CUDA threads are very different from POSIX threads; you can't make arbitrary system calls or synchronize arbitrarily in a CUDA thread.

NVIDIA GPU Memory Structures

Figure 6.10 shows the memory structures of an NVIDIA GPU. We call the on-chip memory that is local to each multithreaded SIMD processor *Local Memory*. It is shared by the SIMD lanes within a multithreaded SIMD processor, but this memory is not shared between multithreaded SIMD processors. We call the off-chip DRAM shared by the whole GPU and all thread blocks *GPU Memory*.

Rather than rely on large caches to contain the entire working sets of an application, GPUs traditionally use smaller streaming caches and rely on extensive multithreading of threads of SIMD instructions to hide the long latency to DRAM,

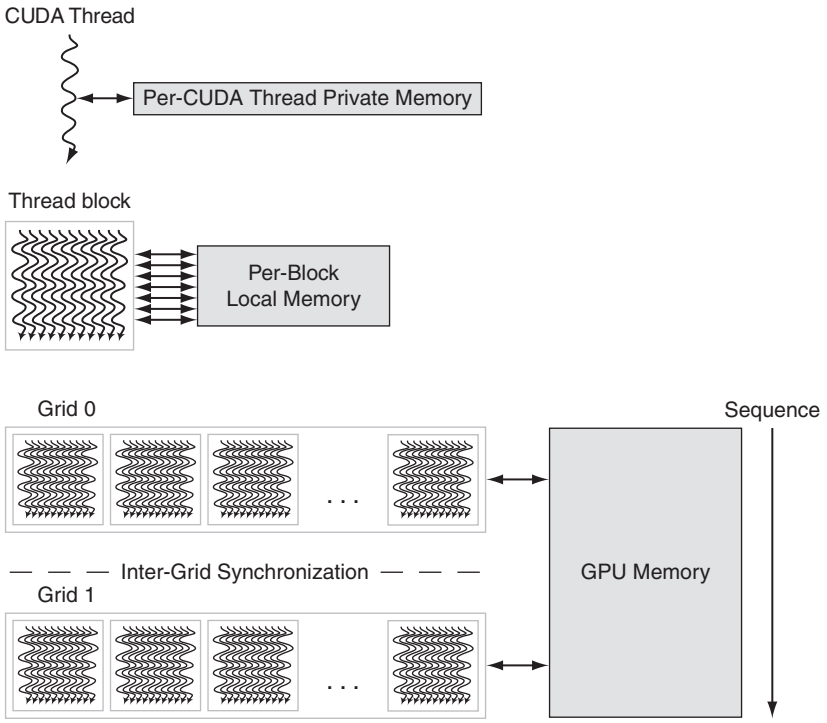


FIGURE 6.10 GPU Memory structures. GPU Memory is shared by the vectorized loops. All threads of SIMD instructions within a thread block share Local Memory.

since their working sets can be hundreds of megabytes. Thus, they will not fit in the last-level cache of a multicore microprocessor. Given the use of hardware multithreading to hide DRAM latency, the chip area used for caches in system processors is spent instead on computing resources and on the large number of registers to hold the state of the many threads of SIMD instructions.

Elaboration: While hiding memory latency is the underlying philosophy, note that the latest GPUs and vector processors have added caches. For example, the recent Fermi architecture has added caches, but they are thought of as either bandwidth filters to reduce demands on GPU Memory or as accelerators for the few variables whose latency cannot be hidden by multithreading. Local memory for stack frames, function calls, and register spilling is a good match to caches, since latency matters when calling a function. Caches can also save energy, since on-chip cache accesses take much less energy than accesses to multiple, external DRAM chips.

Putting GPUs into Perspective

At a high level, multicore computers with SIMD instruction extensions do share similarities with GPUs. Figure 6.11 summarizes the similarities and differences. Both are MIMDs whose processors use multiple SIMD lanes, although GPUs have more processors and many more lanes. Both use hardware multithreading to improve processor utilization, although GPUs have hardware support for many more threads. Both use caches, although GPUs use smaller streaming caches and multicore computers use large multilevel caches that try to contain whole working sets completely. Both use a 64-bit address space, although the physical main memory is much smaller in GPUs. While GPUs support memory protection at the page level, they do not yet support demand paging.

SIMD processors are also similar to vector processors. The multiple SIMD processors in GPUs act as independent MIMD cores, just as many vector computers have multiple vector processors. This view would consider the Fermi GTX 580 as a 16-core machine with hardware support for multithreading, where each core has 16 lanes. The biggest difference is multithreading, which is fundamental to GPUs and missing from most vector processors.

GPUs and CPUs do not go back in computer architecture genealogy to a shared ancestor; there is no Missing Link that explains both. As a result of this uncommon heritage, GPUs have not used the terms common in the computer architecture community, which has led to confusion about what GPUs are and how they work. To help resolve the confusion, Figure 6.12 (from left to right) lists the more descriptive term used in this section, the closest term from mainstream computing, the official NVIDIA GPU term in case you are interested, and then a short description of the term. This "GPU Rosetta Stone" may help relate this section and ideas to more conventional GPU descriptions, such as those found in **Appendix B**.

While GPUs are moving toward mainstream computing, they can't abandon their responsibility to continue to excel at graphics. Thus, the design of GPUs may make more sense when architects ask, given the hardware invested to do graphics

Feature	Multicore with SIMD	GPU
SIMD processors	4 to 8	8 to 16
SIMD lanes/processor	2 to 4	8 to 16
Multithreading hardware support for SIMD threads	2 to 4	16 to 32
Largest cache size	8 MiB	0.75 MiB
Size of memory address	64-bit	64-bit
Size of main memory	8 GiB to 256 GiB	4 GiB to 6 GiB
Memory protection at level of page	Yes	Yes
Demand paging	Yes	No
Cache coherent	Yes	No

FIGURE 6.11 Similarities and differences between multicore with Multimedia SIMD extensions and recent GPUs.

Type	More descriptive name	Closest old term outside of GPUs	Official CUDA/ NVIDIA GPU term	Book definition
Program abstractions	Vectorizable Loop	Vectorizable Loop	Grid	A vectorizable loop, executed on the GPU, made up of one or more Thread Blocks (bodies of vectorized loop) that can execute in parallel.
	Body of Vectorized Loop	Body of a (Strip-Mined) Vectorized Loop	Thread Block	A vectorized loop executed on a multithreaded SIMD Processor, made up of one or more threads of SIMD instructions. They can communicate via Local Memory.
	Sequence of SIMD Lane Operations	One iteration of a Scalar Loop	CUDA Thread	A vertical cut of a thread of SIMD instructions corresponding to one element executed by one SIMD Lane. Result is stored depending on mask and predicate register.
Machine object	A Thread of SIMD Instructions	Thread of Vector Instructions	Warp	A traditional thread, but it contains just SIMD instructions that are executed on a multithreaded SIMD Processor. Results stored depending on a per-element mask.
	SIMD Instruction	Vector Instruction	PTX Instruction	A single SIMD instruction executed across SIMD Lanes.
Processing hardware	Multithreaded SIMD Processor	(Multithreaded) Vector Processor	Streaming Multiprocessor	A multithreaded SIMD Processor executes threads of SIMD instructions, independent of other SIMD Processors.
	Thread Block Scheduler	Scalar Processor	Giga Thread Engine	Assigns multiple Thread Blocks (bodies of vectorized loop) to multithreaded SIMD Processors.
	SIMD Thread Scheduler	Thread scheduler in a Multithreaded CPU	Warp Scheduler	Hardware unit that schedules and issues threads of SIMD instructions when they are ready to execute; includes a scoreboard to track SIMD Thread execution.
	SIMD Lane	Vector lane	Thread Processor	A SIMD Lane executes the operations in a thread of SIMD instructions on a single element. Results stored depending on mask.
Memory hardware	GPU Memory	Main Memory	Global Memory	DRAM memory accessible by all multithreaded SIMD Processors in a GPU.
	Local Memory	Local Memory	Shared Memory	Fast local SRAM for one multithreaded SIMD Processor, unavailable to other SIMD Processors.
	SIMD Lane Registers	Vector Lane Registers	Thread Processor Registers	Registers in a single SIMD Lane allocated across a full thread block (body of vectorized loop).

FIGURE 6.12 Quick guide to GPU terms. We use the first column for hardware terms. Four groups cluster these 12 terms. From top to bottom: Program Abstractions, Machine Objects, Processing Hardware, and Memory Hardware.

well, how can we supplement it to improve the performance of a wider range of applications?

Having covered two different styles of MIMD that have a shared address space, we next introduce parallel processors where each processor has its own private address space, which makes it considerably easier to build much larger systems. The Internet services that you use every day depend on these large-scale systems.

Elaboration: While the GPU was introduced as having a separate memory from the CPU, both AMD and Intel have announced "fused" products that combine GPUs and CPUs to share a single memory. The challenge will be to maintain the high bandwidth memory in a fused architecture that has been a foundation of GPUs.

Check Yourself

True or false: GPUs rely on graphics DRAM chips to reduce memory latency and thereby increase performance on graphics applications.

6.7 Clusters, Warehouse Scale Computers, and Other Message-Passing Multiprocessors

The alternative approach to sharing an address space is for the processors to each have their own private physical address space. Figure 6.13 shows the classic organization of a multiprocessor with multiple private address spaces. This alternative multiprocessor must communicate via explicit **message passing**, which traditionally is the name of such style of computers. Provided the system has routines to **send** and **receive messages**, coordination is built in with message passing, since one processor knows when a message is sent, and the receiving processor knows when a message arrives. If the sender needs confirmation that the message has arrived, the receiving processor can then send an acknowledgment message back to the sender.

message passing Communicating between multiple processors by explicitly sending and receiving information.

send message routine A routine used by a processor in machines with private memories to pass a message to another processor.

receive message routine A routine used by a processor in machines with private memories to accept a message from another processor.

There have been several attempts to build large-scale computers based on high-performance message-passing networks, and they do offer better absolute

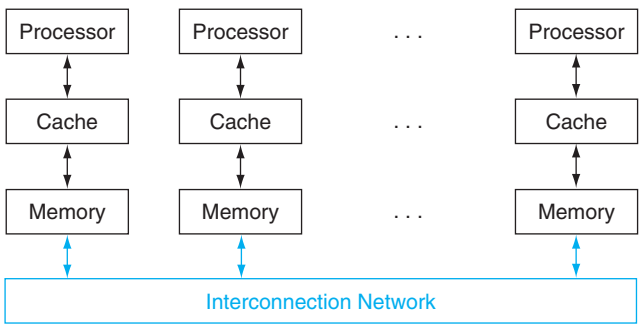


FIGURE 6.13 Classic organization of a multiprocessor with multiple private address spaces, traditionally called a message-passing multiprocessor. Note that unlike the SMP in Figure 6.7, the interconnection network is not between the caches and memory but is instead between processor-memory nodes.

communication performance than clusters built using local area networks. Indeed, many supercomputers today use custom networks. The problem is that they are much more expensive than local area networks like Ethernet. Few applications today outside of high-performance computing can justify the higher communication performance, given the much higher costs.

Hardware/ Software Interface

Computers that rely on message passing for communication rather than cache coherent shared memory are much easier for hardware designers to build (see Section 5.8). There is an advantage for programmers as well, in that communication is explicit, which means there are fewer performance surprises than with the implicit communication in cache-coherent shared memory computers. The downside for programmers is that it's harder to port a sequential program to a message-passing computer, since every communication must be identified in advance or the program doesn't work. Cache-coherent shared memory allows the hardware to figure out what data need to be communicated, which makes porting easier. There are differences of opinion as to which is the shortest path to high performance, given the pros and cons of implicit communication, but there is no confusion in the marketplace today. Multicore microprocessors use shared physical memory and nodes of a cluster communicate with each other using message passing.

Some concurrent applications run well on parallel hardware, independent of whether it offers shared addresses or message passing. In particular, task-level parallelism and applications with little communication—like Web search, mail servers, and file servers—do not require shared addressing to run well. As a result, **clusters** have become the most widespread example today of the message-passing parallel computer. Given the separate memories, each node of a cluster runs a distinct copy of the operating system. In contrast, the cores inside a microprocessor are connected using a high-speed network inside the chip, and a multichip shared-memory system uses the memory interconnect for communication. The memory interconnect has higher bandwidth and lower latency, allowing much better communication performance for shared memory multiprocessors.

clusters Collections of computers connected via I/O over standard network switches to form a message-passing multiprocessor.

The weakness of separate memories for user memory from a parallel programming perspective turns into a strength in system dependability (see Section 5.5). Since a cluster consists of independent computers connected through a local area network, it is much easier to replace a computer without bringing down the system in a cluster than in a shared memory multiprocessor. Fundamentally, the shared address means that it is difficult to isolate a processor and replace it without heroic work by the operating system and in the physical design of the server. It is also easy for clusters to scale down gracefully when a server fails, thereby improving **dependability**. Since the cluster software is a layer that runs on top of the local operating systems running on each computer, it is much easier to disconnect and replace a broken computer.



DEPENDABILITY

Given that clusters are constructed from whole computers and independent, scalable networks, this isolation also makes it easier to expand the system without bringing down the application that runs on top of the cluster.

Their lower cost, higher availability, and rapid, incremental expandability make clusters attractive to service Internet providers, despite their poorer communication performance when compared to large-scale shared-memory multiprocessors. The search engines that hundreds of millions of us use every day depend upon this technology. Amazon, Facebook, Google, Microsoft, and others all have multiple datacenters each with clusters of tens of thousands of servers. Clearly, the use of multiple processors in Internet service companies has been hugely successful.

Warehouse-Scale Computers

Internet services, such as those described above, necessitated the construction of new buildings to house, power, and cool 100,000 servers. Although they may be classified as just large clusters, their architecture and operation are more sophisticated. They act as one giant computer and cost on the order of $150M for the building, the electrical and cooling infrastructure, the servers, and the networking equipment that connects and houses 50,000 to 100,000 servers. We consider them a new class of computer, called *Warehouse-Scale Computers* (WSC).

Anyone can build a fast CPU. The trick is to build a fast system.

Seymour Cray, considered the father of the supercomputer.

Hardware/ Software Interface

The most popular framework for batch processing in a WSC is MapReduce [Dean, 2008] and its open-source twin Hadoop. Inspired by the Lisp functions of the same name, Map first applies a programmer-supplied function to each logical input record. Map runs on thousands of servers to produce an intermediate result of key-value pairs. Reduce collects the output of those distributed tasks and collapses them using another programmer-defined function. With appropriate software support, both are highly parallel yet easy to understand and to use. Within 30 minutes, a novice programmer can run a MapReduce task on thousands of servers.

For example, one MapReduce program calculates the number of occurrences of every English word in a large collection of documents. Below is a simplified version of that program, which shows only the inner loop and assumes just one occurrence of all English words found in a document:

```
map(String key, String value):
      // key: document name
      // value: document contents
      for each word w in value:
      EmitIntermediate(w, "1"); // Produce list of all words
reduce(String key, Iterator values):
// key: a word
// values: a list of counts
      int result = 0;
      for each v in values:
      result += ParseInt(v); // get integer from key-value pair
      Emit(AsString(result));
```

The function `EmitIntermediate` used in the Map function emits each word in the document and the value one. Then the Reduce function sums all the values per word for each document using `ParseInt()` to get the number of occurrences per word in all documents. The MapReduce runtime environment schedules map tasks and reduce tasks to the servers of a WSC.

At this extreme scale, which requires innovation in power distribution, cooling, monitoring, and operations, the WSC is a modern descendant of the 1970s supercomputers—making Seymour Cray the godfather of today's WSC architects. His extreme computers handled computations that could be done nowhere else, but were so expensive that only a few companies could afford them. This time the target is providing information technology for the world instead of high-performance computing for scientists and engineers. Hence, WSCs surely play a more important societal role today than Cray's supercomputers did in the past.

While they share some common goals with servers, WSCs have three major distinctions:

software as a service (SaaS) Rather than selling software that is installed and run on customers' own computers, software is run at a remote site and made available over the Internet typically via a Web interface to customers. SaaS customers are charged based on use versus on ownership.

1. *Ample, easy parallelism*: A concern for a server architect is whether the applications in the targeted marketplace have enough parallelism to justify the amount of parallel hardware and whether the cost is too high for sufficient communication hardware to exploit this parallelism. A WSC architect has no such concern. First, batch applications like MapReduce benefit from the large number of independent data sets that need independent processing, such as billions of Web pages from a Web crawl. Second, interactive Internet service applications, also known as **Software as a Service (SaaS)**, can benefit from millions of independent users of interactive Internet services. Reads and writes are rarely dependent in SaaS, so SaaS rarely needs to synchronize. For example, search uses a read-only index and email is normally reading and writing independent information. We call this type of easy parallelism *Request-Level Parallelism*, as many independent efforts can proceed in parallel naturally with little need for communication or synchronization.
2. *Operational Costs Count*: Traditionally, server architects design their systems for peak performance within a cost budget and worry about energy only to make sure they don't exceed the cooling capacity of their enclosure. They usually ignored operational costs of a server, assuming that they pale in comparison to purchase costs. WSCs have longer lifetimes—the building and electrical and cooling infrastructure are often amortized over 10 or more years—so the operational costs add up: energy, power distribution, and cooling represent more than 30% of the costs of a WSC over 10 years.
3. *Scale and the Opportunities/Problems Associated with Scale*: To construct a single WSC, you must purchase 100,000 servers along with the supporting infrastructure, which means volume discounts. Hence, WSCs are so massive



internally that you get economy of scale even if there are few WSCs. These economies of scale led to *cloud computing*, as the lower per unit costs of a WSC meant that cloud companies could rent servers at a profitable rate and still be below what it costs outsiders to do it themselves. The flip side of the economic opportunity of scale is the need to cope with the failure frequency of scale. Even if a server had a Mean Time To Failure of an amazing 25 years (200,000 hours), the WSC architect would need to design for five server failures every day. Section 5.15 mentioned annualized disk failure rate (AFR) was measured at Google at 2% to 4%. If there were four disks per server and their annual failure rate was 2%, the WSC architect should expect to see one disk fail every *hour*. Thus, fault tolerance is even more important for the WSC architect than for the server architect.

The economies of scale uncovered by WSC have realized the long dreamed of goal of computing as a utility. Cloud computing means anyone anywhere with good ideas, a business model, and a credit card can tap thousands of servers to deliver their vision almost instantly around the world. Of course, there are important obstacles that could limit the growth of cloud computing—such as security, privacy, standards, and the rate of growth of Internet bandwidth—but we foresee them being addressed so that WSCs and cloud computing can flourish.

To put the growth rate of cloud computing into perspective, in 2012 Amazon Web Services announced that it adds enough new server capacity *every day* to support all of Amazon's global infrastructure as of 2003, when Amazon was a $5.2Bn annual revenue enterprise with 6000 employees.

Now that we understand the importance of message-passing multiprocessors, especially for cloud computing, we next cover ways to connect the nodes of a WSC together. Thanks to **Moore's Law** and the increasing number of cores per chip, we now need networks inside a chip as well, so these topologies are important in the small as well as in the large.



Elaboration: The MapReduce framework shuffles and sorts the key-value pairs at the end of the Map phase to produce groups that all share the same key. These groups are next passed to the Reduce phase.

Elaboration: Another form of large-scale computing is *grid computing*, where the computers are spread across large areas, and then the programs that run across them must communicate via long haul networks. The most popular and unique form of grid computing was pioneered by the SETI@home project. As millions of PCs are idle at any one time doing nothing useful, they could be harvested and put to good use if someone developed software that could run on those computers and then gave each PC an independent piece of the problem to work on. The first example was the *Search for ExtraTerrestrial Intelligence* (SETI), which was launched at UC Berkeley in 1999. Over 5 million computer users in more than 200 countries have signed up for SETI@home, with more than 50% outside the US. By the end of 2011, the average performance of the SETI@home grid was 3.5 PetaFLOPS.

Check Yourself

1. True or false: Like SMPs, message-passing computers rely on locks for synchronization.
2. True or false: Clusters have separate memories and thus need many copies of the operating system.

6.8 Introduction to Multiprocessor Network Topologies

Multicore chips require on-chip networks to connect cores together, and clusters require local area networks to connect servers together. This section reviews the pros and cons of different interconnection network topologies.

Network costs include the number of switches, the number of links on a switch to connect to the network, the width (number of bits) per link, and length of the links when the network is mapped into silicon. For example, some cores or servers may be adjacent and others may be on the other side of the chip or the other side of the datacenter. Network performance is multifaceted as well. It includes the latency on an unloaded network to send and receive a message, the throughput in terms of the maximum number of messages that can be transmitted in a given time period, delays caused by contention for a portion of the network, and variable performance depending on the pattern of communication. Another obligation of the network may be fault tolerance, since systems may be required to operate in the presence of broken components. Finally, in this era of energy-limited systems, the energy efficiency of different organizations may trump other concerns.

Networks are normally drawn as graphs, with each edge of the graph representing a link of the communication network. In the figures in this section, the processor-memory node is shown as a black square and the switch is shown as a colored circle. We assume here that all links are *bidirectional*; that is, information can flow in either direction. All networks consist of *switches* whose links go to processor-memory nodes and to other switches. The first network connects a sequence of nodes together:



This topology is called a *ring*. Since some nodes are not directly connected, some messages will have to hop along intermediate nodes until they arrive at the final destination.

Unlike a bus—a shared set of wires that allows broadcasting to all connected devices—a ring is capable of many simultaneous transfers.

Because there are numerous topologies to choose from, performance metrics are needed to distinguish these designs. Two are popular. The first is *total* **network bandwidth**, which is the bandwidth of each link multiplied by the number of links. This represents the peak bandwidth. For the ring network above, with P processors, the total network bandwidth would be P times the bandwidth of one link; the total network bandwidth of a bus is just the bandwidth of that bus.

network bandwidth Informally, the peak transfer rate of a network; can refer to the speed of a single link or the collective transfer rate of all links in the network.

To balance this best bandwidth case, we include another metric that is closer to the worst case: the **bisection bandwidth**. This metric is calculated by dividing the machine into two halves. Then you sum the bandwidth of the links that cross that imaginary dividing line. The bisection bandwidth of a ring is two times the link bandwidth. It is one times the link bandwidth for the bus. If a single link is as fast as the bus, the ring is only twice as fast as a bus in the worst case, but it is P times faster in the best case.

bisection bandwidth The bandwidth between two equal parts of a multiprocessor. This measure is for a worst-case split of the multiprocessor.

Since some network topologies are not symmetric, the question arises of where to draw the imaginary line when bisecting the machine. Bisection bandwidth is a worst-case metric, so the answer is to choose the division that yields the most pessimistic network performance. Stated alternatively, calculate all possible bisection bandwidths and pick the smallest. We take this pessimistic view because parallel programs are often limited by the weakest link in the communication chain.

At the other extreme from a ring is a **fully connected network**, where every processor has a bidirectional link to every other processor. For fully connected networks, the total network bandwidth is $P \times (P-1)/2$, and the bisection bandwidth is $(P/2)^2$.

fully connected network A network that connects processor-memory nodes by supplying a dedicated communication link between every node.

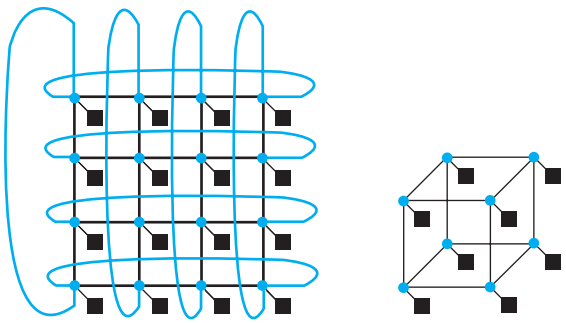
The tremendous improvement in performance of fully connected networks is offset by the tremendous increase in cost. This consequence inspires engineers to invent new topologies that are between the cost of rings and the performance of fully connected networks. The evaluation of success depends in large part on the nature of the communication in the workload of parallel programs run on the computer.

The number of different topologies that have been discussed in publications would be difficult to count, but only a few have been used in commercial parallel processors. Figure 6.14 illustrates two of the popular topologies.

An alternative to placing a processor at every node in a network is to leave only the switch at some of these nodes. The switches are smaller than processor-memory-switch nodes, and thus may be packed more densely, thereby lessening distance and increasing performance. Such networks are frequently called **multistage networks** to reflect the multiple steps that a message may travel. Types of multistage networks are as numerous as single-stage networks; Figure 6.15 illustrates two of the popular multistage organizations. A **fully connected** or **crossbar network** allows any node to communicate with any other node in one pass through the network. An *Omega network* uses less hardware than the crossbar network ($2n \log_2 n$ versus n^2 switches), but contention can occur between messages, depending on the pattern

multistage network A network that supplies a small switch at each node.

crossbar network A network that allows any node to communicate with any other node in one pass through the network.



a. 2-D grid or mesh of 16 nodes b. *n*-cube tree of 8 nodes ($8 = 2^3$ so $n = 3$)

FIGURE 6.14 Network topologies that have appeared in commercial parallel processors. The colored circles represent switches and the black squares represent processor-memory nodes. Even though a switch has many links, generally only one goes to the processor. The Boolean *n*-cube topology is an *n*-dimensional interconnect with 2^n nodes, requiring *n* links per switch (plus one for the processor) and thus *n* nearest-neighbor nodes. Frequently, these basic topologies have been supplemented with extra arcs to improve performance and reliability.

of communication. For example, the Omega network in Figure 6.15 cannot send a message from P_0 to P_6 at the same time that it sends a message from P_1 to P_4.

Implementing Network Topologies

This simple analysis of all the networks in this section ignores important practical considerations in the construction of a network. The distance of each link affects the cost of communicating at a high clock rate—generally, the longer the distance, the more expensive it is to run at a high clock rate. Shorter distances also make it easier to assign more wires to the link, as the power to drive many wires is less if the wires are short. Shorter wires are also cheaper than longer wires. Another practical limitation is that the three-dimensional drawings must be mapped onto chips that are essentially two-dimensional media. The final concern is energy. Energy concerns may force multicore chips to rely on simple grid topologies, for example. The bottom line is that topologies that appear elegant when sketched on the blackboard may be impractical when constructed in silicon or in a datacenter.

Now that we understand the importance of clusters and have seen topologies that we can follow to connect them together, we next look at the hardware and software of the interface of the network to the processor.

Check Yourself True or false: For a ring with P nodes, the ratio of the total network bandwidth to the bisection bandwidth is P/2.

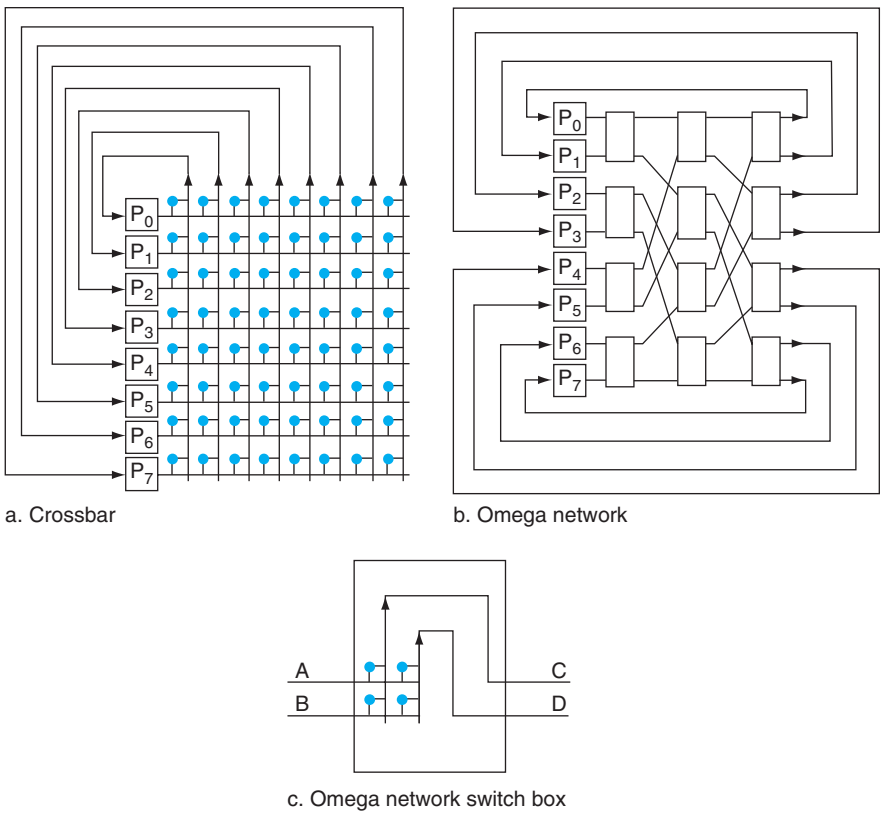


FIGURE 6.15 Popular multistage network topologies for eight nodes. The switches in these drawings are simpler than in earlier drawings because the links are unidirectional; data come in at the left and exit out the right link. The switch box in c can pass A to C and B to D or B to C and A to D. The crossbar uses n^2 switches, where n is the number of processors, while the Omega network uses $2n \log_2 n$ of the large switch boxes, each of which is logically composed of four of the smaller switches. In this case, the crossbar uses 64 switches versus 12 switch boxes, or 48 switches, in the Omega network. The crossbar, however, can support any combination of messages between processors, while the Omega network cannot.



Communicating to the Outside World: Cluster Networking

This online section describes the networking hardware and software used to connect the nodes of a cluster together. The example is 10 gigabit/second Ethernet connected to the computer using *Peripheral Component Interconnect Express* (PCIe). It shows both software and hardware optimizations how to improve network performance, including zero copy messaging, user space communication, using polling instead of I/O interrupts, and hardware calculation of checksums. While the example is networking, the techniques in this section apply to storage controllers and other I/O devices as well.



Communicating to the Outside World: Cluster Networking

This online section describes the networking hardware and software used to connect the nodes of cluster together. As there are whole books and courses just on networking, this section only introduces the main terms and concepts. While our example is networking, the techniques we describe apply to storage controllers and other I/O devices as well.

Ethernet has dominated local-area networks for decades, so it is not surprising that clusters primarily rely on Ethernet as the cluster interconnect. It became commercially popular at 10 Megabits per second link speed in the 1980s, but today 1 Gigabit per second Ethernet is standard and 10 Gigabit per second is being deployed in datacenters. Figure e6.9.1 shows a network interface card (NIC) for 10 Gigabit Ethernet.



Computers offer high-speed links to plug in fast I/O devices like this NIC. While there used to be separate chips to connect the microprocessor to the memory and high-speed I/O devices, thanks to ***Moore's Law*** these functions have been absorbed into the main chip in recent offerings like Intel's Sandy Bridge. A popular high-speed link today is **PCIe**, which stands for **Peripheral Component Interconnect Express**. It is called a *link* in that the basic building block, called a *serial lane*, consists of only four wires: two for receiving data and two for transmitting data. This small number contrasts with an earlier version of PCI that consisted of 64

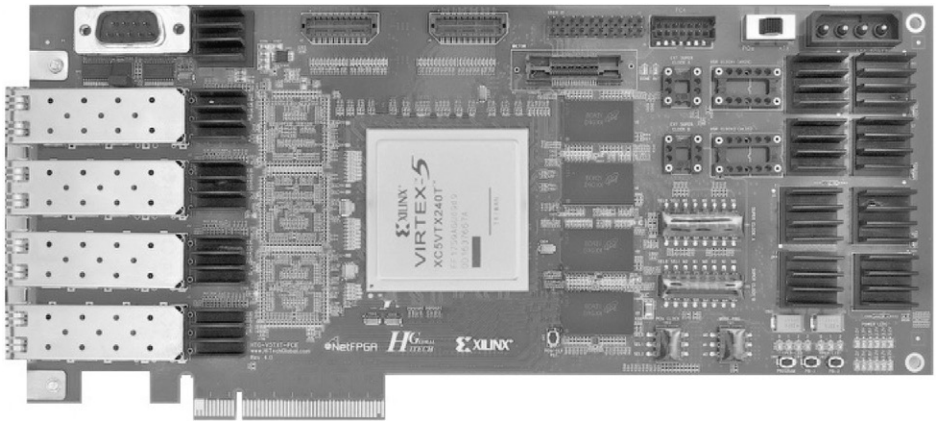


FIGURE e6.9.1 The NetFPGA 10-Gigabit Ethernet card (see http://netfpga.org/), which connects up to four 10-Gigabit/sec Ethernet links. It is an FPGA-based open platform for network research and classroom experimentation. The DMA engine and the four "MAC chips" in Figure e6.9.2 are just portions of the Xilinx Virtex FPGA in the middle of the board. The four PHY chips in Figure e6.9.2 are the four black squares just to the right of the four white rectangles on the left edge of the board, which is where the Ethernet cables are plugged in.

wires, which was called a *parallel bus*. PCIe allows anywhere from one to 32 lanes to be used to connect to I/O devices, depending on its needs. This NIC uses PCI 1.1, so each lane transfers at 2 Gigabits/second.

The NIC in Figure e6.9.1 connects to the host computer over an eight-lane PCIe link, which offers 16 Gigabits/second in both directions. To communicate, a NIC must both send or transmit messages and receive them, often abbreviated as TX and RX, respectively. For this NIC, each 10 G link uses separate transmit and receive queues, each of which can store two full-length Ethernet packets, used between the Ethernet links and the NIC. Figure e6.9.2 is a block diagram of the NIC showing the TX and RX queues. The NIC also has two 32-entry queues for transmitting and receiving between the host computer and the NIC.

memory-mapped I/O An I/O scheme in which portions of the address space are assigned to I/O devices, and reads and writes to those addresses are interpreted as commands to the I/O device.

To give a command to the NIC, the processor must be able to address the device and to supply one or more command words. In **memory-mapped I/O**, portions of the address space are assigned to I/O devices. During initialization (at boot time), PCIe devices can request to be assigned an address region of a specified length. All subsequent processor reads and writes to that address region are forwarded over PCIe to that device. Reads and writes to those addresses are interpreted as commands to the I/O device.

For example, a write operation can be used to send data to the network interface where the data will be interpreted as a command. When the processor issues the address and data, the memory system ignores the operation because the address indicates a portion of the memory space used for I/O. The NIC, however, sees the operation and records the data. User programs are prevented from issuing I/O operations directly, because the OS does not provide access to the address space assigned to the I/O devices, and thus the addresses are protected by the address translation. Memory-mapped I/O can also be used to transmit data by writing or reading to select addresses. The device uses the address to determine the type of command, and the data may be provided by a write or obtained by a read. In any event, the address encodes both the device identity and the type of transmission between processor and device.

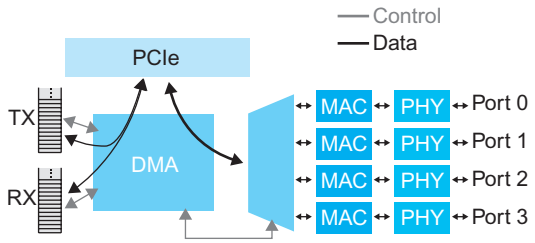


FIGURE e6.9.2 Block diagram of the NetFPGA Ethernet card in Figure e6.9.1 showing the control paths and the data paths. The control path allows the DMA engine to read the status of the queues, such as empty vs. on-empty, and the content of the next available queue entry. The DMA engine also controls port multiplexing. The data path simply passes through the DMA block to the TX/RX queues or to main memory. The "MAC chips" are described below. The PHY chips, which refer to the physical layer, connect the "MAC chips" to physical networking medium, such as copper wire or optical fiber.

While the processor could transfer the data from the user space into the I/O space by itself, the overhead for transferring data from or to a high-speed network could be intolerable, since it could consume a large fraction of the processor. Thus, computer designers long ago invented a mechanism for offloading the processor and having the device controller transfer data directly to or from the memory without involving the processor. This mechanism is called **direct memory access** (DMA).

direct memory access (DMA) A mechanism that provides a device controller with the ability to transfer data directly to or from the memory without involving the processor.

DMA is implemented with a specialized controller that transfers data between the network interface and memory independent of the processor, and in this case the DMA engine is inside the NIC.

To notify the operating system (and eventually the application that will receive the packet) that a transfer is complete, the DMA sends an *I/O interrupt*.

An I/O interrupt is just like the exceptions we saw in Chapters 4 and 5, with two important distinctions:

1. An I/O interrupt is asynchronous with respect to the instruction execution. That is, the interrupt is not associated with any instruction and does not prevent the instruction completion, so it is very different from either page fault exceptions or exceptions such as arithmetic overflow. Our control unit needs only to check for a pending I/O interrupt at the time it starts a new instruction.
2. In addition to the fact that an I/O interrupt has occurred, we would like to convey further information, such as the identity of the device generating the interrupt. Furthermore, the interrupts represent devices that may have different priorities and whose interrupt requests have different urgencies associated with them.

interrupt-driven I/O An I/O scheme that employs interrupts to indicate to the processor that an I/O device needs attention.

To communicate information to the processor, such as the identity of the device raising the interrupt, a system can use either vectored interrupts or an exception identification register, called the *supervisor exception cause* (SCAUSE) register in RISC-V (see Section 4.9). When the processor recognizes the interrupt, the device can send either the vector address or a status field to place in the Cause register. As a result, when the OS gets control, it knows the identity of the device that caused the interrupt and can immediately interrogate the device. An interrupt mechanism eliminates the need for the processor to keep checking the device and instead allows the processor to focus on executing programs.

The Role of the Operating System in Networking

The operating system acts as the interface between the hardware and the program that requests I/O. The network responsibilities of the operating system arise from three characteristics of networks:

1. Multiple programs using the processor share the network.
2. Networks often use interrupts to communicate information about the operations. Because interrupts cause a transfer to kernel or supervisor mode, they must be handled by the operating system (OS).

3. The low-level control of a network is complex, because it requires managing a set of concurrent events and because the requirements for correct device control are often very detailed.

Hardware/Software Interface

These three characteristics of networks specifically and I/O systems in general lead to several different functions the OS must provide:

- The OS guarantees that a user's program accesses only the portions of an I/O device to which the user has rights. For example, the OS must not allow a program to read or write a file on disk if the owner of the file has not granted access to this program. In a system with shared I/O devices, protection could not be provided if user programs could perform I/O directly.
- The OS provides abstractions for accessing devices by supplying routines that handle low-level device operations.
- The OS handles the interrupts generated by I/O devices, just as it handles the exceptions generated by a program.
- The OS tries to provide equitable access to the shared I/O resources, as well as schedule accesses to enhance system throughput.

device driver A program that controls an I/O device that is attached to the computer.

The software inside the operating system that interfaces to a specific I/O device like this NIC is called a **device driver**. The driver for this NIC follows five steps when transmitting or receiving a message. Figure e6.9.3 shows the relationship of these steps as an Ethernet packet is sent from one node of the cluster and received by another node in the cluster.

First, the transmit steps:

1. The driver first prepares a packet buffer in host memory. It copies a packet from the user address space into a buffer that it allocates in the operating system address space.
2. Next, it "talks" to the NIC. The driver writes an *I/O descriptor* to the appropriate NIC register that gives the address of the buffer and its length.
3. The DMA in the NIC next copies the outgoing Ethernet packet from the host buffer over PCIe.
4. When the transmission is complete, the DMA interrupts the processor to notify the processor that the packet has been successfully transmitted.
5. Finally, the driver de-allocates the transmit buffer.

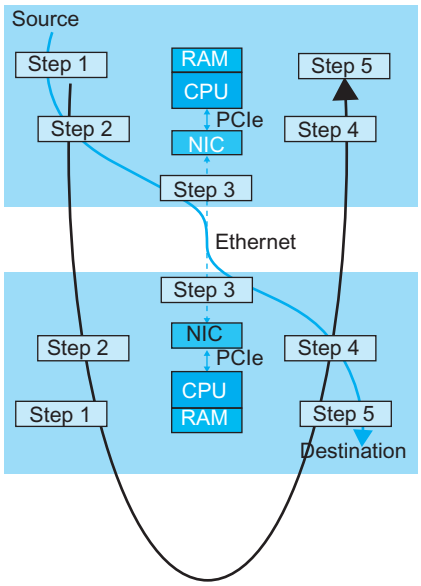


FIGURE e6.9.3 Relationship of the five steps of the driver when transmitting an Ethernet packet from one node and receiving that packet on another node.

Next, the receive steps:

1. First, the driver prepares a packet buffer in host memory, allocating a new buffer in which to place the received packet.
2. Next, it "talks" to the NIC. The driver writes an I/O descriptor to the appropriate NIC register that gives the address of the buffer and its length.
3. The DMA in the NIC next copies the incoming Ethernet packet over PCIe into the allocated host buffer.
4. When the transmission is complete, the DMA interrupts the processor to notify the host of the newly received packet and its size.
5. Finally, the driver copies the received packet into the user address space.

As you can see in Figure e6.9.3, the first three steps are time-critical when transmitting a packet (since the last two occur after the packet is sent), and the last three steps are time-critical when receiving a packet (since the first two occur before a packet arrives). However, these non-critical steps must be completed before individual nodes run out of resources, such as memory space. Failure to do so negatively affects network performance.

Improving Network Performance

The importance of networking in clusters means it is certainly worthwhile to try to improve performance. We show both software and hardware techniques.

Starting with software optimizations, one performance target is reducing the number of times the packet is copied, which you may have noticed happening repeatedly in the five steps of the driver above. The *zero-copy* optimization allows the DMA engine to get the message directly from the user program data space during transmission and be placed where the user wants it when the message is received, rather than go through intermediary buffers in the operating system along the way.

A second software optimization is to cut out the operating system almost entirely by moving the communication into the user address space. By not invoking the operating system and not causing a context switch, we can reduce the software overhead considerably.

In this more radical scenario, a third step would be to drop interrupts. One reason is that modern processors normally go into lower power mode while waiting for an interrupt, and it takes time to come out of low power to service the interrupt as well for the disruption to the pipeline, which increases latency. The alternative to interrupts is for the processor to periodically check status bits to see if I/O operation is complete, which is called **polling**. Hence, we can require the user program to poll the NIC continuously to see when the DMA unit has delivered a message, and as a side effect the processor does not go into low-power mode.

polling The process of periodically checking the status of an I/O device to determine the need to service the device.

Looking at hardware optimizations, one potential target for improvement is in calculating the values of the fields of the Ethernet packet. The 48-bit Ethernet address, called the *Media Access Control address* or *MAC address*, is a unique number assigned to each Ethernet NIC. To improve performance, the "MAC chip"—actually just a portion of the FPGA on this NIC—calculates the value for the preamble fields and the CRC field (see Section 5.5). The driver is left with placing the MAC destination address, MAC source address, message type, the data payload, and padding if needed. (Ethernet requires that the minimum packet, including the header and CRC fields but not the preamble, be 64 bytes.) Note that even the least expensive Ethernet NICs do CRC calculation in hardware today.

A second hardware optimization, available on the most recent Intel processors such as Ivy Bridge, improves the performance of the NIC with respect to the memory hierarchy. *Direct Data IO (DDIO)* allowing up to 10% of the last-level cache is used as a fast scratchpad for the DMA engine. Data are copied directly into the last-level cache rather than to DRAM by the DMA, and only written to DRAM upon eviction from the cache. This optimization helps with latency, but also with bandwidth; some memory regions used for control might be written by the NIC repeatedly, and these writes no longer need to go to DRAM. Thus, DDIO offers benefits similar to those of a write back cache versus a write through cache (Chapter 5).

Let's look at an object store that follows a client-server architecture and uses most of the optimizations above: zero copy messaging, user space communication, polling instead of interrupts, and hardware calculation of preamble and CRC. The driver

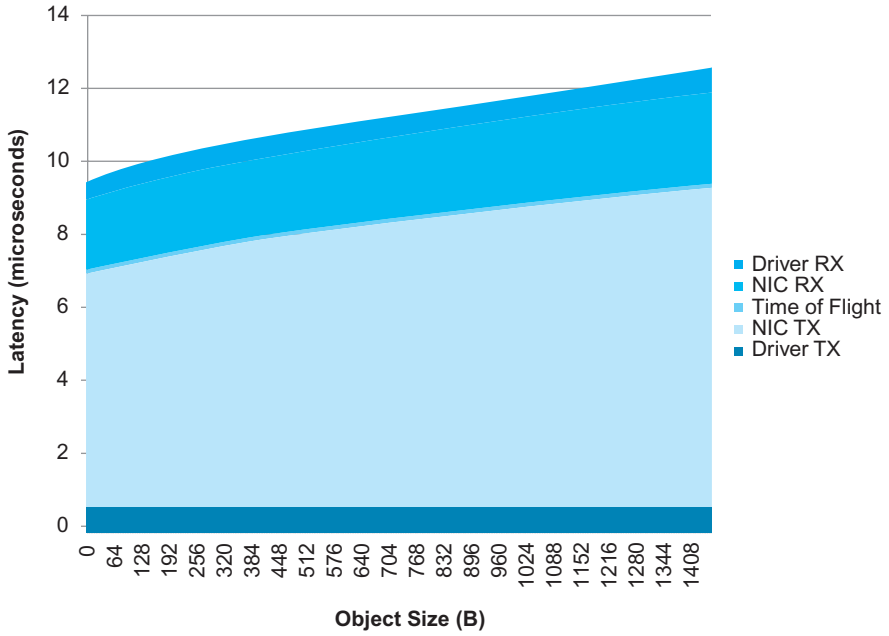


FIGURE e6.9.4 Time to send an object broken into transmit driver and NIC hardware time vs. receive driver and NIC hardware time. NIC transmit time is much larger than the NIC receive time because transmit requires more PCIe round-trips. The NIC does PCIe reads to read the descriptor and data, but on receive the NIC does PCIe writes of data, length of data, and interrupt. PCIe reads incur a round trip latency because NIC waits for the reply, but PCIe writes require no response because PCIe is reliable, so PCIe writes can be sent back-to-back.

operates in user address space as a library that the application invokes. It grants this application exclusive and direct access to the NIC. All of the I/O register space on the NIC is mapped into the application, and all of the driver state is kept in the application. The OS kernel doesn't even see the NIC as such, which avoids the overheads of context switching, the standard kernel network software stack, and interrupts.

Figure e6.9.4 shows the time to send an object from one node to another. It varies from about 9.5 to 12.5 microseconds, depending on the size of the object. Here is the time for each step in microseconds:

0.7 – for the client "driver" (library) to make the request (Driver TX in Figure e6.9.4).

6.4 to 8.7 – for the NIC hardware to transmit the client's request over the PCIe bus to the Ethernet, depending on the size of the object (NIC TX).

0.02 – to send object over the 10 G Ethernet (Time of Flight). The time of flight is limited by speed of light to 5 ns per meter. The three-meter cables used in this measurement mean the time of flight is 15 ns, which is too small to be clearly visible in the figure.

1.8 to 2.5 – for the NIC hardware to receive the object, depending on its size (NIC RX).

0.6 – for the server "driver" to transmit the message with the requested object to the app (Driver RX).

Now that we have seen how to measure the performance of network at a low level of detail, let's raise the perspective to see how to benchmark multiprocessors of all kinds with much higher-level programs.

Elaboration There are three versions of PCIe. This NIC uses PCIe 1.1, which transfers at 2 gigabits per second per lane, so this NIC transfers at up to 16 gigabits per second in each direction. PCIe 2.0, which is found on most PC motherboards today, doubles the lane bandwidth to 4 gigabits per second. PCIe 3.0 doubles again to 8 gigabits per second, and it is starting to be found on some motherboards. We applaud the standard committee's logical rate of bandwidth improvement, which has been about $2^{\text{version number}}$ gigabits/second. The limitations of the Virtex 5 FPGA prevented the NIC from using faster versions of PCIe.

Elaboration While Ethernet is the foundation of cluster communication, clusters commonly use higher-level protocols for reliable communication. Transmission Control Protocol and Internet Protocol (TCP/IP), although invented for planet-wide communication, is often used inside a warehouse-scale computer, due in part to its dependability. While IP makes no delivery guarantees in the protocol, TCP does. The sender keeps the packet sent until it gets the acknowledgment message back that it was received correctly from the receiver. The receiver knows that the message was not corrupted along the way, by double-checking the contents with the TCP CRC field. To ensure that IP delivers to the right destination, the IP header includes a checksum to make sure the destination number remains unchanged. The success of the Internet is due in large part to the elegance and popularity of TCP/IP, which allows independent local-area networks to communicate dependably. Given its importance in the Internet and in clusters, many have accelerated TCP/IP, using techniques like those listed in this section [Regnier, 2004].

Elaboration Adding DMA is another path to the memory system—one that does not go through the address translation mechanism or the cache hierarchy. This difference generates some problems both in virtual memory and in caches. These problems are usually solved with a combination of hardware techniques and software support. The difficulties in having DMA in a virtual memory system arise because pages have both a physical and a virtual address. DMA also creates problems for systems with caches, because there can be two copies of a data item: one in the cache and one in memory. Because the DMA issues memory requests directly to the memory rather than through the processor cache, the value of a memory location seen by the DMA unit and the processor may differ. Consider a read from a NIC that the DMA unit places directly into memory. If some of the locations into which the DMA writes are in the cache, the processor will receive the old value when it does a read. Similarly, if the cache is write-back, the DMA may read a value directly from memory when a newer value is in the

cache, and the value has not been written back. This is called the *stale data problem* or coherence problem (see Chapter 5). Similar solutions for coherence are used with DMA.

Elaboration Virtual Machine support clearly can negatively impact networking performance. As a result, microprocessor designers have been adding hardware to reduce the performance overhead of virtual machines for networking in particular and I/O in general. Intel offers *Virtualization Technology for Directed I/O* (*VT-d*) to help virtualize I/O. It is an I/O memory management unit that enables guest virtual machines to directly use I/O devices, such as Ethernet. It supports *DMA remapping*, which allows the DMA to read or write the data directly in the I/O buffers of the guest virtual machine, rather than into the host I/O buffers and then copy them into the guest I/O buffers. It also supports *interrupt remapping*, which lets the virtual machine monitor route interrupt requests directly to the proper virtual machine.

Check Yourself

Two options for networking are using interrupts or polling, and using DMA or using the processor via load and store instructions.

1. If we want the lowest latency for small packets, which combination is likely best?
2. If we want the lowest latency for large packets, which combination is likely best?

After covering the performance of network at a low level of detail in this online section, the next section shows how to benchmark multiprocessors of all kinds with much higher-level programs.

6.10 Multiprocessor Benchmarks and Performance Models

As we saw in Chapter 1, benchmarking systems is always a sensitive topic, because it is a highly visible way to try to determine which system is better. The results affect not only the sales of commercial systems, but also the reputation of the designers of those systems. Hence, all participants want to win the competition, but they also want to be sure that if someone else wins, they deserve it because they have a genuinely better system. This desire leads to rules to ensure that the benchmark results are not simply engineering tricks for that benchmark, but are instead advances that improve performance of real applications.

To avoid possible tricks, a typical rule is that you can't change the benchmark. The source code and data sets are fixed, and there is a single proper answer. Any deviation from those rules makes the results invalid.

Many multiprocessor benchmarks follow these traditions. A common exception is to be able to increase the size of the problem so that you can run the benchmark on systems with a widely different number of processors. That is, many benchmarks allow weak scaling rather than require strong scaling, even though you must take care when comparing results for programs running different problem sizes.

Figure 6.16 gives a summary of several parallel benchmarks, also described below:

- *Linpack* is a collection of linear algebra routines, and the routines for performing Gaussian elimination constitute what is known as the Linpack benchmark. The DGEMM routine in the example on page 209 represents a small fraction of the source code of the Linpack benchmark, but it accounts for most of the execution time for the benchmark. It allows weak scaling, letting the user pick any size problem. Moreover, it allows the user to rewrite Linpack in almost any form and in any language, as long as it computes the proper result and performs the same number of floating point operations for a given problem size. Twice a year, the 500 computers with the fastest Linpack performance are published at www.top500.org. The first on this list is considered by the press to be the world's fastest computer.

- *SPECrate* is a throughput metric based on the SPEC CPU benchmarks, such as SPEC CPU 2006 (see Chapter 1). Rather than report performance of the individual programs, SPECrate runs many copies of the program simultaneously. Thus, it measures task-level parallelism, as there is no communication between the tasks. You can run as many copies of the programs as you want, so this is again a form of weak scaling.

Benchmark	Scaling?	Reprogram?	Description
Linpack	Weak	Yes	Dense matrix linear algebra [Dongarra, 1979]
SPECrate	Weak	No	Independent job parallelism [Henning, 2007]
Stanford Parallel Applications for Shared Memory SPLASH 2 [Woo et al., 1995]	Strong (although offers two problem sizes)	No	Complex 1D FFT Blocked LU Decomposition Blocked Sparse Cholesky Factorization Integer Radix Sort Barnes-Hut Adaptive Fast Multipole Ocean Simulation Hierarchical Radiosity Ray Tracer Volume Renderer Water Simulation with Spatial Data Structure Water Simulation without Spatial Data Structure
NAS Parallel Benchmarks [Bailey et al., 1991]	Weak	Yes (C or Fortran only)	EP: embarrassingly parallel MG: simplified multigrid CG: unstructured grid for a conjugate gradient method FT: 3-D partial differential equation solution using FFTs IS: large integer sort
PARSEC Benchmark Suite [Bienia et al., 2008]	Weak	No	Blackscholes—Option pricing with Black-Scholes PDE Bodytrack—Body tracking of a person Canneal—Simulated cache-aware annealing to optimize routing Dedup—Next-generation compression with data deduplication Facesim—Simulates the motions of a human face Ferret—Content similarity search server Fluidanimate—Fluid dynamics for animation with SPH method Freqmine—Frequent itemset mining Streamcluster—Online clustering of an input stream Swaptions—Pricing of a portfolio of swaptions Vips—Image processing x264—H.264 video encoding
Berkeley Design Patterns [Asanovic et al., 2006]	Strong or Weak	Yes	Finite-State Machine Combinational Logic Graph Traversal Structured Grid Dense Matrix Sparse Matrix Spectral Methods (FFT) Dynamic Programming N-Body MapReduce Backtrack/Branch and Bound Graphical Model Inference Unstructured Grid

FIGURE 6.16 Examples of parallel benchmarks.

- *SPLASH* and *SPLASH 2* (Stanford Parallel Applications for Shared Memory) were efforts by researchers at Stanford University in the 1990s to put together a parallel benchmark suite similar in goals to the SPEC CPU benchmark suite. It includes both kernels and applications, including many from the high-performance computing community. This benchmark requires strong scaling, although it comes with two data sets.

- The *NAS (NASA Advanced Supercomputing) parallel benchmarks* were another attempt from the 1990s to benchmark multiprocessors. Taken from computational fluid dynamics, they consist of five kernels. They allow weak scaling by defining a few data sets. Like Linpack, these benchmarks can be rewritten, but the rules require that the programming language can only be C or Fortran.
- The recent *PARSEC (Princeton Application Repository for Shared Memory Computers) benchmark suite* consists of multithreaded programs that use **Pthreads** (POSIX threads) and OpenMP (*Open MultiProcessing*; see Section 6.5). They focus on emerging computational domains and consist of nine applications and three kernels. Eight rely on data parallelism, three rely on pipelined parallelism, and one on unstructured parallelism.
- On the cloud front, the goal of the *Yahoo! Cloud Serving Benchmark* (YCSB) is to compare performance of cloud data services. It offers a framework that makes it easy for a client to benchmark new data services, using Cassandra and HBase as representative examples [Cooper, 2010].

Pthreads A UNIX API for creating and manipulating threads. It is structured as a library.

The downside of such traditional restrictions to benchmarks is that innovation is chiefly limited to the architecture and compiler. Better data structures, algorithms, programming languages, and so on often cannot be used, since that would give a misleading result. The system could win because of, say, the algorithm, and not because of the hardware or the compiler.

While these guidelines are understandable when the foundations of computing are relatively stable—as they were in the 1990s and the first half of this decade—they are undesirable during a programming revolution. For this revolution to succeed, we need to encourage innovation at all levels.

Researchers at the University of California at Berkeley have advocated one approach. They identified 13 design patterns that they claim will be part of applications of the future. Frameworks or kernels implement these design patterns. Examples are sparse matrices, structured grids, finite-state machines, map reduce, and graph traversal. By keeping the definitions at a high level, they hope to encourage innovations at any level of the system. Thus, the system with the fastest sparse matrix solver is welcome to use any data structure, algorithm, and programming language, in addition to novel architectures and compilers.

Performance Models

A topic related to benchmarks is performance models. As we have seen with the increasing architectural diversity in this chapter—multithreading, SIMD, GPUs—it would be especially helpful if we had a simple model that offered insights into the performance of different architectures. It need not be perfect, just insightful.

The 3Cs for cache performance from Chapter 5 is an example performance model. It is not a perfect performance model, since it ignores potentially important

factors like block size, block allocation policy, and block replacement policy. Moreover, it has quirks. For example, a miss can be ascribed due to capacity in one design, and to a conflict miss in another cache of the same size. Yet 3Cs model has been popular for 25 years, because it offers insight into the behavior of programs, helping both architects and programmers improve their creations based on insights from that model.

To find such a model for parallel computers, let's start with small kernels, like those from the 13 Berkeley design patterns in Figure 6.16. While there are versions with different data types for these kernels, floating point is popular in several implementations. Hence, peak floating-point performance is a limit on the speed of such kernels on a given computer. For multicore chips, peak floating-point performance is the collective peak performance of all the cores on the chip. If there were multiple microprocessors in the system, you would multiply the peak per chip by the total number of chips.

The demands on the memory system can be estimated by dividing this peak floating-point performance by the average number of floating-point operations per byte accessed:

$$\frac{\text{Floating-Point Operations/Sec}}{\text{Floating-Point Operations/Byte}} = \text{Bytes/Sec}$$

arithmetic intensity The ratio of floating-point operations in a program to the number of data bytes accessed by a program from main memory.

The ratio of floating-point operations per byte of memory accessed is called the **arithmetic intensity**. It can be calculated by taking the total number of floating-point operations for a program divided by the total number of data bytes transferred to main memory during program execution. Figure 6.17 shows the arithmetic intensity of several of the Berkeley design patterns from Figure 6.16.

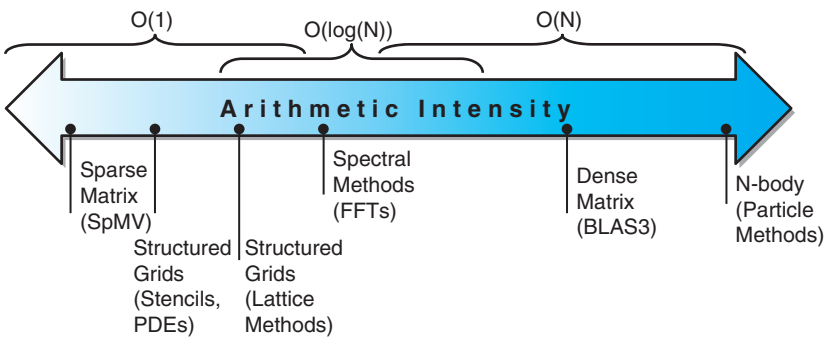


FIGURE 6.17 Arithmetic intensity, specified as the number of floating-point operations to run the program divided by the number of bytes accessed in main memory [Williams, Waterman, and Patterson, 2009]. Some kernels have an arithmetic intensity that scales with problem size, such as Dense Matrix, but there are many kernels with arithmetic intensities independent of problem size. For kernels in this former case, weak scaling can lead to different results, since it puts much less demand on the memory system.

The Roofline Model

This simple model ties floating-point performance, arithmetic intensity, and memory performance together in a two-dimensional graph [Williams, Waterman, and Patterson, 2009]. Peak floating-point performance can be found using the hardware specifications mentioned above. The working sets of the kernels we consider here do not fit in on-chip caches, so peak memory performance may be defined by the memory system behind the caches. One way to find the peak memory performance is the Stream benchmark. (See the *Elaboration* on page 373 in Chapter 5.)

Figure 6.18 shows the model, which is done once for a computer, not for each kernel. The vertical Y-axis is achievable floating-point performance from 0.5 to 64.0 GFLOPs/second. The horizontal X-axis is arithmetic intensity, varying from 1/8 FLOPs/DRAM byte accessed to 16 FLOPs/DRAM byte accessed. Note that the graph is a log-log scale.

For a given kernel, we can find a point on the X-axis based on its arithmetic intensity. If we draw a vertical line through that point, the performance of the kernel on that computer must lie somewhere along that line. We can plot a horizontal line showing peak floating-point performance of the computer. Obviously, the actual floating-point performance can be no higher than the horizontal line, since that is a hardware limit.

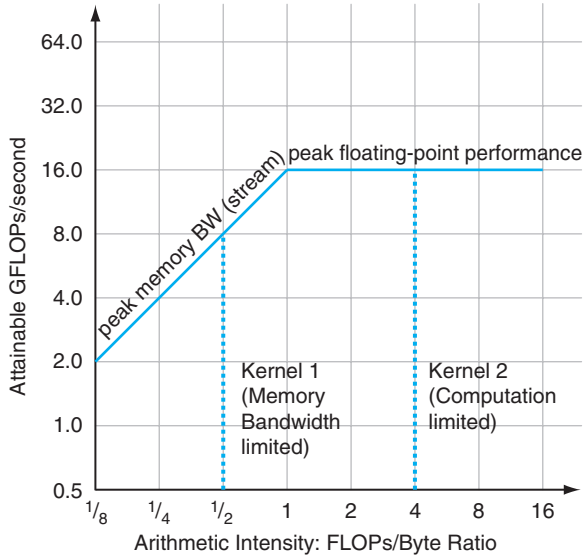


FIGURE 6.18 Roofline Model [Williams, Waterman, and Patterson, 2009]. This example has a peak floating-point performance of 16 GFLOPS/sec and a peak memory bandwidth of 16 GB/sec from the Stream benchmark. (Since Stream is actually four measurements, this line is the average of the four.) The dotted vertical line in color on the left represents Kernel 1, which has an arithmetic intensity of 0.5 FLOPs/byte. It is limited by memory bandwidth to no more than 8 GFLOPS/sec on this Opteron X2. The dotted vertical line to the right represents Kernel 2, which has an arithmetic intensity of 4 FLOPs/byte. It is limited only computationally to 16 GFLOPS/s. (These data are based on the AMD Opteron X2 (Revision F) using dual cores running at 2 GHz in a dual socket system.)

How could we plot the peak memory performance, which is measured in bytes/second? Since the X-axis is FLOPs/byte and the Y-axis FLOPs/second, bytes/second is just a diagonal line at a 45-degree angle in this figure. Hence, we can plot a third line that gives the maximum floating-point performance that the memory system of that computer can support for a given arithmetic intensity. We can express the limits as a formula to plot the line in the graph in Figure 6.18:

$$\text{Attainable GFLOPs/sec} = \text{Min}\ (\text{Peak Memory BW} \times \text{Arithmetic Intensity}, \text{Peak Floating-Point Performance})$$

The horizontal and diagonal lines give this simple model its name and indicate its value. The "roofline" sets an upper bound on performance of a kernel depending on its arithmetic intensity. Given a roofline of a computer, you can apply it repeatedly, since it doesn't vary by kernel.

If we think of arithmetic intensity as a pole that hits the roof, either it hits the slanted part of the roof, which means performance is ultimately limited by memory bandwidth, or it hits the flat part of the roof, which means performance is computationally limited. In Figure 6.18, kernel 1 is an example of the former, and kernel 2 is an example of the latter.

Note that the "ridge point," where the diagonal and horizontal roofs meet, offers an interesting insight into the computer. If it is far to the right, then only kernels with very high arithmetic intensity can achieve the maximum performance of that computer. If it is far to the left, then almost any kernel can potentially hit the maximum performance.

Comparing Two Generations of Opterons

The AMD Opteron X4 (Barcelona) with four cores is the successor to the Opteron X2 with two cores. To simplify board design, they use the same socket. Hence, they have the same DRAM channels and thus the same peak memory bandwidth. In addition to doubling the number of cores, the Opteron X4 also has twice the peak floating-point performance per core: Opteron X4 cores can issue two floating-point SSE2 instructions per clock cycle, while Opteron X2 cores issue at most one. As the two systems we're comparing have similar clock rates—2.2 GHz for Opteron X2 versus 2.3 GHz for Opteron X4—the Opteron X4 has about four times the peak floating-point performance of the Opteron X2 with the same DRAM bandwidth. The Opteron X4 also has a 2MiB L3 cache, which is not found in the Opteron X2.

In Figure 6.19 the roofline models for both systems are compared. As we would expect, the ridge point moves to the right, from 1 in the Opteron X2 to 5 in the Opteron X4. Hence, to see a performance gain in the next generation, kernels need an arithmetic intensity higher than 1, or their working sets must fit in the caches of the Opteron X4.

The roofline model gives an upper bound to performance. Suppose your program is far below that bound. What optimizations should you perform, and in what order?

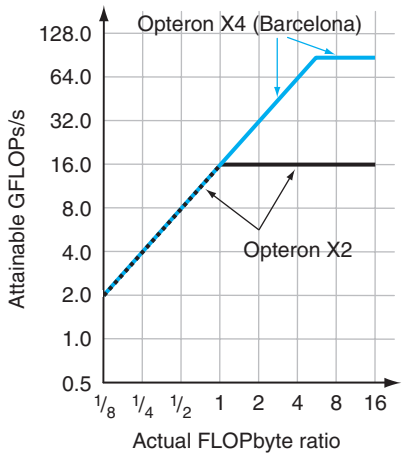


FIGURE 6.19 Roofline models of two generations of Opterons. The Opteron X2 roofline, which is the same as in Figure 6.18, is in black, and the Opteron X4 roofline is in color. The bigger ridge point of Opteron X4 means that kernels that were computationally bound on the Opteron X2 could be memory-performance bound on the Opteron X4.

To reduce computational bottlenecks, the following two optimizations can help almost any kernel:



1. *Floating-point operation mix*. Peak floating-point performance for a computer typically requires an equal number of nearly simultaneous additions and multiplications. That balance is necessary either because the computer supports a fused multiply-add instruction (see the *Elaboration* on page 214 in Chapter 3) or because the floating-point unit has an equal number of floating-point adders and floating-point multipliers. The best performance also requires that a significant fraction of the instruction mix is floating-point operations and not integer instructions.
2. *Improve* ***instruction-level parallelism*** *and apply SIMD*. For modern architectures, the highest performance comes when fetching, executing, and committing three to four instructions per clock cycle (see Section 4.10). The goal for this step is to improve the code from the compiler to increase ILP. One way is by unrolling loops, as we saw in Section 4.12. For the x86 architectures, a single AVX instruction can operate on four double precision operands, so they should be used whenever possible (see Sections 3.7 and 3.8).



To reduce memory bottlenecks, the following two optimizations can help:

1. *Software prefetching*. Usually the highest performance requires keeping many memory operations in flight, which is easier to do by performing **predicting** accesses via software prefetch instructions rather than waiting until the data are required by the computation.



2. *Memory affinity*. Microprocessors today include a memory controller on the same chip with the microprocessor, which improves performance of the **memory hierarchy**. If the system has multiple chips, this means that some addresses go to the DRAM that is local to one chip, and the rest require accesses over the chip interconnect to access the DRAM that is local to another chip. This split results in non-uniform memory accesses, which we described in Section 6.5. Accessing memory through another chip lowers performance. This second optimization tries to allocate data and the threads tasked to operate on that data to the same memory-processor pair, so that the processors rarely have to access the memory of the other chips.

The roofline model can help decide which of these two optimizations to perform and the order in which to perform them. We can think of each of these optimizations as a "ceiling" below the appropriate roofline, meaning that you cannot break through a ceiling without performing the associated optimization.

The computational roofline can be found from the manuals, and the memory roofline can be found from running the Stream benchmark. The computational ceilings, such as floating-point balance, can also come from the manuals for that computer. A memory ceiling, such as memory affinity, requires running experiments on each computer to determine the gap between them. The good news is that this process only need be done once per computer, for once someone characterizes a computer's ceilings, everyone can use the results to prioritize their optimizations for that computer.

Figure 6.20 adds ceilings to the roofline model in Figure 6.18, showing the computational ceilings in the top graph and the memory bandwidth ceilings on the bottom graph. Although the higher ceilings are not labeled with both optimizations, they are implied in this figure; to break through the highest ceiling, you need to have already broken through all the ones below.

The width of the gap between the ceiling and the next higher limit is the reward for trying that optimization. Thus, Figure 6.20 suggests that optimization 2, which improves ILP, has a large benefit for improving computation on that computer, and optimization 4, which improves memory affinity, has a large benefit for improving memory bandwidth on that computer.

Figure 6.21 combines the ceilings of Figure 6.20 into a single graph. The arithmetic intensity of a kernel determines the optimization region, which in turn suggests which optimizations to try. Note that the computational optimizations and the memory bandwidth optimizations overlap for much of the arithmetic intensity. Three regions are shaded differently in Figure 6.21 to indicate the different optimization strategies. For example, Kernel 2 falls in the blue trapezoid on the right, which suggests working only on the computational optimizations. Kernel 1 falls in the blue-gray parallelogram in the middle, which suggests trying both types of optimizations. Moreover, it suggests starting with optimizations 2 and 4. Note that the Kernel 1 vertical lines fall below the floating-point imbalance optimization, so optimization 1 may be unnecessary. If a kernel fell in the gray triangle on the lower left, it would suggest trying just memory optimizations.

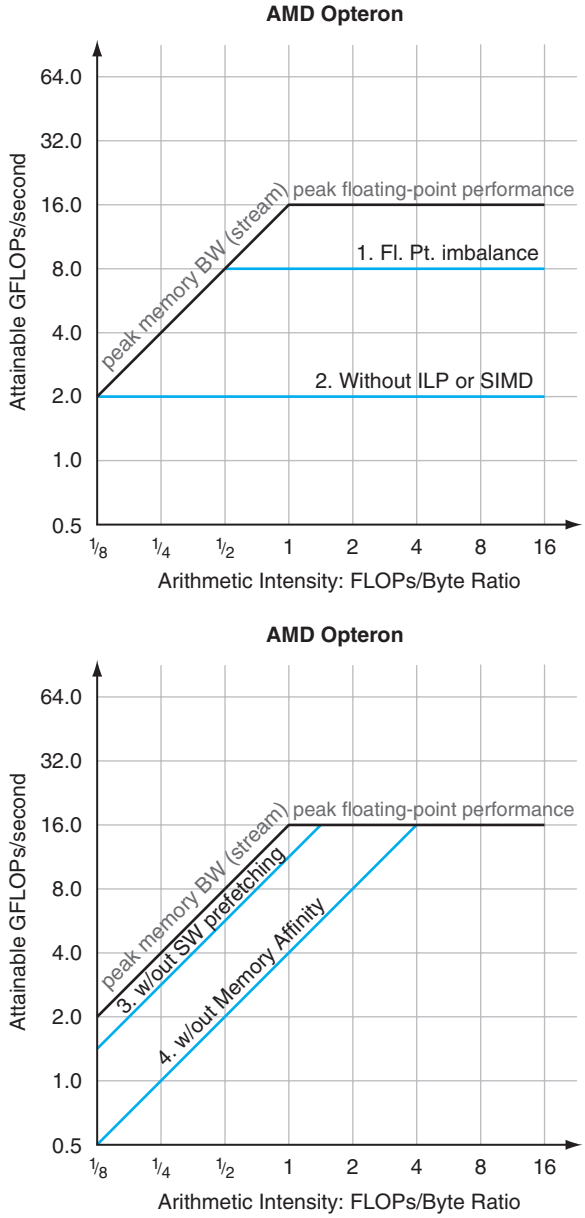


FIGURE 6.20 Roofline model with ceilings. The top graph shows the computational "ceilings" of 8 GFLOPs/sec if the floating-point operation mix is imbalanced and 2 GFLOPs/sec if the optimizations to increase ILP and SIMD are also missing. The bottom graph shows the memory bandwidth ceilings of 11 GB/sec without software prefetching and 4.8 GB/sec if memory affinity optimizations are also missing.

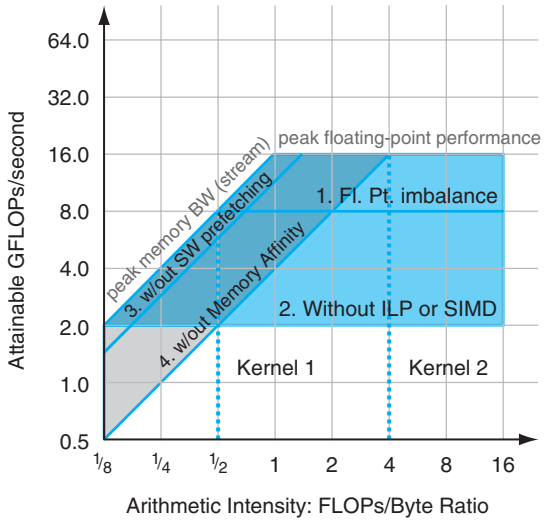


FIGURE 6.21 Roofline model with ceilings, overlapping areas shaded, and the two kernels from Figure 6.18. Kernels whose arithmetic intensity land in the blue trapezoid on the right should focus on computation optimizations, and kernels whose arithmetic intensity land in the gray triangle in the lower left should focus on memory bandwidth optimizations. Those that land in the blue-gray parallelogram in the middle need to worry about both. As Kernel 1 falls in the parallelogram in the middle, try optimizing ILP and SIMD, memory affinity, and software prefetching. Kernel 2 falls in the trapezoid on the right, so try optimizing ILP and SIMD and the balance of floating-point operations.

Thus far, we have been assuming that the arithmetic intensity is fixed, but that is not really the case. First, there are kernels where the arithmetic intensity increases with problem size, such as for Dense Matrix and N-body problems (see Figure 6.17). Indeed, this can be a reason that programmers have more success with weak scaling than with strong scaling. Second, the effectiveness of the **memory hierarchy** affects the number of accesses that go to memory, so optimizations that improve cache performance also improve arithmetic intensity. One example is improving temporal locality by unrolling loops and then grouping together statements with similar addresses. Many computers have special cache instructions that allocate data in a cache but do not first fill the data from memory at that address, since it will soon be over-written. Both these optimizations reduce memory traffic, thereby moving the arithmetic intensity pole to the right by a factor of, say, 1.5. This shift right could put the kernel in a different optimization region.



While the examples above show how to help programmers improve performance, architects can also use the model to decide where they should optimize hardware to improve the performance of the kernels that they think will be important.

The next section uses the roofline model to demonstrate the performance difference between a multicore microprocessor and a GPU and to see whether these differences reflect performance of real programs.

Elaboration: The ceilings are ordered so that lower ceilings are easier to optimize. Clearly, a programmer can optimize in any order, but following this sequence reduces the chances of wasting effort on an optimization that has no benefit due to other constraints. Like the 3Cs model, as long as the roofline model delivers on insights, a model can have assumptions that may prove optimistic. For example, roofline assumes the load is balanced between all processors.

Elaboration: An alternative to the Stream benchmark is to use the raw DRAM bandwidth as the roofline. While the raw bandwidth definitely is a hard upper bound, actual memory performance is often so far from that boundary that it's not that useful. That is, no program can go close to that bound. The downside to using Stream is that very careful programming may exceed the Stream results, so the memory roofline may not be as hard a limit as the computational roofline. We stick with Stream because few programmers will be able to deliver more memory bandwidth than Stream discovers.

Elaboration: Although the roofline model shown is for multicore processors, it clearly would work for a uniprocessor as well.

Check Yourself True or false: The main drawback with conventional approaches to benchmarks for parallel computers is that the rules that ensure fairness also slow software innovation.

6.11 Real Stuff: Benchmarking and Rooflines of the Intel Core i7 960 and the NVIDIA Tesla GPU

A group of Intel researchers published a paper [Lee et al., 2010] comparing a quad-core Intel Core i7 960 with multimedia SIMD extensions to the previous generation GPU, the NVIDIA Tesla GTX 280. Figure 6.22 lists the characteristics of the two systems. Both products were purchased in Fall 2009. The Core i7 is in Intel's 45-nanometer semiconductor technology while the GPU is in TSMC's 65-nanometer technology. Although it might have been fairer to have a comparison by a neutral party or by both interested parties, the purpose of this section is *not* to determine how much faster one product is than another, but to try to understand the relative value of features of these two contrasting architecture styles.

The rooflines of the Core i7 960 and GTX 280 in Figure 6.23 illustrate the differences in the computers. Not only does the GTX 280 have much higher memory bandwidth and double-precision floating-point performance, but also its double-precision ridge point is considerably to the left. The double-precision ridge point is 0.6 for the GTX 280 versus 3.1 for the Core i7. As mentioned above, it is much easier to hit peak computational performance the further the ridge point of

	Core i7-960	GTX 280	GTX 480	Ratio 280/i7	Ratio 480/i7
Number of processing elements (cores or SMs)	4	30	15	7.5	3.8
Clock frequency (GHz)	3.2	1.3	1.4	0.41	0.44
Die size	263	576	520	2.2	2.0
Technology	Intel 45 nm	TSMC 65 nm	TSMC 40 nm	1.6	1.0
Power (chip, not module)	130	130	167	1.0	1.3
Transistors	700 M	1400 M	3030 M	2.0	4.4
Memory brandwith (GBytes/sec)	32	141	177	4.4	5.5
Single-precision SIMD width	4	8	32	2.0	8.0
Double-precision SIMD width	2	1	16	0.5	8.0
Peak single-precision scalar FLOPS (GFLOPs/sec)	26	117	63	4.6	2.5
Peak single-precision SIMD FLOPS (GFLOPs/sec)	102	311 to 933	515 or 1344	3.0–9.1	6.6–13.1
(SP 1 add or multiply)	N.A.	(311)	(515)	(3.0)	(6.6)
(SP 1 instruction fused multiply-adds)	N.A.	(622)	(1344)	(6.1)	(13.1)
(Rare SP dual issue fused multiply-add and multiply)	N.A.	(933)	N.A.	(9.1)	–
Peak double-precision SIMD FLOPS (GFLOPs/sec)	51	78	515	1.5	10.1

FIGURE 6.22 Intel Core i7-960, NVIDIA GTX 280, and GTX 480 specifications. The rightmost columns show the ratios of the Tesla GTX 280 and the Fermi GTX 480 to Core i7. Although the case study is between the Tesla 280 and i7, we include the Fermi 480 to show its relationship to the Tesla 280 since it is described in this chapter. Note that these memory bandwidths are higher than in Figure 6.23 because these are DRAM pin bandwidths and those in Figure 6.23 are at the processors as measured by a benchmark program. (From Table 2 in Lee et al. [2010].)

the roofline is to the left. For single-precision performance, the ridge point moves far to the right for both computers, so it's considerably harder to hit the roof of single-precision performance. Note that the arithmetic intensity of the kernel is based on the bytes that go to main memory, not the bytes that go to cache memory. Thus, as mentioned above, caching can change the arithmetic intensity of a kernel on a particular computer, if most references really go to the cache. Note also that this bandwidth is for unit-stride accesses in both architectures. Real gather-scatter addresses can be slower on the GTX 280 and on the Core i7, as we shall see.

The researchers selected the benchmark programs by analyzing the computational and memory characteristics of four recently proposed benchmark suites and then "formulated the set of *throughput computing kernels* that capture these characteristics." Figure 6.24 shows the performance results, with larger numbers meaning faster. The Rooflines help explain the relative performance in this case study.

Given that the raw performance specifications of the GTX 280 vary from 2.5× slower (clock rate) to 7.5× faster (cores per chip) while the performance varies

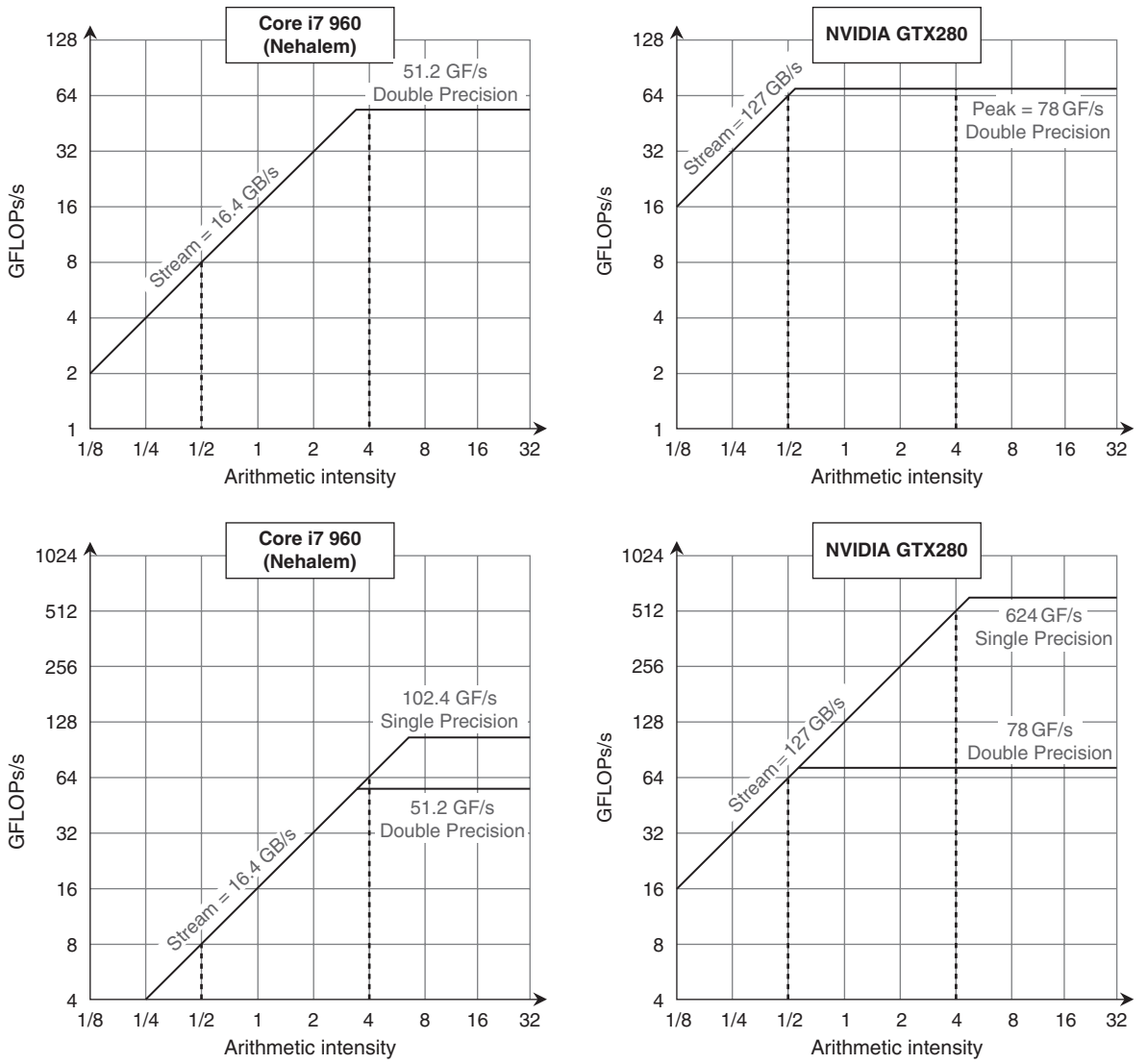


FIGURE 6.23 Roofline model [Williams, Waterman, and Patterson, 2009]. These rooflines show double-precision floating-point performance in the top row and single-precision performance in the bottom row. (The DP FP performance ceiling is also in the bottom row to give perspective.) The Core i7 960 on the left has a peak DP FP performance of 51.2 GFLOPs/sec, a SP FP peak of 102.4 GFLOPs/sec, and a peak memory bandwidth of 16.4 GBytes/sec. The NVIDIA GTX 280 has a DP FP peak of 78 GFLOPs/sec, SP FP peak of 624 GFLOPs/sec, and 127 GBytes/sec of memory bandwidth. The dashed vertical line on the left represents an arithmetic intensity of 0.5 FLOP/byte. It is limited by memory bandwidth to no more than 8 DP GFLOPs/sec or 8 SP GFLOPs/sec on the Core i7. The dashed vertical line to the right has an arithmetic intensity of 4 FLOP/byte. It is limited only computationally to 51.2 DP GFLOPs/sec and 102.4 SP GFLOPs/sec on the Core i7 and 78 DP GFLOPs/sec and 624 SP GFLOPs/sec on the GTX 280. To hit the highest computation rate on the Core i7 you need to use all four cores and SSE instructions with an equal number of multiplies and adds. For the GTX 280, you need to use fused multiply-add instructions on all multithreaded SIMD processors.

Kernel	Units	Core i7-960	GTX 280	GTX 280/ i7-960
SGEMM	GFLOPs/sec	94	364	3.9
MC	Billion paths/sec	0.8	1.4	1.8
Conv	Million pixels/sec	1250	3500	2.8
FFT	GFLOPs/sec	71.4	213	3.0
SAXPY	GBytes/sec	16.8	88.8	5.3
LBM	Million lookups/sec	85	426	5.0
Solv	Frames/sec	103	52	0.5
SpMV	GFLOPs/sec	4.9	9.1	1.9
GJK	Frames/sec	67	1020	15.2
Sort	Million elements/sec	250	198	0.8
RC	Frames/sec	5	8.1	1.6
Search	Million queries/sec	50	90	1.8
Hist	Million pixels/sec	1517	2583	1.7
Bilat	Million pixels/sec	83	475	5.7

FIGURE 6.24 Raw and relative performance measured for the two platforms. In this study, SAXPY is just used as a measure of memory bandwidth, so the right unit is GBytes/sec and not GFLOP/sec. (Based on Table 3 in [Lee et al., 2010].)

from 2.0× slower (Solv) to 15.2× faster (GJK), the Intel researchers decided to find the reasons for the differences:

- *Memory bandwidth*. The GPU has 4.4× the memory bandwidth, which helps explain why LBM and SAXPY run 5.0 and 5.3× faster; their working sets are hundreds of megabytes and hence don't fit into the Core i7 cache. (So as to access memory intensively, they purposely did not use cache blocking as in Chapter 5.) Hence, the slope of the rooflines explains their performance. SpMV also has a large working set, but it only runs 1.9× faster because the double-precision floating point of the GTX 280 is only 1.5× as fast as the Core i7.
- *Compute bandwidth*. Five of the remaining kernels are compute bound: SGEMM, Conv, FFT, MC, and Bilat. The GTX is faster by 3.9, 2.8, 3.0, 1.8, and 5.7×, respectively. The first three of these use single-precision floating-point arithmetic, and GTX 280 single precision is 3 to 6× faster. MC uses double precision, which explains why it's only 1.8× faster since DP performance is only 1.5× faster. Bilat uses transcendental functions, which the GTX 280 supports directly. The Core i7 spends two-thirds of its time calculating transcendental functions for Bilat, so the GTX 280 is 5.7× faster. This observation helps point out the value of hardware support for operations that occur in your workload: double-precision floating point and perhaps even transcendentals.

- *Cache benefits*. *Ray casting* (RC) is only 1.6× faster on the GTX because cache blocking with the Core i7 caches prevents it from becoming memory bandwidth bound (see Sections 5.4 and 5.14), as it is on GPUs. Cache blocking can help Search, too. If the index trees are small so that they fit in the cache, the Core i7 is twice as fast. Larger index trees make them memory bandwidth bound. Overall, the GTX 280 runs search 1.8× faster. Cache blocking also helps Sort. While most programmers wouldn't run Sort on a SIMD processor, it can be written with a 1-bit Sort primitive called *split*. However, the split algorithm executes many more instructions than a scalar sort does. As a result, the Core i7 runs 1.25× as fast as the GTX 280. Note that caches also help other kernels on the Core i7, since cache blocking allows SGEMM, FFT, and SpMV to become compute bound. This observation re-emphasizes the importance of cache blocking optimizations in Chapter 5.

- *Gather-Scatter*. The multimedia SIMD extensions are of little help if the data are scattered throughout main memory; optimal performance comes only when accesses to data are aligned on 16-byte boundaries. Thus, GJK gets little benefit from SIMD on the Core i7. As mentioned above, GPUs offer gather-scatter addressing that is found in a vector architecture but omitted from most SIMD extensions. The memory controller even batches accesses to the same DRAM page together (see Section 5.2). This combination means the GTX 280 runs GJK a startling 15.2× as fast as the Core i7, which is larger than any single physical parameter in Figure 6.22. This observation reinforces the importance of gather-scatter to vector and GPU architectures that is missing from SIMD extensions.

- *Synchronization*. The performance of synchronization is limited by atomic updates, which are responsible for 28% of the total runtime on the Core i7 despite its having a hardware fetch-and-increment instruction. Thus, Hist is only 1.7× faster on the GTX 280. Solv solves a batch of independent constraints in a small amount of computation followed by barrier synchronization. The Core i7 benefits from the atomic instructions and a memory consistency model that ensures the right results even if not all previous accesses to memory hierarchy have completed. Without the memory consistency model, the GTX 280 version launches some batches from the system processor, which leads to the GTX 280 running 0.5× as fast as the Core i7. This observation points out how synchronization performance can be important for some data parallel problems.

It is striking how often weaknesses in the Tesla GTX 280 that were uncovered by kernels selected by Intel researchers were already being addressed in the successor architecture to Tesla: Fermi has faster double-precision floating-point performance, faster atomic operations, and caches. It was also interesting that the gather-scatter support of vector architectures that predate the SIMD instructions by decades was so important to the effective usefulness of these SIMD extensions, which some had predicted before the comparison. The Intel researchers noted that six of the 14 kernels would exploit SIMD better with more efficient gather-scatter support on the Core i7. This study certainly establishes the importance of cache blocking as well.

Now that we have seen a wide range of results of benchmarking different multiprocessors, let's return to our DGEMM example to see in detail how much we have to change the C code to exploit multiple processors.

6.12 Going Faster: Multiple Processors and Matrix Multiply

This section is the final and largest step in our incremental performance journey of adapting DGEMM to the underlying hardware of the Intel Core i7 (Sandy Bridge). Each Core i7 has eight cores, and the computer we have been using has two Core i7s. Thus, we have 16 cores on which to run DGEMM.

Figure 6.25 shows the OpenMP version of DGEMM that utilizes those cores. Note that line 30 is the *single* line added to Figure 5.48 to make this code run on multiple processors: an OpenMP pragma that tells the compiler to use multiple threads in the outermost loop. It tells the computer to spread the work of the outermost loop across all the threads.

Figure 6.26 plots a classic multiprocessor speed-up graph, showing the performance improvement versus a single thread as the number of threads increase. This graph makes it easy to see the challenges of strong scaling versus weak scaling. When everything fits in the first-level data cache, as is the case for 32×32 matrices, adding threads actually hurts performance. The 16-threaded version of DGEMM is almost half as fast as the single-threaded version in this case. In contrast, the two largest matrices get a $14\times$ speedup from 16 threads, and hence the classic two "up and to the right" lines in Figure 6.26.

Figure 6.27 shows the absolute performance increase as we increase the number of threads from one to 16. DGEMM now operates at 174 GLOPS for 960×960 matrices. As our unoptimized C version of DGEMM in Figure 3.22 ran this code at just 0.8 GFLOPS, the optimizations in Chapters 3 to 6 that tailor the code to the underlying hardware result in a speed-up of over 200 times!

Next up is our warnings of the fallacies and pitfalls of multiprocessing. The computer architecture graveyard is filled with parallel processing projects that have ignored them.

Elaboration: These results are with Turbo mode turned off. We are using a dual chip system in this system, so not surprisingly, we can get the full Turbo speed-up (3.3/2.6 = 1.27) with either one thread (only one core on one of the chips) or two threads (one core per chip). As we increase the number of threads and hence the number of active cores, the benefit of Turbo mode decreases, as there is less of the power budget to spend on the active cores. For four threads the average Turbo speed-up is 1.23, for eight it is 1.13, and for 16 it is 1.11.

```
 1 #include <x86intrin.h>
 2 #define UNROLL (4)
 3 #define BLOCKSIZE 32
 4 void do_block (int n, int si, int sj, int sk,
 5                 double *A, double *B, double *C)
 6 {
 7   for ( int i = si; i < si+BLOCKSIZE; i+=UNROLL*4 )
 8     for ( int j = sj; j < sj+BLOCKSIZE; j++ ) {
 9       __m256d c[4];
10       for ( int x = 0; x < UNROLL; x++ )
11         c[x] = _mm256_load_pd(C+i+x*4+j*n);
12      /* c[x] = C[i][j] */
13       for( int k = sk; k < sk+BLOCKSIZE; k++ )
14       {
15         __m256d b = _mm256_broadcast_sd(B+k+j*n);
16      /* b = B[k][j] */
17         for (int x = 0; x < UNROLL; x++)
18           c[x] = _mm256_add_pd(c[x], /* c[x]+=A[i][k]*b */
19                  _mm256_mul_pd(_mm256_load_pd(A+n*k+x*4+i), b));
20       }
21
22       for ( int x = 0; x < UNROLL; x++ )
23         _mm256_store_pd(C+i+x*4+j*n, c[x]);
24         /* C[i][j] = c[x] */
25     }
26 }
27
28 void dgemm (int n, double* A, double* B, double* C)
29 {
30 #pragma omp parallel for
31   for ( int sj = 0; sj < n; sj += BLOCKSIZE )
32     for ( int si = 0; si < n; si += BLOCKSIZE )
33       for ( int sk = 0; sk < n; sk += BLOCKSIZE )
34         do_block(n, si, sj, sk, A, B, C);
35 }
```

FIGURE 6.25 OpenMP version of DGEMM from Figure 5.48. Line 30 is the only OpenMP code, making the outermost for loop operate in parallel. This line is the only difference from Figure 5.48.

Elaboration: Although the Sandy Bridge supports two hardware threads per core, we do not get more performance from 32 threads. The reason is that a single AVX hardware is shared between the two threads multiplexed onto one core, so assigning two threads per core actually hurts performance due to the multiplexing overhead.

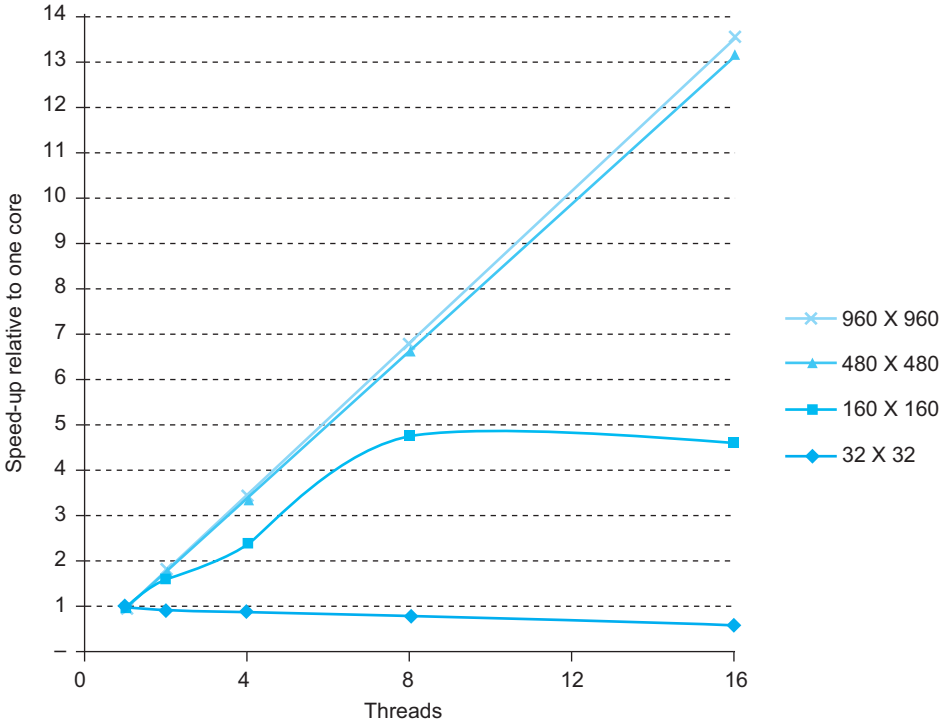


FIGURE 6.26 Performance improvements relative to a single thread as the number of threads increase. The most honest way to present such graphs is to make performance relative to the best version of a single processor program, which we did. This plot is relative to the performance of the code in Figure 5.48 *without* including OpenMP pragmas.

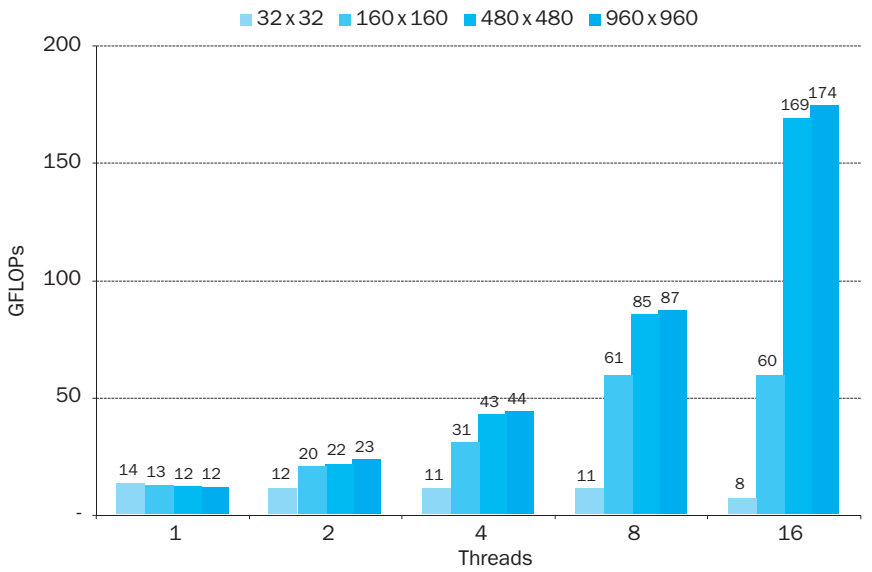


FIGURE 6.27 DGEMM performance versus the number of threads for four matrix sizes. The performance improvement compared unoptimized code in Figure 3.22 for the 960 × 960 matrix with 16 threads is an astounding 212 times faster!

6.13 Fallacies and Pitfalls

For over a decade prophets have voiced the contention that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers in such a manner as to permit cooperative solution. …Demonstration is made of the continued validity of the single processor approach …

Gene Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," Spring Joint Computer Conference, 1967

The many assaults on parallel processing have uncovered numerous fallacies and pitfalls. We cover four here.

Fallacy: Amdahl's Law doesn't apply to parallel computers.

In 1987, the head of a research organization claimed that a multiprocessor machine had broken Amdahl's Law. To try to understand the basis of the media reports, let's see the quote that gave us Amdahl's Law [1967, p. 483]:

> *A fairly obvious conclusion which can be drawn at this point is that the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude.*

This statement must still be true; the neglected portion of the program must limit performance. One interpretation of the law leads to the following lemma: portions of every program must be sequential, so there must be an economic upper bound to the number of processors—say, 100. By showing linear speed-up with 1000 processors, this lemma is disproved; hence the claim that Amdahl's Law was broken.

The approach of the researchers was just to use weak scaling: rather than going 1000 times faster on the same data set, they computed 1000 times more work in comparable time. For their algorithm, the sequential portion of the program was constant, independent of the size of the input, and the rest was fully parallel—hence, linear speed-up with 1000 processors.

Amdahl's Law obviously applies to parallel processors. What this research does point out is that one of the main uses of faster computers is to run larger problems. Just be sure that users really care about those problems versus being a justification to buying an expensive computer by finding a problem that simply keeps lots of processors busy.

Fallacy: Peak performance tracks observed performance.

The supercomputer industry once used this metric in marketing, and the fallacy is exacerbated with parallel machines. Not only are marketers using the nearly unattainable peak performance of a uniprocessor node, but also they are then multiplying it by the total number of processors, assuming perfect speed-up! Amdahl's Law suggests how difficult it is to reach either peak; multiplying the two together multiplies the sins. The roofline model helps put peak performance in perspective.

> *Pitfall: Not developing the software to take advantage of, or optimize for, a multiprocessor architecture.*

There is a long history of parallel software lagging behind parallel hardware, possibly because the software problems are much harder. We give one example to show the subtlety of the issues, but there are many examples we could choose!

One frequently encountered problem occurs when software designed for a uniprocessor is adapted to a multiprocessor environment. For example, the Silicon Graphics operating system originally protected the page table with a single lock, assuming that page allocation is infrequent. In a uniprocessor, this does not represent a performance problem. In a multiprocessor, it can become a major performance bottleneck for some programs. Consider a program that uses a large number of pages that are initialized at start-up, which UNIX does for statically allocated pages. Suppose the program is parallelized so that multiple processes allocate the pages. Because page allocation requires the use of the page table, which is locked whenever it is in use, even an OS kernel that allows multiple threads in the OS will be serialized if the processes all try to allocate their pages at once (which is exactly what we might expect at initialization time!).

This page table serialization eliminates parallelism in initialization and has a significant impact on overall parallel performance. This performance bottleneck persists even for task-level parallelism. For example, suppose we split the parallel processing program apart into separate jobs and run them, one job per processor, so that there is no sharing between the jobs. (This is exactly what one user did, since he reasonably believed that the performance problem was due to unintended sharing or interference in his application.) Unfortunately, the lock still serializes all the jobs—so even the independent job performance is poor.

This pitfall indicates the kind of subtle but significant performance bugs that can arise when software runs on multiprocessors. Like many other key software components, the OS algorithms and data structures must be rethought in a multiprocessor context. Placing locks on smaller portions of the page table effectively eliminated the problem.

Fallacy: You can get good vector performance without providing memory bandwidth.

As we saw in the Roofline model, memory bandwidth is quite important to all architectures. DAXPY requires 1.5 memory references per floating-point operation, and this ratio is typical of many scientific codes. Even if the floating-point operations took no time, a Cray-1 could not increase the DAXPY performance of the vector sequence used, since it was memory limited. The Cray-1 performance on Linpack jumped when the compiler used blocking to change the computation so that values could be kept in the vector registers. This approach lowered the number of memory references per FLOP and improved the performance by nearly a factor of two! Thus, the memory bandwidth on the Cray-1 became sufficient for a loop that formerly required more bandwidth, which is just what the Roofline model would predict.

6.14 Concluding Remarks

We are dedicating all of our future product development to multicore designs. We believe this is a key inflection point for the industry. …This is not a race. This is a sea change in computing…

Paul Otellini, Intel President, Intel Developers Forum, 2004

The dream of building computers by simply aggregating processors has been around since the earliest days of computing. Progress in building and using effective and efficient parallel processors, however, has been slow. This rate of progress has been limited by difficult software problems as well as by a long process of evolving the architecture of multiprocessors to enhance usability and improve efficiency. We have discussed many of the software challenges in this chapter, including the difficulty of writing programs that obtain good speed-up due to Amdahl's Law. The wide variety of different architectural approaches and the limited success and short life of many of the parallel architectures of the past have compounded the software difficulties. We discuss the history of the development of these multiprocessors in online **Section 6.15**. To go into even greater depth on topics in this chapter, see Chapter 4 of *Computer Architecture: A Quantitative Approach, Fifth Edition* for more on GPUs and comparisons between GPUs and CPUs and Chapter 6 for more on WSCs.

As we said in Chapter 1, despite this long and checkered past, the information technology industry has now tied its future to parallel computing. Although it is easy to make the case that this effort will fail like many in the past, there are reasons to be hopeful:

- Clearly, *software as a service* (SaaS) is growing in importance, and clusters have proven to be a very successful way to deliver such services. By providing redundancy at a higher level, including geographically distributed datacenters, such services have delivered 24 × 7 × 365 availability for customers around the world.
- We believe that Warehouse-Scale Computers are changing the goals and principles of server design, just as the needs of mobile clients are changing the goals and principles of microprocessor design. Both are revolutionizing the software industry as well. Performance per dollar and performance per joule drive both mobile client hardware and the WSC hardware, and parallelism is the key to delivering on those sets of goals.
- SIMD and vector operations are a good match to multimedia applications, which are playing a larger role in the post-PC era. They share the advantage of being easier for the programmer than classic parallel MIMD programming and being more energy-efficient than MIMD. To put into perspective the importance of SIMD versus MIMD, Figure 6.28 plots the number of cores for MIMD versus the number of 32-bit and 64-bit operations per clock cycle in SIMD mode for x86 computers over time. For x86 computers, we expect to see two additional cores per chip about every 2 years and the SIMD width to double about every 4 years. Given these assumptions, over the next decade the potential speed-up from SIMD parallelism is twice that of

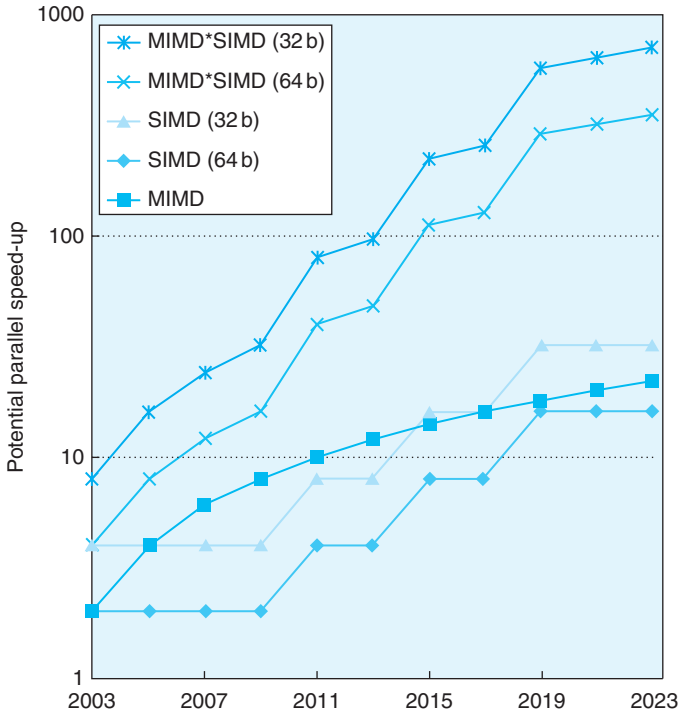


FIGURE 6.28 Potential speed-up via parallelism from MIMD, SIMD, and both MIMD and SIMD over time for x86 computers. This figure assumes that two cores per chip for MIMD will be added every 2 years and the number of operations for SIMD will double every 4 years.

MIMD parallelism. Given the effectiveness of SIMD for multimedia and its increasing importance in the post-PC era, that emphasis may be appropriate. Hence, it's as least as important to understand SIMD parallelism as MIMD parallelism, even though the latter has received much more attention.

- The use of parallel processing in domains such as scientific and engineering computation is popular. This application domain has an almost limitless thirst for more computation. It also has many applications that have lots of natural concurrency. Once again, clusters dominate this application area. For example, using the 2012 Top 500 report, clusters are responsible for more than 80% of the 500 fastest Linpack results.
- All desktop and server microprocessor manufacturers are building multiprocessors to achieve higher performance, so, unlike in the past, there is no easy path to higher performance for sequential applications. As we said earlier, sequential programs are now slow programs. Hence, programmers who need higher performance *must* parallelize their codes or write new parallel processing programs.

- In the past, microprocessors and multiprocessors were subject to different definitions of success. When scaling uniprocessor performance, microprocessor architects were happy if single thread performance went up by the square root of the increased silicon area. Thus, they were pleased with sublinear performance in terms of resources. Multiprocessor success used to be defined as *linear* speed-up as a function of the number of processors, assuming that the cost of purchase or cost of administration of n processors was n times as much as one processor. Now that parallelism is happening on-chip via multicore, we can use the traditional microprocessor metric of being successful with sublinear performance improvement.
- The success of just-in-time runtime compilation and autotuning makes it feasible to think of software adapting itself to take advantage of the increasing number of cores per chip, which provides flexibility that is not available when limited to static compilers.
- Unlike in the past, the open source movement has become a critical portion of the software industry. This movement is a meritocracy, where better engineering solutions can win the mind share of the developers over legacy concerns. It also embraces innovation, inviting change to old software and welcoming new languages and software products. Such an open culture could be extremely helpful during this time of rapid change.

To motivate readers to embrace this revolution, we demonstrated the potential of parallelism concretely for matrix multiply on the Intel Core i7 (Sandy Bridge) in the Going Faster sections of Chapters 3 to 6:

- Data-level parallelism in Chapter 3 improved performance by a factor of 3.85 by executing four 64-bit floating-point operations in parallel using the 256-bit operands of the AVX instructions, demonstrating the value of SIMD.
- Instruction-level parallelism in Chapter 4 pushed performance up by another factor of 2.3 by unrolling loops four times to give the out-of-order execution hardware more instructions to schedule.
- Cache optimizations in Chapter 5 improved performance of matrices that didn't fit into the L1 data cache by another factor of 2.0 to 2.5 by using cache blocking to reduce cache misses.
- Thread-level parallelism in this chapter improved performance of matrices that don't fit into a single L1 data cache by another factor of 4 to 14 by utilizing all 16 cores of our multicore chips, demonstrating the value of MIMD. We did this by adding a single line using an OpenMP pragma.

Using the ideas in this book and tailoring the software to this computer added 24 lines of code to DGEMM. For the matrix sizes of 32×32, 160×160, 480×480, and 960×960, the overall performance speed-up from these ideas realized in those two-dozen lines of code is factors of 8, 39, 129, and 212!

This parallel revolution in the hardware/software interface is perhaps the greatest challenge facing the field in the last 60 years. You can also think of it as an outstanding opportunity, as our Going Faster sections demonstrate. This revolution will provide many new research and business prospects inside and outside the IT field, and the companies that dominate the multicore era may not be the same ones that dominated the uniprocessor era. After understanding the underlying hardware trends and learning to adapt software to them, perhaps you will be one of the innovators who will seize the opportunities that are certain to appear in the uncertain times ahead. We look forward to benefiting from your inventions!



Historical Perspective and Further Reading

This section online gives the rich and often disastrous history of multiprocessors over the last 50 years.

References

B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears. Benchmarking cloud serving systems with YCSB, In: Proceedings of the 1st ACM Symposium on Cloud computing, June 10–11, 2010, Indianapolis, Indiana, USA, doi:10.1145/1807128.1807152.

G. Regnier, S. Makineni, R. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong. TCP onloading for data center servers. IEEE Computer, 37(11):48–58, 2004.

6.16 Exercises

6.1 First, write down a list of your daily activities that you typically do on a weekday. For instance, you might get out of bed, take a shower, get dressed, eat breakfast, dry your hair, brush your teeth. Make sure to break down your list so you have a minimum of 10 activities.

6.1.1 [5] <§6.2> Now consider which of these activities is already exploiting some form of parallelism (e.g., brushing multiple teeth at the same time, versus one at a time, carrying one book at a time to school, versus loading them all into your backpack and then carry them "in parallel"). For each of your activities, discuss if they are already working in parallel, but if not, why they are not.



Historical Perspective and Further Reading



There is a tremendous amount of history in multiprocessors; in this section, we divide our discussion by both time period and architecture. We start with the SIMD approach and the Illiac IV. We then turn to a short discussion of some other early experimental multiprocessors and progress to a discussion of some of the great debates in parallel processing. Next we describe the historical roots of the present **multiprocessors** and conclude by discussing recent advances.

SIMD Computers: Attractive Idea, Many Attempts, No Lasting Successes

> *The cost of a general multiprocessor is, however, very high and further design options were considered which would decrease the cost without seriously degrading the power or efficiency of the system. The options consist of recentralizing one of the three major components.… Centralizing the [control unit] gives rise to the basic organization of [an] … array processor such as the Illiac IV.*
>
> Bouknight et al. [1972]

The SIMD model was one of the earliest models of parallel computing, dating back to the first large-scale multiprocessor, the Illiac IV. The key idea in that multiprocessor, as in more recent SIMD multiprocessors, is to have a single instruction that operates on many data items at once, using many functional units (see Figure e6.15.1).

Although successful in pushing several technologies that proved useful in later projects, it failed as a computer. Costs escalated from the $8 million estimate in 1966 to $31 million by 1972, despite construction of only a quarter of the planned multiprocessor. Actual performance was at best 15 MFLOPS, versus initial predictions of 1000 MFLOPS for the full system [Hord, 1982]. Delivered to NASA Ames Research in 1972, the computer required three more years of engineering before it was usable.

These events slowed the investigation of SIMD, with Danny Hillis [1989] resuscitating this style in the Connection Machine, which had 65,636 1-bit processors.

Real SIMD computers need to have a mixture of SISD and SIMD instructions. There is an SISD host computer to perform operations such as branches and address calculations that do not need parallel operation. The SIMD instructions are broadcast to all the execution units, each of which has its own set of registers. For flexibility, individual execution units can be disabled during an SIMD instruction. In addition, massively parallel SIMD multiprocessors rely on interconnection or communication networks to exchange data between processing elements.



FIGURE e6.15.1 The Illiac IV control unit followed by its 64 processing elements. It was perhaps the most infamous of supercomputers. The project started in 1965 and ran its first real application in 1976. The 64 processors used a 13-MHz clock, and their combined main memory size was 1 MB: 64 × 16 KB. The Illiac IV was the first machine to teach us that software for parallel machines dominates hardware issues. *Photo courtesy of NASA Ames Research Center.*

SIMD works best in dealing with arrays in `for` loops. Hence, to have the opportunity for massive parallelism in SIMD, there must be massive amounts of data, or data parallelism. SIMD is at its weakest in case statements, in which each execution unit must perform a different operation on its data, depending on what data it has. The execution units with the wrong data are disabled so that the proper units can continue. Such situations essentially run at 1/*n*th performance, where *n* is the number of cases.

The basic tradeoff in SIMD multiprocessors is performance of a processor versus the number of processors. Recent multiprocessors emphasize a large degree of parallelism over performance of the individual processors. The Connection Multiprocessor 2, for example, offered 65,536 single-bit-wide processors, while the Illiac IV had 64 64-bit processors.

After being resurrected in the 1980s, originally by Thinking Machines and then by MasPar, the SIMD model has once again been put to bed as a general-purpose multiprocessor architecture, for two main reasons. First, it is too inflexible. A number of important problems cannot use such a style of multiprocessor, and the architecture does not scale down in a competitive fashion; that is, small-scale SIMD multiprocessors often have worse cost performance than that of the alternatives. Second, SIMD cannot take advantage of the tremendous performance and cost advantages of microprocessor technology. Instead of leveraging this low-cost technology, designers of SIMD multiprocessors must build custom processors for their multiprocessors.

Although SIMD computers have departed from the scene as general-purpose alternatives, this style of architecture will continue to have a role in special-purpose designs. Many special-purpose tasks are highly data parallel and require a limited set of functional units. Thus, designers can build in support for certain operations, as well as hardwired interconnection paths among functional units. Such organizations are often called array processors, and they are useful for tasks like image and signal processing.

Multimedia Extensions as SIMD Extensions to Instruction Sets

Many recent architectures have laid claim to being the first to offer multimedia extensions, in which a set of new instructions takes advantage of a single wide ALU that can be partitioned so that it will act as several narrower ALUs operating in parallel. It's unlikely that any appeared before 1957, however, when the Lincoln Lab's TX-2 computer offered instructions that operated on the ALU as either one 36-bit operation, two 18-bit operations, or four 9-bit operations. Ivan Sutherland, considered the Father of Computer Graphics, built his historic Sketchpad system on the TX-2. Sketchpad did, in fact, take advantage of these SIMD instructions, despite TX-2 appearing before invention of the term SIMD.

Other Early Experiments



DEPENDABILITY

It is difficult to distinguish the first MIMD multiprocessor. Surprisingly, the first computer from the Eckert-Mauchly Corporation, for example, had duplicate units to improve **availability**.

Two of the best-documented multiprocessor projects were undertaken in the 1970s at Carnegie Mellon University. The first of these was C.mmp, which consisted of 16 PDP-11s connected by a crossbar switch to 16 memory units. It was among the first multiprocessors with more than a few processors, and it had a shared memory programming model. Much of the focus of the research in the C.mmp project was on software, especially in the OS area. A later multiprocessor, Cm*, was

a cluster-based multiprocessor with a distributed memory and a nonuniform access time. The absence of caches and a long remote access latency made data placement critical. Many of the ideas in these multiprocessors would be reused in the 1980s, when the microprocessor made it much cheaper to build multiprocessors.

Great Debates in Parallel Processing

> *The turning away from the conventional organization came in the middle 1960s, when the law of diminishing returns began to take effect in the effort to increase the operational speed of a computer.… Electronic circuits are ultimately limited in their speed of operation by the speed of light … and many of the circuits were already operating in the nanosecond range.*
>
> W. Jack Bouknight et al.
> The Illiac IV System [1972]

> *… sequential computers are approaching a fundamental physical limit on their potential computational power. Such a limit is the speed of light …*
>
> Angel L. DeCegama
> The Technology of Parallel Processing, Volume I [1989]

> *… today's multiprocessors … are nearing an impasse as technologies approach the speed of light. Even if the components of a sequential processor could be made to work this fast, the best that could be expected is no more than a few million instructions per second.*
>
> David Mitchell
> The Transputer: The Time Is Now [1989]

The quotes above give the classic arguments for abandoning the current form of computing, and Amdahl [1967] gave the classic reply in support of continued focus on the IBM 360 architecture. Arguments for the advantages of parallel execution can be traced back to the 19th century [Menabrea, 1842]! Despite this, the effectiveness of the multiprocessor in reducing the latency of individual important programs is still being explored. Aside from these debates about the advantages and limitations of parallelism, several hot debates have focused on how to build multiprocessors.

From today's perspective, it is clear that the speed of light was not the brick wall; the brick wall was, instead, the power consumption of CMOS as the clock rates increased.

It's hard to predict the future, yet in 1989 Gordon Bell made two predictions for 1995. We included these predictions in the first edition of the book, when the outcome was completely unclear. We discuss them in this section, together with an assessment of the accuracy of the prediction.

The first was that a computer capable of sustaining a tera FLOPS—one million MFLOPS—would be constructed by 1995, using either a multicomputer with 4K to 32K nodes or a Connection Multiprocessor with several million processing elements.

To put this prediction in perspective, each year the Gordon Bell Prize acknowledges advances in parallelism, including the fastest real program (highest MFLOPS). In 1989, the winner used an eight-processor Cray Y-MP to run at 1680 MFLOPS. On the basis of these numbers, multiprocessors and programs would have to have improved by a factor of 3.6 each year for the fastest program to achieve 1 TFLOPS in 1995. In 1999, the first Gordon Bell prize winner crossed the 1 TFLOPS bar. Using a 5832-processor IBM RS/6000 SST system designed specially for Livermore Laboratories, they achieved 1.18 TFLOPS on a shock wave simulation. This ratio represents a year-to-year improvement of 1.93, which is still quite impressive.

What has been recognized since the 1990s is that although we may have the technology to build a TFLOPS multiprocessor, it is not clear that the machine is cost-effective, except perhaps for a few very specialized and critically important applications related to national security. We estimated in 1990 that achieving 1 TFLOPS would require a machine with about 5000 processors and would cost about $100 million. The 5832-processor IBM system at Livermore cost $110 million. As might be expected, improvements in the performance of individual microprocessors both in cost and performance directly affect the cost and performance of large-scale multiprocessors, but a 5000-processor system will cost more than 5000 times the price of a desktop system using the same processor. Since that time, much faster multiprocessors have been built, but the major improvements have increasingly come from the processors in the past 5 years, rather than fundamental breakthroughs in parallel architecture.

The second Bell prediction concerned the number of data streams in supercomputers shipped in 1995. Danny Hillis believed that although supercomputers with a small number of data streams might be the best sellers, the biggest multiprocessors would be multiprocessors with many data streams, and these would perform the bulk of the computations. Bell bet Hillis that in the last quarter of calendar year 1995, more sustained MFLOPS would be shipped in multiprocessors using few data streams (<100) rather than many data streams (>1000). This bet concerned only supercomputers, defined as multiprocessors costing more than $1 million and used for scientific applications. Sustained MFLOPS was defined for this bet as the number of floating-point operations per month, so availability of multiprocessors affects their rating.

In 1989, when this bet was made, it was totally unclear who would win. In 1995, a survey of the current publicly known supercomputers showed only six multiprocessors in existence in the world with more than 1000 data streams, so Bell's prediction was a clear winner. In fact, in 1995, much smaller microprocessor-based multiprocessors (<20 processors) were becoming dominant.

In 1995, a survey of the 500 highest-performance multiprocessors in use (based on Linpack ratings), called the Top 500, showed that the largest number of multiprocessors were bus-based shared memory multiprocessors! By 2005,

various clusters or multicomputers played a large role. For example, in the top 25 systems, 11 were custom clusters, such as the IBM Blue Gene system or the Cray XT3, 10 were clusters of shared memory multiprocessors (both using distributed and centralized memory), and the remaining four were clusters built using PCs with an off-the-shelf interconnect.

More Recent Advances and Developments

With the primary exception of the parallel vector multiprocessors and more recently of the IBM Blue Gene design, all other modern MIMD computers have been built from off-the-shelf microprocessors using a bus and logically central memory or an interconnection network and a distributed memory. A number of experimental multiprocessors built in the 1980s further refined and enhanced the concepts that form the basis for many of today's multiprocessors.

The Development of Bus-Based Coherent Multiprocessors

Although very large mainframes were built with multiple processors in the 1960s and 1970s, multiprocessors did not become highly successful until the 1980s. Bell [1985] suggests the key was that the smaller size of the microprocessor allowed the memory bus to replace the interconnection network hardware and that portable operating systems meant that multiprocessor projects no longer required the invention of a new operating system. In this paper, Bell defined the terms multiprocessor and multicomputer and set the stage for two different approaches to building larger-scale multiprocessors. The first bus-based multiprocessor with snooping caches was the Synapse N + 1 in 1984.

The early 1990s saw the beginning of an expansion of such systems with the use of very wide, high-speed buses (the SGI Challenge system used a 256-bit, packet-oriented bus supporting up to eight processor boards and 32 processors) and later the use of multiple buses and crossbar interconnects, for example, in the Sun SPARCCenter and Enterprise systems. In 2001, the Sun Enterprise servers represented the primary example of large-scale (>16 processors), symmetric multiprocessors in active use.

Toward Large-Scale Multiprocessors

In the effort to build large-scale multiprocessors, two different directions were explored: message-passing multicomputers and scalable shared memory multiprocessors. Although there had been many attempts to build mesh and hypercube-connected multiprocessors, one of the first multiprocessors to successfully bring together all the pieces was the Cosmic Cube built at Caltech [Seitz, 1985]. It introduced important advances in routing and interconnect technology and substantially reduced the cost of the interconnect, which helped make the multicomputer viable. The Intel iPSC 860, a hypercube-connected collection of i860s,

was based on these ideas. More recent multiprocessors, such as the Intel Paragon, have used networks with lower dimensionality and higher individual links. The Paragon also employed a separate i860 as a communications controller in each node, although a number of users have found it better to use both i860 processors for computation as well as communication. The Thinking Multiprocessors CM-5 made use of off-the-shelf microprocessors. It provided user-level access to the communication channel, significantly improving communication latency. In 1995, these two multiprocessors represented the state of the art in message-passing multicomputers.

Clusters

Clusters were probably "invented" in the 1960s by customers who could not fit all their work on one computer, or who needed a backup machine in case of failure of the primary machine [Pfister, 1998]. Tandem introduced a 16-node cluster in 1975. Digital followed with VAX clusters, introduced in 1984. They were originally independent computers that shared I/O devices, requiring a distributed operating system to coordinate activity. Soon they had communication links between computers, in part so that the computers could be geographically distributed to increase availability in case of a disaster at a single site. Users log on to the cluster and are unaware of which machine they are using. DEC (now HP) sold more than 25,000 clusters by 1993. Other early companies were Tandem (now HP) and IBM (still IBM). Today, virtually every company has cluster products. Most of these products are aimed at availability, with performance scaling as a secondary benefit.

Scientific computing on clusters emerged as a competitor to MPPs. In 1993, the Beowulf project started with the goal of fulfilling NASA's desire for a 1-GFLOPS computer for less than $50,000. In 1994, a 16-node cluster built from off-the-shelf PCs using 80486s achieved that goal. This emphasis led to a variety of software interfaces to make it easier to submit, coordinate, and debug large programs or a large number of independent programs.

Efforts were made to reduce latency of communication in clusters as well as to increase bandwidth, and several research projects worked on that problem. (One commercial result of the low-latency research was the VI interface standard, which has been embraced by Infiniband, discussed below.) Low latency then proved useful in other applications. For example, in 1997 a cluster of 100 UltraSPARC desktop computers at U.C. Berkeley, connected by 160 MB/sec per link Myrinet switches, was used to set world records in database sort (sorting 8.6 GB of data originally on disk in 1 minute) and in cracking an encrypted message (taking just 3.5 hours to decipher a 40-bit DES key).

This research project, called Network of Workstations, also developed the Inktomi search engine, which led to a start-up company with the same name.

Google followed the example of Inktomi to build search engines from clusters of desktop computers rather than large-scale SMPs, which was the strategy of the leading search engine, Alta Vista, that Google took over. In 2013, virtually all Internet services rely on clusters to serve their millions of customers.

Clusters are also very popular with scientists. One reason is their low cost, which enables individual scientists or small groups to own a cluster dedicated to their programs. Such clusters can get results faster than waiting in the long job queues of the shared MPPs at supercomputer centers, which can stretch to weeks.

For those interested in learning more, Pfister [1998] has written an entertaining book on clusters.

Recent Trends in Large-Scale Multiprocessors

In the mid-to-late 1990s, it became clear that the hoped-for growth in the market for ultralarge-scale parallel computing was unlikely to occur. Without this market growth, it became increasingly obvious that the high-end parallel computing market was too small to support the costs of highly customized hardware and software designed for a small market. Perhaps the most important trend to come out of this observation was that clustering would be used to reach the highest levels of performance. There are now three general classes of large-scale multiprocessors:

1. Clusters that integrate standard desktop motherboards using interconnection technology, such as Myrinet or Infiniban
2. Multicomputers built from standard microprocessors configured into processing elements and connected with a custom interconnect, such as the IBM Blue Gene
3. Clusters of small-scale shared memory computers, possibly with vector support, including the Earth Simulator

The IBM Blue Gene is the most interesting of these designs, since its rationale parallels the underlying causes of the recent trend toward multicore in uniprocessor architectures. Blue Gene started as a research project within IBM aimed at the protein sequencing and folding problem. The Blue Gene designers observed that power was becoming an increasing concern in large-scale multiprocessors and that the performance/watt of processors from the embedded space was much better than those in the high-end uniprocessor space. If parallelism was the route to high performance, why not start with the most efficient building block and simply have more of them?

Thus, Blue Gene is constructed using a custom chip that includes an embedded PowerPC microprocessor offering half the performance of a high-end PowerPC, but at a much smaller fraction of the area and the power. This allows more system functions, including the global interconnect, to be integrated onto the same die.

The result is a highly replicable and efficient building block, allowing Blue Gene to reach much larger processor counts more efficiently. Instead of using stand-alone microprocessors or standard desktop boards as building blocks, Blue Gene uses processor cores. No doubt such an approach provides much greater efficiency. Whether the market can support the cost of a customized design and special software remains an open question.

In 2006, a Blue Gene processor at Lawrence Livermore with 32K processors held a factor of 2.6 lead in Linpack performance over the third-place system, which consisted of 20 SGI Altix 512-processor systems interconnected with Infiniband as a cluster.

Blue Gene's predecessor was an experimental machine, QCDOD, which pioneered the concept of a machine using a lower-power embedded microprocessor and tightly integrated interconnect to drive down the cost and power consumption of a node.

Looking Further

There is an almost unbounded amount of information on multiprocessors and multicomputers: conferences, journal papers, and even books seem to appear faster than any single person can absorb the ideas. No doubt many of these papers will go unnoticed—not unlike the past. Most of the major architecture conferences contain papers on multiprocessors. An annual conference, Supercomputing XY (where X and Y are the last two digits of the year), brings together users, architects, software developers, and vendors and publishes the proceedings in book, CD-ROM, and online (see www.scXY.org) form. Two major journals, *Journal of Parallel and Distributed Computing* and the *IEEE Transactions on Parallel and Distributed Systems*, contain papers on all aspects of parallel processing. Several books focusing on parallel processing are included in the following references, with Culler et al. [1998] being the most recent, large-scale effort. For years, Eugene Miya of NASA Ames has collected an online bibliography of parallel processing papers. The bibliography, which now contains more than 35,000 entries, is available online at: www.ira.uka.de/bibliography/Parallel/Eugene/index.html.

Asanovic et al. [2006] surveyed the wide-ranging challenges for the industry in this multicore challenge. That report may be helpful in understanding the depth of the various challenges.

In addition to documenting the discovery of concepts now used in practice, these references also provide descriptions of many ideas that have been explored and found wanting, as well as ideas whose time has just not yet come. Given the move toward multicore and multiprocessors as the future of high-performance computer architecture, we expect that many new approaches will be explored in the years ahead. A few of them will manage to solve the hardware and software problems that have been the key to using multiprocessing for the past 40 years!

Further Reading

Almasi, G. S. and A. Gottlieb [1989]. *Highly Parallel Computing*, Benjamin/Cummings, Redwood City, CA.

A textbook covering parallel computers.

Amdahl, G. M. [1967]. "Validity of the single processor approach to achieving large scale computing capabilities," *Proc. AFIPS Spring Joint Computer Conf.*, Atlantic City, NJ (April), 483–85.

Written in response to the claims of the Illiac IV, this three-page article describes Amdahl's law and gives the classic reply to arguments for abandoning the current form of computing.

Andrews, G. R. [1991]. *Concurrent Programming: Principles and Practice*, Benjamin/Cummings, Redwood City, CA.

A text that gives the principles of parallel programming.

Archibald, J. and J. -L. Baer [1986]. "Cache coherence protocols: Evaluation using a multiprocessor simulation model", *ACM Trans. on Computer Systems* 4 4 (November), 273–98.

Classic survey paper of shared-bus cache coherence protocols.

Arpaci-Dusseau, A., R. Arpaci-Dusseau, D. Culler, J. Hellerstein, and D. Patterson [1997]. "High-performance sorting on networks of workstations," Proc. ACM SIG MOD/PODS Conference on Management of Data, Tucson, AZ (May), 12–15.

How a world record sort was performed on a cluster, including architecture critique of the workstation and network interface. By April 1, 1997, they pushed the record to 8.6 GB in 1 minute and 2.2 seconds to sort 100 MB.

Asanovic, K., R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. [2006]. "The landscape of parallel computing research: A view from Berkeley." Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley (December 18).

Nicknamed the "Berkeley View," this report lays out the landscape of the multicore challenge.

Bailey, D. H., E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. [1991]. "The NAS parallel benchmarks—summary and preliminary results," *Proceedings of the 1991 ACM/IEEE conference on Super-computing* (August).

Describes the NAS parallel benchmarks.

Bell, C. G. [1985]. "Multis: A new class of multiprocessor computers", *Science* 228(April 26), 462–467.

Distinguishes shared address and nonshared address multiprocessors based on micro processors.

Bienia, C., S. Kumar, J. P. Singh, and K. Li [2008]. "The PARSEC benchmark suite: characterization and architectural implications," Princeton University Technical Report TR-81 1-008 (January).

Describes the PARSEC parallel benchmarks. Also see http://parsec.cs.princeton.edu/.

Bouknight, W. J., Denenberg, S. A., McIntyre, D. E., Randall, J. M., Sameh, A. H., and Slotnick, D. L. [1972]. The Illiac IV system, *Proceedings of the IEEE*, 60(4), 369–388.

This describes the most infamous SIMD supercomputer.

Culler, D. E. and J. P. Singh, with A. Gupta [1998]. *Parallel Computer Architecture*, Morgan Kaufmann, San Francisco.

A textbook on parallel computers.

Dongarra, J. J., J. R. Bunch, G. B. Moler, G. W. Stewart [1979]. *LINPACK Users' Guide*, Society for Industrial Mathematics.

The original document describing Linpack, which became a widely used parallel bench mark.

Falk, H. [1976]. "Reaching for the gigaflop", *IEEE Spectrum* 13: 10 (October), 65–70.

Chronicles the sad story of the Illiac IV: four times the cost and less than one-tenth the performance of original goals.

Flynn, M. J. [1966]. "Very high-speed computing systems", *Proc. IEEE* 54 12 (December), 1901–09.

Classic article showing SISD/SIMD/MISD/MIMD classifications.

Hennessy, J. and D. Patterson [2003]. Chapters 6 and 8 in *Computer Architecture: A Quantitative Approach*, third edition, Morgan Kaufmann Publishers, San Francisco.

A more in-depth coverage of a variety of multiprocessor and cluster topics, including programs and measurements.

Henning, J. L. [2007]. "SPEC CPU suite growth: an historical perspective", *Computer Architecture News* Vol. 35, no. 1 (March).

Gives the history of SPEC, including the use of SPECrate to measure performance on independent jobs, which is being used as a parallel benchmark.

Hillis, W. D. [1989]. *The connection machine*. The MIT Press.

PhD Dissertation that makes case for 1-bit SIMD computer.

Hord, R. M. [1982]. *The Illiac-IV, the First Supercomputer*, Computer Science Press, Rockville, MD.

A historical accounting of the Illiac IV project.

Hwang, K. [1993]. *Advanced Computer Architecture with Parallel Programming*, McGraw-Hill, New York.

Another textbook covering parallel computers.

Kozyrakis, C. and D. Patterson [2003]. "Scalable vector processors for embedded systems", *IEEE Micro* 23:6 (November–December), 36–45.

Examination of a vector architecture for the MIPS instruction set in media and signal processing.

Menabrea, L. F. [1842]. "Sketch of the analytical engine invented by Charles Babbage", *Bibliothèque Universelle de Genève* (October).

Certainly the earliest reference on multiprocessors, this mathematician made this comment while translating papers on Babbage's mechanical computer.

Pfister, G. F. [1998]. *In Search of Clusters: The Coming Battle in Lowly Parallel Computing*, second edition, Prentice Hall, Upper Saddle River, NJ.

An entertaining book that advocates clusters and is critical of NUMA multiprocessors.

Regnier, G., S. Makineni, I. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, and A. Foong [2004]. TCP onloading for data center servers. *Computer*, 37(11), 48–58.

A paper describing benefits of doing TCP/IP inside servers vs. external hardware.

Seitz, C. [1985]. "The Cosmic Cube", *Comm. ACM* 28 1 (January), 22–31.

A tutorial article on a parallel processor connected via a hypertree. The Cosmic Cube is the ancestor of the Intel supercomputers.

Slotnick, D. L. [1982]. "The conception and development of parallel processors—a personal memoir", *Annals of the History of Computing* 4: 1 (January), 20–30.

Recollections of the beginnings of parallel processing by the architect of the Illiac I V.

Williams, S., J. Carter, L. Oliker, J. Shalf, and K. Yelick [2008]. "Lattice Boltzmann simulation optimization on leading multicore platforms," *International Parallel & Distributed Processing Symposium (IPDPS)*.

Paper containing the results of the four multicores for LBMHD.

Williams, S., L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel [2007]. "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," *Supercomputing* (SC).

Paper containing the results of the four multicores for SPmV.

Williams, S. [2008]. *Autotuning Performance of Multicore Computers*, Ph.D. Dissertation, U.C. Berkeley.

Dissertation containing the roofline model.

Woo, S. C., M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. "The SPLASH-2 programs: characterization and methodological considerations," *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA '95)*, May, 24–36.

Paper describing the second version of the Stanford parallel benchmarks.

6.1.2 [5] <§6.2> Next, consider which of the activities could be carried out concurrently (e.g., eating breakfast and listening to the news). For each of your activities, describe which other activity could be paired with this activity.

6.1.3 [5] <§6.2> For Exercise 6.1.2, what could we change about current systems (e.g., showers, clothes, TVs, cars) so that we could perform more tasks in parallel?

6.1.4 [5] <§6.2> Estimate how much shorter time it would take to carry out these activities if you tried to carry out as many tasks in parallel as possible.

6.2 You are trying to bake three blueberry pound cakes. Cake ingredients are as follows:

1 cup butter, softened
1 cup sugar
4 large eggs
1 teaspoon vanilla extract
1/2 teaspoon salt
1/4 teaspoon nutmeg
1 1/2 cups flour
1 cup blueberries

The recipe for a single cake is as follows:

> Step 1: Preheat oven to 325°F (160°C). Grease and flour your cake pan.
>
> Step 2: In large bowl, beat together with a mixer butter and sugar at medium speed until light and fluffy. Add eggs, vanilla, salt and nutmeg. Beat until thoroughly blended. Reduce mixer speed to low and add flour, 1/2 cup at a time, beating just until blended.
>
> Step 3: Gently fold in blueberries. Spread evenly in prepared baking pan. Bake for 60 minutes.

6.2.1 [5] <§6.2> Your job is to cook three cakes as efficiently as possible. Assuming that you only have one oven large enough to hold one cake, one large bowl, one cake pan, and one mixer, come up with a schedule to make three cakes as quickly as possible. Identify the bottlenecks in completing this task.

6.2.2 [5] <§6.2> Assume now that you have three bowls, three cake pans and three mixers. How much faster is the process now that you have additional resources?

6.2.3 [5] <§6.2> Assume now that you have two friends that will help you cook, and that you have a large oven that can accommodate all three cakes. How will this change the schedule you arrived at in Exercise 6.2.1 above?

6.2.4 [5] <§6.2> Compare the cake-making task to computing three iterations of a loop on a parallel computer. Identify data-level parallelism and task-level parallelism in the cake-making loop.

6.3 Many computer applications involve searching through a set of data and sorting the data. A number of efficient searching and sorting algorithms have been devised in order to reduce the runtime of these tedious tasks. In this problem we will consider how best to parallelize these tasks.

6.3.1 [10] <§6.2> Consider the following binary search algorithm (a classic divide and conquer algorithm) that searches for a value *X* in a sorted N-element array A and returns the index of matched entry:

```
BinarySearch(A[0..N-1], X) {
    low = 0
    high = N -1
    while (low <= high) {
        mid = (low + high) / 2
        if (A[mid] >X)
            high = mid -1
        else if (A[mid] <X)
            low = mid + 1
        else
            return mid // found
    }
    return -1 // not found
}
```

Assume that you have Y cores on a multi-core processor to run BinarySearch. Assuming that Y is much smaller than N, express the speed-up factor you might expect to obtain for values of Y and N. Plot these on a graph.

6.3.2 [5] <§6.2> Next, assume that Y is equal to N. How would this affect your conclusions in your previous answer? If you were tasked with obtaining the best speed-up factor possible (i.e., strong scaling), explain how you might change this code to obtain it.

6.4 Consider the following piece of C code:

```
for (j=2;j<=1000;j++)
    D[j] = D[j-1]+D[j-2];
```

The RISC-V code corresponding to the above fragment is:

```
        li      x5, 8000
        add     x12, x10, x5
        addi    x11, x10, 16
LOOP:   fld     f0, -16(x11)
        fld     f1, -8(x11)

        fadd.d  f2, f0, f1
        fsd     f2, 0(x11)
        addi    x11, x11, 8
        ble     x11, x12, LOOP
```

The latency of an instruction is the number of cycles that must come between that instruction and an instruction using the result. Assume floating point instructions have the following associated latencies (in cycles):

fadd.d	fld	fsd
4	6	1

6.4.1 [10] <§6.2> How many cycles does it take to execute this code?

6.4.2 [10] <§6.2> Re-order the code to reduce stalls. Now, how many cycles does it take to execute this code? (Hint: You can remove additional stalls by changing the offset on the `fsd` instruction.)

6.4.3 [10] <§6.2> When an instruction in a later iteration of a loop depends upon a data value produced in an earlier iteration of the same loop, we say that there is a *loop-carried dependence* between iterations of the loop. Identify the loop-carried dependences in the above code. Identify the dependent program variable and assembly-level registers. You can ignore the loop induction variable `j`.

6.4.4 [15] <§6.2> Rewrite the code by using registers to carry the data between iterations of the loop (as opposed to storing and re-loading the data from main memory). Show where this code stalls and calculate the number of cycles required to execute. Note that for this problem you will need to use the assembler pseudo-instruction "`fmv.d rd, rs1`", which writes the value of floating-point register rs1 into floating-point register rd. Assume that `fmv.d` executes in a single cycle.

6.4.5 [10] <§6.2> Loop unrolling was described in Chapter 4. Unroll and optimize the loop above so that each unrolled loop handles three iterations of the original loop. Show where this code stalls and calculate the number of cycles required to execute.

6.4.6 [10] <§6.2> The unrolling from Exercise 6.4.5. works nicely because we happen to want a multiple of three iterations. What happens if the number of iterations is not known at compile time? How can we efficiently handle a number of iterations that isn't a multiple of the number of iterations per unrolled loop?

6.4.7 [15] <§6.2> Consider running this code on a two-node distributed memory message passing system. Assume that we are going to use message passing as described in Section 6.7, where we introduce a new operation send (`x, y`) that sends to node x the value y, and an operation `receive( )` that waits for the value being sent to it. Assume that send operations take one cycle to issue (i.e., later instructions on the same node can proceed on the next cycle), but take several cycles to be received on the receiving node. Receive instructions stall execution on the node where they are executed until they receive a message. Can you use such a system to speed up the code for this exercise? If so, what is the maximum latency for receiving information that can be tolerated? If not, why not?

6.5 Consider the following recursive mergesort algorithm (another classic divide and conquer algorithm). Mergesort was first described by John Von Neumann in 1945. The basic idea is to divide an unsorted list *x* of *m* elements into two sublists of about half the size of the original list. Repeat this operation on each sublist, and continue until we have lists of size 1 in length. Then starting with sublists of length 1, "merge" the two sublists into a single sorted list.

```
Mergesort(m)
    var list left, right, result
    if length(m) ≤ 1
        return m
    else
        var middle = length(m) / 2
        for each x in m up to middle
            add x to left
        for each x in m after middle
            add x to right
        left = Mergesort(left)
        right = Mergesort(right)
        result = Merge(left, right)
        return result
```

The merge step is carried out by the following code:

```
Merge(left,right)
    var list result
    while length(left) >0 and length(right) > 0
        if first(left) ≤ first(right)
            append first(left) to result
            left = rest(left)
        else
            append first(right) to result
            right = rest(right)
    if length(left) >0
        append rest(left) to result
    if length(right) >0
        append rest(right) to result
    return result
```

6.5.1 [10] <§6.2> Assume that you have Y cores on a multicore processor to run Mergesort. Assuming that Y is much smaller than length (m), express the speed-up factor you might expect to obtain for values of Y and length (m). Plot these on a graph.

6.5.2 [10] <§6.2> Next, assume that Y is equal to length (m). How would this affect your conclusions in your previous answer? If you were tasked with obtaining the best speed-up factor possible (i.e., strong scaling), explain how you might change this code to obtain it.

6.6 Matrix multiplication plays an important role in a number of applications. Two matrices can only be multiplied if the number of columns of the first matrix is equal to the number of rows in the second.

Let's assume we have an $m \times n$ matrix A and we want to multiply it by an $n \times p$ matrix B. We can express their product as an $m \times p$ matrix denoted by AB (or $A \cdot B$). If we assign $C = AB$, and $c_{i,j}$ denotes the entry in C at position (i, j), then for each element i and j with $1 \leq i \leq m$ and $1 \leq j \leq p$ $c_{i,j} = \sum_{k=1}^{n} a_{i,k} \times b_{k,j}$. Now we want to see if we can parallelize the computation of C. Assume that matrices are laid out in memory sequentially as follows: $a_{1,1}$, $a_{2,1}$, $a_{3,1}$, $a_{4,1}$, …, etc.

6.6.1 [10] <§6.5> Assume that we are going to compute C on both a single-core shared-memory machine and a four-core shared-memory machine. Compute the speed-up we would expect to obtain on the four-core machine, ignoring any memory issues.

6.6.2 [10] <§6.5> Repeat Exercise 6.6.1, assuming that updates to C incur a cache miss due to false sharing when consecutive elements are in a row (i.e., index i) are updated.

6.6.3 [10] <§6.5> How would you fix the false sharing issue that can occur?

6.7 Consider the following portions of two different programs running at the same time on four processors in a *symmetric multicore processor* (SMP). Assume that before this code is run, both x and y are 0.

Core 1: `x = 2;`

Core 2: `y = 2;`

Core 3: `w = x + y + 1;`

Core 4: `z = x + y;`

6.7.1 [10] <§6.5> What are all the possible resulting values of `w`, `x`, `y`, and `z`? For each possible outcome, explain how we might arrive at those values. You will need to examine all possible interleavings of instructions.

6.7.2 [5] <§6.5> How could you make the execution more deterministic so that only one set of values is possible?

6.8 The dining philosopher's problem is a classic problem of synchronization and concurrency. The general problem is stated as philosophers sitting at a round table doing one of two things: eating or thinking. When they are eating, they are not thinking, and when they are thinking, they are not eating. There is a bowl of pasta in the center. A fork is placed in between each philosopher. The result is that each philosopher has one fork to her left and one fork to her right. Given the nature of eating pasta, the philosopher needs two forks to eat, and can only use the forks on her immediate left and right. The philosophers do not speak to one another.

6.8.1 [10] <§6.7> Describe the scenario where none of philosophers ever eats (i.e., starvation). What is the sequence of events that happen that lead up to this problem?

6.8.2 [10] <§6.7> Describe how we can solve this problem by introducing the concept of a priority. Can we guarantee that we will treat all the philosophers fairly? Explain.

Now assume we hire a waiter who is in charge of assigning forks to philosophers. Nobody can pick up a fork until the waiter says they can. The waiter has global knowledge of all forks. Further, if we impose the policy that philosophers will always request to pick up their left fork before requesting to pick up their right fork, then we can guarantee to avoid deadlock.

6.8.3 [10] <§6.7> We can implement requests to the waiter as either a queue of requests or as a periodic retry of a request. With a queue, requests are handled in the order they are received. The problem with using the queue is that we may not always be able to service the philosopher whose request is at the head of the queue (due to the unavailability of resources). Describe a scenario with five philosophers where a queue is provided, but service is not granted even though there are forks available for another philosopher (whose request is deeper in the queue) to eat.

6.8.4 [10] <§6.7> If we implement requests to the waiter by periodically repeating our request until the resources become available, will this solve the problem described in Exercise 6.8.3? Explain.

6.9 Consider the following three CPU organizations:

CPU SS: A two-core superscalar microprocessor that provides out-of-order issue capabilities on two *function units* (FUs). Only a single thread can run on each core at a time.

CPU MT: A fine-grained multithreaded processor that allows instructions from two threads to be run concurrently (i.e., there are two functional units), though only instructions from a single thread can be issued on any cycle.

CPU SMT: An SMT processor that allows instructions from two threads to be run concurrently (i.e., there are two functional units), and instructions from either or both threads can be issued to run on any cycle.

Assume we have two threads X and Y to run on these CPUs that include the following operations:

Thread X	Thread Y
A1 – takes three cycles to execute	B1 – take two cycles to execute
A2 – no dependences	B2 – conflicts for a functional unit with B1
A3 – conflicts for a functional unit with A1	B3 – depends on the result of B2
A4 – depends on the result of A3	B4 – no dependences and takes two cycles to execute

Assume all instructions take a single cycle to execute unless noted otherwise or they encounter a hazard.

6.9.1 [10] <§6.4> Assume that you have one SS CPU. How many cycles will it take to execute these two threads? How many issue slots are wasted due to hazards?

6.9.2 [10] <§6.4> Now assume you have two SS CPUs. How many cycles will it take to execute these two threads? How many issue slots are wasted due to hazards?

6.9.3 [10] <§6.4> Assume that you have one MT CPU. How many cycles will it take to execute these two threads? How many issue slots are wasted due to hazards?

6.9.4 [10] <§6.4> Assume you have one SMT CPU. How many cycles will it take to execute the two threads? How many issue slots are wasted due to hazards?

6.10 Virtualization software is being aggressively deployed to reduce the costs of managing today's high-performance servers. Companies like VMWare, Microsoft, and IBM have all developed a range of virtualization products. The general concept, described in Chapter 5, is that a hypervisor layer can be introduced between the hardware and the operating system to allow multiple operating systems to share the same physical hardware. The hypervisor layer is then responsible for allocating CPU and memory resources, as well as handling services typically handled by the operating system (e.g., I/O).

Virtualization provides an abstract view of the underlying hardware to the hosted operating system and application software. This will require us to rethink how multi-core and multiprocessor systems will be designed in the future to support the sharing of CPUs and memories by a number of operating systems concurrently.

6.10.1 [30] <§6.4> Select two hypervisors on the market today, and compare and contrast how they virtualize and manage the underlying hardware (CPUs and memory).

6.10.2 [15] <§6.4> Discuss what changes may be necessary in future multi-core CPU platforms in order to better match the resource demands placed on these systems. For instance, can multithreading play an effective role in alleviating the competition for computing resources?

6.11 We would like to execute the loop below as efficiently as possible. We have two different machines, a MIMD machine and a SIMD machine.

```
for (i=0; i<2000; i++)
  for (j=0; j<3000; j++)
       X_array[i][j] = Y_array[j][i] + 200;
```

6.11.1 [10] <§6.3> For a four CPU MIMD machine, show the sequence of RISC-V instructions that you would execute on each CPU. What is the speed-up for this MIMD machine?

6.11.2 [20] <§6.3> For an eight-wide SIMD machine (i.e., eight parallel SIMD functional units), write an assembly program in using your own SIMD extensions to RISC-V to execute the loop. Compare the number of instructions executed on the SIMD machine to the MIMD machine.

6.12 A systolic array is an example of an MISD machine. A systolic array is a pipeline network or "wavefront" of data processing elements. Each of these elements does not need a program counter since execution is triggered by the arrival of data. Clocked systolic arrays compute in "lock-step" with each processor undertaking alternate compute and communication phases.

6.12.1 [10] <§6.3> Consider proposed implementations of a systolic array (you can find these on the Internet or in technical publications). Then attempt to program the loop provided in Exercise 6.11 using this MISD model. Discuss any difficulties you encounter.

6.12.2 [10] <§6.3> Discuss the similarities and differences between an MISD and SIMD machines. Answer this question in terms of data-level parallelism.

6.13 Assume we want to execute the DAXPY loop shown on page 501 in RISC-V vector assembly on the NVIDIA 8800 GTX GPU described in this chapter. In this problem, we will assume that all math operations are performed on single-precision floating-point numbers (we will rename the loop SAXPY). Assume that instructions take the following number of cycles to execute.

Loads	Stores	Add.S	Mult.S
5	2	3	4

6.13.1 [20] <§6.6> Describe how you will constructs warps for the SAXPY loop to exploit the eight cores provided in a single multiprocessor.

6.14 Download the CUDA Toolkit and SDK from https://developer.nvidia.com/cuda-toolkit. Make sure to use the "emurelease" (Emulation Mode) version of the code. (You will not need actual NVIDIA hardware for this assignment.) Build the example programs provided in the SDK, and confirm that they run on the emulator.

6.14.1 [90] <§6.6> Using the "template" SDK sample as a starting point, write a CUDA program to perform the following vector operations:

1) $a - b$ (vector-vector subtraction)

2) a · b (vector dot product)

The dot product of two vectors $a = [a_1, a_2, \ldots, a_n]$ and $b = [b_1, b_2, \ldots, b_n]$ is defined as:

$$a \cdot b = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

Submit code for each program that demonstrates each operation and verifies the correctness of the results.

6.14.2 [90] <§6.6> If you have GPU hardware available, complete a performance analysis on your program, examining the computation time for the GPU and a CPU version of your program for a range of vector sizes. Explain any results you see.

6.15 AMD has recently announced integrating a graphics processing unit with their x86 cores into a single package (though with different clocks for each of the cores). This is an example of a heterogeneous multiprocessor system. One of the key design points is to allow for fast data communication between the CPU and the GPU. Before AMD's Fusion architecture, communications were needed between discrete CPU and GPU chips. Presently, the plan is to use multiple (at least 16) PCI express channels to facilitate intercommunication.

6.15.1 [25] <§6.6> Compare the bandwidth and latency associated with these two interconnect technologies.

6.16 Refer to Figure 6.14b, which shows an n-cube interconnect topology of order 3 that interconnects eight nodes. One attractive feature of an n-cube interconnection network topology is its ability to sustain broken links and still provide connectivity.

6.16.1 [10] <§6.8> Develop an equation that computes how many links in the n-cube (where n is the order of the cube) can fail and we can still guarantee an unbroken link will exist to connect any node in the n-cube.

6.16.2 [10] <§6.8> Compare the resiliency to failure of n-cube to a fully connected interconnection network. Plot a comparison of reliability as a function of the added number of links for the two topologies.

6.17 Benchmarking is a field of study that involves identifying representative workloads to run on specific computing platforms in order to be able to objectively compare performance of one system to another. In this exercise we will compare two classes of benchmarks: the Whetstone CPU benchmark and the PARSEC Benchmark suite. Select one program from PARSEC. All programs should be freely available on the Internet. Consider running multiple copies of Whetstone versus running the PARSEC Benchmark on any of the systems described in Section 6.11.

6.17.1 [60] <§6.10> What is inherently different between these two classes of workload when run on these multi-core systems?

6.17.2 [60] <§6.10> In terms of the Roofline Model, how dependent will the results you obtain when running these benchmarks be on the amount of sharing and synchronization present in the workload used?

6.18 When performing computations on sparse matrices, latency in the memory hierarchy becomes much more of a factor. Sparse matrices lack the spatial locality in the datastream typically found in matrix operations. As a result, new matrix representations have been proposed.

One of the earliest sparse matrix representations is the Yale Sparse Matrix Format. It stores an initial sparse $m \times n$ matrix, M in row form using three one-dimensional arrays. Let R be the number of nonzero entries in M. We construct an array A of length R that contains all nonzero entries of M (in left-to-right top-to-bottom order). We also construct a second array IA of length $m+1$ (i.e., one entry per row, plus one). $IA(i)$ contains the index in A of the first nonzero element of row i. Row i of the original matrix extends from $A(IA(i))$ to $A(IA(i+1)-1)$. The third array, JA, contains the column index of each element of A, so it also is of length R.

6.18.1 [15] <§6.10> Consider the sparse matrix X below and write C code that would store this code in Yale Sparse Matrix Format.

```
Row 1 [1, 2, 0, 0, 0, 0]
Row 2 [0, 0, 1, 1, 0, 0]
Row 3 [0, 0, 0, 0, 9, 0]
Row 4 [2, 0, 0, 0, 0, 2]
Row 5 [0, 0, 3, 3, 0, 7]
Row 6 [1, 3, 0, 0, 0, 1]
```

6.18.2 [10] <§6.10> In terms of storage space, assuming that each element in matrix X is single-precision floating point, compute the amount of storage used to store the matrix above in Yale Sparse Matrix Format.

6.18.3 [15] <§6.10> Perform matrix multiplication of matrix X by matrix Y shown below.

```
[2, 4, 1, 99, 7, 2]
```

Put this computation in a loop, and time its execution. Make sure to increase the number of times this loop is executed to get good resolution in your timing measurement. Compare the runtime of using a naïve representation of the matrix, and the Yale Sparse Matrix Format.

6.18.4 [15] <§6.10> Can you find a more efficient sparse matrix representation (in terms of space and computational overhead)?

6.19 In future systems, we expect to see heterogeneous computing platforms constructed out of heterogeneous CPUs. We have begun to see some appear in the embedded processing market in systems that contain both floating-point DSPs and microcontroller CPUs in a multichip module package.

Assume that you have three classes of CPU:

CPU A—A moderate-speed multi-core CPU (with a floating-point unit) that can execute multiple instructions per cycle.

CPU B—A fast single-core integer CPU (i.e., no floating-point unit) that can execute a single instruction per cycle.

CPU C—A slow vector CPU (with floating-point capability) that can execute multiple copies of the same instruction per cycle.

Assume that our processors run at the following frequencies:

CPU A	CPU B	CPU C
1 GHz	3 GHz	250 MHz

CPU A can execute two instructions per cycle, CPU B can execute one instruction per cycle, and CPU C can execute eight instructions (through the same instruction) per cycle. Assume all operations can complete execution in a single cycle of latency without any hazards.

All three CPUs have the ability to perform integer arithmetic, though CPU B cannot perform floating point arithmetic. CPU A and B have an instruction set similar to a RISC-V processor. CPU C can only perform floating point add and subtract operations, as well as memory loads and stores. Assume all CPUs have access to shared memory and that synchronization has zero cost.

The task at hand is to compare two matrices X and Y that each contain 1024×1024 floating-point elements. The output should be a count of the number of indices where the value in X was larger or equal to the value in Y.

6.19.1 [10] <§6.11> Describe how you would partition the problem on the three different CPUs to obtain the best performance.

6.19.2 [10] <§6.11> What kind of instruction would you add to the vector CPU C to obtain better performance?

6.20 This question looks at the amount of queuing that is occurring in the system given a maximum transaction processing rate, and the latency observed on average for a transaction. The latency includes both the service time (which is computed by the maximum rate) and the queue time.

Assume a quad-core computer system can process database queries at a steady state maximum rate of rate requests per second. Also assume that each transaction takes, on average, lat ms to process. For each of the pairs in the table, answer the following questions:

Average Transaction Latency	Maximum transaction processing rate
1 ms	5000/sec
2 ms	5000/sec
1 ms	10,000/sec
2 ms	10,000/sec

For each of the pairs in the table, answer the following questions:

6.20.1 [10] <§6.11> On average, how many requests are being processed at any given instant?

6.20.2 [10] <§6.11> If we move to an eight-core system, ideally, what will happen to the system throughput (i.e., how many queries/second will the computer process)?

6.20.3 [10] <§6.11> Discuss why we rarely obtain this kind of speed-up by simply increasing the number of cores.

Answers to Check Yourself

§6.1, page 494: False. Task-level parallelism can help sequential applications and sequential applications can be made to run on parallel hardware, although it is more challenging.
§6.2, page 499: False. *Weak* scaling can compensate for a serial portion of the program that would otherwise limit scalability, but not so for strong scaling.
§6.3, page 504: True, but they are missing useful vector features like gather-scatter and vector length registers that improve the efficiency of vector architectures. (As an elaboration in this section mentions, the AVX2 SIMD extensions offers indexed loads via a gather operation but *not* scatter for indexed stores. The Haswell generation x86 microprocessor is the first to support AVX2.)
§6.4, page 509: 1. True. 2. True.
§6.5, page 513: False. Since the shared address is a *physical* address, multiple tasks each in their own *virtual* address spaces can run well on a shared memory multiprocessor.
§6.6, page 521: False. Graphics DRAM chips are prized for their higher bandwidth.
§6.7, page 526: 1. False. Sending and receiving a message is an implicit synchronization, as well as a way to share data. 2. True.
§6.8, page 528: True.
§6.10, page 540: True. We likely need innovation at all levels of the hardware and software stack for parallel computing to succeed.

A

APPENDIX

The Basics of Logic Design

I always loved that word, Boolean.

Claude Shannon
IEEE Spectrum, April 1992
(Shannon's master's thesis showed that the algebra invented by George Boole in the 1800s could represent the workings of electrical switches.)

A.1 **Introduction** A-3
A.2 **Gates, Truth Tables, and Logic Equations** A-4
A.3 **Combinational Logic** A-9
A.4 **Using a Hardware Description Language** A-20
A.5 **Constructing a Basic Arithmetic Logic Unit** A-26
A.6 **Faster Addition: Carry Lookahead** A-37
A.7 **Clocks** A-47

A.8 Memory Elements: Flip-Flops, Latches, and Registers A-49
A.9 Memory Elements: SRAMs and DRAMs A-57
A.10 Finite-State Machines A-66
A.11 Timing Methodologies A-71
A.12 Field Programmable Devices A-77
A.13 Concluding Remarks A-78
A.14 Exercises A-79

A.1 Introduction

This appendix provides a brief discussion of the basics of logic design. It does not replace a course in logic design, nor will it enable you to design significant working logic systems. If you have little or no exposure to logic design, however, this appendix will provide sufficient background to understand all the material in this book. In addition, if you are looking to understand some of the motivation behind how computers are implemented, this material will serve as a useful introduction. If your curiosity is aroused but not sated by this appendix, the references at the end provide several additional sources of information.

Section A.2 introduces the basic building blocks of logic, namely, *gates*. Section A.3 uses these building blocks to construct simple *combinational* logic systems, which contain no memory. If you have had some exposure to logic or digital systems, you will probably be familiar with the material in these first two sections. Section A.5 shows how to use the concepts of Sections A.2 and A.3 to design an ALU for the RISC-V processor. Section A.6 shows how to make a fast adder, and

may be safely skipped if you are not interested in this topic. Section A.7 is a short introduction to the topic of clocking, which is necessary to discuss how memory elements work. Section A.8 introduces memory elements, and Section A.9 extends it to focus on random access memories; it describes both the characteristics that are important to understanding how they are used, as discussed in Chapter 4, and the background that motivates many of the aspects of memory hierarchy design discussed in Chapter 5. Section A.10 describes the design and use of finite-state machines, which are sequential logic blocks. If you intend to read **Appendix C**, you should thoroughly understand the material in Sections A.2 through A.10. If you intend to read only the material on control in Chapter 4, you can skim the appendices; however, you should have some familiarity with all the material except Section A.11. Section A.11 is intended for those who want a deeper understanding of clocking methodologies and timing. It explains the basics of how edge-triggered clocking works, introduces another clocking scheme, and briefly describes the problem of synchronizing asynchronous inputs.

Throughout this appendix, where it is appropriate, we also include segments to demonstrate how logic can be represented in Verilog, which we introduce in Section A.4. A more extensive and complete Verilog tutorial is available online on the Companion Web site for this book.

A.2 Gates, Truth Tables, and Logic Equations

The electronics inside a modern computer are *digital*. Digital electronics operate with only two voltage levels of interest: a high voltage and a low voltage. All other voltage values are temporary and occur while transitioning between the values. (As we discuss later in this section, a possible pitfall in digital design is sampling a signal when it not clearly either high or low.) The fact that computers are digital is also a key reason they use binary numbers, since a binary system matches the underlying abstraction inherent in the electronics. In various logic families, the values and relationships between the two voltage values differ. Thus, rather than refer to the voltage levels, we talk about signals that are (logically) true, or 1, or are **asserted**; or signals that are (logically) false, or 0, or are **deasserted**. The values 0 and 1 are called *complements* or *inverses* of one another.

asserted signal A signal that is (logically) true, or 1.

deasserted signal A signal that is (logically) false, or 0.

Logic blocks are categorized as one of two types, depending on whether they contain memory. Blocks without memory are called *combinational*; the output of a combinational block depends only on the current input. In blocks with memory, the outputs can depend on both the inputs and the value stored in memory, which is called the *state* of the logic block. In this section and the next, we will focus

only on **combinational logic**. After introducing different memory elements in Section A.8, we will describe how **sequential logic**, which is logic including state, is designed.

combinational logic
A logic system whose blocks do not contain memory and hence compute the same output given the same input.

sequential logic
A group of logic elements that contain memory and hence whose value depends on the inputs as well as the current contents of the memory.

Truth Tables

Because a combinational logic block contains no memory, it can be completely specified by defining the values of the outputs for each possible set of input values. Such a description is normally given as a *truth table*. For a logic block with n inputs, there are 2^n entries in the truth table, since there are that many possible combinations of input values. Each entry specifies the value of all the outputs for that particular input combination.

Truth Tables

EXAMPLE

Consider a logic function with three inputs, A, B, and C, and three outputs, D, E, and F. The function is defined as follows: D is true if at least one input is true, E is true if exactly two inputs are true, and F is true only if all three inputs are true. Show the truth table for this function.

ANSWER

The truth table will contain $2^3 = 8$ entries. Here it is:

Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

Truth tables can completely describe any combinational logic function; however, they grow in size quickly and may not be easy to understand. Sometimes we want to construct a logic function that will be 0 for many input combinations, and we use a shorthand of specifying only the truth table entries for the nonzero outputs. This approach is used in Chapter 4 and **Appendix C**.

Boolean Algebra

Another approach is to express the logic function with logic equations. This is done with the use of *Boolean algebra* (named after Boole, a 19th-century mathematician). In Boolean algebra, all the variables have the values 0 or 1 and, in typical formulations, there are three operators:

- The OR operator is written as +, as in $A + B$. The result of an OR operator is 1 if either of the variables is 1. The OR operation is also called a *logical sum*, since its result is 1 if either operand is 1.
- The AND operator is written as $\cdot$, as in $A \cdot B$. The result of an AND operator is 1 only if both inputs are 1. The AND operator is also called *logical product*, since its result is 1 only if both operands are 1.
- The unary operator NOT is written as $\overline{A}$. The result of a NOT operator is 1 only if the input is 0. Applying the operator NOT to a logical value results in an inversion or negation of the value (i.e., if the input is 0 the output is 1, and vice versa).

There are several laws of Boolean algebra that are helpful in manipulating logic equations.

- Identity law: $A + 0 = A$ and $A \cdot 1 = A$
- Zero and One laws: $A + 1 = 1$ and $A \cdot 0 = 0$
- Inverse laws: $A + \overline{A} = 1$ and $A \cdot \overline{A} = 0$
- Commutative laws: $A + B = B + A$ and $A \cdot B = B \cdot A$
- Associative laws: $A + (B + C) = (A + B) + C$ and $A \cdot (B \cdot C) = (A \cdot B) \cdot C$
- Distributive laws: $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$ and $A + (B \cdot C) = (A + B) \cdot (A + C)$

In addition, there are two other useful theorems, called DeMorgan's laws, that are discussed in more depth in the exercises.

Any set of logic functions can be written as a series of equations with an output on the left-hand side of each equation and a formula consisting of variables and the three operators above on the right-hand side.

Logic Equations

EXAMPLE

Show the logic equations for the logic functions, *D*, *E*, and *F*, described in the previous example.

ANSWER

Here's the equation for *D:*

$$D = A + B + C$$

F is equally simple:

$$F = A \cdot B \cdot C$$

E is a little tricky. Think of it in two parts: what must be true for *E* to be true (two of the three inputs must be true), and what cannot be true (all three cannot be true). Thus we can write *E* as

$$E = ((A \cdot B) + (A \cdot C) + (B \cdot C)) \cdot \overline{(A \cdot B \cdot C)}$$

We can also derive *E* by realizing that *E* is true only if exactly two of the inputs are true. Then we can write *E* as an OR of the three possible terms that have two true inputs and one false input:

$$E = (A \cdot B \cdot \overline{C}) + (A \cdot C \cdot \overline{B}) + (B \cdot C \cdot \overline{A})$$

Proving that these two expressions are equivalent is explored in the exercises.

In Verilog, we describe combinational logic whenever possible using the assign statement, which is described beginning on page A-23. We can write a definition for *E* using the Verilog exclusive-OR operator as `assign E = (A & (B ^ C)) | (B & C & ~A)`, which is yet another way to describe this function. *D* and *F* have even simpler representations, which are just like the corresponding C code: `assign D = A | B | C` and `assign F = A & B & C`.

Gates

gate A device that implements basic logic functions, such as AND or OR.

Logic blocks are built from **gates** that implement basic logic functions. For example, an AND gate implements the AND function, and an OR gate implements the OR function. Since both AND and OR are commutative and associative, an AND or an OR gate can have multiple inputs, with the output equal to the AND or OR of all the inputs. The logical function NOT is implemented with an inverter that always has a single input. The standard representation of these three logic building blocks is shown in Figure A.2.1.

Rather than draw inverters explicitly, a common practice is to add "bubbles" to the inputs or outputs of a gate to cause the logic value on that input line or output line to be inverted. For example, Figure A.2.2 shows the logic diagram for the function $\overline{A} + B$, using explicit inverters on the left and bubbled inputs and outputs on the right.

Any logical function can be constructed using AND gates, OR gates, and inversion; several of the exercises give you the opportunity to try implementing some common logic functions with gates. In the next section, we'll see how an implementation of any logic function can be constructed using this knowledge.

NOR gate An inverted OR gate.

NAND gate An inverted AND gate.

In fact, all logic functions can be constructed with only a single gate type, if that gate is inverting. The two common inverting gates are called **NOR** and **NAND** and correspond to inverted OR and AND gates, respectively. NOR and NAND gates are called *universal*, since any logic function can be built using this one gate type. The exercises explore this concept further.

Check Yourself

Are the following two logical expressions equivalent? If not, find a setting of the variables to show they are not:

- $(A \cdot B \cdot \overline{C}) + (A \cdot C \cdot \overline{B}) + (B \cdot C \cdot \overline{A})$
- $B \cdot (A \cdot \overline{C} + C \cdot \overline{A})$



FIGURE A.2.1 Standard drawing for an AND gate, OR gate, and an inverter, shown from left to right. The signals to the left of each symbol are the inputs, while the output appears on the right. The AND and OR gates both have two inputs. Inverters have a single input.

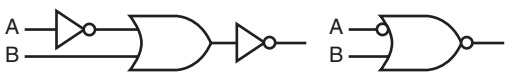


FIGURE A.2.2 Logic gate implementation of $\overline{\overline{A} + B}$ using explicit inverts on the left and bubbled inputs and outputs on the right. This logic function can be simplified to $A \cdot \overline{B}$ or in Verilog, `A & ~ B`.

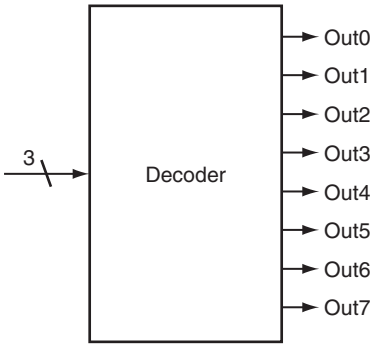
A.3 Combinational Logic

In this section, we look at a couple of larger logic building blocks that we use heavily, and we discuss the design of structured logic that can be automatically implemented from a logic equation or truth table by a translation program. Last, we discuss the notion of an array of logic blocks.

Decoders

One logic block that we will use in building larger components is a **decoder**. The most common type of decoder has an n-bit input and 2^n outputs, where only one output is asserted for each input combination. This decoder translates the n-bit input into a signal that corresponds to the binary value of the n-bit input. The outputs are thus usually numbered, say, Out0, Out1, …, Out$2^n - 1$. If the value of the input is i, then Out$i$ will be true and all other outputs will be false. Figure A.3.1 shows a 3-bit decoder and the truth table. This decoder is called a *3-to-8 decoder* since there are three inputs and eight (2^3) outputs. There is also a logic element called an *encoder* that performs the inverse function of a decoder, taking 2^n inputs and producing an n-bit output.

decoder A logic block that has an n-bit input and $2n$ outputs, where only one output is asserted for each input combination.



a. A 3-bit decoder

Inputs			Outputs							
12	11	10	Out7	Out6	Out5	Out4	Out3	Out2	Out1	Out0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

b. The truth table for a 3-bit decoder

FIGURE A.3.1 A 3-bit decoder has three inputs, called 12, 11, and 10, and $2^3 = 8$ outputs, called Out0 to Out7. Only the output corresponding to the binary value of the input is true, as shown in the truth table. The label 3 on the input to the decoder says that the input signal is 3 bits wide.

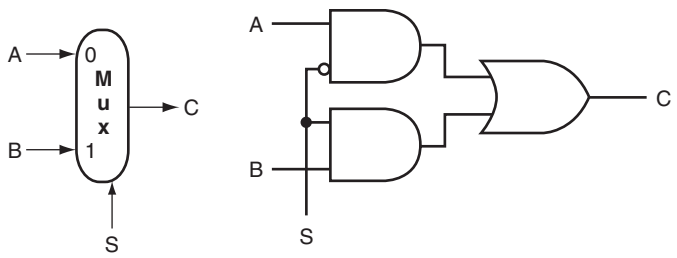


FIGURE A.3.2 A two-input multiplexor on the left and its implementation with gates on the right. The multiplexor has two data inputs (*A* and *B*), which are labeled *0* and *1*, and one selector input (*S*), as well as an output *C*. Implementing multiplexors in Verilog requires a little more work, especially when they are wider than two inputs. We show how to do this beginning on page A-23.

Multiplexors

selector value Also called **control value**. The control signal that is used to select one of the input values of a multiplexor as the output of the multiplexor.

One basic logic function that we use quite often in Chapter 4 is the *multiplexor*. A multiplexor might more properly be called a *selector*, since its output is one of the inputs that is selected by a control. Consider the two-input multiplexor. The left side of Figure A.3.2 shows this multiplexor has three inputs: two data values and a **selector** (or **control**) **value**. The selector value determines which of the inputs becomes the output. We can represent the logic function computed by a two-input multiplexor, shown in gate form on the right side of Figure A.3.2, as $C = (A \cdot S) + (B \cdot S)$.

Multiplexors can be created with an arbitrary number of data inputs. When there are only two inputs, the selector is a single signal that selects one of the inputs if it is true (1) and the other if it is false (0). If there are n data inputs, there will need to be $\lceil \log_2 n \rceil$ selector inputs. In this case, the multiplexor basically consists of three parts:

1. A decoder that generates n signals, each indicating a different input value
2. An array of n AND gates, each combining one of the inputs with a signal from the decoder
3. A single large OR gate that incorporates the outputs of the AND gates

To associate the inputs with selector values, we often label the data inputs numerically (i.e., 0, 1, 2, 3, …, $n-1$) and interpret the data selector inputs as a binary number. Sometimes, we make use of a multiplexor with undecoded selector signals.

Multiplexors are easily represented combinationally in Verilog by using *if* expressions. For larger multiplexors, *case* statements are more convenient, but care must be taken to synthesize combinational logic.

Two-Level Logic and PLAs

As pointed out in the previous section, any logic function can be implemented with only AND, OR, and NOT functions. In fact, a much stronger result is true. Any logic function can be written in a canonical form, where every input is either a true or complemented variable and there are only two levels of gates—one being AND and the other OR—with a possible inversion on the final output. Such a representation is called a *two-level representation*, and there are two forms, called **sum of products** and *product of sums*. A sum-of-products representation is a logical sum (OR) of products (terms using the AND operator); a product of sums is just the opposite. In our earlier example, we had two equations for the output E:

sum of products A form of logical representation that employs a logical sum (OR) of products (terms joined using the AND operator).

$$E = ((A \cdot B) + (A \cdot C) + (B \cdot C)) \cdot \overline{(A \cdot B \cdot C)}$$

and

$$E = (A \cdot B \cdot \overline{C}) + (A \cdot C \cdot \overline{B}) + (B \cdot C \cdot \overline{A})$$

This second equation is in a sum-of-products form: it has two levels of logic and the only inversions are on individual variables. The first equation has three levels of logic.

Elaboration: We can also write E as a product of sums:

$$E = \overline{(\overline{A} + \overline{B} + C) \cdot (\overline{A} + \overline{C} + B) \cdot (\overline{B} + C + A)}$$

To derive this form, you need to use *DeMorgan's theorems*, which are discussed in the exercises.

In this text, we use the sum-of-products form. It is easy to see that any logic function can be represented as a sum of products by constructing such a representation from the truth table for the function. Each truth table entry for which the function is true corresponds to a product term. The product term consists of a logical product of all the inputs or the complements of the inputs, depending on whether the entry in the truth table has a 0 or 1 corresponding to this variable. The logic function is the logical sum of the product terms where the function is true. This is more easily seen with an example.

EXAMPLE

Sum of Products

Show the sum-of-products representation for the following truth table for *D*.

Inputs		Outputs	
A	B	C	D
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

ANSWER

There are four product terms, since the function is true (1) for four different input combinations. These are:

$$\overline{A} \cdot \overline{B} \cdot C$$
$$\overline{A} \cdot B \cdot C$$
$$A \cdot \overline{B} \cdot \overline{C}$$
$$A \cdot B \cdot C$$

Thus, we can write the function for *D* as the sum of these terms:

$$D = (\overline{A} \cdot \overline{B} \cdot C) + (\overline{A} \cdot B \cdot \overline{C}) + (A \cdot \overline{B} \cdot \overline{C}) + (A \cdot B \cdot C)$$

Note that only those truth table entries for which the function is true generate terms in the equation.

programmable logic array (PLA) A structured-logic element composed of a set of inputs and corresponding input complements and two stages of logic: the first generates product terms of the inputs and input complements, and the second generates sum terms of the product terms. Hence, PLAs implement logic functions as a sum of products.

minterms Also called **product terms**. A set of logic inputs joined by conjunction (AND operations); the product terms form the first logic stage of the *programmable logic array* (PLA).

We can use this relationship between a truth table and a two-level representation to generate a gate-level implementation of any set of logic functions. A set of logic functions corresponds to a truth table with multiple output columns, as we saw in the example on page A-5. Each output column represents a different logic function, which may be directly constructed from the truth table.

The sum-of-products representation corresponds to a common structured-logic implementation called a **programmable logic array (PLA)**. A PLA has a set of inputs and corresponding input complements (which can be implemented with a set of inverters), and two stages of logic. The first stage is an array of AND gates that form a set of **product terms** (sometimes called **minterms**); each product term can consist of any of the inputs or their complements. The second stage is an array of OR gates, each of which forms a logical sum of any number of the product terms. Figure A.3.3 shows the basic form of a PLA.

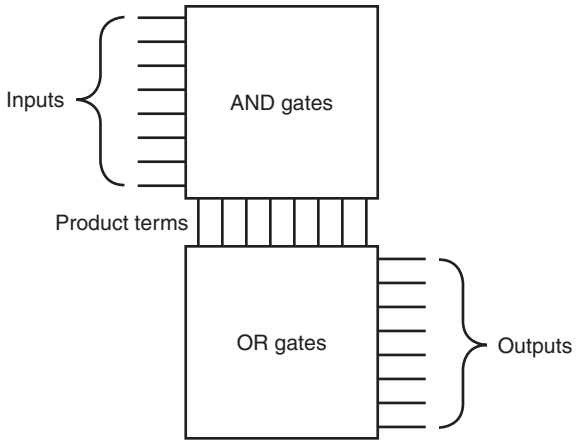


FIGURE A.3.3 The basic form of a PLA consists of an array of AND gates followed by an array of OR gates. Each entry in the AND gate array is a product term consisting of any number of inputs or inverted inputs. Each entry in the OR gate array is a sum term consisting of any number of these product terms.

A PLA can directly implement the truth table of a set of logic functions with multiple inputs and outputs. Since each entry where the output is true requires a product term, there will be a corresponding row in the PLA. Each output corresponds to a potential row of OR gates in the second stage. The number of OR gates corresponds to the number of truth table entries for which the output is true. The total size of a PLA, such as that shown in Figure A.3.3, is equal to the sum of the size of the AND gate array (called the *AND plane*) and the size of the OR gate array (called the *OR plane*). Looking at Figure A.3.3, we can see that the size of the AND gate array is equal to the number of inputs times the number of different product terms, and the size of the OR gate array is the number of outputs times the number of product terms.

A PLA has two characteristics that help make it an efficient way to implement a set of logic functions. First, only the truth table entries that produce a true value for at least one output have any logic gates associated with them. Second, each different product term will have only one entry in the PLA, even if the product term is used in multiple outputs. Let's look at an example.

PLAs

EXAMPLE

Consider the set of logic functions defined in the example on page A-5. Show a PLA implementation of this example for *D*, *E*, and *F*.

ANSWER

Here is the truth table we constructed earlier:

Inputs			Outputs		
A	*B*	*C*	*D*	*E*	*F*
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

Since there are seven unique product terms with at least one true value in the output section, there will be seven columns in the AND plane. The number of rows in the AND plane is three (since there are three inputs), and there are also three rows in the OR plane (since there are three outputs). Figure A.3.4 shows the resulting PLA, with the product terms corresponding to the truth table entries from top to bottom.

read-only memory (ROM) A memory whose contents are designated at creation time, after which the contents can only be read. ROM is used as structured logic to implement a set of logic functions by using the terms in the logic functions as address inputs and the outputs as bits in each word of the memory.

programmable ROM (PROM) A form of read-only memory that can be programmed when a designer knows its contents.

Rather than drawing all the gates, as we do in Figure A.3.4, designers often show just the position of AND gates and OR gates. Dots are used on the intersection of a product term signal line and an input line or an output line when a corresponding AND gate or OR gate is required. Figure A.3.5 shows how the PLA of Figure A.3.4 would look when drawn in this way. The contents of a PLA are fixed when the PLA is created, although there are also forms of PLA-like structures, called *PLAs*, that can be programmed electronically when a designer is ready to use them.

ROMs

Another form of structured logic that can be used to implement a set of logic functions is a **read-only memory (ROM)**. A ROM is called a memory because it has a set of locations that can be read; however, the contents of these locations are fixed, usually at the time the ROM is manufactured. There are also **programmable ROMs (PROMs)** that can be programmed electronically, when a designer knows their contents. There are also erasable PROMs; these devices require a slow erasure process using ultraviolet light, and thus are used as read-only memories, except during the design and debugging process.

A ROM has a set of input address lines and a set of outputs. The number of addressable entries in the ROM determines the number of address lines: if the

ROM contains 2^m addressable entries, called the *height*, then there are m input lines. The number of bits in each addressable entry is equal to the number of output bits and is sometimes called the *width* of the ROM. The total number of bits in the ROM is equal to the height times the width. The height and width are sometimes collectively referred to as the *shape* of the ROM.

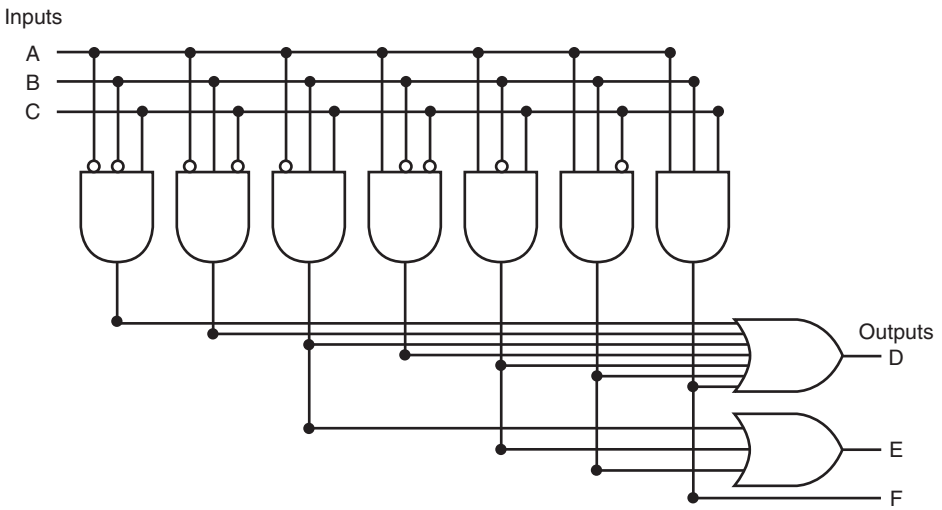


FIGURE A.3.4 The PLA for implementing the logic function described in the example.

A ROM can encode a collection of logic functions directly from the truth table. For example, if there are n functions with m inputs, we need a ROM with m address lines (and 2^m entries), with each entry being n bits wide. The entries in the input portion of the truth table represent the addresses of the entries in the ROM, while the contents of the output portion of the truth table constitute the contents of the ROM. If the truth table is organized so that the sequence of entries in the input portion constitutes a sequence of binary numbers (as have all the truth tables we have shown so far), then the output portion gives the ROM contents in order as well. In the example starting on page A-13, there were three inputs and three outputs. This leads to a ROM with $2^3 = 8$ entries, each 3 bits wide. The contents of those entries in increasing order by address are directly given by the output portion of the truth table that appears on page A-14.

ROMs and PLAs are closely related. A ROM is fully decoded: it contains a full output word for every possible input combination. A PLA is only partially decoded. This means that a ROM will always contain more entries. For the earlier truth table on page A-14, the ROM contains entries for all eight possible inputs, whereas the PLA contains only the seven active product terms. As the number of inputs grows,

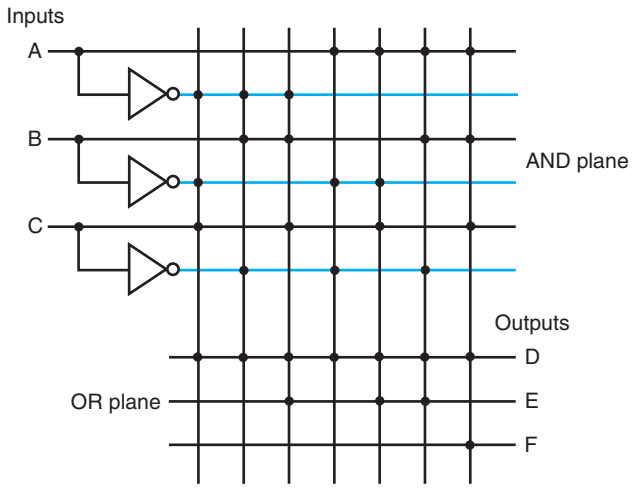


FIGURE A.3.5 A PLA drawn using dots to indicate the components of the product terms and sum terms in the array. Rather than use inverters on the gates, usually all the inputs are run the width of the AND plane in both true and complement forms. A dot in the AND plane indicates that the input, or its inverse, occurs in the product term. A dot in the OR plane indicates that the corresponding product term appears in the corresponding output.

the number of entries in the ROM grows exponentially. In contrast, for most real logic functions, the number of product terms grows much more slowly (see the examples in **Appendix C**). This difference makes PLAs generally more efficient for implementing combinational logic functions. ROMs have the advantage of being able to implement any logic function with the matching number of inputs and outputs. This advantage makes it easier to change the ROM contents if the logic function changes, since the size of the ROM need not change.

In addition to ROMs and PLAs, modern logic synthesis systems will also translate small blocks of combinational logic into a collection of gates that can be placed and wired automatically. Although some small collections of gates are usually not area-efficient, for small logic functions they have less overhead than the rigid structure of a ROM and PLA and so are preferred.

For designing logic outside of a custom or semicustom integrated circuit, a common choice is a field programming device; we describe these devices in Section A.12.

Don't Cares

Often in implementing some combinational logic, there are situations where we do not care what the value of some output is, either because another output is true or because a subset of the input combinations determines the values of the outputs. Such situations are referred to as *don't cares*. Don't cares are important because they make it easier to optimize the implementation of a logic function.

There are two types of don't cares: output don't cares and input don't cares, both of which can be represented in a truth table. *Output don't cares* arise when we don't care about the value of an output for some input combination. They appear as Xs in the output portion of a truth table. When an output is a don't care for some input combination, the designer or logic optimization program is free to make the output true or false for that input combination. *Input don't cares* arise when an output depends on only some of the inputs, and they are also shown as Xs, though in the input portion of the truth table.

Don't Cares

EXAMPLE

Consider a logic function with inputs A, B, and C defined as follows:

- If A or C is true, then output D is true, whatever the value of B.
- If A or B is true, then output E is true, whatever the value of C.
- Output F is true if exactly one of the inputs is true, although we don't care about the value of F, whenever D and E are both true.

Show the full truth table for this function and the truth table using don't cares. How many product terms are required in a PLA for each of these?

ANSWER

Here's the full truth table, without don't cares:

Inputs			Outputs		
A	*B*	*C*	*D*	*E*	*F*
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	1	1
0	1	1	1	1	0
1	0	0	1	1	1
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	1	0

This requires seven product terms without optimization. The truth table written with output don't cares looks like this:

Inputs			Outputs		
A	*B*	*C*	*D*	*E*	*F*
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	1	1
0	1	1	1	1	X
1	0	0	1	1	X
1	0	1	1	1	X
1	1	0	1	1	X
1	1	1	1	1	X

If we also use the input don't cares, this truth table can be further simplified to yield the following:

Inputs			Outputs		
A	*B*	*C*	*D*	*E*	*F*
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	1	1
X	1	1	1	1	X
1	X	X	1	1	X

This simplified truth table requires a PLA with four minterms, or it can be implemented in discrete gates with one two-input AND gate and three OR gates (two with three inputs and one with two inputs). This compares to the original truth table that had seven minterms and would have required four AND gates.

Logic minimization is critical to achieving efficient implementations. One tool useful for hand minimization of random logic is *Karnaugh maps*. Karnaugh maps represent the truth table graphically, so that product terms that may be combined are easily seen. Nevertheless, hand optimization of significant logic functions using Karnaugh maps is impractical, both because of the size of the maps and their complexity. Fortunately, the process of logic minimization is highly mechanical and can be performed by design tools. In the process of minimization, the tools take advantage of the don't cares, so specifying them is important. The textbook references at the end of this appendix provide further discussion on logic minimization, Karnaugh maps, and the theory behind such minimization algorithms.

Arrays of Logic Elements

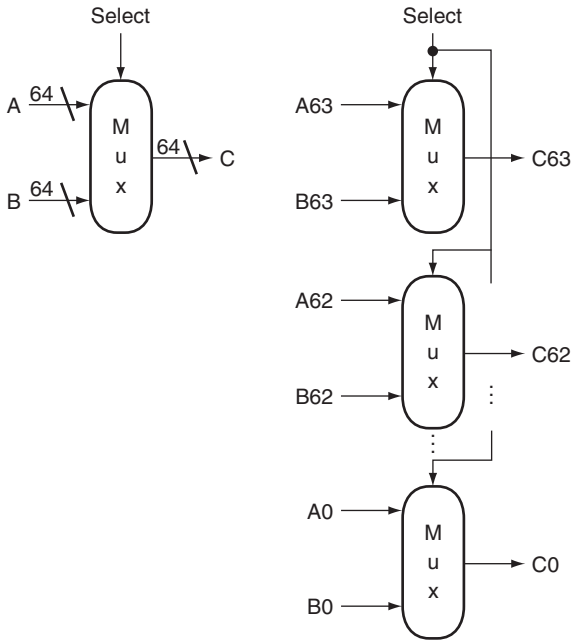
Many of the combinational operations to be performed on data have to be done to an entire word (64 bits) of data. Thus we often want to build an array of logic

elements, which we can represent simply by showing that a given operation will happen to an entire collection of inputs. Inside a machine, much of the time we want to select between a pair of *buses*. A **bus** is a collection of data lines that is treated together as a single logical signal. (The term *bus* is also used to indicate a shared collection of lines with multiple sources and uses.)

bus In logic design, a collection of data lines that is treated together as a single logical signal; also, a shared collection of lines with multiple sources and uses.

For example, in the RISC-V instruction set, the result of an instruction that is written into a register can come from one of two sources. A multiplexor is used to choose which of the two buses (each 64 bits wide) will be written into the Result register. The 1-bit multiplexor, which we showed earlier, will need to be replicated 64 times.

We indicate that a signal is a bus rather than a single 1-bit line by showing it with a thicker line in a figure. Most buses are 64 bits wide; those that are not are explicitly labeled with their width. When we show a logic unit whose inputs and outputs are buses, this means that the unit must be replicated a sufficient number of times to accommodate the width of the input. Figure A.3.6 shows how we draw a multiplexor that selects between a pair of 64-bit buses and how this expands in terms of 1-bit-wide multiplexors. Sometimes we need to construct an array of logic elements where the inputs for some elements in the array are outputs from earlier elements. For example, this is how a multibit-wide ALU is constructed. In such cases, we must explicitly show how to create wider arrays, since the individual elements of the array are no longer independent, as they are in the case of a 64-bit-wide multiplexor.



a. A 64-bit wide 2-to-1 multiplexor

b. The 64-bit wide multiplexor is actually an array of 64 1-bit multiplexors

FIGURE A.3.6 A multiplexor is arrayed 64 times to perform a selection between two 64-bit inputs. Note that there is still only one data selection signal used for all 64 1-bit multiplexors.

Check Yourself

Parity is a function in which the output depends on the number of 1s in the input. For an even parity function, the output is 1 if the input has an even number of ones. Suppose a ROM is used to implement an even parity function with a 4-bit input. Which of A, B, C, or D represents the contents of the ROM?

Address	A	B	C	D
0	0	1	0	1
1	0	1	1	0
2	0	1	0	1
3	0	1	1	0
4	0	1	0	1
5	0	1	1	0
6	0	1	0	1
7	0	1	1	0
8	1	0	0	1
9	1	0	1	0
10	1	0	0	1
11	1	0	1	0
12	1	0	0	1
13	1	0	1	0
14	1	0	0	1
15	1	0	1	0

A.4 Using a Hardware Description Language

hardware description language A programming language for describing hardware, used for generating simulations of a hardware design and also as input to synthesis tools that can generate actual hardware.

Verilog One of the two most common hardware description languages.

VHDL One of the two most common hardware description languages.

Today most digital design of processors and related hardware systems is done using a **hardware description language**. Such a language serves two purposes. First, it provides an abstract description of the hardware to simulate and debug the design. Second, with the use of logic synthesis and hardware compilation tools, this description can be compiled into the hardware implementation.

In this section, we introduce the hardware description language Verilog and show how it can be used for combinational design. In the rest of the appendix, we expand the use of Verilog to include design of sequential logic. In the optional sections of Chapter 4 that appear online, we use Verilog to describe processor implementations. In the optional section from Chapter 5 that appears online, we use system Verilog to describe cache controller implementations. System Verilog adds structures and some other useful features to Verilog.

Verilog is one of the two primary hardware description languages; the other is **VHDL**. Verilog is somewhat more heavily used in industry and is based on C, as opposed to VHDL, which is based on Ada. The reader generally familiar with C will find the basics of Verilog, which we use in this appendix, easy to follow.

Readers already familiar with VHDL should find the concepts simple, provided they have been exposed to the syntax of C.

Verilog can specify both a behavioral and a structural definition of a digital system. A **behavioral specification** describes how a digital system functionally operates. A **structural specification** describes the detailed organization of a digital system, usually using a hierarchical description. A structural specification can be used to describe a hardware system in terms of a hierarchy of basic elements such as gates and switches. Thus, we could use Verilog to describe the exact contents of the truth tables and datapath of the last section.

behavioral specification Describes how a digital system operates functionally.

structural specification Describes how a digital system is organized in terms of a hierarchical connection of elements.

With the arrival of **hardware synthesis tools**, most designers now use Verilog or VHDL to structurally describe only the datapath, relying on logic synthesis to generate the control from a behavioral description. In addition, most CAD systems provide extensive libraries of standardized parts, such as ALUs, multiplexors, register files, memories, and programmable logic blocks, as well as basic gates.

hardware synthesis tools Computer-aided design software that can generate a gate-level design based on behavioral descriptions of a digital system.

Obtaining an acceptable result using libraries and logic synthesis requires that the specification be written with an eye toward the eventual synthesis and the desired outcome. For our simple designs, this primarily means making clear what we expect to be implemented in combinational logic and what we expect to require in sequential logic. In most of the examples we use in this section and the remainder of this appendix, we have written the Verilog with the eventual synthesis in mind.

Datatypes and Operators in Verilog

There are two primary datatypes in Verilog:

1. A **wire** specifies a combinational signal.
2. A **reg** (register) holds a value, which can vary with time. A reg need not necessarily correspond to an actual register in an implementation, although it often will.

wire In Verilog, specifies a combinational signal.

reg In Verilog, a register.

A register or wire, named X, that is 64 bits wide is declared as an array: `reg [63:0] X` or `wire [63:0] X`, which also sets the index of 0 to designate the least significant bit of the register. Because we often want to access a subfield of a register or wire, we can refer to a contiguous set of bits of a register or wire with the notation [`starting bit: ending bit`], where both indices must be constant values.

An array of registers is used for a structure like a register file or memory. Thus, the declaration

```
reg [63:0] registerfile[0:31]
```

specifies a variable registerfile that is equivalent to a RISC-V registerfile, where register 0 is the first. When accessing an array, we can refer to a single element, as in C, using the notation `registerfile[regnum]`.

The possible values for a register or wire in Verilog are

- 0 or 1, representing logical false or true
- X, representing unknown, the initial value given to all registers and to any wire not connected to something
- Z, representing the high-impedance state for tristate gates, which we will not discuss in this appendix

Constant values can be specified as decimal numbers as well as binary, octal, or hexadecimal. We often want to say exactly how large a constant field is in bits. This is done by prefixing the value with a decimal number specifying its size in bits. For example:

- `4'b0100` specifies a 4-bit binary constant with the value 4, as does `4'd4`.
- `-8'h4` specifies an 8-bit constant with the value −4 (in two's complement representation)

Values can also be concatenated by placing them within `{ }` separated by commas. The notation `{x{bitfield}}` replicates `bitfield` `x` times. For example:

- `{32{2'b01}}` creates a 64-bit value with the pattern 0101 … 01.
- `{A[31:16],B[15:0]}` creates a value whose upper 16 bits come from `A` and whose lower 16 bits come from `B`.

Verilog provides the full set of unary and binary operators from C, including the arithmetic operators (+, −, *. /), the logical operators (&, |, ~), the comparison operators (= =, ! =, >, < , < =, > =), the shift operators (<<, >>), and C's conditional operator (?, which is used in the form `condition ? expr1 :expr2` and returns `expr1` if the condition is true and `expr2` if it is false). Verilog adds a set of unary logic reduction operators (&, |, ^) that yield a single bit by applying the logical operator to all the bits of an operand. For example, `&A` returns the value obtained by ANDing all the bits of `A` together, and `^A` returns the reduction obtained by using exclusive OR on all the bits of `A`.

Check Yourself

Which of the following define exactly the same value?

1. `8'bimoooo`
2. `8'hF0`
3. `8'd240`
4. `{{4{1'b1}},{4{1'b0}}}`
5. `{4'b1,4'b0)`

Structure of a Verilog Program

A Verilog program is structured as a set of modules, which may represent anything from a collection of logic gates to a complete system. Modules are similar to classes in C++, although not nearly as powerful. A module specifies its input and output ports, which describe the incoming and outgoing connections of a module. A module may also declare additional variables. The body of a module consists of:

- `initial` constructs, which can initialize `reg` variables
- Continuous assignments, which define only combinational logic
- `always` constructs, which can define either sequential or combinational logic
- Instances of other modules, which are used to implement the module being defined

Representing Complex Combinational Logic in Verilog

A continuous assignment, which is indicated with the keyword `assign`, acts like a combinational logic function: the output is continuously assigned the value, and a change in the input values is reflected immediately in the output value. Wires may only be assigned values with continuous assignments. Using continuous assignments, we can define a module that implements a half-adder, as Figure A.4.1 shows.

Assign statements are one sure way to write Verilog that generates combinational logic. For more complex structures, however, assign statements may be awkward or tedious to use. It is also possible to use the `always` block of a module to describe a combinational logic element, although care must be taken. Using an `always` block allows the inclusion of Verilog control constructs, such as *if-then-else, case* statements, *for* statements, and *repeat* statements, to be used. These statements are similar to those in C with small changes.

An `always` block specifies an optional list of signals on which the block is sensitive (in a list starting with @). The `always` block is re-evaluated if any of the

```
module half_adder (A,B,Sum,Carry);
   input A,B; //two 1-bit inputs
   output Sum, Carry; //two 1-bit outputs
   assign Sum = A ^ B; //sum is A xor B
   assign Carry = A & B; //Carry is A and B
endmodule
```

FIGURE A.4.1 A Verilog module that defines a half-adder using continuous assignments.

sensitivity list The list of signals that specifies when an `always` block should be re-evaluated.

listed signals changes value; if the list is omitted, the `always` block is constantly re-evaluated. When an `always` block is specifying combinational logic, the **sensitivity list** should include all the input signals. If there are multiple Verilog statements to be executed in an `always` block, they are surrounded by the keywords `begin` and `end`, which take the place of the { and } in C. An `always` block thus looks like this:

```
always @(list of signals that cause reevaluation) begin
   Verilog statements including assignments and other
control statements end
```

`Reg` variables may only be assigned inside an `always` block, using a procedural assignment statement (as distinguished from continuous assignment we saw earlier). There are, however, two different types of procedural assignments. The assignment operator = executes as it does in C; the right-hand side is evaluated, and the left-hand side is assigned the value. Furthermore, it executes like the normal C assignment statement: that is, it is completed before the next statement is executed. Hence, the assignment operator = has the name **blocking assignment**. This blocking can be useful in the generation of sequential logic, and we will return to it shortly. The other form of assignment (**nonblocking**) is indicated by <=. In nonblocking assignment, all right-hand sides of the assignments in an `always` group are evaluated and the assignments are done simultaneously. As a first example of combinational logic implemented using an `always` block, Figure A.4.2 shows the implementation of a 4-to-1 multiplexor, which uses a `case` construct to make it easy to write. The `case` construct looks like a C `switch` statement. Figure A.4.3 shows a definition of a RISC-V ALU, which also uses a `case` statement.

blocking assignment In Verilog, an assignment that completes before the execution of the next statement.

nonblocking assignment An assignment that continues after evaluating the right-hand side, assigning the left-hand side the value only after all right-hand sides are evaluated.

Since only reg variables may be assigned inside `always` blocks, when we want to describe combinational logic using an `always` block, care must be taken to ensure that the reg does not synthesize into a register. A variety of pitfalls are described in the elaboration below.

Elaboration: Continuous assignment statements always yield combinational logic, but other Verilog structures, even when in `always` blocks, can yield unexpected results during logic synthesis. The most common problem is creating sequential logic by implying the existence of a latch or register, which results in an implementation that is both slower and more costly than perhaps intended. To ensure that the logic that you intend to be combinational is synthesized that way, make sure you do the following:

1. Place all combinational logic in a continuous assignment or an `always` block.
2. Make sure that all the signals used as inputs appear in the sensitivity list of an `always` block.
3. Ensure that every path through an `always` block assigns a value to the exact same set of bits.

The last of these is the easiest to overlook; read through the example in Figure A.5.15 to convince yourself that this property is adhered to.

```
module Mult4to1 (In1,In2,In3,In4,Sel,Out);
   input [63:0] In1, In2, In3, In4; //four 64-bit inputs
   input [1:0] Sel; //selector signal
   output reg [63:0] Out; //64-bit output
   always @(In1, In2, In3, In4, Sel)
   case (Sel) // a 4->1 multiplexor
      0: Out <= In1;
      1: Out <= In2;
      2: Out <= In3;
      default: Out <= In4;
   endcase
endmodule
```

FIGURE A.4.2 A Verilog definition of a 4-to-1 multiplexor with 64-bit inputs, using a `case` statement. The `case` statement acts like a C `switch` statement, except that in Verilog only the code associated with the selected case is executed (as if each case state had a break at the end) and there is no fall-through to the next statement.

```
module RISCVALU (ALUctl, A, B, ALUOut, Zero);
   input [3:0] ALUctl;
   input [63:0] A,B;
   output reg [63:0] ALUOut;
   output Zero;
   assign Zero = (ALUOut==0); //Zero is true if ALUOut is 0; goes anywhere
   always @(ALUctl, A, B) //reevaluate if these change
      case (ALUctl)
         0: ALUOut <= A & B;
         1: ALUOut <= A | B;
         2: ALUOut <= A + B;
         6: ALUOut <= A - B;
         7: ALUOut <= A < B ? 1:0;
         12: ALUOut <= ~(A | B); // result is nor
         default: ALUOut <= 0; //default to 0, should not happen;
      endcase
endmodule
```

FIGURE A.4.3 A Verilog behavioral definition of a RISC-V ALU. This could be synthesized using a module library containing basic arithmetic and logical operations.

Check Yourself Assuming all values are initially zero, what are the values of `A` and `B` after executing this Verilog code inside an `always` block?

```
C = 1;
A <= C;
B = C;
```

A.5 Constructing a Basic Arithmetic Logic Unit

*ALU n. [**A**rthritic **L**ogic **U**nit or (rare) **A**rithmetic **L**ogic **U**nit] A random-number generator supplied as standard with all computer systems.*

Stan Kelly-Bootle, *The Devil's DP Dictionary*, 1981

The **arithmetic logic unit (ALU)** is the brawn of the computer, the device that performs the arithmetic operations like addition and subtraction or logical operations like AND and OR. This section constructs an ALU from four hardware building blocks (AND and OR gates, inverters, and multiplexors) and illustrates how combinational logic works. In the next section, we will see how addition can be sped up through more clever designs.

Because the RISC-V registers are 64 bits wide, we need a 64-bit-wide ALU. Let's assume that we will connect 64 1-bit ALUs to create the desired ALU. We'll therefore start by constructing a 1-bit ALU.

A 1-Bit ALU

The logical operations are easiest, because they map directly onto the hardware components in Figure A.2.1.

The 1-bit logical unit for AND and OR looks like Figure A.5.1. The multiplexor on the right then selects *a* AND *b* or *a* OR *b*, depending on whether the value of *Operation* is 0 or 1. The line that controls the multiplexor is shown in color to distinguish it from the lines containing data. Notice that we have renamed the control and output lines of the multiplexor to give them names that reflect the function of the ALU.

The next function to include is addition. An adder must have two inputs for the operands and a single-bit output for the sum. There must be a second output to pass on the carry, called *CarryOut*. Since the CarryOut from the neighbor adder must be included as an input, we need a third input. This input is called *CarryIn*. Figure A.5.2 shows the inputs and the outputs of a 1-bit adder. Since we know what addition is supposed to do, we can specify the outputs of this "black box" based on its inputs, as Figure A.5.3 demonstrates.

We can express the output functions CarryOut and Sum as logical equations, and these equations can in turn be implemented with logic gates. Let's do CarryOut. Figure A.5.4 shows the values of the inputs when CarryOut is a 1.

We can turn this truth table into a logical equation:

$$\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b) + (a \cdot b \cdot \text{CarryIn})$$

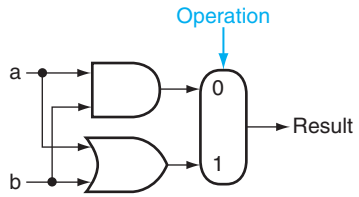


FIGURE A.5.1 The 1-bit logical unit for AND and OR.

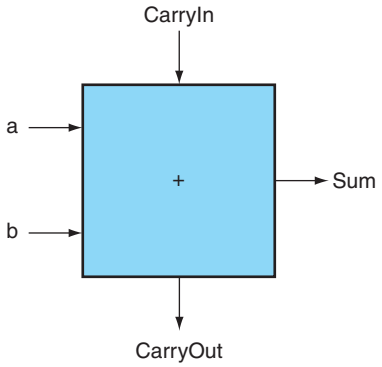


FIGURE A.5.2 A 1-bit adder. This adder is called a full adder; it is also called a (3,2) adder because it has three inputs and two outputs. An adder with only the a and b inputs is called a (2,2) adder or half-adder.

Inputs			Outputs		
a	b	CarryIn	CarryOut	Sum	Comments
0	0	0	0	0	$0 + 0 + 0 = 00_{two}$
0	0	1	0	1	$0 + 0 + 1 = 01_{two}$
0	1	0	0	1	$0 + 1 + 0 = 01_{two}$
0	1	1	1	0	$0 + 1 + 1 = 10_{two}$
1	0	0	0	1	$1 + 0 + 0 = 01_{two}$
1	0	1	1	0	$1 + 0 + 1 = 10_{two}$
1	1	0	1	0	$1 + 1 + 0 = 10_{two}$
1	1	1	1	1	$1 + 1 + 1 = 11_{two}$

FIGURE A.5.3 Input and output specification for a 1-bit adder.

If a · b · CarryIn is true, then all of the other three terms must also be true, so we can leave out this last term corresponding to the fourth line of the table. We can thus simplify the equation to

$$\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b)$$

Figure A.5.5 shows that the hardware within the adder black box for CarryOut consists of three AND gates and one OR gate. The three AND gates correspond exactly to the three parenthesized terms of the formula above for CarryOut, and the OR gate sums the three terms.

Inputs		
a	b	CarryIn
0	1	1
1	0	1
1	1	0
1	1	1

FIGURE A.5.4 Values of the inputs when CarryOut is a 1.

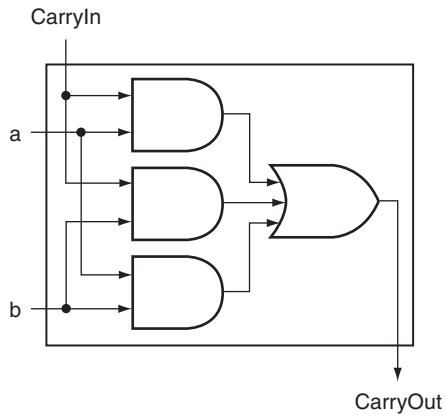


FIGURE A.5.5 Adder hardware for the CarryOut signal. The rest of the adder hardware is the logic for the Sum output given in the equation on this page.

The Sum bit is set when exactly one input is 1 or when all three inputs are 1. The Sum results in a complex Boolean equation (recall that $\overline{a}$ means NOT a):

$$\text{Sum} = (a \cdot \overline{b} \cdot \overline{\text{CarryIn}}) + (\overline{a} \cdot b \cdot \overline{\text{CarryIn}}) + (\overline{a} \cdot \overline{b} \cdot \text{CarryIn}) + (a \cdot b \cdot \text{CarryIn})$$

The drawing of the logic for the Sum bit in the adder black box is left as an exercise for the reader.

Figure A.5.6 shows a 1-bit ALU derived by combining the adder with the earlier components. Sometimes designers also want the ALU to perform a few more simple operations, such as generating 0. The easiest way to add an operation is to expand the multiplexor controlled by the Operation line and, for this example, to connect 0 directly to the new input of that expanded multiplexor.

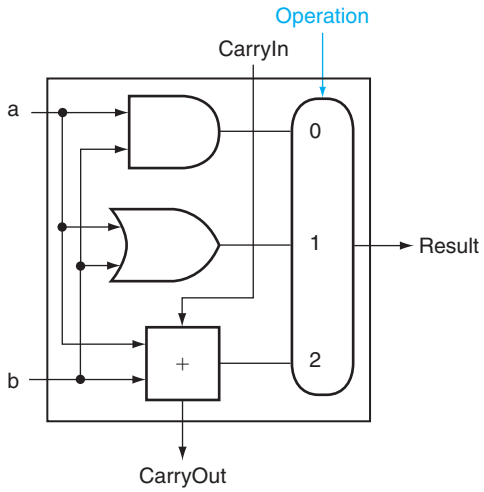


FIGURE A.5.6 A 1-bit ALU that performs AND, OR, and addition (see Figure A.5.5).

A 64-Bit ALU

Now that we have completed the 1-bit ALU, the full 64-bit ALU is created by connecting adjacent "black boxes." Using *xi* to mean the *i*th bit of *x*, Figure A.5.7 shows a 64-bit ALU. Just as a single stone can cause ripples to radiate to the shores of a quiet lake, a single carry out of the least significant bit (Result0) can ripple all the way through the adder, causing a carry out of the most significant bit (Result63). Hence, the adder created by directly linking the carries of 1-bit adders is called a *ripple carry* adder. We'll see a faster way to connect the 1-bit adders starting on page A-38.

Subtraction is the same as adding the negative version of an operand, and this is how adders perform subtraction. Recall that the shortcut for negating a two's complement number is to invert each bit (sometimes called the *one's complement*) and then add 1. To invert each bit, we simply add a 2:1 multiplexor that chooses between b and $\overline{b}$, as Figure A.5.8 shows.

Suppose we connect 64 of these 1-bit ALUs, as we did in Figure A.5.7. The added multiplexor gives the option of b or its inverted value, depending on Binvert, but

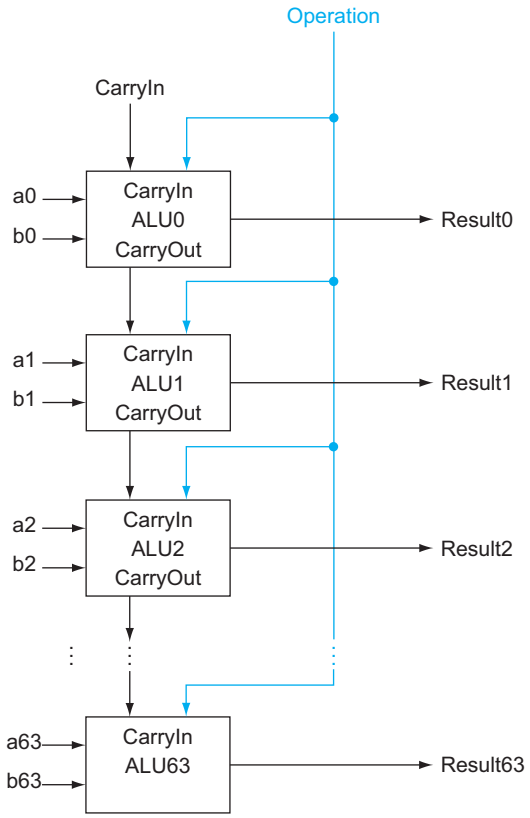


FIGURE A.5.7 A 64-bit ALU constructed from 64 1-bit ALUs. CarryOut of the less significant bit is connected to the CarryIn of the more significant bit. This organization is called ripple carry.

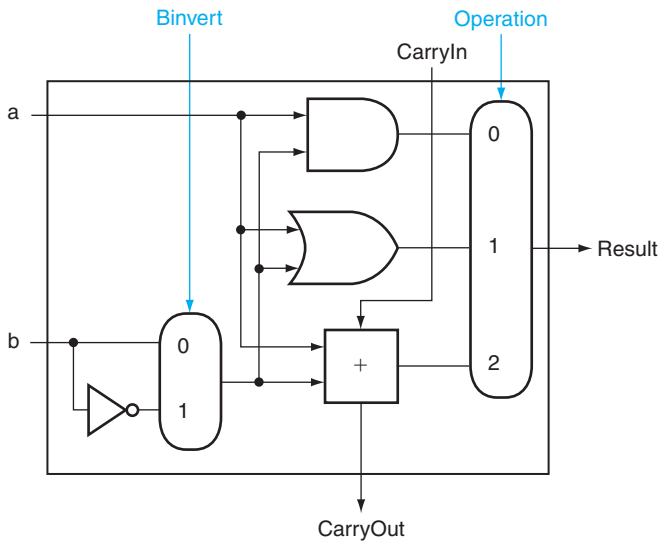


FIGURE A.5.8 A 1-bit ALU that performs AND, OR, and addition on a and b or a and $\overline{b}$. By selecting $\overline{b}$ (Binvert = 1) and setting CarryIn to 1 in the least significant bit of the ALU, we get two's complement subtraction of b from a instead of addition of b to a.

this is only one step in negating a two's complement number. Notice that the least significant bit still has a CarryIn signal, even though it's unnecessary for addition. What happens if we set this CarryIn to 1 instead of 0? The adder will then calculate $a + b + 1$. By selecting the inverted version of b, we get exactly what we want:

$$a + \overline{b} + 1 = a + (\overline{b} + 1) = a + (-b) = a - b$$

The simplicity of the hardware design of a two's complement adder helps explain why two's complement representation has become the universal standard for integer computer arithmetic.

We also wish to add a NOR function. Instead of adding a separate gate for NOR, we can reuse much of the hardware already in the ALU, like we did for subtract. The insight comes from the following truth about NOR:

$$\overline{(a + b)} = \overline{a} \cdot \overline{b}$$

That is, NOT (a OR b) is equivalent to NOT a AND NOT b. This fact is called DeMorgan's theorem and is explored in the exercises in more depth.

Since we have AND and NOT b, we only need to add NOT a to the ALU. Figure A.5.9 shows that change.

Tailoring the 64-Bit ALU to RISC-V

These four operations—add, subtract, AND, OR—are found in the ALU of almost every computer, and the operations of most RISC-V instructions can be performed by this ALU. But the design of the ALU is incomplete.

One instruction that still needs support is the set less than instruction (`slt`). Recall that the operation produces 1 if rs1 < rs2, and 0 otherwise. Consequently, `slt` will set all but the least significant bit to 0, with the least significant bit set according to the comparison. For the ALU to perform `slt`, we first need to expand the three-input multiplexor in Figure A.5.9 to add an input for the `slt` result. We call that new input *Less* and use it only for `slt`.

The top drawing of Figure A.5.10 shows the new 1-bit ALU with the expanded multiplexor. From the description of `slt` above, we must connect 0 to the Less input for the upper 63 bits of the ALU, since those bits are always set to 0. What remains to consider is how to compare and set the *least significant bit* for set less than instructions.

What happens if we subtract b from a? If the difference is negative, then a < b since

$$(a - b) < 0 \Rightarrow ((a - b) + b) < (0 + b) \Rightarrow a < b$$

We want the least significant bit of a set less than operation to be a 1 if a < b; that is, a 1 if a − b is negative and a 0 if it's positive. This desired result corresponds exactly to the sign bit values: 1 means negative and 0 means positive. Following this line of argument, we need only connect the sign bit from the adder output to the least significant bit to get set less than. (Alas, this argument only holds if the subtraction does not overflow; we will explore a complete implementation in the exercises.)

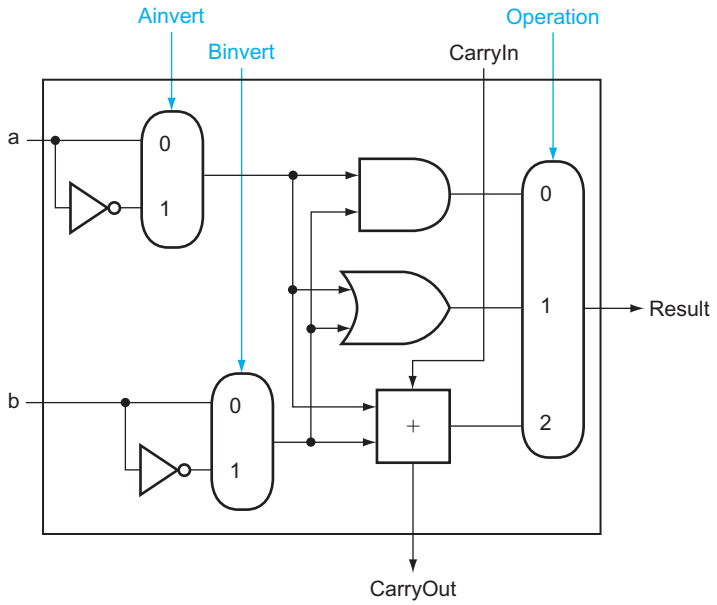


FIGURE A.5.9 A 1-bit ALU that performs AND, OR, and addition on a and b or $\bar{a}$ and $\bar{b}$. By selecting $\bar{a}$ (Ainvert = 1) and $\bar{b}$ (Binvert = 1), we get a NOR b instead of a AND b.

Unfortunately, the Result output from the most significant ALU bit in the top of Figure A.5.10 for the `slt` operation is *not* the output of the adder; the ALU output for the slt operation is obviously the input value Less.

Thus, we need a new 1-bit ALU for the most significant bit that has an extra output bit: the adder output. The bottom drawing of Figure A.5.10 shows the design, with this new adder output line called *Set*. As long as we need a special ALU for the most significant bit, we added the overflow detection logic since it is also associated with that bit. Figure A.5.11 shows the 64-bit ALU.

Notice that every time we want the ALU to subtract, we set both CarryIn and Binvert to 1. For adds or logical operations, we want both control lines to be 0. We can therefore simplify control of the ALU by combining the CarryIn and Binvert to a single control line called *Bnegate*.

To further tailor the ALU to the RISC-V instruction set, we must support conditional branch instructions such as Branch if Equal (`beq`), which branches if two registers are equal. The easiest way to test equality with the ALU is to subtract b from a and then test to see if the result is 0, since

$$(a - b = 0) \Rightarrow a = b$$

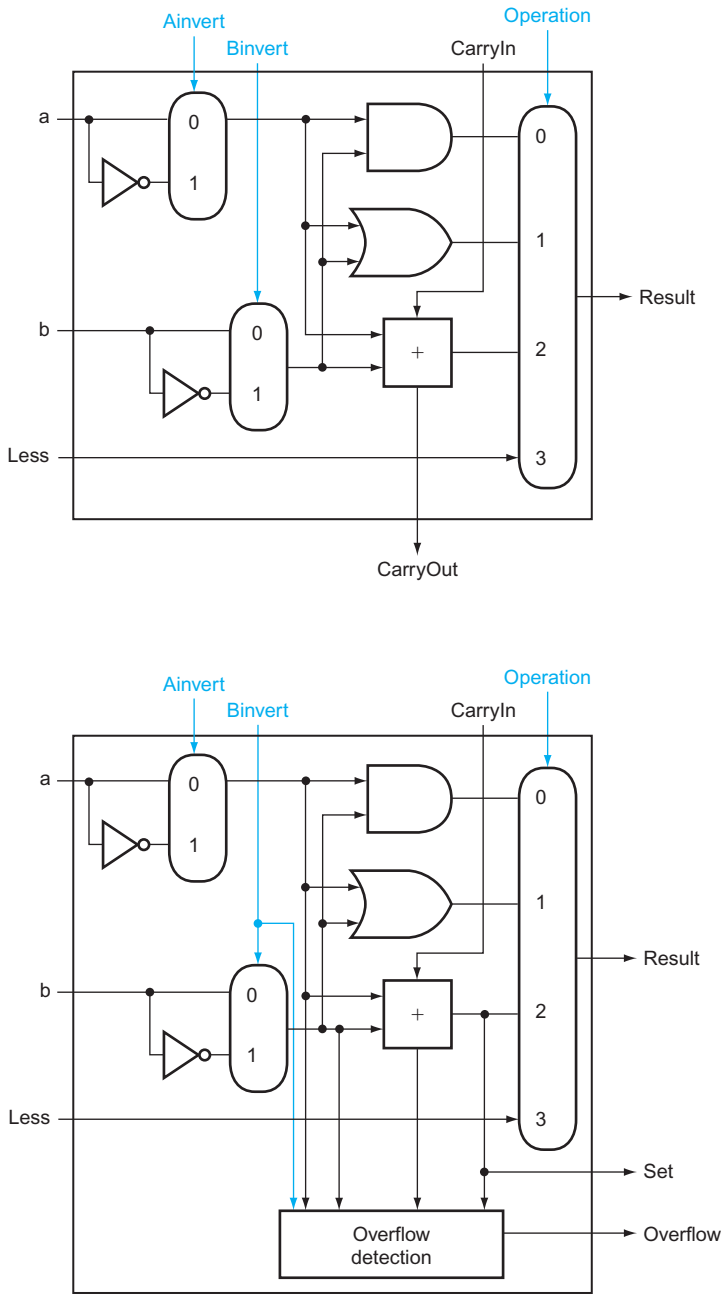


FIGURE A.5.10 (Top) A 1-bit ALU that performs AND, OR, and addition on a and b or $\bar{b}$, and (bottom) a 1-bit ALU for the most significant bit. The top drawing includes a direct input that is connected to perform the set on less than operation (see Figure A.5.11); the bottom has a direct output from the adder for the less than comparison called Set. (See Exercise A.24 at the end of this appendix to see how to calculate overflow with fewer inputs.)

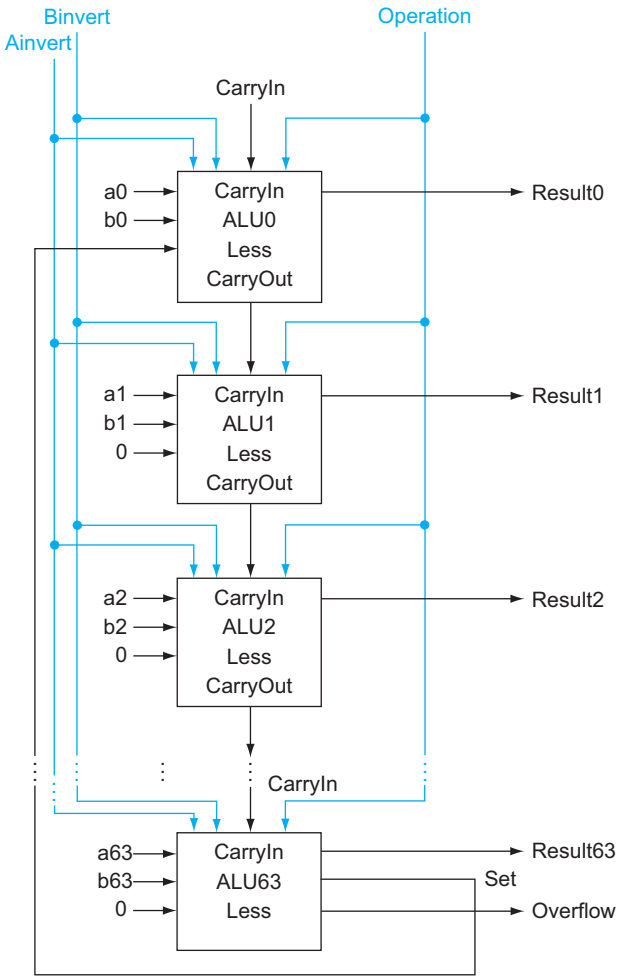


FIGURE A.5.11 A 64-bit ALU constructed from the 63 copies of the 1-bit ALU in the top of Figure A.5.10 and one 1-bit ALU in the bottom of that figure. The Less inputs are connected to 0 except for the least significant bit, which is connected to the Set output of the most significant bit. If the ALU performs a − b and we select the input 3 in the multiplexor in Figure A.5.10, then Result = 0 … 001 if a < b, and Result = 0 … 000 otherwise.

Thus, if we add hardware to test if the result is 0, we can test for equality. The simplest way is to OR all the outputs together and then send that signal through an inverter:

$$\text{Zero} = \overline{(\text{Result63} + \text{Result62} + \cdots + \text{Result2} + \text{Result1} + \text{Result0})}$$

Figure A.5.12 shows the revised 64-bit ALU. We can think of the combination of the 1-bit Ainvert line, the 1-bit Bnegate line, and the 2-bit Operation lines as 4-bit control lines for the ALU, telling it to perform add, subtract, AND, OR, NOR, or

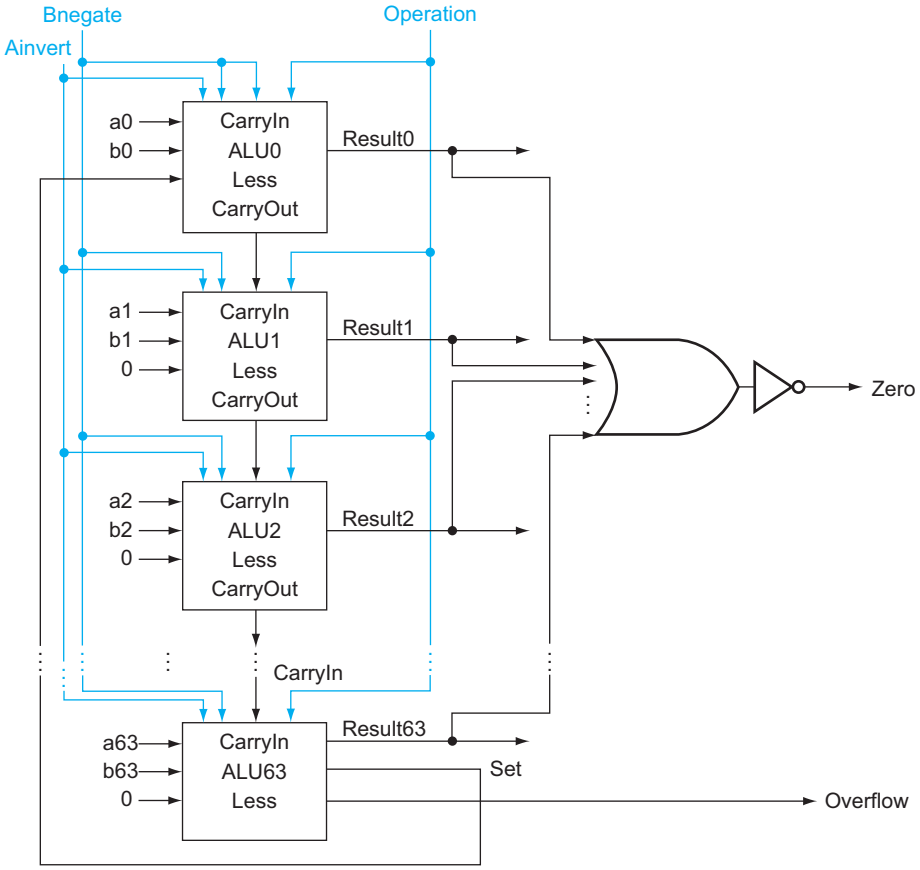


FIGURE A.5.12 The final 64-bit ALU. This adds a Zero detector to Figure A.5.11.

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set less than
1100	NOR

FIGURE A.5.13 The values of the three ALU control lines, Ainvert, Bnegate, and Operation, and the corresponding ALU operations.

set less than. Figure A.5.13 shows the ALU control lines and the corresponding ALU operation.

Finally, now that we have seen what is inside a 64-bit ALU, we will use the universal symbol for a complete ALU, as shown in Figure A.5.14.

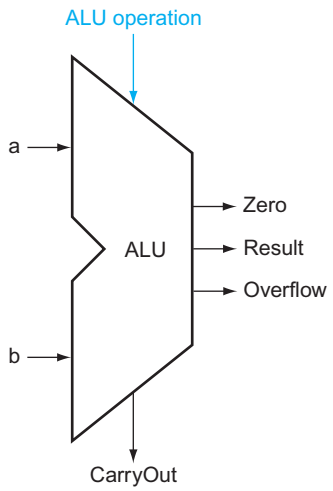


FIGURE A.5.14 The symbol commonly used to represent an ALU, as shown in Figure A.5.12. This symbol is also used to represent an adder, so it is normally labeled either with ALU or Adder.

```
module RISCVALU (ALUctl, A, B, ALUOut, Zero);
   input [3:0] ALUctl;
   input [63:0] A,B;
   output reg [63:0] ALUOut;
   output Zero;
   assign Zero = (ALUOut==0); //Zero is true if ALUOut is 0
   always @(ALUctl, A, B) begin //reevaluate if these change
      case (ALUctl)
         0: ALUOut <= A & B;
         1: ALUOut <= A | B;
         2: ALUOut <= A + B;
         6: ALUOut <= A - B;
         7: ALUOut <= A < B ? 1 : 0;
         12: ALUOut <= ~(A | B); // result is nor
         default: ALUOut <= 0;
      endcase
    end
endmodule
```

FIGURE A.5.15 A Verilog behavioral definition of a RISC-V ALU.

Defining the RISC-V ALU in Verilog

Figure A.5.15 shows how a combinational RISC-V ALU might be specified in Verilog; such a specification would probably be compiled using a standard parts library that provided an adder, which could be instantiated. For completeness, we show the ALU control for RISC-V in Figure A.5.16, which is used in Chapter 4, where we build a Verilog version of the RISC-V datapath.

```
module ALUControl (ALUOp, FuncCode, ALUCtl);
   input [1:0] ALUOp;
   input [5:0] FuncCode;
   output [3:0] reg ALUCtl;
   always case (FuncCode)
   32: ALUOp<=2; // add
   34: ALUOp<=6; // subtract
   36: ALUOP<=0; // and
   37: ALUOp<=1; // or
   39: ALUOp<=12; // nor
   42: ALUOp<=7; // slt
   default: ALUOp<=15; // should not happen
   endcase
endmodule
```

FIGURE A.5.16 The RISC-V ALU control: a simple piece of combinational control logic.

The next question is, "How quickly can this ALU add two 64-bit operands?" We can determine the a and b inputs, but the CarryIn input depends on the operation in the adjacent 1-bit adder. If we trace all the way through the chain of dependencies, we connect the most significant bit to the least significant bit, so the most significant bit of the sum must wait for the *sequential* evaluation of all 64 1-bit adders. This sequential chain reaction is too slow to be used in time-critical hardware. The next section explores how to speed-up addition. This topic is not crucial to understanding the rest of the appendix and may be skipped.

Check Yourself

Suppose you wanted to add the operation NOT (a AND b), called NAND. How could the ALU change to support it?

1. No change. You can calculate NAND quickly using the current ALU since $(a \cdot b) = \overline{a} + \overline{b}$ and we already have NOT a, NOT b, and OR.
2. You must expand the big multiplexor to add another input, and then add new logic to calculate NAND.

A.6 Faster Addition: Carry Lookahead

The key to speeding up addition is 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.

A key to understanding fast-carry schemes is to remember that, unlike software, hardware executes in parallel whenever inputs change.

Fast Carry Using "Infinite" Hardware

As we mentioned earlier, any equation can be represented in two levels of logic. Since the only external inputs are the two operands and the CarryIn to the least significant bit of the adder, in theory we could calculate the CarryIn values to all the remaining bits of the adder in just two levels of logic.

For example, the CarryIn for bit 2 of the adder is exactly the CarryOut of bit 1, so the formula is

$$\text{CarryIn2} = (\text{b1} \cdot \text{CarryIn1}) + (\text{a1} \cdot \text{CarryIn1}) + (\text{a1} \cdot \text{b1})$$

Similarly, CarryIn1 is defined as

$$\text{CarryIn1} = (\text{b0} \cdot \text{CarryIn0}) + (\text{a0} \cdot \text{CarryIn0}) + (\text{a0} \cdot \text{b0})$$

Using the shorter and more traditional abbreviation of *ci* for CarryIn*i*, we can rewrite the formulas as

$$\text{c2} = (\text{b1} \cdot \text{c1}) + (\text{a1} \cdot \text{c1}) + (\text{a1} \cdot \text{b1})$$
$$\text{c1} = (\text{b0} \cdot \text{c0}) + (\text{a0} \cdot \text{c0}) + (\text{a0} \cdot \text{b0})$$

Substituting the definition of c1 for the first equation results in this formula:

$$\begin{aligned}\text{c2} = {} & (\text{a1} \cdot \text{a0} \cdot \text{b0}) + (\text{a1} \cdot \text{a0} \cdot \text{c0}) + (\text{a1} \cdot \text{b0} \cdot \text{c0}) \\ & + (\text{b1} \cdot \text{a0} \cdot \text{b0}) + (\text{b1} \cdot \text{a0} \cdot \text{c0}) + (\text{b1} \cdot \text{b0} \cdot \text{c0}) + (\text{a1} \cdot \text{b1})\end{aligned}$$

You can imagine how the equation expands as we get to higher bits in the adder; it grows rapidly with the number of bits. This complexity is reflected in the cost of the hardware for fast carry, making this simple scheme prohibitively expensive for wide adders.

Fast Carry Using the First Level of Abstraction: Propagate and Generate

Most fast-carry schemes limit the complexity of the equations to simplify the hardware, while still making substantial speed improvements over ripple carry. One such scheme is a *carry-lookahead adder*. In Chapter 1, we said computer systems cope with complexity by using levels of abstraction. A carry-lookahead adder relies on levels of abstraction in its implementation.

Let's factor our original equation as a first step:

$$\begin{aligned} c_{i+1} &= (b_i \cdot c_i) + (a_i \cdot c_i) + (a_i \cdot b_i) \\ &= (a_i \cdot b_i) + (a_i + b_i) \cdot c_i \end{aligned}$$

If we were to rewrite the equation for c2 using this formula, we would see some repeated patterns:

$$c2 = (a1 \cdot b1) + (a1 \cdot b1) \cdot ((a0 \cdot b0) + (a0 + b0) \cdot c0)$$

Note the repeated appearance of $(a_i \cdot b_i)$ and $(a_i + b_i)$ in the formula above. These two important factors are traditionally called *generate* (g_i) and *propagate* (p_i):

$$\begin{aligned} g_i &= a_i \cdot b_i \\ p_i &= a_i + b_i \end{aligned}$$

Using them to define c_{i+1}, we get

$$c_{i+1} = g_i + p_i \cdot c_i$$

To see where the signals get their names, suppose g_i is 1. Then

$$c_{i+1} = g_i + p_i \cdot c_i = 1 + p_i \cdot c_i = 1$$

That is, the adder *generates* a CarryOut (c_{i+1}) independent of the value of CarryIn (c_i). Now suppose that g_i is 0 and p_i is 1. Then

$$c_{i+1} = g_i + p_i \cdot c_i = 0 + 1 \cdot c_i = c_i$$

That is, the adder *propagate*s CarryIn to a CarryOut. Putting the two together, CarryIn_{i+1} is a 1 if either g_i is 1 or both p_i is 1 and CarryIn$_i$ is 1.

As an analogy, imagine a row of dominoes set on edge. The end domino can be tipped over by pushing one far away, provided there are no gaps between the two. Similarly, a carry out can be made true by a generate far away, provided all the propagates between them are true.

Relying on the definitions of propagate and generate as our first level of abstraction, we can express the CarryIn signals more economically. Let's show it for 4 bits:

$$\begin{aligned} c1 &= g0 + (p0 \cdot c0) \\ c2 &= g1 + (p1 \cdot g0) + (p1 \cdot p0 \cdot c0) \\ c3 &= g2 + (p2 \cdot g1) + (p2 \cdot p1 \cdot g0) + (p2 \cdot p1 \cdot p0 \cdot c0) \\ c4 &= g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0) \\ &\quad + (p3 \cdot p2 \cdot p1 \cdot p0 \cdot c0) \end{aligned}$$

These equations just represent common sense: CarryIn*i* is a 1 if some earlier adder generates a carry and all intermediary adders propagate a carry. Figure A.6.1 uses plumbing to try to explain carry lookahead.

Even this simplified form leads to large equations and, hence, considerable logic even for a 16-bit adder. Let's try moving to two levels of abstraction.

Fast Carry Using the Second Level of Abstraction

First, we consider this 4-bit adder with its carry-lookahead logic as a single building block. If we connect them in ripple carry fashion to form a 16-bit adder, the add will be faster than the original with a little more hardware.

To go faster, we'll need carry lookahead at a higher level. To perform carry lookahead for 4-bit adders, we need to propagate and generate signals at this higher level. Here they are for the four 4-bit adder blocks:

$$
\begin{aligned}
P0 &= p3 \cdot p2 \cdot p1 \cdot p0 \\
P1 &= p7 \cdot p6 \cdot p5 \cdot p4 \\
P2 &= p11 \cdot p10 \cdot p9 \cdot p8 \\
P3 &= p15 \cdot p14 \cdot p13 \cdot p12
\end{aligned}
$$

That is, the "super" propagate signal for the 4-bit abstraction (P*i*) is true only if each of the bits in the group will propagate a carry.

For the "super" generate signal (G*i*), we care only if there is a carry out of the most significant bit of the 4-bit group. This obviously occurs if generate is true for that most significant bit; it also occurs if an earlier generate is true *and* all the intermediate propagates, including that of the most significant bit, are also true:

$$
\begin{aligned}
G0 &= g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0) \\
G1 &= g7 + (p7 \cdot g6) + (p7 \cdot p6 \cdot g5) + (p7 \cdot p6 \cdot p5 \cdot g4) \\
G2 &= g11 + (p11 \cdot g10) + (p11 \cdot p10 \cdot g9) + (p11 \cdot p10 \cdot p9 \cdot g8) \\
G3 &= g15 + (p15 \cdot g14) + (p15 \cdot p14 \cdot g13) + (p15 \cdot p14 \cdot p13 \cdot g12)
\end{aligned}
$$

Figure A.6.2 updates our plumbing analogy to show P0 and G0.

Then the equations at this higher level of abstraction for the carry in for each 4-bit group of the 16-bit adder (C1, C2, C3, C4 in Figure A.6.3) are very similar to the carry out equations for each bit of the 4-bit adder (c1, c2, c3, c4) on page A-40:

$$
\begin{aligned}
C1 &= G0 + (P0 \cdot c0) \\
C2 &= G1 + (P1 \cdot G0) + (P1 \cdot P0 \cdot c0) \\
C3 &= G2 + (P2 \cdot G1) + (P2 \cdot P1 \cdot G0) + (P2 \cdot P1 \cdot P0 \cdot c0) \\
C4 &= G3 + (P3 \cdot G2) + (P3 \cdot P2 \cdot G1) + (P3 \cdot P2 \cdot P1 \cdot G0) \\
&\quad + (P3 \cdot P2 \cdot P1 \cdot P0 \cdot c0)
\end{aligned}
$$

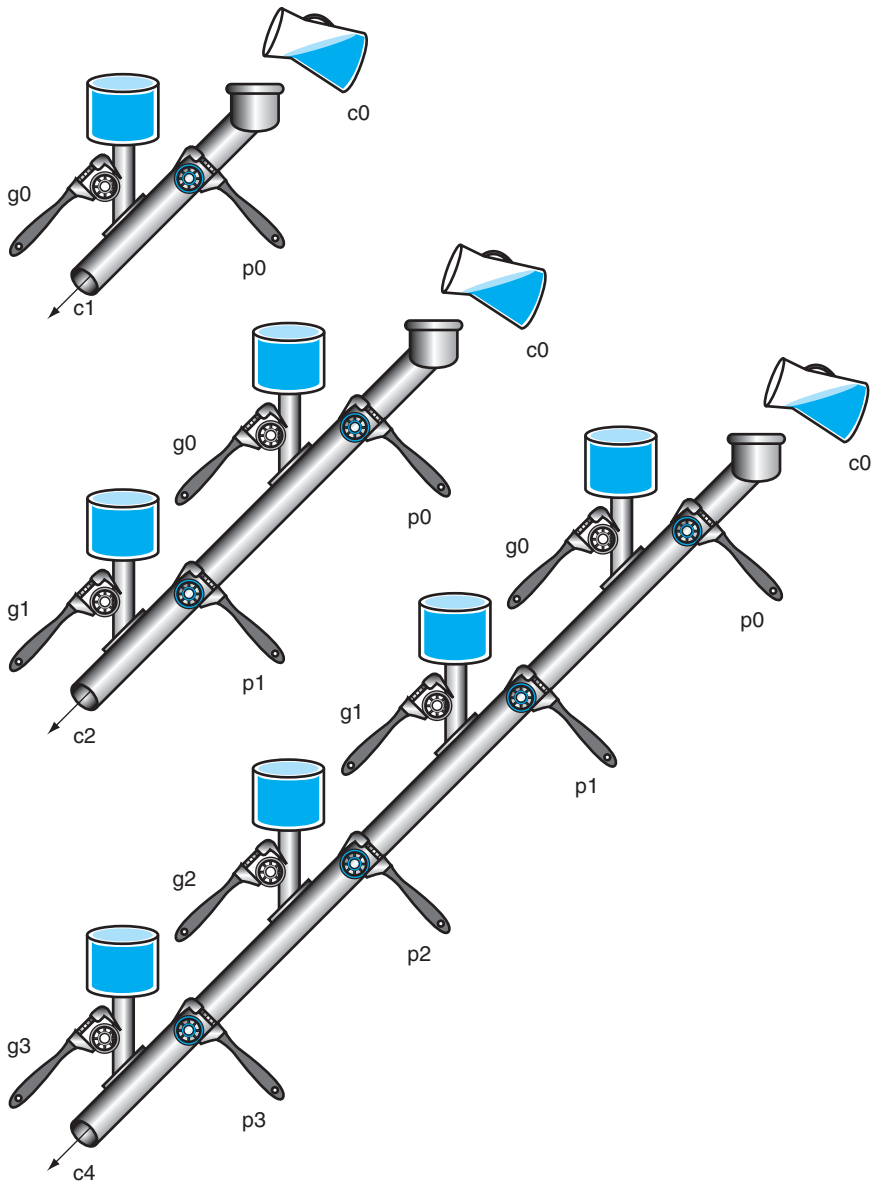


FIGURE A.6.1 A plumbing analogy for carry lookahead for 1 bit, 2 bits, and 4 bits using water pipes and valves. The wrenches are turned to open and close valves. Water is shown in color. The output of the pipe (c_{i+1}) will be full if either the nearest generate value (g*i*) is turned on or if the *i* propagate value (p*i*) is on and there is water further upstream, either from an earlier generate or a propagate with water behind it. CarryIn (c0) can result in a carry out without the help of any generates, but with the help of *all* propagates.

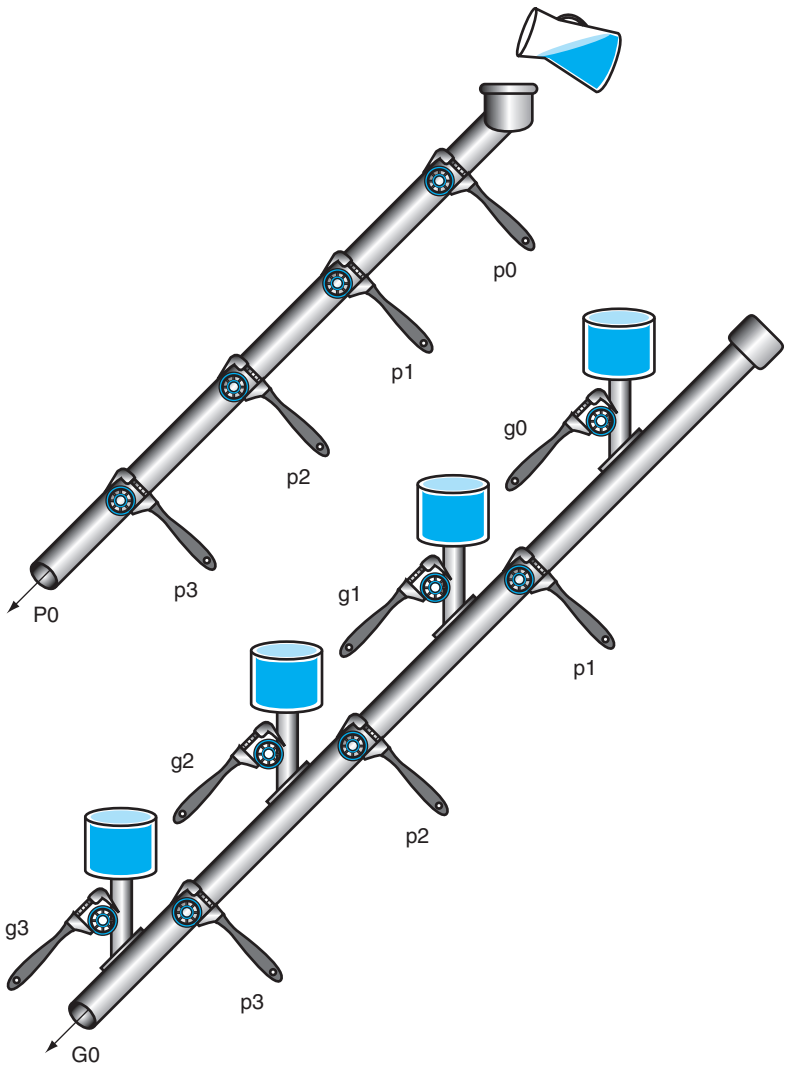


FIGURE A.6.2 A plumbing analogy for the next-level carry-lookahead signals P0 and G0. P0 is open only if all four propagates (p*i*) are open, while water flows in G0 only if at least one generate (g*i*) is open and all the propagates downstream from that generate are open.

Figure A.6.3 shows 4-bit adders connected with such a carry-lookahead unit. The exercises explore the speed differences between these carry schemes, different notations for multibit propagate and generate signals, and the design of a 64-bit adder.

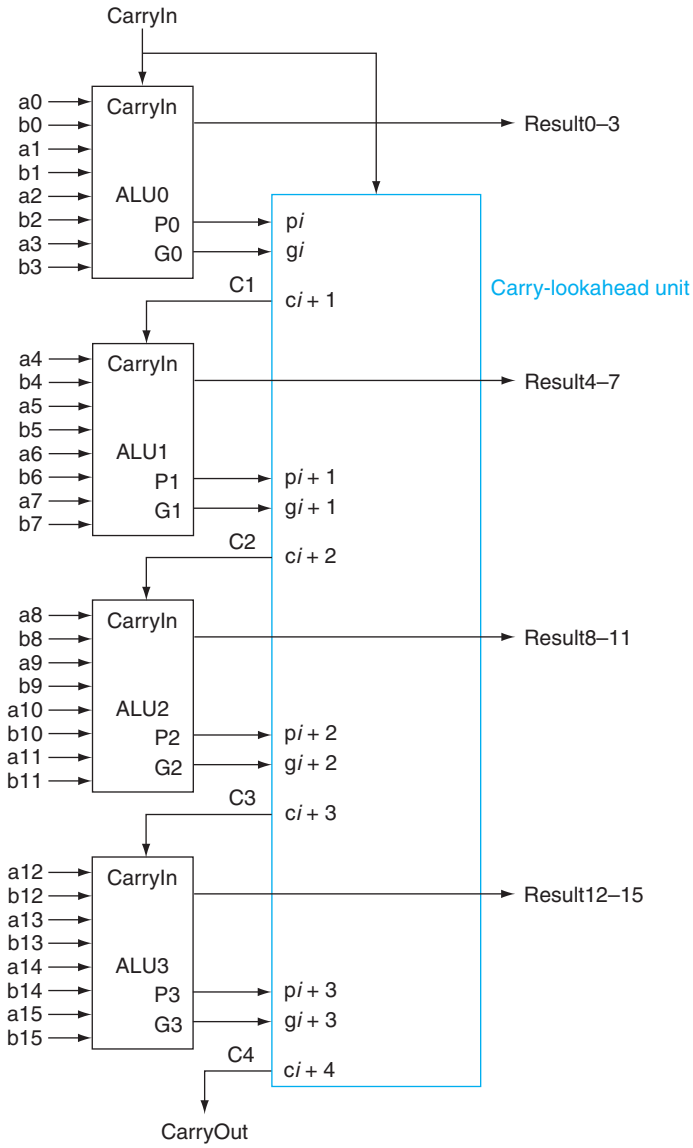


FIGURE A.6.3 Four 4-bit ALUs using carry lookahead to form a 16-bit adder. Note that the carries come from the carry-lookahead unit, not from the 4-bit ALUs.

EXAMPLE

Both Levels of the Propagate and Generate

Determine the g*i*, p*i*, P*i*, and G*i* values of these two 16-bit numbers:

```
a:      0001  1010  0011  0011two
b:      1110  0101  1110  1011two
```

Also, what is CarryOut15 (C4)?

ANSWER

Aligning the bits makes it easy to see the values of generate g*i* (a*i*·b*i*) and propagate p*i* (a*i* + b*i*):

```
a:      0001  1010  0011  0011
b:      1110  0101  1110  1011
gi:     0000  0000  0010  0011
pi:     1111  1111  1111  1011
```

where the bits are numbered 15 to 0 from left to right. Next, the "super" propagates (P3, P2, P1, P0) are simply the AND of the lower-level propagates:

$$P3 = 1 \cdot 1 \cdot 1 \cdot 1 = 1$$
$$P2 = 1 \cdot 1 \cdot 1 \cdot 1 = 1$$
$$P1 = 1 \cdot 1 \cdot 1 \cdot 1 = 1$$
$$P0 = 1 \cdot 0 \cdot 1 \cdot 1 = 0$$

The "super" generates are more complex, so use the following equations:

$$\begin{aligned} G0 &= g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0) \\ &= 0 + (1 \cdot 0) + (1 \cdot 0 \cdot 1) + (1 \cdot \mathbf{0} \cdot 1 \cdot 1) = 0 + 0 + 0 + 0 = 0 \\ G1 &= g7 + (p7 \cdot g6) + (p7 \cdot p6 \cdot g5) + (p7 \cdot p6 \cdot p5 \cdot g4) \\ &= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 1) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 1 + 0 = 1 \\ G2 &= g11 + (p11 \cdot g10) + (p11 \cdot p10 \cdot g9) + (p11 \cdot p10 \cdot p9 \cdot g8) \\ &= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 0 + 0 = 0 \\ G3 &= g15 + (p15 \cdot g14) + (p15 \cdot p14 \cdot g13) + (p15 \cdot p14 \cdot p13 \cdot g12) \\ &= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 0 + 0 = 0 \end{aligned}$$

Finally, CarryOut15 is

$$\begin{aligned} C4 &= G3 + (P3 \cdot G2) + (P3 \cdot P2 \cdot G1) + (P3 \cdot P2 \cdot P1 \cdot G0) \\ &\quad + (P3 \cdot P2 \cdot P1 \cdot P0 \cdot c0) \\ &= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 1) + (1 \cdot 1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0 \cdot 0) \\ &= 0 + 0 + 1 + 0 + 0 = 1 \end{aligned}$$

Hence, there *is* a carry out when adding these two 16-bit numbers.

The reason carry lookahead can make carries faster is that all logic begins evaluating the moment the clock cycle begins, and the result will not change once the output of each gate stops changing. By taking the shortcut of going through fewer gates to send the carry in signal, the output of the gates will stop changing sooner, and hence the time for the adder can be less.

To appreciate the importance of carry lookahead, we need to calculate the relative performance between it and ripple carry adders.

Speed of Ripple Carry versus Carry Lookahead

EXAMPLE

One simple way to model time for logic is to assume each AND or OR gate takes the same time for a signal to pass through it. Time is estimated by simply counting the number of gates along the path through a piece of logic. Compare the number of *gate delays* for paths of two 16-bit adders, one using ripple carry and one using two-level carry lookahead.

ANSWER

Figure A.5.5 on page A-28 shows that the carry out signal takes two gate delays per bit. Then the number of gate delays between a carry in to the least significant bit and the carry out of the most significant is $32 \times 2 = 64$.

For carry lookahead, the carry out of the most significant bit is just C4, defined in the example. It takes two levels of logic to specify C4 in terms of P*i* and G*i* (the OR of several AND terms). P*i* is specified in one level of logic (AND) using p*i*, and G*i* is specified in two levels using p*i* and g*i*, so the worst case for this next level of abstraction is two levels of logic. p*i* and g*i* are each one level of logic, defined in terms of a*i* and b*i*. If we assume one gate delay for each level of logic in these equations, the worst case is $2 + 2 + 1 = 5$ gate delays.

Hence, for the path from carry in to carry out, the 16-bit addition by a carry-lookahead adder is six times faster, using this very simple estimate of hardware speed.

Summary

Carry lookahead offers a faster path than waiting for the carries to ripple through all 32 1-bit adders. This faster path is paved by two signals, generate and propagate.

The former creates a carry regardless of the carry input, and the latter passes a carry along. Carry lookahead also gives another example of how abstraction is important in computer design to cope with complexity.

Check Yourself Using the simple estimate of hardware speed above with gate delays, what is the relative performance of a ripple carry 8-bit add versus a 64-bit add using carry-lookahead logic?

1. A 64-bit carry-lookahead adder is three times faster: 8-bit adds are 16 gate delays and 64-bit adds are seven gate delays.
2. They are about the same speed, since 64-bit adds need more levels of logic in the 16-bit adder.
3. Eight-bit adds are faster than 64 bits, even with carry lookahead.

Elaboration: We have now accounted for all but one of the arithmetic and logical operations for the core RISC-V instruction set: the ALU in Figure A.5.14 omits support of shift instructions. It would be possible to widen the ALU multiplexor to include a left shift by 1 bit or a right shift by 1 bit. But hardware designers have created a circuit called a *barrel shifter*, which can shift from 1 to 63 bits in no more time than it takes to add two 64-bit numbers, so shifting is normally done outside the ALU.

Elaboration: The logic equation for the Sum output of the full adder on page A-28 can be expressed more simply by using a more powerful gate than AND and OR. An *exclusive OR* gate is true if the two operands disagree; that is,

$$x \neq y \Rightarrow 1 \text{ and } x == y \Rightarrow 0$$

In some technologies, exclusive OR is more efficient than two levels of AND and OR gates. Using the symbol $\oplus$ to represent exclusive OR, here is the new equation:

$$\text{Sum} = a \oplus b \oplus \text{CarryIn}$$

Also, we have drawn the ALU the traditional way, using gates. Computers are designed today in CMOS transistors, which are basically switches. CMOS ALU and barrel shifters take advantage of these switches and have many fewer multiplexors than shown in our designs, but the design principles are similar.

Elaboration: Using lowercase and uppercase to distinguish the hierarchy of generate and propagate symbols breaks down when you have more than two levels. An alternate notation that scales is $g_{i,j}$ and $p_{i,j}$ for the generate and propagate signals for bits i to j. Thus, $g_{1..1}$ is generated for bit 1, $g_{4..1}$ is for bits 4 to 1, and $g_{16..1}$ is for bits 16 to 1.

A.7 Clocks

Before we discuss memory elements and sequential logic, it is useful to discuss briefly the topic of clocks. This short section introduces the topic and is similar to the discussion found in Section 4.2. More details on clocking and timing methodologies are presented in Section A.11.

Clocks are needed in sequential logic to decide when an element that contains state should be updated. A clock is simply a free-running signal with a fixed *cycle time*; the *clock frequency* is simply the inverse of the cycle time. As shown in Figure A.7.1, the *clock cycle time* or *clock period* is divided into two portions: when the clock is high and when the clock is low. In this text, we use only **edge-triggered clocking**. This means that all state changes occur on a clock edge. We use an edge-triggered methodology because it is simpler to explain. Depending on the technology, it may or may not be the best choice for a **clocking methodology**.

edge-triggered clocking A clocking scheme in which all state changes occur on a clock edge.

clocking methodology The approach used to determine when data are valid and stable relative to the clock.

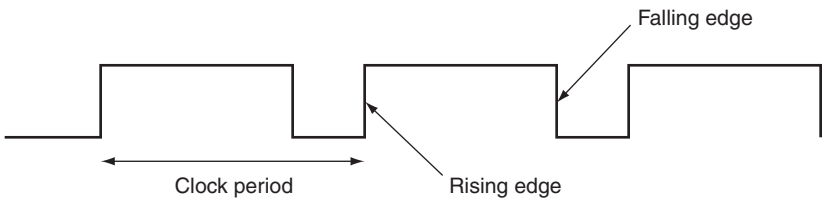


FIGURE A.7.1 A clock signal oscillates between high and low values. The clock period is the time for one full cycle. In an edge-triggered design, either the rising or falling edge of the clock is active and causes state to be changed.

In an edge-triggered methodology, either the rising edge or the falling edge of the clock is *active* and causes state changes to occur. As we will see in the next section, the **state elements** in an edge-triggered design are implemented so that the contents of the state elements only change on the active clock edge. The choice of which edge is active is influenced by the implementation technology and does not affect the concepts involved in designing the logic.

state element A memory element.

The clock edge acts as a sampling signal, causing the value of the data input to a state element to be sampled and stored in the state element. Using an edge trigger means that the sampling process is essentially instantaneous, eliminating problems that could occur if signals were sampled at slightly different times.

The major constraint in a clocked system, also called a **synchronous system**, is that the signals that are written into state elements must be *valid* when the active

synchronous system A memory system that employs clocks and where data signals are read only when the clock indicates that the signal values are stable.

clock edge occurs. A signal is valid if it is stable (i.e., not changing), and the value will not change again until the inputs change. Since combinational circuits cannot have feedback, if the inputs to a combinational logic unit are not changed, the outputs will eventually become valid.

Figure A.7.2 shows the relationship among the state elements and the combinational logic blocks in a synchronous, sequential logic design. The state elements, whose outputs change only after the clock edge, provide valid inputs to the combinational logic block. To ensure that the values written into the state elements on the active clock edge are valid, the clock must have a long enough period so that all the signals in the combinational logic block stabilize, and then the clock edge samples those values for storage in the state elements. This constraint sets a lower bound on the length of the clock period, which must be long enough for all state element inputs to be valid.

In the rest of this appendix, as well as in Chapter 4, we usually omit the clock signal, since we are assuming that all state elements are updated on the same clock edge. Some state elements will be written on every clock edge, while others will be written only under certain conditions (such as a register being updated). In such cases, we will have an explicit write signal for that state element. The write signal must still be gated with the clock so that the update occurs only on the clock edge if the write signal is active. We will see how this is done and used in the next section.

One other advantage of an edge-triggered methodology is that it is possible to have a state element that is used as both an input and output to the same combinational logic block, as shown in Figure A.7.3. In practice, care must be taken to prevent races in such situations and to ensure that the clock period is long enough; this topic is discussed further in Section A.11.

Now that we have discussed how clocking is used to update state elements, we can discuss how to construct the state elements.

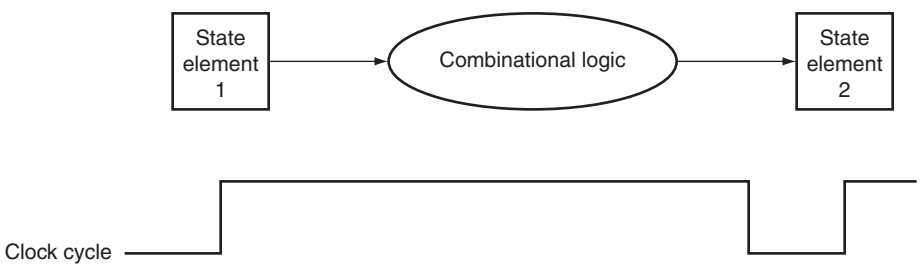


FIGURE A.7.2 The inputs to a combinational logic block come from a state element, and the outputs are written into a state element. The clock edge determines when the contents of the state elements are updated.

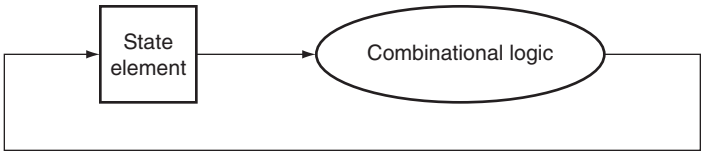


FIGURE A.7.3 An edge-triggered methodology allows a state element to be read and written in the same clock cycle without creating a race that could lead to undetermined data values. Of course, the clock cycle must still be long enough so that the input values are stable when the active clock edge occurs.

Elaboration Occasionally, designers find it useful to have a small number of state elements that change on the opposite clock edge from the majority of the state elements. Doing so requires extreme care, because such an approach has effects on both the inputs and the outputs of the state element. Why then would designers ever do this? Consider the case where the amount of combinational logic before and after a state element is small enough so that each could operate in one-half clock cycle, rather than the more usual full clock cycle. Then the state element can be written on the clock edge corresponding to a half clock cycle, since the inputs and outputs will both be usable after one-half clock cycle. One common place where this technique is used is in **register files**, where simply reading or writing the register file can often be done in half the normal clock cycle. Chapter 4 makes use of this idea to reduce the pipelining overhead.

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.

A.8 Memory Elements: Flip-Flops, Latches, and Registers

In this section and the next, we discuss the basic principles behind memory elements, starting with flip-flops and latches, moving on to register files, and finishing with memories. All memory elements store state: the output from any memory element depends both on the inputs and on the value that has been stored inside the memory element. Thus all logic blocks containing a memory element contain state and are sequential.

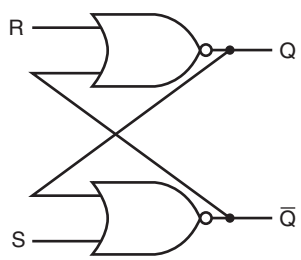


FIGURE A.8.1 A pair of cross-coupled NOR gates can store an internal value. The value stored on the output Q is recycled by inverting it to obtain $\overline{Q}$ and then inverting $\overline{Q}$ to obtain Q. If either R or $\overline{Q}$ is asserted, Q will be deasserted and vice versa.

The simplest type of memory elements are *unclocked*; that is, they do not have any clock input. Although we only use clocked memory elements in this text, an unclocked latch is the simplest memory element, so let's look at this circuit first. Figure A.8.1 shows an *S-R latch* (set-reset latch), built from a pair of NOR gates (OR gates with inverted outputs). The outputs Q and $\overline{Q}$ represent the value of the stored state and its complement. When neither S nor R are asserted, the cross-coupled NOR gates act as inverters and store the previous values of Q and $\overline{Q}$.

For example, if the output, Q, is true, then the bottom inverter produces a false output (which is $\overline{Q}$), which becomes the input to the top inverter, which produces a true output, which is Q, and so on. If S is asserted, then the output Q will be asserted and $\overline{Q}$ will be deasserted, while if R is asserted, then the output $\overline{Q}$ will be asserted and Q will be deasserted. When S and R are both deasserted, the last values of Q and $\overline{Q}$ will continue to be stored in the cross-coupled structure. Asserting S and R simultaneously can lead to incorrect operation: depending on how S and R are deasserted, the latch may oscillate or become metastable (this is described in more detail in Section A.11).

This cross-coupled structure is the basis for more complex memory elements that allow us to store data signals. These elements contain additional gates used to store signal values and to cause the state to be updated only in conjunction with a clock. The next section shows how these elements are built.

flip-flop A memory element for which the output is equal to the value of the stored state inside the element and for which the internal state is changed only on a clock edge.

latch A memory element in which the output is equal to the value of the stored state inside the element and the state is changed whenever the appropriate inputs change and the clock is asserted.

D flip-flop A flip-flop with one data input that stores the value of that input signal in the internal memory when the clock edge occurs.

Flip-Flops and Latches

Flip-flops and **latches** are the simplest memory elements. In both flip-flops and latches, the output is equal to the value of the stored state inside the element. Furthermore, unlike the S-R latch described above, all the latches and flip-flops we will use from this point on are clocked, which means that they have a clock input and the change of state is triggered by that clock. The difference between a flip-flop and a latch is the point at which the clock causes the state to actually change. In a clocked latch, the state is changed whenever the appropriate inputs change and the clock is asserted, whereas in a flip-flop, the state is changed only on a clock edge. Since throughout this text we use an edge-triggered timing methodology where state is only updated on clock edges, we need only use flip-flops. Flip-flops are often built from latches, so we start by describing the operation of a simple clocked latch and then discuss the operation of a flip-flop constructed from that latch.

For computer applications, the function of both flip-flops and latches is to store a signal. A *D latch* or **D flip-flop** stores the value of its data input signal in the internal memory. Although there are many other types of latch and flip-flop, the D type is the only basic building block that we will need. A D latch has two inputs and two outputs. The inputs are the data value to be stored (called D) and a clock signal (called C) that indicates when the latch should read the value on the D input and store it. The outputs are simply the value of the internal state (Q)

and its complement ($\overline{Q}$). When the clock input C is asserted, the latch is said to be *open*, and the value of the output (Q) becomes the value of the input D. When the clock input C is deasserted, the latch is said to be *closed*, and the value of the output (Q) is whatever value was stored the last time the latch was open.

Figure A.8.2 shows how a D latch can be implemented with two additional gates added to the cross-coupled NOR gates. Since when the latch is open the value of Q changes as D changes, this structure is sometimes called a *transparent latch*. Figure A.8.3 shows how this D latch works, assuming that the output Q is initially false and that D changes first.

As mentioned earlier, we use flip-flops as the basic building block, rather than latches. Flip-flops are not transparent: their outputs change *only* on the clock edge. A flip-flop can be built so that it triggers on either the rising (positive) or falling (negative) clock edge; for our designs we can use either type. Figure A.8.4 shows how a falling-edge D flip-flop is constructed from a pair of D latches. In a D flip-flop, the output is stored when the clock edge occurs. Figure A.8.5 shows how this flip-flop operates.

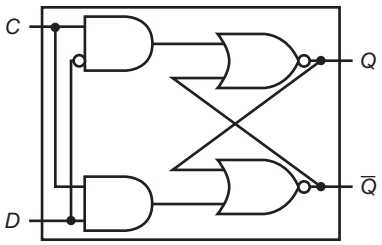


FIGURE A.8.2 A D latch implemented with NOR gates. A NOR gate acts as an inverter if the other input is 0. Thus, the cross-coupled pair of NOR gates acts to store the state value unless the clock input, C, is asserted, in which case the value of input D replaces the value of Q and is stored. The value of input D must be stable when the clock signal C changes from asserted to deasserted.

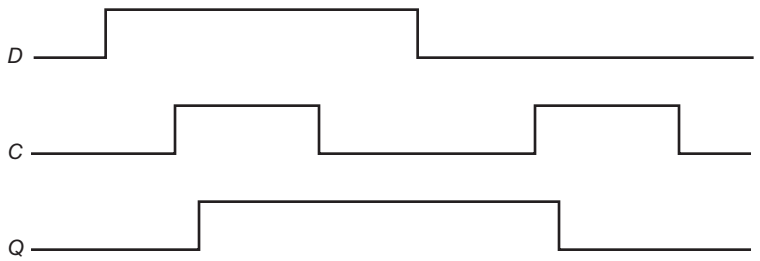


FIGURE A.8.3 Operation of a D latch, assuming the output is initially deasserted. When the clock, C, is asserted, the latch is open and the Q output immediately assumes the value of the D input.

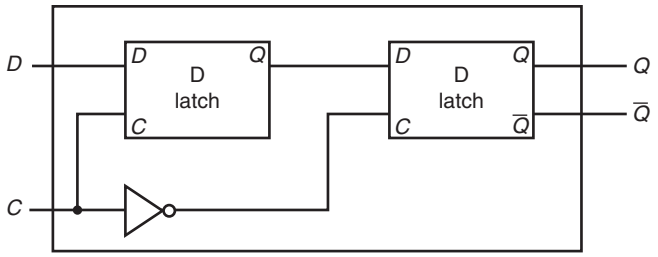


FIGURE A.8.4 A D flip-flop with a falling-edge trigger. The first latch, called the master, is open and follows the input *D* when the clock input, *C*, is asserted. When the clock input, *C*, falls, the first latch is closed, but the second latch, called the slave, is open and gets its input from the output of the master latch.

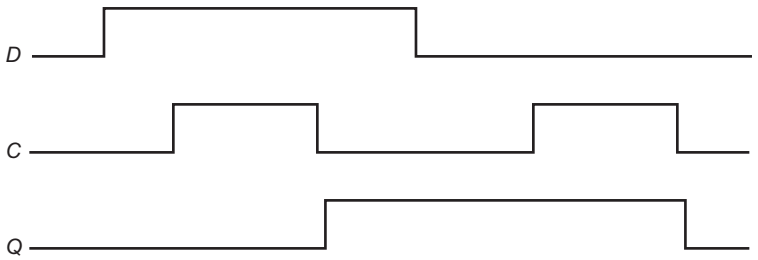


FIGURE A.8.5 Operation of a D flip-flop with a falling-edge trigger, assuming the output is initially deasserted. When the clock input (*C*) changes from asserted to deasserted, the *Q* output stores the value of the *D* input. Compare this behavior to that of the clocked D latch shown in Figure A.8.3. In a clocked latch, the stored value and the output, *Q*, both change whenever *C* is high, as opposed to only when *C* transitions.

Here is a Verilog description of a module for a rising-edge D flip-flop, assuming that C is the clock input and D is the data input:

```
module DFF(clock,D,Q,Qbar);
    input clock, D;
    output reg Q;
    output Qbar;
    assign Qbar= ~ Q;
    always @(posedge clock)
       Q=D;
endmodule
```

setup time The minimum time that the input to a memory device must be valid before the clock edge.

Because the *D* input is sampled on the clock edge, it must be valid for a period of time immediately before and immediately after the clock edge. The minimum time that the input must be valid before the clock edge is called the **setup time**; the

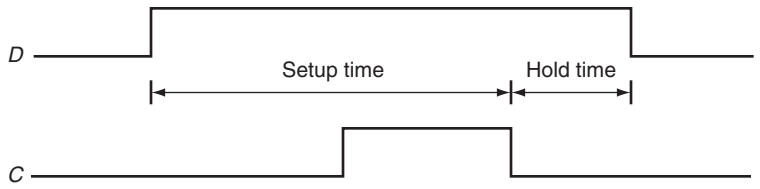


FIGURE A.8.6 Setup and hold time requirements for a D flip-flop with a falling-edge trigger. The input must be stable for a period of time before the clock edge, as well as after the clock edge. The minimum time the signal must be stable before the clock edge is called the setup time, while the minimum time the signal must be stable after the clock edge is called the hold time. Failure to meet these minimum requirements can result in a situation where the output of the flip-flop may not be predictable, as described in Section A.11. Hold times are usually either 0 or very small and thus not a cause of worry.

minimum time during which it must be valid after the clock edge is called the **hold time**. Thus the inputs to any flip-flop (or anything built using flip-flops) must be valid during a window that begins at time t_{setup} before the clock edge and ends at t_{hold} after the clock edge, as shown in Figure A.8.6. Section A.11 talks about clocking and timing constraints, including the propagation delay through a flip-flop, in more detail.

hold time The minimum time during which the input must be valid after the clock edge.

We can use an array of D flip-flops to build a register that can hold a multibit datum, such as a byte or word. We used registers throughout our datapaths in Chapter 4.

Register Files

One structure that is central to our datapath is a *register file*. A register file consists of a set of registers that can be read and written by supplying a register number to be accessed. A register file can be implemented with a decoder for each read or write port and an array of registers built from D flip-flops. Because reading a register does not change any state, we need only supply a register number as an input, and the only output will be the data contained in that register. For writing a register we will need three inputs: a register number, the data to write, and a clock that controls the writing into the register. In Chapter 4, we used a register file that has two read ports and one write port. This register file is drawn as shown in Figure A.8.7. The read ports can be implemented with a pair of multiplexors, each of which is as wide as the number of bits in each register of the register file. Figure A.8.8 shows the implementation of two register read ports for a 64-bit-wide register file.

Implementing the write port is slightly more complex, since we can only change the contents of the designated register. We can do this by using a decoder to generate a signal that can be used to determine which register to write. Figure A.8.9 shows how to implement the write port for a register file. It is important to remember that the flip-flop changes state only on the clock edge. In Chapter 4, we hooked up write signals for the register file explicitly and assumed the clock shown in Figure A.8.9 is attached implicitly.

What happens if the same register is read and written during a clock cycle? Because the write of the register file occurs on the clock edge, the register will be

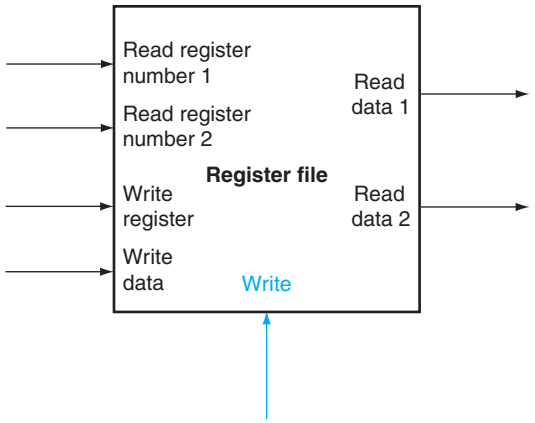


FIGURE A.8.7 A register file with two read ports and one write port has five inputs and two outputs. The control input Write is shown in color.

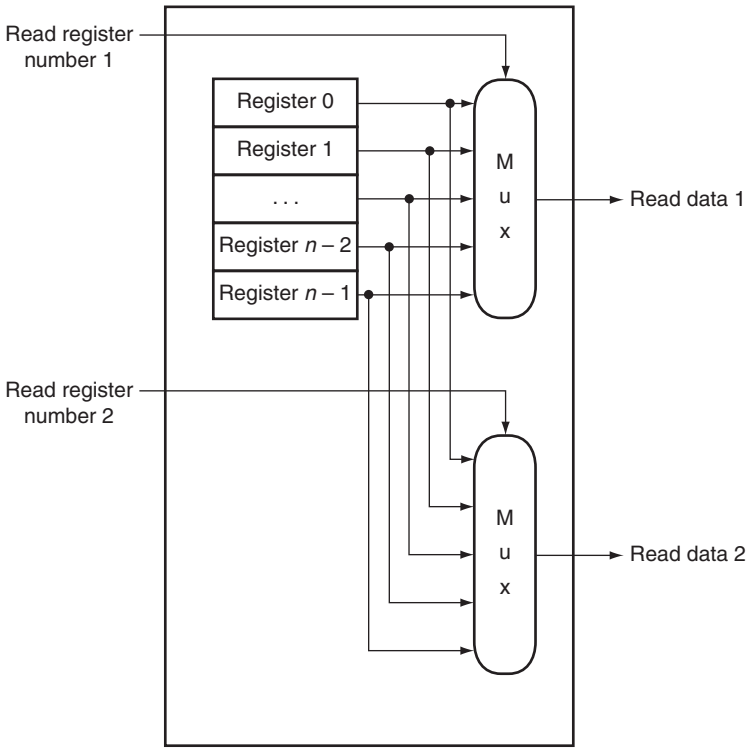


FIGURE A.8.8 The implementation of two read ports for a register file with *n* registers can be done with a pair of *n*-to-1 multiplexors, each 64 bits wide. The register read number signal is used as the multiplexor selector signal. Figure A.8.9 shows how the write port is implemented.

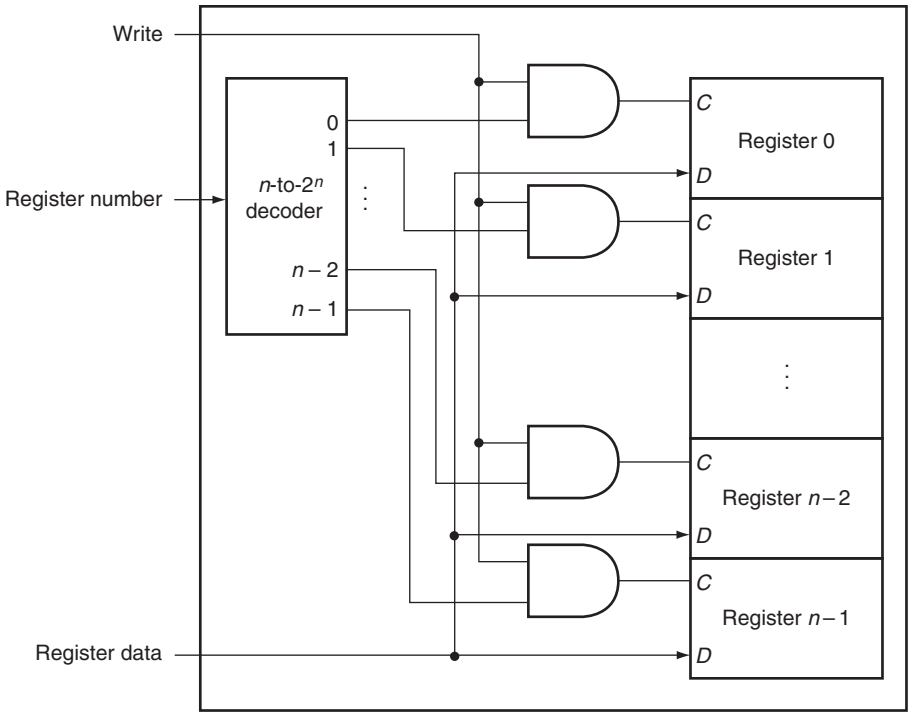


FIGURE A.8.9 The write port for a register file is implemented with a decoder that is used with the write signal to generate the C input to the registers. All three inputs (the register number, the data, and the write signal) will have setup and hold-time constraints that ensure that the correct data are written into the register file.

valid during the time it is read, as we saw earlier in Figure A.7.2. The value returned will be the value written in an earlier clock cycle. If we want a read to return the value currently being written, additional logic in the register file or outside of it is needed. Chapter 4 makes extensive use of such logic.

Specifying Sequential Logic in Verilog

To specify sequential logic in Verilog, we must understand how to generate a clock, how to describe when a value is written into a register, and how to specify sequential control. Let us start by specifying a clock. A clock is not a predefined object in Verilog; instead, we generate a clock by using the Verilog notation `#n` before a statement; this causes a delay of `n` simulation time steps before the execution of the statement. In most Verilog simulators, it is also possible to generate a clock as an external input, allowing the user to specify at simulation time the number of clock cycles during which to run a simulation.

The code in Figure A.8.10 implements a simple clock that is high or low for one simulation unit and then switches state. We use the delay capability and blocking assignment to implement the clock.

```
reg clock;
always #1 clock = ~clock;
```

FIGURE A.8.10 A specification of a clock.

Next, we must be able to specify the operation of an edge-triggered register. In Verilog, this is done by using the sensitivity list on an `always` block and specifying as a trigger either the positive or negative edge of a binary variable with the notation `posedge` or `negedge`, respectively. Hence, the following Verilog code causes register `A` to be written with the value b at the positive edge clock:

```
reg [63:0] A;
wire [63:0] b;

always @(posedge clock) A <= b;
```

```
module registerfile (Read1,Read2,WriteReg,WriteData,RegWrite,
Data1,Data2,clock);
   input [5:0] Read1,Read2,WriteReg; // the register numbers
to read or write
   input [63:0] WriteData; // data to write
   input RegWrite, // the write control
     clock; // the clock to trigger write
   output [63:0] Data1, Data2; // the register values read
   reg [63:0] RF [31:0]; // 32 registers each 32 bits long

   assign Data1 = RF[Read1];
   assign Data2 = RF[Read2];

   always begin
      // write the register with new value if Regwrite is
high
      @(posedge clock) if (RegWrite) RF[WriteReg] <=
WriteData;
   end
endmodule
```

FIGURE A.8.11 A RISC-V register file written in behavioral Verilog. This register file writes on the rising clock edge.

Throughout this chapter and the Verilog sections of Chapter 4, we will assume a positive edge-triggered design. Figure A.8.11 shows a Verilog specification of a RISC-V register file that assumes two reads and one write, with only the write being clocked.

Check Yourself

In the Verilog for the register file in Figure A.8.11, the output ports corresponding to the registers being read are assigned using a continuous assignment, but the register being written is assigned in an `always` block. Which of the following is the reason?

a. There is no special reason. It was simply convenient.

b. Because Data1 and Data2 are output ports and WriteData is an input port.

c. Because reading is a combinational event, while writing is a sequential event.

A.9 Memory Elements: SRAMs and DRAMs

Registers and register files provide the basic building blocks for small memories, but larger amounts of memory are built using either **SRAMs (static random access memories)** or *DRAMs* (dynamic random access memories). We first discuss SRAMs, which are somewhat simpler, and then turn to DRAMs.

static random access memory (SRAM) A memory where data are stored statically (as in flip-flops) rather than dynamically (as in DRAM). SRAMs are faster than DRAMs, but less dense and more expensive per bit.

SRAMs

SRAMs are simply integrated circuits that are memory arrays with (usually) a single access port that can provide either a read or a write. SRAMs have a fixed access time to any datum, though the read and write access characteristics often differ. An SRAM chip has a specific configuration in terms of the number of addressable locations, as well as the width of each addressable location. For example, a 4M × 8 SRAM provides 4M entries, each of which is 8 bits wide. Thus it will have 22 address lines (since $4M = 2^{22}$), an 8-bit data output line, and an 8-bit single data input line. As with ROMs, the number of addressable locations is often called the *height*, with the number of bits per unit called the *width*. For a variety of technical reasons, the newest and fastest SRAMs are typically available in narrow configurations: × 1 and × 4. Figure A.9.1 shows the input and output signals for a 2M × 16 SRAM.

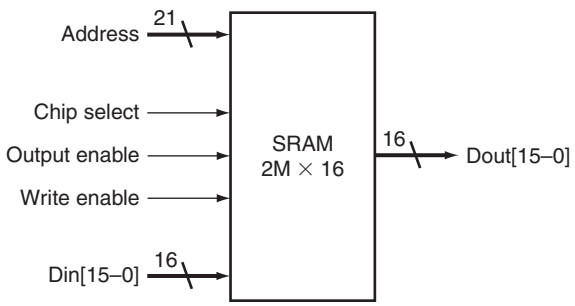


FIGURE A.9.1 A 32K × 8 SRAM showing the 21 address lines ($32K = 2^{15}$) and 16 data inputs, the three control lines, and the 16 data outputs.

To initiate a read or write access, the Chip select signal must be made active. For reads, we must also activate the Output enable signal that controls whether or not the datum selected by the address is actually driven on the pins. The Output enable is useful for connecting multiple memories to a single-output bus and using Output enable to determine which memory drives the bus. The SRAM read access time is usually specified as the delay from the time that Output enable is true and the address lines are valid until the time that the data are on the output lines. Typical read access times for SRAMs in 2004 varied from about 2–4 ns for the fastest CMOS parts, which tend to be somewhat smaller and narrower, to 8–20 ns for the typical largest parts, which in 2004 had more than 32 million bits of data. The demand for low-power SRAMs for consumer products and digital appliances has grown greatly in the past 5 years; these SRAMs have much lower stand-by and access power, but usually are 5–10 times slower. Most recently, synchronous SRAMs—similar to the synchronous DRAMs, which we discuss in the next section—have also been developed.

For writes, we must supply the data to be written and the address, as well as signals to cause the write to occur. When both the Write enable and Chip select are true, the data on the data input lines are written into the cell specified by the address. There are setup-time and hold-time requirements for the address and data lines, just as there were for D flip-flops and latches. In addition, the Write enable signal is not a clock edge but a pulse with a minimum width requirement. The time to complete a write is specified by the combination of the setup times, the hold times, and the Write enable pulse width.

Large SRAMs cannot be built in the same way we build a register file because, unlike a register file where a 32-to-1 multiplexor might be practical, the 64K-to-1 multiplexor that would be needed for a 64K × 1 SRAM is totally impractical. Rather than use a giant multiplexor, large memories are implemented with a shared output line, called a *bit line*, which multiple memory cells in the memory array can assert. To allow multiple sources to drive a single line, a *three-state buffer* (or *tristate buffer*) is used. A three-state buffer has two inputs—a data signal and an Output enable—and a single output, which is in one of three states: asserted, deasserted, or high impedance. The output of a tristate buffer is equal to the data input signal, either asserted or deasserted, if the Output enable is asserted, and is otherwise in a *high-impedance state* that allows another three-state buffer whose Output enable is asserted to determine the value of a shared output.

Figure A.9.2 shows a set of three-state buffers wired to form a multiplexor with a decoded input. It is critical that the Output enable at most one of the three-state buffers be asserted; otherwise, the three-state buffers may try to set the output line differently. By using three-state buffers in the individual cells of the SRAM, each cell that corresponds to a particular output can share the same output line. The use of a set of distributed three-state buffers is a more efficient implementation than a large centralized multiplexor. The three-state buffers are incorporated into the flip-flops that form the basic cells of the SRAM. Figure A.9.3 shows how a small 4 × 2 SRAM might be built, using D latches with an input called Enable that controls the three-state output.

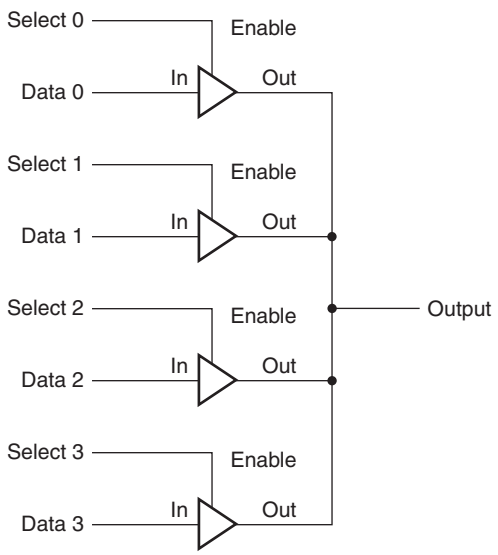


FIGURE A.9.2 Four three-state buffers are used to form a multiplexor. Only one of the four Select inputs can be asserted. A three-state buffer with a deasserted Output enable has a high-impedance output that allows a three-state buffer whose Output enable is asserted to drive the shared output line.

The design in Figure A.9.3 eliminates the need for an enormous multiplexor; however, it still requires a very large decoder and a correspondingly large number of word lines. For example, in a 4M × 8 SRAM, we would need a 22-to-4M decoder and 4M word lines (which are the lines used to enable the individual flip-flops)! To circumvent this problem, large memories are organized as rectangular arrays and use a two-step decoding process. Figure A.9.4 shows how a 4M × 8 SRAM might be organized internally using a two-step decode. As we will see, the two-level decoding process is quite important in understanding how DRAMs operate.

Recently we have seen the development of both synchronous SRAMs (SSRAMs) and synchronous DRAMs (SDRAMs). The key capability provided by synchronous RAMs is the ability to transfer a *burst* of data from a series of sequential addresses within an array or row. The burst is defined by a starting address, supplied in the usual fashion, and a burst length. The speed advantage of synchronous RAMs comes from the ability to transfer the bits in the burst without having to specify additional address bits. Instead, a clock is used to transfer the successive bits in the burst. The elimination of the need to specify the address for the transfers within the burst significantly improves the rate for transferring the block of data. Because of this capability, synchronous SRAMs and DRAMs are rapidly becoming the RAMs of choice for building memory systems in computers. We discuss the use of synchronous DRAMs in a memory system in more detail in the next section and in Chapter 5.

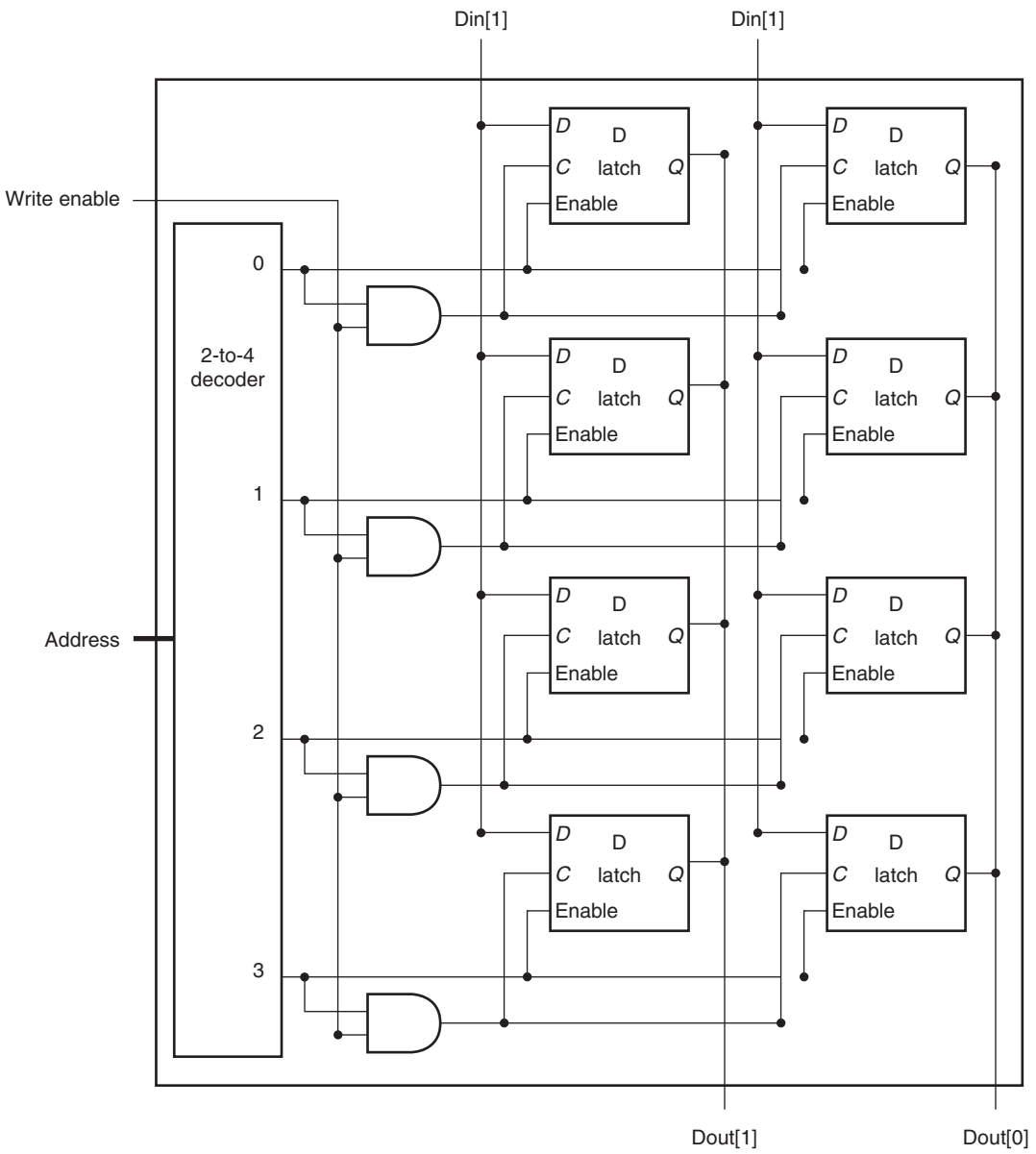


FIGURE A.9.3 The basic structure of a 4 × 2 SRAM consists of a decoder that selects which pair of cells to activate. The activated cells use a three-state output connected to the vertical bit lines that supply the requested data. The address that selects the cell is sent on one of a set of horizontal address lines, called word lines. For simplicity, the Output enable and Chip select signals have been omitted, but they could easily be added with a few AND gates.

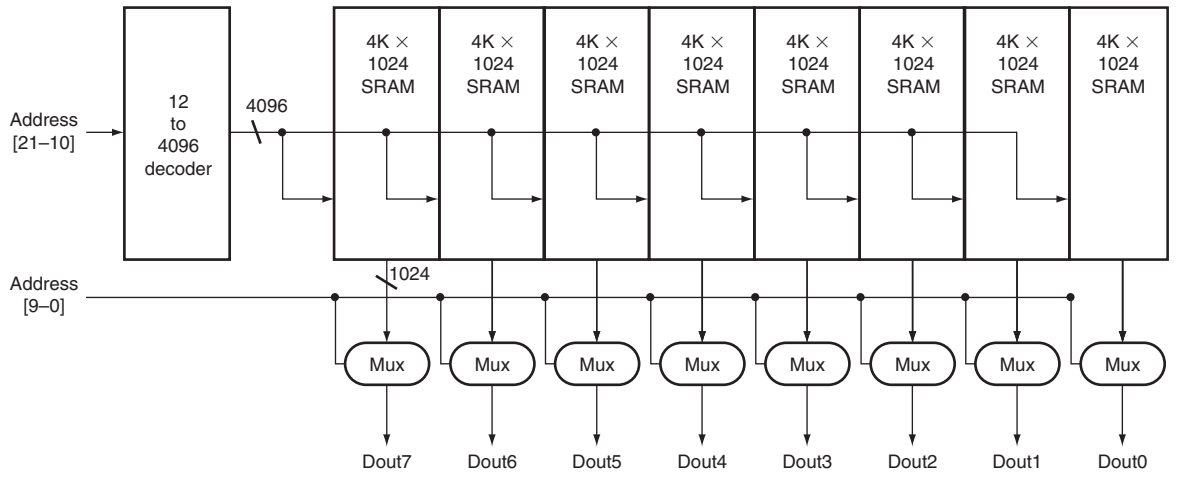


FIGURE A.9.4 Typical organization of a 4M × 8 SRAM as an array of 4K × 1024 arrays. The first decoder generates the addresses for eight 4K × 1024 arrays; then a set of multiplexors is used to select 1 bit from each 1024-bit-wide array. This is a much easier design than a single-level decode that would need either an enormous decoder or a gigantic multiplexor. In practice, a modern SRAM of this size would probably use an even larger number of blocks, each somewhat smaller.

DRAMs

In a static RAM (SRAM), the value stored in a cell is kept on a pair of inverting gates, and as long as power is applied, the value can be kept indefinitely. In a dynamic RAM (DRAM), the value kept in a cell is stored as a charge in a capacitor. A single transistor is then used to access this stored charge, either to read the value or to overwrite the charge stored there. Because DRAMs use only a single transistor per bit of storage, they are much denser and cheaper per bit. By comparison, SRAMs require four to six transistors per bit. Because DRAMs store the charge on a capacitor, it cannot be kept indefinitely and must periodically be *refreshed*. That is why this memory structure is called *dynamic*, as opposed to the static storage in a SRAM cell.

To refresh the cell, we merely read its contents and write it back. The charge can be kept for several milliseconds, which might correspond to close to a million clock cycles. Today, single-chip memory controllers often handle the refresh function independently of the processor. If every bit had to be read out of the DRAM and then written back individually, with large DRAMs containing multiple megabytes, we would constantly be refreshing the DRAM, leaving no time for accessing it. Fortunately, DRAMs also use a two-level decoding structure, and this allows us to refresh an entire row (which shares a word line) with a read cycle followed immediately by a write cycle. Typically, refresh operations consume 1% to 2% of the active cycles of the DRAM, leaving the remaining 98% to 99% of the cycles available for reading and writing data.

Elaboration: How does a DRAM read and write the signal stored in a cell? The transistor inside the cell is a switch, called a *pass transistor*, that allows the value stored on the capacitor to be accessed for either reading or writing. Figure A.9.5 shows how the single-transistor cell looks. The pass transistor acts like a switch: when the signal on the word line is asserted, the switch is closed, connecting the capacitor to the bit line. If the operation is a write, then the value to be written is placed on the bit line. If the value is a 1, the capacitor will be charged. If the value is a 0, then the capacitor will be discharged. Reading is slightly more complex, since the DRAM must detect a very small charge stored in the capacitor. Before activating the word line for a read, the bit line is charged to the voltage that is halfway between the low and high voltage. Then, by activating the word line, the charge on the capacitor is read out onto the bit line. This causes the bit line to move slightly toward the high or low direction, and this change is detected with a sense amplifier, which can detect small changes in voltage.

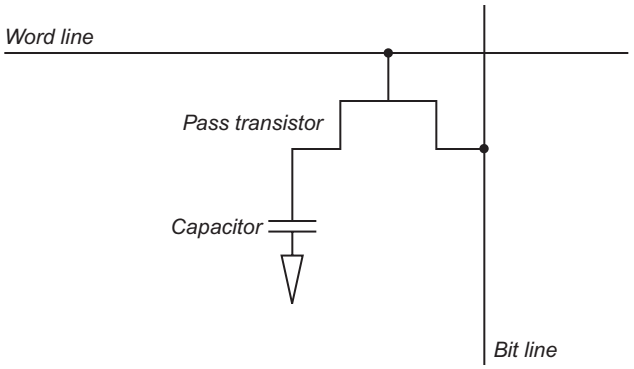


FIGURE A.9.5 A single-transistor DRAM cell contains a capacitor that stores the cell contents and a transistor used to access the cell.

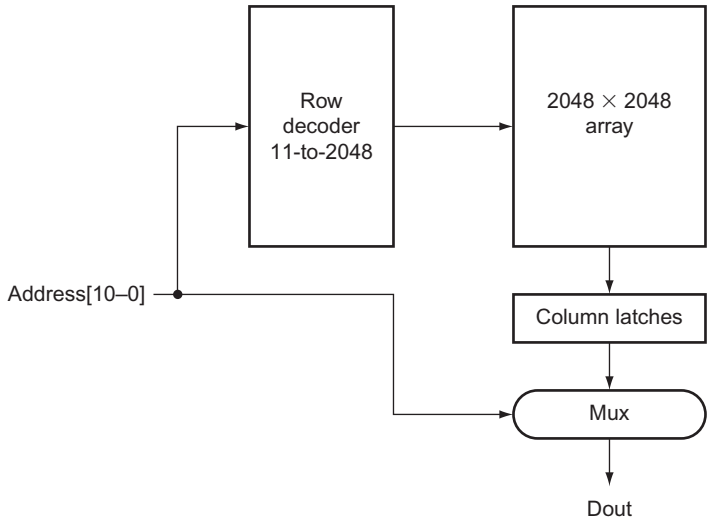


FIGURE A.9.6 A 4M × 1 DRAM is built with a 2048 × 2048 array. The row access uses 11 bits to select a row, which is then latched in 2048 1-bit latches. A multiplexor chooses the output bit from these 2048 latches. The RAS and CAS signals control whether the address lines are sent to the row decoder or column multiplexor.

DRAMs use a two-level decoder consisting of a *row access* followed by a *column access*, as shown in Figure A.9.6. The row access chooses one of a number of rows and activates the corresponding word line. The contents of all the columns in the active row are then stored in a set of latches. The column access then selects the data from the column latches. To save pins and reduce the package cost, the same address lines are used for both the row and column address; a pair of signals called RAS (*Row Access Strobe*) and CAS (*Column Access Strobe*) are used to signal the DRAM that either a row or column address is being supplied. Refresh is performed by simply reading the columns into the column latches and then writing the same values back. Thus, an entire row is refreshed in one cycle. The two-level addressing scheme, combined with the internal circuitry, makes DRAM access times much longer (by a factor of 5–10) than SRAM access times. In 2004, typical DRAM access times ranged from 45 to 65 ns; 256 Mbit DRAMs are in full production, and the first customer samples of 1 GB DRAMs became available in the first quarter of 2004. The much lower cost per bit makes DRAM the choice for main memory, while the faster access time makes SRAM the choice for caches.

You might observe that a 64M × 4 DRAM actually accesses 8K bits on every row access and then throws away all but four of those during a column access. DRAM designers have used the internal structure of the DRAM as a way to provide higher bandwidth out of a DRAM. This is done by allowing the column address to change without changing the row address, resulting in an access to other bits in the column latches. To make this process faster and more precise, the address inputs were clocked, leading to the dominant form of DRAM in use today: synchronous DRAM or SDRAM.

Since about 1999, SDRAMs have been the memory chip of choice for most cache-based main memory systems. SDRAMs provide fast access to a series of bits within a row by sequentially transferring all the bits in a burst under the control of a clock signal. In 2004, DDRRAMs (Double Data Rate RAMs), which are called double data rate because they transfer data on both the rising and falling edge of an externally supplied clock, were the most heavily used form of SDRAMs. As we discuss in Chapter 5, these high-speed transfers can be used to boost the bandwidth available out of main memory to match the needs of the processor and caches.

Error Correction

Because of the potential for data corruption in large memories, most computer systems use some sort of error-checking code to detect possible corruption of data. One simple code that is heavily used is a *parity code*. In a parity code the number of 1s in a word is counted; the word has odd parity if the number of 1s is odd and

even otherwise. When a word is written into memory, the parity bit is also written (1 for odd, 0 for even). Then, when the word is read out, the parity bit is read and checked. If the parity of the memory word and the stored parity bit do not match, an error has occurred.

A 1-bit parity scheme can detect at most 1 bit of error in a data item; if there are 2 bits of error, then a 1-bit parity scheme will not detect any errors, since the parity will match the data with two errors. (Actually, a 1-bit parity scheme can detect any odd number of errors; however, the probability of having three errors is much lower than the probability of having two, so, in practice, a 1-bit parity code is limited to detecting a single bit of error.) Of course, a parity code cannot tell which bit in a data item is in error.

A 1-bit parity scheme is an **error detection code**; there are also *error correction codes* (ECC) that will detect and allow correction of an error. For large main memories, many systems use a code that allows the detection of up to 2 bits of error and the correction of a single bit of error. These codes work by using more bits to encode the data; for example, the typical codes used for main memories require 7 or 8 bits for every 128 bits of data.

error detection code A code that enables the detection of an error in data, but not the precise location and, hence, correction of the error.

Elaboration: A 1-bit parity code is a *distance-2 code*, which means that if we look at the data plus the parity bit, no 1-bit change is sufficient to generate another legal combination of the data plus parity. For example, if we change a bit in the data, the parity will be wrong, and vice versa. Of course, if we change 2 bits (any 2 data bits or 1 data bit and the parity bit), the parity will match the data and the error cannot be detected. Hence, there is a distance of two between legal combinations of parity and data.

To detect more than one error or correct an error, we need a *distance-3 code*, which has the property that any legal combination of the bits in the error correction code and the data has at least 3 bits differing from any other combination. Suppose we have such a code and we have one error in the data. In that case, the code plus data will be one bit away from a legal combination, and we can correct the data to that legal combination. If we have two errors, we can recognize that there is an error, but we cannot correct the errors. Let's look at an example. Here are the data words and a distance-3 error correction code for a 4-bit data item.

Data Word	Code bits	Data	Code bits
0000	000	1000	111
0001	011	1001	100
0010	101	1010	010
0011	110	1011	001
0100	110	1100	001
0101	101	1101	010
0110	011	1110	100
0111	000	1111	111

To see how this works, let's choose a data word, say 0110, whose error correction code is 011. Here are the four 1-bit error possibilities for this data: 1110, 0010, 0100, and 0111. Now look at the data item with the same code (011), which is the entry with the value 0001. If the error correction decoder received one of the four possible data words with an error, it would have to choose between correcting to 0110 or 0001. While these four words with error have only 1 bit changed from the correct pattern of 0110, they each have 2 bits that are different from the alternate correction of 0001. Hence, the error correction mechanism can easily choose to correct to 0110, since a single error is a much higher probability. To see that two errors can be detected, simply notice that all the combinations with 2 bits changed have a different code. The one reuse of the same code is with 3 bits different, but if we correct a 2-bit error, we will correct to the wrong value, since the decoder will assume that only a single error has occurred. If we want to correct 1-bit errors and detect, but not erroneously correct, 2-bit errors, we need a distance-4 code.

Although we distinguished between the code and data in our explanation, in truth, an error correction code treats the combination of code and data as a single word in a larger code (7 bits in this example). Thus, it deals with errors in the code bits in the same fashion as errors in the data bits.

While the above example requires $n - 1$ bits for n bits of data, the number of bits required grows slowly, so that for a distance-3 code, a 64-bit word needs 7 bits and a 128-bit word needs 8. This type of code is called a *Hamming code*, after R. Hamming, who described a method for creating such codes.

A.10 Finite-State Machines

finite-state machine A sequential logic function consisting of a set of inputs and outputs, a next-state function that maps the current state and the inputs to a new state, and an output function that maps the current state and possibly the inputs to a set of asserted outputs.

next-state function A combinational function that, given the inputs and the current state, determines the next state of a finite-state machine.

As we saw earlier, digital logic systems can be classified as combinational or sequential. Sequential systems contain state stored in memory elements internal to the system. Their behavior depends both on the set of inputs supplied and on the contents of the internal memory, or state of the system. Thus, a sequential system cannot be described with a truth table. Instead, a sequential system is described as a **finite-state machine** (or often just *state machine*). A finite-state machine has a set of states and two functions, called the **next-state function** and the *output function*. The set of states corresponds to all the possible values of the internal storage. Thus, if there are n bits of storage, there are $2n$ states. The next-state function is a combinational function that, given the inputs and the current state, determines the next state of the system. The output function produces a set of outputs from the current state and the inputs. Figure A.10.1 shows this diagrammatically.

The state machines we discuss here and in Chapter 4 are *synchronous*. This means that the state changes together with the clock cycle, and a new state is computed once every clock. Thus, the state elements are updated only on the clock edge. We use this methodology in this section and throughout Chapter 4, and we do not usually show the clock explicitly. We use state machines throughout Chapter 4 to control the execution of the processor and the actions of the datapath.

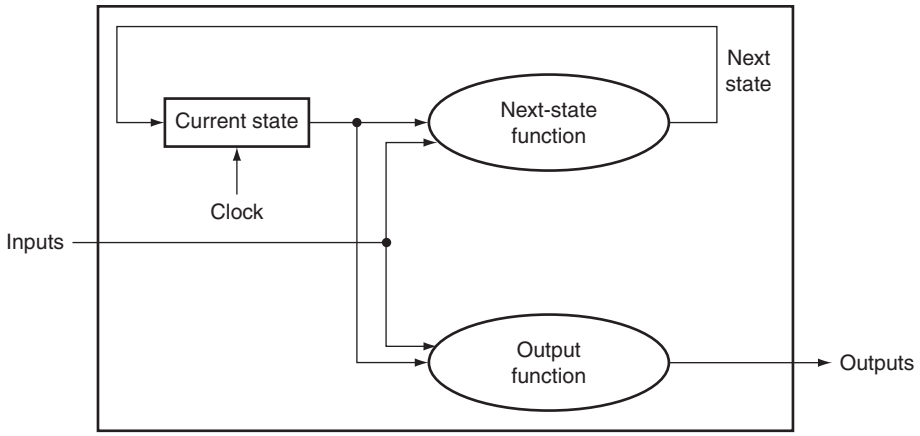


FIGURE A.10.1 A state machine consists of internal storage that contains the state and two combinational functions: the next-state function and the output function. Often, the output function is restricted to take only the current state as its input; this does not change the capability of a sequential machine, but does affect its internals.

To illustrate how a finite-state machine operates and is designed, let's look at a simple and classic example: controlling a traffic light. (Chapters 4 and 5 contain more detailed examples of using finite-state machines to control processor execution.) When a finite-state machine is used as a controller, the output function is often restricted to depend on just the current state. Such a finite-state machine is called a *Moore machine*. This is the type of finite-state machine we use throughout this book. If the output function can depend on both the current state and the current input, the machine is called a *Mealy machine*. These two machines are equivalent in their capabilities, and one can be turned into the other mechanically. The basic advantage of a Moore machine is that it can be faster, while a Mealy machine may be smaller, since it may need fewer states than a Moore machine. In Chapter 5, we discuss the differences in more detail and show a Verilog version of finite-state control using a Mealy machine.

Our example concerns the control of a traffic light at an intersection of a north-south route and an east-west route. For simplicity, we will consider only the green and red lights; adding the yellow light is left for an exercise. We want the lights to cycle no faster than 30 seconds in each direction, so we will use a 0.033-Hz clock so that the machine cycles between states at no faster than once every 30 seconds. There are two output signals:

- *NSlite:* When this signal is asserted, the light on the north-south road is green; when this signal is deasserted, the light on the north-south road is red.

- *EWlite:* When this signal is asserted, the light on the east-west road is green; when this signal is deasserted, the light on the east-west road is red.

In addition, there are two inputs:

- *NScar:* Indicates that a car is over the detector placed in the roadbed in front of the light on the north-south road (going north or south).
- *EWcar:* Indicates that a car is over the detector placed in the roadbed in front of the light on the east-west road (going east or west).

The traffic light should change from one direction to the other only if a car is waiting to go in the other direction; otherwise, the light should continue to show green in the same direction as the last car that crossed the intersection.

To implement this simple traffic light we need two states:

- *NSgreen:* The traffic light is green in the north-south direction.
- *EWgreen:* The traffic light is green in the east-west direction.

We also need to create the next-state function, which can be specified with a table:

	Inputs		
	NScar	EWcar	Next state
NSgreen	0	0	NSgreen
NSgreen	0	1	EWgreen
NSgreen	1	0	NSgreen
NSgreen	1	1	EWgreen
EWgreen	0	0	EWgreen
EWgreen	0	1	EWgreen
EWgreen	1	0	NSgreen
EWgreen	1	1	NSgreen

Notice that we didn't specify in the algorithm what happens when a car approaches from both directions. In this case, the next-state function given above changes the state to ensure that a steady stream of cars from one direction cannot lock out a car in the other direction.

The finite-state machine is completed by specifying the output function.

Before we examine how to implement this finite-state machine, let's look at a graphical representation, which is often used for finite-state machines. In this representation, nodes are used to indicate states. Inside the node we place a list of the outputs that are active for that state. Directed arcs are used to show the next-state function, with labels on the arcs specifying the input condition as logic functions. Figure A.10.2 shows the graphical representation for this finite-state machine.

	Outputs	
	NSlite	EWlite
NSgreen	1	0
EWgreen	0	1

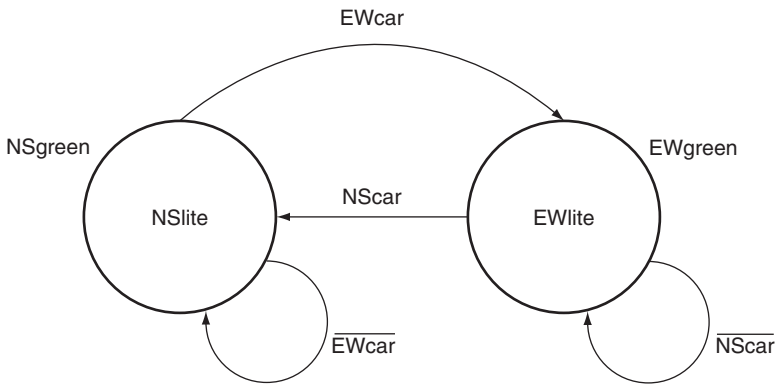


FIGURE A.10.2 The graphical representation of the two-state traffic light controller. We simplified the logic functions on the state transitions. For example, the transition from NSgreen to EWgreen in the next-state table is $(\overline{\text{NScar}} \cdot \text{EWcar}) + (\text{NScar} \cdot \text{EWcar})$, which is equivalent to EWcar.

A finite-state machine can be implemented with a register to hold the current state and a block of combinational logic that computes the next-state function and the output function. Figure A.10.3 shows how a finite-state machine with 4 bits of state, and thus up to 16 states, might look. To implement the finite-state machine in this way, we must first assign state numbers to the states. This process is called *state assignment*. For example, we could assign NSgreen to state 0 and EWgreen to state 1. The state register would contain a single bit. The next-state function would be given as

$$\text{NextState} = (\overline{\text{CurrentState}} \cdot \text{EWcar}) + (\text{CurrentState} \cdot \overline{\text{NScar}})$$

where CurrentState is the contents of the state register (0 or 1) and NextState is the output of the next-state function that will be written into the state register at the end of the clock cycle. The output function is also simple:

$$\text{NSlite} = \overline{\text{CurrentState}}$$
$$\text{EWlite} = \text{CurrentState}$$

The combinational logic block is often implemented using structured logic, such as a PLA. A PLA can be constructed automatically from the next-state and output function tables. In fact, there are *computer-aided design* (CAD) programs

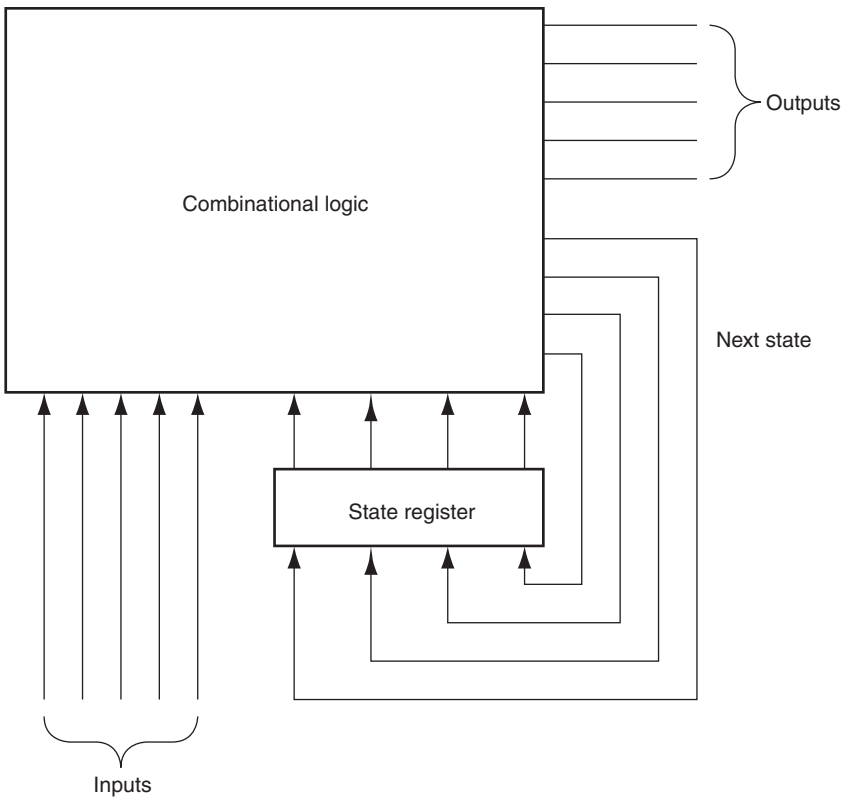


FIGURE A.10.3 A finite-state machine is implemented with a state register that holds the current state and a combinational logic block to compute the next state and output functions. The latter two functions are often split apart and implemented with two separate blocks of logic, which may require fewer gates.

that take either a graphical or textual representation of a finite-state machine and produce an optimized implementation automatically. In Chapters 4 and 5, finite-state machines were used to control processor execution. **Appendix C** discusses the detailed implementation of these controllers with both PLAs and ROMs.

To show how we might write the control in Verilog, Figure A.10.4 shows a Verilog version designed for synthesis. Note that for this simple control function, a Mealy machine is not useful, but this style of specification is used in Chapter 5 to implement a control function that is a Mealy machine and has fewer states than the Moore machine controller.

```
module TrafficLite (EWCar,NSCar,EWLite,NSLite,clock);

   input EWCar, NSCar,clock;
output EWLite,NSLite;

reg state;

initial state=0;  //set initial state

//following two assignments set the output, which is based
only on the state variable
assign NSLite = ~ state; //NSLite on if state = 0;
assign EWLite = state; //EWLite on if state = 1

always @(posedge clock) // all state updates on a positive
clock edge
   case (state)
      0: state = EWCar; //change state only if EWCar

      1: state = ~ NSCar; // change state only if NSCar

   endcase
endmodule
```

FIGURE A.10.4 A Verilog version of the traffic light controller.

Check Yourself

What is the smallest number of states in a Moore machine for which a Mealy machine could have fewer states?

a. Two, since there could be a one-state Mealy machine that might do the same thing.

b. Three, since there could be a simple Moore machine that went to one of two different states and always returned to the original state after that. For such a simple machine, a two-state Mealy machine is possible.

c. You need at least four states to exploit the advantages of a Mealy machine over a Moore machine.

A.11 Timing Methodologies

Throughout this appendix and in the rest of the text, we use an edge-triggered timing methodology. This timing methodology has an advantage in that it is simpler to explain and understand than a level-triggered methodology. In this section, we explain this timing methodology in a little more detail and also introduce level-sensitive clocking. We conclude this section by briefly discussing

the issue of asynchronous signals and synchronizers, an important problem for digital designers.

The purpose of this section is to introduce the major concepts in clocking methodology. The section makes some important simplifying assumptions; if you are interested in understanding timing methodology in more detail, consult one of the references listed at the end of this appendix.

We use an edge-triggered timing methodology because it is simpler to explain and has fewer rules required for correctness. In particular, if we assume that all clocks arrive at the same time, we are guaranteed that a system with edge-triggered registers between blocks of combinational logic can operate correctly without races if we simply make the clock long enough. A *race* occurs when the contents of a state element depend on the relative speed of different logic elements. In an edge-triggered design, the clock cycle must be long enough to accommodate the path from one flip-flop through the combinational logic to another flip-flop where it must satisfy the setup-time requirement. Figure A.11.1 shows this requirement for a system using rising edge-triggered flip-flops. In such a system the clock period (or cycle time) must be at least as large as

$$t_{\text{prop}} + t_{\text{combinational}} + t_{\text{setup}}$$

for the worst-case values of these three delays, which are defined as follows:

- t_{prop} is the time for a signal to propagate through a flip-flop; it is also sometimes called clock-to-*Q*.
- $t_{\text{combinational}}$ is the longest delay for any combinational logic (which by definition is surrounded by two flip-flops).
- t_{setup} is the time before the rising clock edge that the input to a flip-flop must be valid.

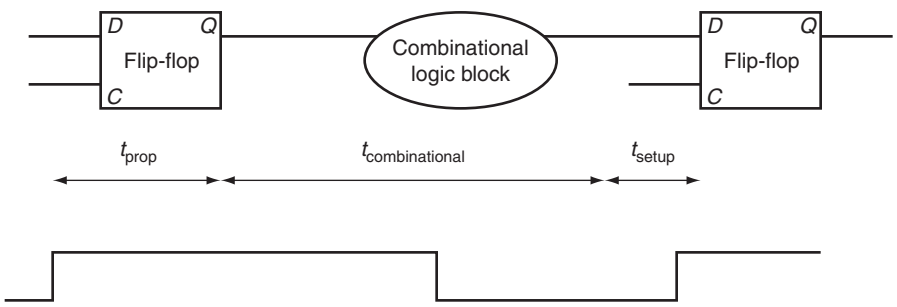


FIGURE A.11.1 In an edge-triggered design, the clock must be long enough to allow signals to be valid for the required setup time before the next clock edge. The time for a flip-flop input to propagate to the flip-flip outputs is t_{prop}; the signal then takes $t_{\text{combinational}}$ to travel through the combinational logic and must be valid t_{setup} before the next clock edge.

We make one simplifying assumption: the hold-time requirements are satisfied, which is almost never an issue with modern logic.

One additional complication that must be considered in edge-triggered designs is **clock skew**. Clock skew is the difference in absolute time between when two state elements see a clock edge. Clock skew arises because the clock signal will often use two different paths, with slightly different delays, to reach two different state elements. If the clock skew is large enough, it may be possible for a state element to change and cause the input to another flip-flop to change before the clock edge is seen by the second flip-flop.

clock skew The difference in absolute time between the times when two state elements see a clock edge.

Figure A.11.2 illustrates this problem, ignoring setup time and flip-flop propagation delay. To avoid incorrect operation, the clock period is increased to allow for the maximum clock skew. Thus, the clock period must be longer than

$$t_{\text{prop}} + t_{\text{combinational}} + t_{\text{setup}} + t_{\text{skew}}$$

With this constraint on the clock period, the two clocks can also arrive in the opposite order, with the second clock arriving t_{skew} earlier, and the circuit will work

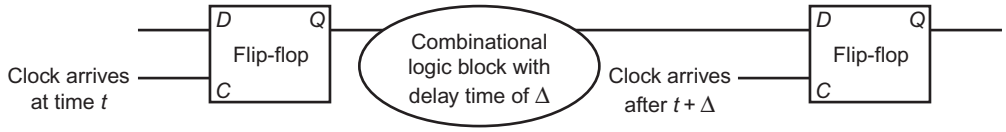


FIGURE A.11.2 Illustration of how clock skew can cause a race, leading to incorrect operation. Because of the difference in when the two flip-flops see the clock, the signal that is stored into the first flip-flop can race forward and change the input to the second flip-flop before the clock arrives at the second flip-flop.

correctly. Designers reduce clock-skew problems by carefully routing the clock signal to minimize the difference in arrival times. In addition, smart designers also provide some margin by making the clock a little longer than the minimum; this allows for variation in components as well as in the power supply. Since clock skew can also affect the hold-time requirements, minimizing the size of the clock skew is important.

Edge-triggered designs have two drawbacks: they require extra logic and they may sometimes be slower. Just looking at the D flip-flop versus the level-sensitive latch that we used to construct the flip-flop shows that edge-triggered design requires more logic. An alternative is to use **level-sensitive clocking**. Because state changes in a level-sensitive methodology are not instantaneous, a level-sensitive scheme is slightly more complex and requires additional care to make it operate correctly.

level-sensitive clocking A timing methodology in which state changes occur at either high or low clock levels but are not instantaneous as such changes are in edge-triggered designs.

Level-Sensitive Timing

In level-sensitive timing, the state changes occur at either high or low levels, but they are not instantaneous as they are in an edge-triggered methodology. Because of the noninstantaneous change in state, races can easily occur. To ensure that a level-sensitive design will also work correctly if the clock is slow enough, designers use *two-phase clocking*. Two-phase clocking is a scheme that makes use of two nonoverlapping clock signals. Since the two clocks, typically called ϕ_1 and ϕ_2, are nonoverlapping, at most one of the clock signals is high at any given time, as Figure A.11.3 shows. We can use these two clocks to build a system that contains level-sensitive latches but is free from any race conditions, just as the edge-triggered designs were.

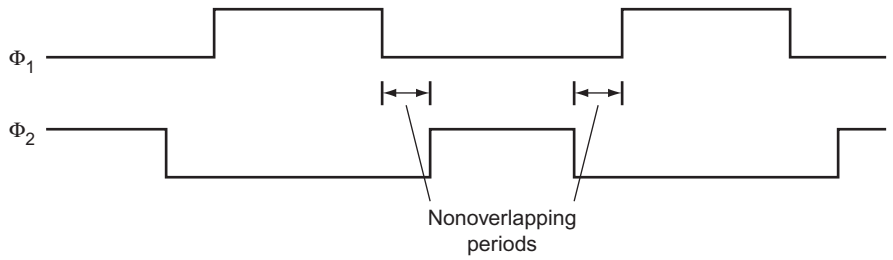


FIGURE A.11.3 A two-phase clocking scheme showing the cycle of each clock and the nonoverlapping periods.

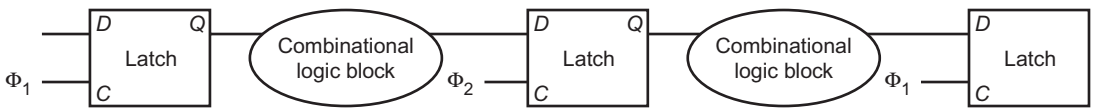


FIGURE A.11.4 A two-phase timing scheme with alternating latches showing how the system operates on both clock phases. The output of a latch is stable on the opposite phase from its C input. Thus, the first block of combinational inputs has a stable input during ϕ_2, and its output is latched by ϕ_2. The second (rightmost) combinational block operates in just the opposite fashion, with stable inputs during ϕ_1. Thus, the delays through the combinational blocks determine the minimum time that the respective clocks must be asserted. The size of the nonoverlapping period is determined by the maximum clock skew and the minimum delay of any logic block.

One simple way to design such a system is to alternate the use of latches that are open on ϕ_1 with latches that are open on ϕ_2. Because both clocks are not asserted at the same time, a race cannot occur. If the input to a combinational block is a ϕ_1 clock, then its output is latched by a ϕ_2 clock, which is open only during ϕ_2 when the input latch is closed and hence has a valid output. Figure A.11.4 shows how a system with two-phase timing and alternating latches operates. As in an edge-triggered design, we must pay attention to clock skew, particularly between the two

clock phases. By increasing the amount of nonoverlap between the two phases, we can reduce the potential margin of error. Thus, the system is guaranteed to operate correctly if each phase is long enough and if there is large enough nonoverlap between the phases.

Asynchronous Inputs and Synchronizers

By using a single clock or a two-phase clock, we can eliminate race conditions if clock-skew problems are avoided. Unfortunately, it is impractical to make an entire system function with a single clock and still keep the clock skew small. While the CPU may use a single clock, I/O devices will probably have their own clock. An asynchronous device may communicate with the CPU through a series of handshaking steps. To translate the asynchronous input to a synchronous signal that can be used to change the state of a system, we need to use a *synchronizer*, whose inputs are the asynchronous signal and a clock and whose output is a signal synchronous with the input clock.

Our first attempt to build a synchronizer uses an edge-triggered D flip-flop, whose *D* input is the asynchronous signal, as Figure A.11.5 shows. Because we communicate with a handshaking protocol, it does not matter whether we detect the asserted state of the asynchronous signal on one clock or the next, since the signal will be held asserted until it is acknowledged. Thus, you might think that this simple structure is enough to sample the signal accurately, which would be the case except for one small problem.

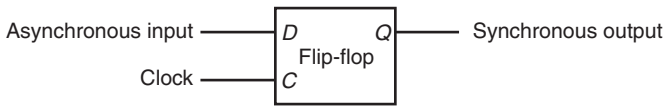


FIGURE A.11.5 A synchronizer built from a D flip-flop is used to sample an asynchronous signal to produce an output that is synchronous with the clock. This "synchronizer" will *not* work properly!

The problem is a situation called **metastability**. Suppose the asynchronous signal is transitioning between high and low when the clock edge arrives. Clearly, it is not possible to know whether the signal will be latched as high or low. That problem we could live with. Unfortunately, the situation is worse: when the signal that is sampled is not stable for the required setup and hold times, the flip-flop may go into a *metastable* state. In such a state, the output will not have a legitimate high or low value, but will be in the indeterminate region between them. Furthermore,

metastability
A situation that occurs if a signal is sampled when it is not stable for the required setup and hold times, possibly causing the sampled value to fall in the indeterminate region between a high and low value.

synchronizer failure A situation in which a flip-flop enters a metastable state and where some logic blocks reading the output of the flip-flop see a 0 while others see a 1.

the flip-flop is not guaranteed to exit this state in any bounded amount of time. Some logic blocks that look at the output of the flip-flop may see its output as 0, while others may see it as 1. This situation is called a **synchronizer failure**.

In a purely synchronous system, synchronizer failure can be avoided by ensuring that the setup and hold times for a flip-flop or latch are always met, but this is impossible when the input is asynchronous. Instead, the only solution possible is to wait long enough before looking at the output of the flip-flop to ensure that its output is stable, and that it has exited the metastable state, if it ever entered it. How long is long enough? Well, the probability that the flip-flop will stay in the metastable state decreases exponentially, so after a very short time the probability that the flip-flop is in the metastable state is very low; however, the probability never reaches 0! So designers wait long enough such that the probability of a synchronizer failure is very low, and the time between such failures will be years or even thousands of years.

For most flip-flop designs, waiting for a period that is several times longer than the setup time makes the probability of synchronization failure very low. If the clock rate is longer than the potential metastability period (which is likely), then a safe synchronizer can be built with two D flip-flops, as Figure A.11.6 shows. If you are interested in reading more about these problems, look into the references.

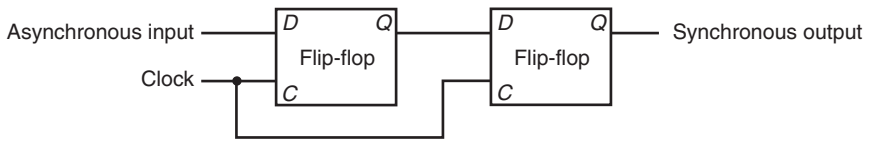


FIGURE A.11.6 This synchronizer will work correctly if the period of metastability that we wish to guard against is less than the clock period. Although the output of the first flip-flop may be metastable, it will not be seen by any other logic element until the second clock, when the second D flip-flop samples the signal, which by that time should no longer be in a metastable state.

Check Yourself

Suppose we have a design with very large clock skew—longer than the register **propagation time**. Is it always possible for such a design to slow the clock down enough to guarantee that the logic operates properly?

propagation time The time required for an input to a flip-flop to propagate to the outputs of the flip-flop.

a. Yes, if the clock is slow enough the signals can always propagate and the design will work, even if the skew is very large.

b. No, since it is possible that two registers see the same clock edge far enough apart that a register is triggered, and its outputs propagated and seen by a second register with the same clock edge.

A.12 Field Programmable Devices

Within a custom or semicustom chip, designers can make use of the flexibility of the underlying structure to easily implement combinational or sequential logic. How can a designer who does not want to use a custom or semicustom IC implement a complex piece of logic taking advantage of the very high levels of integration available? The most popular component used for sequential and combinational logic design outside of a custom or semicustom IC is a **field programmable device (FPD)**. An FPD is an integrated circuit containing combinational logic, and possibly memory devices, that are configurable by the end user.

FPDs generally fall into two camps: **programmable logic devices (PLDs)**, which are purely combinational, and **field programmable gate arrays (FPGAs)**, which provide both combinational logic and flip-flops. PLDs consist of two forms: **simple PLDs (SPLDs)**, which are usually either a PLA or a **programmable array logic (PAL)**, and complex PLDs, which allow more than one logic block as well as configurable interconnections among blocks. When we speak of a PLA in a PLD, we mean a PLA with user programmable and-plane and or-plane. A PAL is like a PLA, except that the or-plane is fixed.

Before we discuss FPGAs, it is useful to talk about how FPDs are configured. Configuration is essentially a question of where to make or break connections. Gate and register structures are static, but the connections can be configured. Notice that by configuring the connections, a user determines what logic functions are implemented. Consider a configurable PLA: by determining where the connections are in the and-plane and the or-plane, the user dictates what logical functions are computed in the PLA. Connections in FPDs are either permanent or reconfigurable. Permanent connections involve the creation or destruction of a connection between two wires. Current FPLDs all use an **antifuse** technology, which allows a connection to be built at programming time that is then permanent. The other way to configure CMOS FPLDs is through a SRAM. The SRAM is downloaded at power-on, and the contents control the setting of switches, which in turn determines which metal lines are connected. The use of SRAM control has the advantage in that the FPD can be reconfigured by changing the contents of the SRAM. The disadvantages of the SRAM-based control are two-fold: the configuration is volatile and must be reloaded on power-on, and the use of active transistors for switches slightly increases the resistance of such connections.

FPGAs include both logic and memory devices, usually structured in a two-dimensional array with the corridors dividing the rows and columns used for

field programmable devices (FPD) An integrated circuit containing combinational logic, and possibly memory devices, that are configurable by the end user.

programmable logic device (PLD) An integrated circuit containing combinational logic whose function is configured by the end user.

field programmable gate array (FPGA) A configurable integrated circuit containing both combinational logic blocks and flip-flops.

simple programmable logic device (SPLD) Programmable logic device, usually containing either a single PAL or PLA.

programmable array logic (PAL) Contains a programmable and-plane followed by a fixed or-plane.

antifuse A structure in an integrated circuit that when programmed makes a permanent connection between two wires.

lookup tables (LUTs) In a field programmable device, the name given to the cells because they consist of a small amount of logic and RAM.

global interconnect between the cells of the array. Each cell is a combination of gates and flip-flops that can be programmed to perform some specific function. Because they are basically small, programmable RAMs, they are also called **lookup tables (LUTs)**. Newer FPGAs contain more sophisticated building blocks such as pieces of adders and RAM blocks that can be used to build register files. Some FPGAs even contain 64-bit RISC-V cores!

In addition to programming each cell to perform a specific function, the interconnections between cells are also programmable, allowing modern FPGAs with hundreds of blocks and hundreds of thousands of gates to be used for complex logic functions. Interconnect is a major challenge in custom chips, and this is even more true for FPGAs, because cells do not represent natural units of decomposition for structured design. In many FPGAs, 90% of the area is reserved for interconnect and only 10% is for logic and memory blocks.

Just as you cannot design a custom or semicustom chip without CAD tools, you also need them for FPDs. Logic synthesis tools have been developed that target FPGAs, allowing the generation of a system using FPGAs from structural and behavioral Verilog.

A.13 Concluding Remarks

This appendix introduces the basics of logic design. If you have digested the material in this appendix, you are ready to tackle the material in Chapters 4 and 5, both of which use the concepts discussed in this appendix extensively.

Further Reading

There are a number of good texts on logic design. Here are some you might like to look into.

Ciletti, M. D. [2002]. *Advanced Digital Design with the Verilog HDL*, Englewood Cliffs, NJ: Prentice Hall.
A thorough book on logic design using Verilog.

Katz, R. H. [2004]. *Modern Logic Design*, 2nd ed., Reading, MA: Addison-Wesley.
A general text on logic design.

Wakerly, J. F. [2000]. *Digital Design: Principles and Practices*, 3rd ed., Englewood Cliffs, NJ: Prentice Hall.
A general text on logic design.

A.14 Exercises

A.1 [10] <§A.2> In addition to the basic laws we discussed in this section, there are two important theorems, called DeMorgan's theorems:

$$\overline{A + B} = \overline{A} \cdot \overline{B} \quad \text{and} \quad \overline{A \cdot B} = \overline{A} + \overline{B}$$

Prove DeMorgan's theorems with a truth table of the form

A	B	$\overline{A}$	$\overline{B}$	$\overline{A + B}$	$\overline{A} \cdot \overline{B}$	$\overline{A \cdot B}$	$\overline{A} + \overline{B}$
0	0	1	1	1	1	1	1
0	1	1	0	0	0	1	1
1	0	0	1	0	0	1	1
1	1	0	0	0	0	0	0

A.2 [15] <§A.2> Prove that the two equations for E in the example starting on page A-7 are equivalent by using DeMorgan's theorems and the axioms shown on page A-7.

A.3 [10] <§A.2> Show that there are $2n$ entries in a truth table for a function with n inputs.

A.4 [10] <§A.2> One logic function that is used for a variety of purposes (including within adders and to compute parity) is *exclusive OR*. The output of a two-input exclusive OR function is true only if exactly one of the inputs is true. Show the truth table for a two-input exclusive OR function and implement this function using AND gates, OR gates, and inverters.

A.5 [15] <§A.2> Prove that the NOR gate is universal by showing how to build the AND, OR, and NOT functions using a two-input NOR gate.

A.6 [15] <§A.2> Prove that the NAND gate is universal by showing how to build the AND, OR, and NOT functions using a two-input NAND gate.

A.7 [10] <§§A.2, A.3> Construct the truth table for a four-input odd-parity function (see page A-65 for a description of parity).

A.8 [10] <§§A.2, A.3> Implement the four-input odd-parity function with AND and OR gates using bubbled inputs and outputs.

A.9 [10] <§§A.2, A.3> Implement the four-input odd-parity function with a PLA.

A.10 [15] <§§A.2, A.3> Prove that a two-input multiplexor is also universal by showing how to build the NAND (or NOR) gate using a multiplexor.

A.11 [5] <§§4.2, A.2, A.3> Assume that X consists of 3 bits, x2 x1 x0. Write four logic functions that are true if and only if

- X contains only one 0
- X contains an even number of 0s
- X when interpreted as an unsigned binary number is less than 4
- X when interpreted as a signed (two's complement) number is negative

A.12 [5] <§§4.2, A.2, A.3> Implement the four functions described in Exercise A.11 using a PLA.

A.13 [5] <§§4.2, A.2, A.3> Assume that X consists of 3 bits, x2 x1 x0, and Y consists of 3 bits, y2 y1 y0. Write logic functions that are true if and only if

- X <Y, where X and Y are thought of as unsigned binary numbers
- X <Y, where X and Y are thought of as signed (two's complement) numbers
- X = Y

Use a hierarchical approach that can be extended to larger numbers of bits. Show how can you extend it to 6-bit comparison.

A.14 [5] <§§A.2, A.3> Implement a switching network that has two data inputs (*A* and *B*), two data outputs (*C* and *D*), and a control input (*S*). If *S* equals 1, the network is in pass-through mode, and *C* should equal *A*, and *D* should equal *B*. If *S* equals 0, the network is in crossing mode, and *C* should equal *B*, and *D* should equal *A*.

A.15 [15] <§§A.2, A.3> Derive the product-of-sums representation for *E* shown on page A-11 starting with the sum-of-products representation. You will need to use DeMorgan's theorems.

A.16 [30] <§§A.2, A.3> Give an algorithm for constructing the sum-of-products representation for an arbitrary logic equation consisting of AND, OR, and NOT. The algorithm should be recursive and should not construct the truth table in the process.

A.17 [5] <§§A.2, A.3> Show a truth table for a multiplexor (inputs *A*, *B*, and *S*; output *C*), using don't cares to simplify the table where possible.

A.18 [5] <§A.3> What is the function implemented by the following Verilog modules:

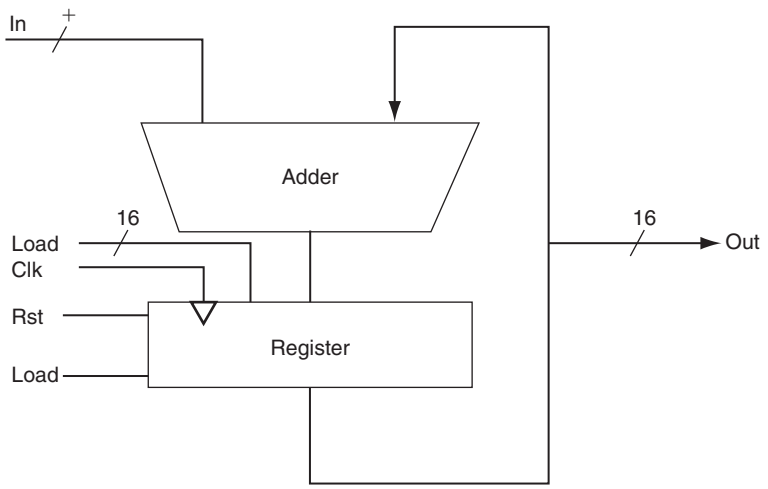
```
module FUNC1 (I0, I1, S, out);
      input I0, I1;
      input S;
      output out;
      out = S? I1: I0;
endmodule

module FUNC2 (out,ctl,clk,reset);
      output [7:0] out;
      input ctl, clk, reset;
      reg [7:0] out;
      always @(posedge clk)
      if (reset) begin
                  out <= 8'b0 ;
      end
      else if (ctl) begin
                  out <= out + 1;
      end
      else begin
                  out <= out - 1;
      end
endmodule
```

A.19 [5] <§A.4> The Verilog code on page A-53 is for a D flip-flop. Show the Verilog code for a D latch.

A.20 [10] <§§A.3, A.4> Write down a Verilog module implementation of a 2-to-4 decoder (and/or encoder).

A.21 [10] <§§A.3, A.4> Given the following logic diagram for an accumulator, write down the Verilog module implementation of it. Assume a positive edge-triggered register and asynchronous Rst.



A.22 [20] <§§B3, A.4, A.5> Section 3.3 presents basic operation and possible implementations of multipliers. A basic unit of such implementations is a shift-and-add unit. Show a Verilog implementation for this unit. Show how can you use this unit to build a 32-bit multiplier.

A.23 [20] <§§B3, A.4, A.5> Repeat Exercise A.22, but for an unsigned divider rather than a multiplier.

A.24 [15] <§A.5> The ALU supported set on less than (slt) using just the sign bit of the adder. Let's try a set on less than operation using the values -7_{ten} and 6_{ten}. To make it simpler to follow the example, let's limit the binary representations to 4 bits: 1001_{two} and 0110_{two}.

$$1001_{two} - 0110_{two} = 1001_{two} + 1010_{two} = 0011_{two}$$

This result would suggest that −7> 6, which is clearly wrong. Hence, we must factor in overflow in the decision. Modify the 1-bit ALU in Figure A.5.10 on page A-33 to handle `slt` correctly. Make your changes on a photocopy of this figure to save time.

A.25 [20] <§A.6> A simple check for overflow during addition is to see if the CarryIn to the most significant bit is not the same as the CarryOut of the most significant bit. Prove that this check is the same as in Figure 3.2.

A.26 [5] <§A.6> Rewrite the equations on page A-44 for a carry-lookahead logic for a 16-bit adder using a new notation. First, use the names for the CarryIn signals of the individual bits of the adder. That is, use c4, c8, c12, … instead of C1, C2, C3, …. In addition, let P*i,j*; mean a propagate signal for bits *i* to *j*, and G*i,j*; mean a generate signal for bits *i* to *j*. For example, the equation

$$C2 = G1 + (P1 \cdot G0) + (P1 \cdot P0 \cdot c0)$$

can be rewritten as

$$c8 = G_{7,4} + (P_{7,4} \cdot G_{3,0}) + (P_{7,4} \cdot P_{3,0} \cdot c0)$$

This more general notation is useful in creating wider adders.

A.27 [15] <§A.6> Write the equations for the carry-lookahead logic for a 64-bit adder using the new notation from Exercise A.26 and using 16-bit adders as building blocks. Include a drawing similar to Figure A.6.3 in your solution.

A.28 [10] <§A.6> Now calculate the relative performance of adders. Assume that hardware corresponding to any equation containing only OR or AND terms, such as the equations for p*i* and g*i* on page A-40, takes one time unit T. Equations that consist of the OR of several AND terms, such as the equations for c1, c2, c3, and c4 on page A-40, would thus take two time units, 2T. The reason is it would take T to produce the AND terms and then an additional T to produce the result of the OR. Calculate the numbers and performance ratio for 4-bit adders for both ripple carry and carry lookahead. If the terms in equations are further defined by other equations, then add the appropriate delays for those intermediate equations, and continue recursively until the actual input bits of the adder are used in an equation. Include a drawing of each adder labeled with the calculated delays and the path of the worst-case delay highlighted.

A.29 [15] <§A.6> This exercise is similar to Exercise A.28, but this time calculate the relative speeds of a 16-bit adder using ripple carry only, ripple carry of 4-bit groups that use carry lookahead, and the carry-lookahead scheme on page A-39.

A.30 [15] <§A.6> This exercise is similar to Exercises A.28 and A.29, but this time calculate the relative speeds of a 64-bit adder using ripple carry only, ripple carry of 4-bit groups that use carry lookahead, ripple carry of 16-bit groups that use carry lookahead, and the carry-lookahead scheme from Exercise A.27.

A.31 [10] <§A.6> Instead of thinking of an adder as a device that adds two numbers and then links the carries together, we can think of the adder as a hardware device that can add three inputs together (a*i*, b*i*, c*i*) and produce two outputs (s, c_{i+1}). When adding two numbers together, there is little we can do with this observation. When we are adding more than two operands, it is possible to reduce the cost of the carry. The idea is to form two independent sums, called S′ (sum bits) and C′ (carry bits). At the end of the process, we need to add C′ and S′ together using a normal adder. This technique of delaying carry propagation until the end of a sum of numbers is called *carry save addition*. The block drawing on the lower right of Figure A.14.1 (see below) shows the organization, with two levels of carry save adders connected by a single normal adder.

Calculate the delays to add four 16-bit numbers using full carry-lookahead adders versus carry save with a carry-lookahead adder forming the final sum. (The time unit T in Exercise A.28 is the same.)

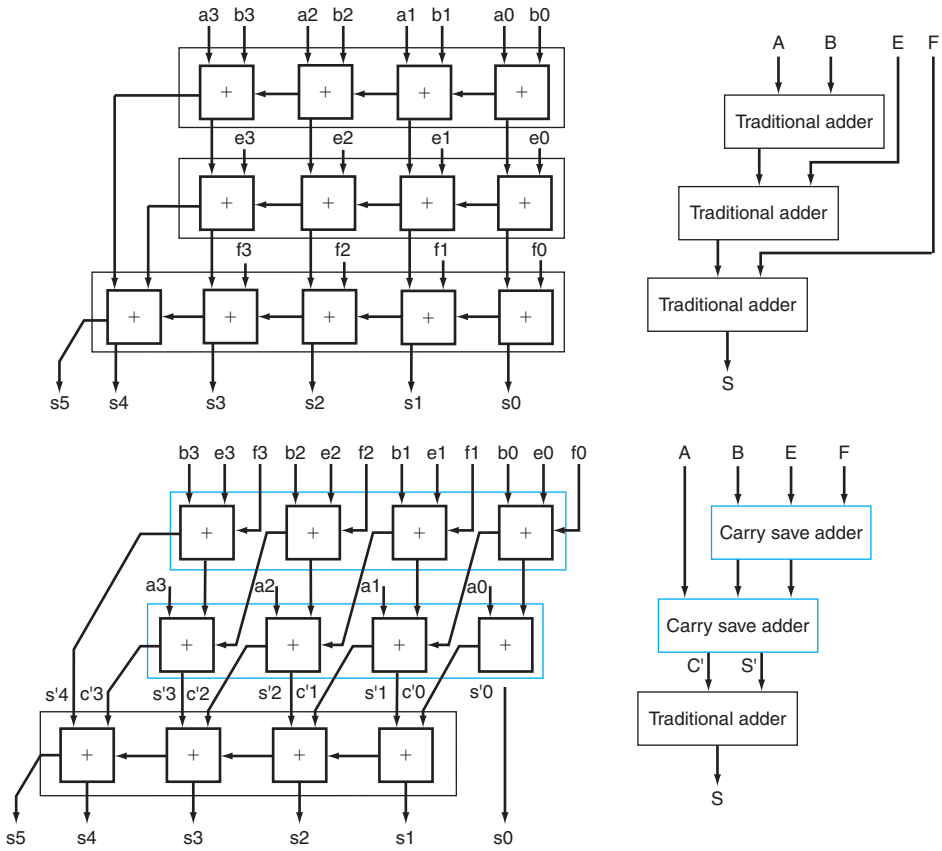


FIGURE A.14.1 Traditional ripple carry and carry save addition of four 4-bit numbers. The details are shown on the left, with the individual signals in lowercase, and the corresponding higher-level blocks are on the right, with collective signals in upper case. Note that the sum of four n-bit numbers can take $n + 2$ bits.

A.32 [20] <§A.6> Perhaps the most likely case of adding many numbers at once in a computer would be when trying to multiply more quickly by using many adders to add many numbers in a single clock cycle. Compared to the multiply algorithm in Chapter 3, a carry save scheme with many adders could multiply more than 10 times faster. This exercise estimates the cost and speed of a combinational multiplier to multiply two positive 16-bit numbers. Assume that you have 16 intermediate terms M15, M14, …, M0, called *partial products*, that contain the multiplicand ANDed with multiplier bits m15, m14, …, m0. The idea is to use carry save adders to reduce the n operands into $2n/3$ in parallel groups of three, and do this repeatedly until you get two large numbers to add together with a traditional adder.

First, show the block organization of the 16-bit carry save adders to add these 16 terms, as shown on the right in Figure A.14.1. Then calculate the delays to add these 16 numbers. Compare this time to the iterative multiplication scheme in Chapter 3 but only assume 16 iterations using a 16-bit adder that has full carry lookahead whose speed was calculated in Exercise A.29.

A.33 [10] <§A.6> There are times when we want to add a collection of numbers together. Suppose you wanted to add four 4-bit numbers (A, B, E, F) using 1-bit full adders. Let's ignore carry lookahead for now. You would likely connect the 1-bit adders in the organization at the top of Figure A.14.1. Below the traditional organization is a novel organization of full adders. Try adding four numbers using both organizations to convince yourself that you get the same answer.

A.34 [5] <§A.6> First, show the block organization of the 16-bit carry save adders to add these 16 terms, as shown in Figure A.14.1. Assume that the time delay through each 1-bit adder is 2T. Calculate the time of adding four 4-bit numbers to the organization at the top versus the organization at the bottom of Figure A.14.1.

A.35 [5] <§A.8> Quite often, you would expect that given a timing diagram containing a description of changes that take place on a data input *D* and a clock input *C* (as in Figures A.8.3 and A.8.6 on pages A-52 and A-54, respectively), there would be differences between the output waveforms (*Q*) for a D latch and a D flip-flop. In a sentence or two, describe the circumstances (e.g., the nature of the inputs) for which there would not be any difference between the two output waveforms.

A.36 [5] <§A.8> Figure A.8.8 on page A-55 illustrates the implementation of the register file for the RISC-V datapath. Pretend that a new register file is to be built, but that there are only two registers and only one read port, and that each register has only 2 bits of data. Redraw Figure A.8.8 so that every wire in your diagram corresponds to only 1 bit of data (unlike the diagram in Figure A.8.8, in which some wires are 5 bits and some wires are 32 bits). Redraw the registers using D flip-flops. You do not need to show how to implement a D flip-flop or a multiplexor.

A.37 [10] <§A.10> A friend would like you to build an "electronic eye" for use as a fake security device. The device consists of three lights lined up in a row, controlled by the outputs Left, Middle, and Right, which, if asserted, indicate that a light should be on. Only one light is on at a time, and the light "moves" from left to right and then from right to left, thus scaring away thieves who believe that the device is monitoring their activity. Draw the graphical representation for the finite-state machine used to specify the electronic eye. Note that the rate of the eye's movement will be controlled by the clock speed (which should not be too great) and that there are essentially no inputs.

A.38 [10] <§A.10> Assign state numbers to the states of the finite-state machine you constructed for Exercise A.37 and write a set of logic equations for each of the outputs, including the next-state bits.

A.39 [15] <§§A.2, A.8, A.10> Construct a 3-bit counter using three D flip-flops and a selection of gates. The inputs should consist of a signal that resets the counter to 0, called *reset*, and a signal to increment the counter, called *inc*. The outputs should be the value of the counter. When the counter has value 7 and is incremented, it should wrap around and become 0.

A.40 [20] <§A.10> A *Gray code* is a sequence of binary numbers with the property that no more than 1 bit changes in going from one element of the sequence to another. For example, here is a 3-bit binary Gray code: 000, 001, 011, 010, 110, 111, 101, and 100. Using three D flip-flops and a PLA, construct a 3-bit Gray code counter that has two inputs: *reset*, which sets the counter to 000, and *inc*, which makes the counter go to the next value in the sequence. Note that the code is cyclic, so that the value after 100 in the sequence is 000.

A.41 [25] <§A.10> We wish to add a yellow light to our traffic light example on page A-68. We will do this by changing the clock to run at 0.25 Hz (a 4-second clock cycle time), which is the duration of a yellow light. To prevent the green and red lights from cycling too fast, we add a 30-second timer. The timer has a single input, called *TimerReset*, which restarts the timer, and a single output, called *TimerSignal*, which indicates that the 30-second period has expired. Also, we must redefine the traffic signals to include yellow. We do this by defining two output signals for each light: green and yellow. If the output NSgreen is asserted, the green light is on; if the output NSyellow is asserted, the yellow light is on. If both signals are off, the red light is on. Do *not* assert both the green and yellow signals at the same time, since American drivers will certainly be confused, even if European drivers understand what this means! Draw the graphical representation for the finite-state machine for this improved controller. Choose names for the states that are *different* from the names of the outputs.

A.42 [15] <§A.10> Write down the next-state and output-function tables for the traffic light controller described in Exercise A.41.

A.43 [15] <§§A.2, A.10> Assign state numbers to the states in the traffic light example of Exercise A.41 and use the tables of Exercise A.42 to write a set of logic equations for each of the outputs, including the next-state outputs.

A.44 [15] <§§A.3, A.10> Implement the logic equations of Exercise A.43 as a PLA.

Answers to Check Yourself

§A.2, page A-8: No. If $A = 1$, $C = 1$, $B = 0$, the first is true, but the second is false.
§A.3, page A-20: C.
§A.4, page A-22: They are all exactly the same.
§A.4, page A-26: `A` = 0, `B` = 1.
§A.5, page A-37: 2.
§A.6, page A-46: 1.
§A.8, page A-57: c.
§A.10, page A-71: b.
§A.11, page A-76: b.

Index

Note: Online information is listed by print page number and a period followed by "e" with online page number (54.e1). Page references preceded by a single letter with hyphen refer to appendices. Page references followed by "*f*," "*t*," and "*b*" refer to figures, tables, and boxes, respectively.

0-9, and symbols

1-bit ALU, A-26–A-29. *See also* Arithmetic logic unit (ALU)
  adder, A-27*f*
  CarryOut, A-28
  for most significant bit, A-33*f*
  illustrated, A-29*f*
  logical unit for AND/OR, A-27*f*
  performing AND, OR, and addition, A-31, A-33*f*
64-bit ALU, A-29–A-31. *See also* Arithmetic logic unit (ALU)
  from 63 copies of 1-bit ALU, A-34*f*
  with 64 1-bit ALUs, A-30*f*
  defining in Verilog, A-36–A-37
  illustrated, A-35*f*
  ripple carry adder, A-29
7090/7094 hardware, 227.e6

A

Absolute references, 127
Abstractions
  hardware/software interface, 22
  principle, 22
  to simplify design, 11
Accumulator architectures, 162.e1–162.e2
Acronyms, 9
Active matrix, 18
add (add), 64*f*
addi (add immediate), 64*f*, 72, 84
Addition, 174–177. *See also* Arithmetic
  binary, 174*b*–175*b*
  floating-point, 198–201, 206
  operands, 175
  significands, 197*b*–198*b*
  speed, 177*b*
Address interleaving, 372–373
Address select logic, C-24, C-25*f*
Address space, 420, 423*b*
  extending, 469*b*
  flat, 469
  ID (ASID), 438
  inadequate, 473.e5–473.e6
  shared, 509–510
  single physical, 509–510
  virtual, 438
Address translation
  for ARM cortex-A53, 460*f*
  defined, 420–421
  fast, 430–432
  for Intel core i7, 460*f*
  TLB for, 430–432
Address-control lines, C-26*f*
Addresses
  base, 69
  byte, 70
  defined, 68
  memory, 78*b*
  virtual, 420–421, 440, 441*b*
Addressing
  base, 118*f*
  in branches, 115–117
  displacement, 118
  immediate, 118*f*
  PC-relative, 115–116, 118*f*
  register, 118*f*
  RISC-V modes, 117–118
  x86 modes, 151
Addressing modes
  desktop architectures, D-5–D-6
Advanced Vector Extensions (AVX), 218–219
AGP, B-9–B-10
Algol-60, 162.e6
Aliasing, 436
Alignment restriction, 70
All-pairs N-body algorithm, B-65
Alpha architecture
  bit count instructions, D-29
  floating-point instructions, D-28–D-29
  instructions, D-27–D-29
  no divide, D-28
  PAL code, D-28
  unaligned load-store, D-28
  VAX floating-point formats, D-29
ALU control, 251–253. *See also* Arithmetic logic unit (ALU)
  bits, 252–253, 252*f*
  logic, C-6–C-7
  mapping to gates, C-4–C-7
  truth tables, C-5*f*
ALU control block, 255
  defined, C-4–C-6
  generating ALU control bits, C-6*f*
ALUOp, 252, C-6*b*–C-7*b*
  bits, 252–253
  control signal, 255
Amazon Web Services (AWS), 417*b*
AMD Opteron X4 (Barcelona), 535, 536*f*
AMD64, 148, 162.e5, 217
Amdahl's law, 393, 495–496
  corollary, 49
  defined, 49
  fallacy, 548
and (and), 64*f*
AND gates, A-12–A-13, C-7
AND operation, 90, A-6
andi (and immediate), 64*f*
Annual failure rate (AFR), 410–411
  versus MTTF of disks, 410*b*–411*b*
Antidependence, 327
Antifuse, A-77
Apple computer, 54.e6
Apple iPad 2 A1395, 20*f*
  logic board of, 20*f*
  processor integrated circuit of, 21*f*
Application binary interface (ABI), 22
Application programming interfaces (APIs)
  defined, B-4
  graphics, B-14
Architectural registers, 337–338
Arithmetic, 172
  addition, 174–177
  addition and subtraction, 174–177
  division, 183–191
  fallacies and pitfalls, 222–225

Arithmetic (*Continued*)
floating-point, 191–216
historical perspective, 227
multiplication, 177–183
parallelism and, 216–217
Streaming SIMD Extensions and advanced vector extensions in x86, 217–218
subtraction, 174–177
subword parallelism, 216–217
subword parallelism and matrix multiply, 218–222
Arithmetic instructions. *See also* Instructions
desktop RISC, D-11*f*
embedded RISC, D-13*f*
logical, 243–244
operands, 67–74
Arithmetic intensity, 533–534
Arithmetic logic unit (ALU). *See also* ALU control; Control units
1-bit, A-26–A-29
64-bit, A-29–A-31
before forwarding, 299*f*
branch datapath, 246–247
hardware, 176
memory-reference instruction use, 237
for register values, 244
R-format operations, 245*f*
signed-immediate input, 302
ARM Cortex-A53, 236, 334–342
address translation for, 460*f*
caches in, 461*f*
data cache miss rates for, 462*f*
memory hierarchies of, 459
performance of, 462–464
specification, 335*f*
TLB hardware for, 460*f*
ARPAnet, 54.e9
Arrays, 407*f*
logic elements, A-18–A-20
multiple dimension, 212
pointers *versus*, 141–144
procedures for setting to zero, 141*f*
ASCII
binary numbers *versus*, 109*b*
character representation, 108*f*
defined, 108–109
symbols, 111
Assemblers, 125–127
defined, 14
function, 125–127
microcode, C-30
number acceptance, 126
object file, 126
Assembly language, 15*f*
defined, 14, 125
floating-point, 207*f*
illustrated, 15*f*
programs, 125
RISC-V, 64*f*, 85*b*–86*b*
translating into machine language, 85*b*–86*b*
Asserted signals, 242, A-4
Associativity
in caches, 397*b*–398*b*
degree, increasing, 396–398, 444
increasing, 401–402
set, tag size *versus*, 401*b*–402*b*
Atomic compare and swap, 123*b*
Atomic exchange, 122
Atomic fetch-and-increment, 123*b*
Atomic memory operation, B-21
Attribute interpolation, B-43–B-44
auipc's effect, 156
Automobiles, computer application in, 4
Average memory access time (AMAT), 394
calculating, 394*b*

B

Bandwidth, 29–30
bisection, 527
external to DRAM, 390
memory, 390
network, 525–526
Barrier synchronization, B-18
defined, B-20
for thread communication, B-34
Base addressing, 69, 118
Base registers, 69
Basic block, 95*b*
Benchmarks, 530–540
defined, 46
Linpack, 227.e2–227.e3, 530
multiprocessor, 530–540
NAS parallel, 532
parallel, 531*f*
PARSEC suite, 532
SPEC CPU, 46–48
SPEC power, 48–49
SPECrate, 530
Stream, 540*b*
Biased notation, 81, 195
Binary numbers, 82
ASCII *versus*, 109*b*
conversion to decimal numbers, 77*b*
defined, 74
Bisection bandwidth, 527
Bit maps
defined, 18
goal, 18
storing, 18
Bit-Interleaved Parity (RAID 3), 458.e4
Bits
ALUOp, 252–253
defined, 14
dirty, 430*b*
guard, 214
patterns, 214*b*–215*b*
reference, 428*b*
rounding, 214
sign, 75
state, C-8–C-10
sticky, 214
valid, 376–378
Blocking assignment, A-24
Blocking factor, 406
Block-Interleaved Parity (RAID 4), 458.e4–458.e5
Blocks
combinational, A-4–A-5
defined, 367–368
finding, 444–445
flexible placement, 394–398
least recently used (LRU), 401
locating in cache, 399–401
miss rate and, 383*f*
multiword, mapping addresses to, 382*b*–383*b*
placement locations, 443
placement strategies, 396
replacement selection, 401
replacement strategies, 446
spatial locality exploitation, 383
state, A-4–A-5
valid data, 376–378
Bonding, 28
Boolean algebra, A-6–A-7
Bounds check shortcut, 96
Branch datapath
ALU, 246–247
operations, 246–247
Branch if Equal (beq), A-32
Branch if greater than or equal, unsigned (bgeu), 95–96

Branch if less than (blt) instruction, 95–96
Branch if less than, unsigned (bltu), 95–96
Branch instructions
  pipeline impact, 308*f*
Branch not taken
  assumption, 307–308
  defined, 246
Branch prediction
  buffers, 310
  as control hazard solution, 274
  defined, 273–274
  dynamic, 274, 310–314
  static, 324
Branch predictors
  accuracy, 312
  correlation, 312–313
  information from, 312–313
  tournament, 313–314
Branch table, 97–98
Branch taken
  cost reduction, 308–309
  defined, 246
Branch target
  addresses, 246
  buffers, 312
Branches. *See also* Conditional branches
  addressing in, 115–117
  compiler creation, 93–94
  decision, moving up, 308–309
  delayed, 274, 308–310
  ending, 95*b*
  execution in ID stage, 309
  pipelined, 310*b*
  target address, 308–309
Branch-on-zero instruction, 260–261
Bubble Sort, 140
Bubbles, 305
Bus-based coherent multiprocessors, 553.e1
Buses, A-18–A-19
Bytes
  addressing, 70
  order, 70

C

C language
  assignment, compiling into RISC-V, 65*b*
  compiling, 144.e1–144.e2, 144–145
  compiling assignment with registers, 67*b*–68*b*
  compiling while loops in, 94*b*–95*b*
  sort algorithms, 141*f*
  translation hierarchy, 124*f*
  translation to RISC-V assembly language, 65
  variables, 104*b*
C.mmp, 553.e3–553.e4
C + + language, 144.e26, 162.e7
Cache blocking and matrix multiply, 465–468
Cache coherence, 454–458
  coherence, 454
  consistency, 454
  enforcement schemes, 456
  implementation techniques, 459.e10–459.e11
  migration, 456
  problem, 454, 455*f*, 458*b*
  protocol example, 459.e11–459.e15
  protocols, 456
  replication, 456
  snooping protocol, 456–458
  snoopy, 459.e16
  state diagram, 459.e15*f*
Cache coherency protocol, 459.e11–459.e15
  finite-state transition diagram, 459.e14*f*
  functioning, 459.e13*f*
  mechanism, 459.e13*f*
  state diagram, 459.e15*f*
  states, 459.e12
  write-back cache, 459.e14*f*
Cache controllers, 459
  coherent cache implementation techniques, 459.e10–459.e11
  implementing, 459.e1
  snoopy cache coherence, 459.e16
  SystemVerilog, 459.e1–459.e4
Cache hits, 460
Cache misses
  block replacement on, 445–446
  capacity, 447–448
  compulsory, 447
  conflict, 447
  defined, 384
  direct-mapped cache, 396
  fully associative cache, 398
  handling, 384–385
  memory-stall clock cycles, 391
  reducing with flexible block placement, 394–398
  set-associative cache, 397
  steps, 385
  in write-through cache, 385
Cache performance, 390–410
  calculating, 392*b*–393*b*
  hit time and, 393–394
  impact on processor performance, 392–393
Cache-aware instructions, 472
Caches, 375–390. *See also* Blocks
  accessing, 378–384
  in ARM cortex-A53, 461*f*
  associativity in, 397*b*–398*b*
  bits in, 382*b*
  bits needed for, 382
  contents illustration, 379*f*
  defined, 19–22, 375–376
  direct-mapped, 376, 377*f*, 382, 394
  empty, 378
  FSM for controlling, 449–454
  fully associative, 395
  GPU, B-38
  inconsistent, 385
  index, 380
  in Intel Core i7, 461*f*
  Intrinsity FastMATH example, 387–389
  locating blocks in, 399–401
  locations, 377*f*
  multilevel, 390, 402–405
  nonblocking, 460
  physically addressed, 436–437
  physically indexed, 436*b*–437*b*
  physically tagged, 436*b*–437*b*
  primary, 402, 409–410
  secondary, 402, 409–410
  set-associative, 395
  simulating, 468*b*
  size, 381–383
  split, 389*b*
  summary, 389–390
  tag field, 380
  tags, 459.e10–459.e11, 459.e1*f*
  virtual memory and TLB integration, 435–437
  virtually addressed, 436
  virtually indexed, 436
  virtually tagged, 436
  write-back, 386–387, 446
  write-through, 385, 387, 446
  writes, 385–387
Callee, 99, 101
Caller, 99

Capabilities, 473.e12
Capacity misses, 447
Carry lookahead, A-37–A-47
  4-bit ALUs using, A-43*f*
  adder, A-38
  fast, with first level of abstraction, A-38–A-40
  fast, with "infinite" hardware, A-38
  fast, with second level of abstraction, A-40–A-45
  plumbing analogy, A-41*f*, A-42*f*
  ripple carry speed *versus*, A-45*b*
  summary, A-45–A-47
Carry save adders, 183
CDC 6600, 54.e6, 347.e2
Cell phones, 6–7
Central processor unit (CPU). *See also* Processors
  classic performance equation, 36–40
  defined, 19
  execution time, 32–34
  performance, 33–35
  system, time, 32
  time, 391
  time measurements, 33–34
  user, time, 32
Cg pixel shader program, B-15
Characters
  ASCII representation, 108–109
  in Java, 111–113
Chips, 19, 25–26
  manufacturing process, 26
Classes
  defined, 144.e14
  packages, 144.e20
Clock cycles
  defined, 33
  memory-stall, 391
  number of registers and, 67
  worst-case delay and, 262
Clock cycles per instruction (CPI), 35–36, 272
  one level of caching, 402
  two levels of caching, 402
Clock rate
  defined, 33
  frequency switched as function of, 41
  power and, 40
Clocking methodology, 241–243, A-47
  edge-triggered, 241, A-47, A-72–A-73
  level-sensitive, A-73–A-74, A-74–A-75
  for predictability, 241
Clocks, A-47–A-49
  edge, A-47, A-49*b*
  in edge-triggered design, A-72*f*
  skew, A-73
  specification, A-56*f*
  synchronous system, A-47–A-48
Cloud computing, 524–525
  defined, 7
Cluster networking, 529.e3–529.e5, 529.e6–529.e9, 529–530, 529.e1
Clusters, 553.e7–553.e8
  defined, 492, 522, 553.e7
  isolation, 523
  organization, 491
  scientific computing on, 553.e7
Cm*, 553.e3–553.e4
CMOS (complementary metal oxide semiconductor), 41
Coarse-grained multithreading, 506–507
Cobol, 162.e6
Code generation, 144.e12
Code motion, 144.e6
Cold-start miss, 447
Collision misses, 447
Column major order, 405
Combinational blocks, A-4–A-5
Combinational control units, C-4–C-8
Combinational elements, 240
Combinational logic, 241, A-3–A-4, A-9–A-20
  arrays, A-18–A-19
  decoders, A-9–A-10
  defined, A-4–A-5
  don't cares, A-17–A-18
  multiplexors, A-10
  ROMs, A-14–A-16
  two-level, A-11–A-14
  Verilog, A-23–A-26
Commercial computer development, 54.e3–54.e9
Commit units
  buffer, 329
  defined, 329
  in update control, 334*b*
Common case fast, 11
Common subexpression elimination, 144.e5
Communication, 23–24
  overhead, reducing, 44–45
  thread, B-34
Compact code, 162.e3–162.e4
Compare and branch zero, 309
Comparisons
  constant operands in, 72–74
  signed *versus* unsigned, 95–96
Compilers, 125
  branch creation, 94*b*
  brief history, 162.e7–162.e8
  conservative, 144.e6
  defined, 14
  front end, 144.e2
  function, 14, 125
  high-level optimizations, 144.e3–144.e4
  ILP exploitation, 347.e4–347.e5
  Just In Time (JIT), 133
  optimization, 141, 162.e8
  speculation, 323–324
  structure, 144.e1*f*
Compiling
  C assignment statements, 65*b*
  C language, 94*b*–95*b*, 144–145, 144.e1, 144.e2
  floating-point programs, 208*b*–209*b*
  if-then-else, 93*b*
  in Java, 144.e18–144.e19
  procedures, 100*b*–101*b*, 102*b*–103*b*
  recursive procedures, 102*b*–103*b*
  while loops, 94*b*–95*b*
Compressed sparse row (CSR) matrix, B-55, B-56
Compulsory misses, 447–449
Computer architects, 11–13
  abstraction to simplify design, 11
  common case fast, 11
  dependability via redundancy, 12
  hierarchy of memories, 12
  Moore's law, 11
  parallelism, 12
  pipelining, 12
  prediction, 12
Computers
  application classes, traditional, 3
  applications, 4
  arithmetic for, 172
  characteristics, 54.e12*f*
  commercial development, 54.e3–54.e9
  component organization, 17*f*
  components, 17*f*
  design measure, 53
  desktop, 5
  embedded, 5–6
  first, 54.e2
  in information revolution, 4
  instruction representation, 81–89

performance measurement, 54.e1–54.e3
post-PC era, 6–7
servers, 5
Condition codes/flags, 96
Conditional branches
changing program counter with, 312*b*
compiling if-then-else into, 93*b*
defined, 92–93
desktop RISC, D-16*f*
embedded RISC, D-16*f*
implementation, 97*b*
in loops, 117
PA-RISC, D-34–D-36, D-35*f*
PC-relative addressing, 115–116
RISC, D-10–D-16
SPARC, D-10–D-12
Conditional move instructions, 313*b*–314*b*
Conflict misses, 447
Constant memory, B-40
Constant operands, 72–74
frequent occurrence, 72
Content Addressable Memory (CAM), 400*b*–401*b*
Context switch, 438*b*
Control
ALU, 251–253
challenge, 315
finalizing, 261
forwarding, 300
FSM, C-8–C-22
implementation, optimizing, C-27
mapping to hardware, C-3–C-4, C-4–C-8, C-8–C-22, C-22–C-28, C-28–C-32, C-32–C-33
memory, C-26*f*
organizing, to reduce logic, C-31–C-32
pipelined, 290–294
Control and status register (CSR) access instructions, 464–465
Control flow graphs, 144.e8, 144.e9
illustrated examples, 144.e8*f*, 144.e9*f*, 144.e11*f*
Control functions
ALU, mapping to gates, C-4–C-7
defining, 256
PLA, implementation, C-7, C-20
ROM, encoding, C-19
for single-cycle implementation, 261–262
Control hazards, 271–274, 307–315
branch delay reduction, 308–310
branch not taken assumption, 307–308
branch prediction as solution, 274
delayed decision approach, 274*b*
dynamic branch prediction, 310–314
logic implementation in Verilog, 345.e8
pipeline stalls as solution, 272*f*
pipeline summary, 314–315
solutions, 272*f*
static multiple-issue processors and, 324
Control lines
asserted, 256
in datapath, 255*f*
execution/address calculation, 291
final three stages, 293*f*
instruction decode/register file read, 291
instruction fetch, 291
memory access, 291
setting of, 256
values, 291
write-back, 291
Control signals
ALUOp, 255
defined, 242
effect of, 256*f*
multi-bit, 256
pipelined datapaths with, 290–294
truth tables, C-14*f*
Control units, 239–240. *See also* Arithmetic logic unit (ALU)
address select logic, C-24, C-25*f*
combinational, implementing, C-4–C-8
with explicit counter, C-23*f*
illustrated, 257*f*
logic equations, C-11–C-12
main, designing, 253–256
as microcode, C-28*f*
next-state outputs, C-10, C-12*b*–C-13*b*
output, 251–253, C-10
RISC-V, C-10*f*
Cooperative thread arrays (CTAs), B-30
Coprocessors
defined, 212*b*
Core RISC-V instruction set
abstract view, 238*f*
desktop RISC, D-9*f*
implementation, 236–237
implementation illustration, 239*f*
overview, 237–240
subset, 236
Cores
defined, 43
number per chip, 43
Correlation predictor, 312–313
Cosmic Cube, 553.e6–553.e7
CPU, 9
Cray computers, 227.e4, 227.e5
Critical word first, 384
Crossbar networks, 527–528
CTSS (Compatible Time-Sharing System), 473.e13
CUDA programming environment, 515, B-5–B-6
barrier synchronization, B-18, B-34
development, B-17, B-17–B-18
hierarchy of thread groups, B-18
kernels, B-19, B-24
key abstractions, B-18
paradigm, B-19–B-22
parallel plus-scan template, B-61*f*
per-block shared memory, B-58
plus-reduction implementation, B-63*f*
programs, B-6, B-24
scalable parallel programming with, B-17–B-23
shared memories, B-18
threads, B-36
Cyclic redundancy check, 415*b*–416*b*
Cylinder, 374

D

D flip-flops, A-50–A-51, A-52
D latches, A-50–A-51, A-51
Data bits, 413*f*
Data flow analysis, 144.e8
Data hazards, 268–271, 294–307. *See also* Hazards
forwarding, 268–269, 294–307
load-use, 269–271, 308
stalls and, 303–307
Data parallel problem decomposition, B-17, B-18*f*
Data race, 121
Data selectors, 237–238
Data transfer instructions. *See also* Instructions
defined, 68–69
load, 69
offset, 69
store, 70–71

Datacenters, 7
Data-level parallelism, 500
Datapath elements
  defined, 243
  sharing, 248–249
Datapaths
  branch, 246–247
  building, 243–251
  control signal truth tables, C-14*f*
  control unit, 257*f*
  defined, 19
  design, 243
  exception handling, 318*f*
  for fetching instructions, 245*f*
  for hazard resolution via forwarding, 302*f*
  for memory instructions, 247
  in operation for branch-if-equal instruction, 260–261
  in operation for load instruction, 259*f*
  in operation for R-type instruction, 258*f*
  operation of, 256–261
  pipelined, 276–294
  for RISC-V architecture, 249
  for R-type instructions, 256–259
  single, creating, 247–251
  single-cycle, 275
  static two-issue, 326*f*
Deasserted signals, 242, A-4
DEC PDP-8, 162.e2*f*
Decimal numbers
  binary number conversion to, 77*b*
  defined, 74
Decision-making instructions, 92–98
Decoders, A-9–A-10
  two-level, A-64
Decoding machine language, 118–120
Defect, 26–27
Delayed branches, 274. *See also* Branches
  as control hazard solution, 274
  embedded RISCs and, D-23
  reducing, 308–310
Delayed decision, 274*b*
DeMorgan's theorems, A-11
Denormalized numbers, 216
Dependability via redundancy, 12
Dependable memory hierarchy, 410–416
  failure, defining, 410–412
Dependences
  between pipeline registers, 238–239
  between pipeline registers and ALU inputs, 297–298
  bubble insertion and, 305
  detection, 297*b*
  name, 327
  sequence, 295
Design
  compromises and, 84
  datapath, 243
  digital, 345
  logic, 240–243
  main control unit, 253–256
  memory hierarchy, challenges, 449*f*
  pipelining instruction sets, 267
Desktop and server RISCs. *See also* Reduced instruction set computer (RISC) architectures
  addressing modes, D-6
  architecture summary, D-4*f*
  arithmetic/logical instructions, D-11*f*
  conditional branches, D-16
  constant extension summary, D-9*f*
  control instructions, D-11*f*
  conventions equivalent to MIPS core, D-12*f*
  data transfer instructions, D-10*f*
  features added to, D-45*f*
  floating-point instructions, D-12*f*
  instruction formats, D-7*f*
  multimedia extensions, D-16–D-18
  multimedia support, D-18*f*
Desktop computers, defined, 5
Device driver, 529.e4
DGEMM (Double precision General Matrix Multiply), 218–219, 342, 344–345, 405, 530
  cache blocked version of, 407*f*
  optimized C version of, 220*f*, 342*f*, 466*f*
  performance, 344*f*, 408*f*
Dicing, 27
Dies, 26–27
Digital design pipeline, 345
Digital signal-processing (DSP) extensions, D-19
DIMMs (dual inline memory modules), 473.e4
Direct Data IO (DDIO), 529.e6
Direct memory access (DMA), 529.e2*f*, 529.e3
Direct3D, B-13
Direct-mapped caches. *See also* Caches
  address portions, 399*f*
  choice of, 400–401
  defined, 376, 394
  illustrated, 377*f*
  memory block location, 395*f*
  misses, 397*b*–398*b*
  single comparator, 399
  total number of bits, 382
Dirty bit, 430*b*
Dirty pages, 430*b*
Disk memory, 373–375
Displacement addressing, 118
Distributed Block-Interleaved Parity (RAID 5), 458.e5–458.e6
Divide algorithm, 186*b*
Dividend, 184
Division, 183–191
  algorithm, 185*f*
  dividend, 184
  divisor, 184
Divisor, 184
divu (Divide Unsigned). *See also* Arithmetic
  faster, 188–189
  floating-point, 206–212
  hardware, 184–187
  hardware, improved version, 187*f*
  operands, 184
  quotient, 184
  remainder, 184
  in RISC-V, 189
  signed, 187–188
  SRT, 189
Don't cares, A-17–A-18
  example, A-17*b*–A-18*b*
  term, 253
Double data rate (DDR), 371–372
Double Data Rate (DDR) SDRAM, 371–372, A-64
Double precision. *See also* Single precision
  defined, 193
  FMA, B-45, B-45–B-46
  GPU, B-45, B-74*b*
  representation, 212–214
Doubleword, 67, 151
Dual inline memory modules (DIMMs), 373
Dynamic branch prediction, 310–314. *See also* Control hazards
  branch prediction buffer, 310
  loops and, 312*b*
Dynamic hardware predictors, 274
Dynamic multiple-issue processors, 322, 328–333. *See also* Multiple issue
  pipeline scheduling, 329–333
  superscalar, 328
Dynamic pipeline scheduling, 329–333

commit unit, 329
concept, 329
hardware-based speculation, 331–333
primary units, 330*f*
reorder buffer, 334*b*
reservation station, 329
Dynamic random access memory (DRAM), 370–373, A-62–A-64
bandwidth external to, 390
cost, 23
defined, 19, A-62
DIMM, 473.e4
Double Date Rate (DDR), 371–372
early board, 473.e4*f*
GPU, B-37–B-38
growth of capacity, 25*f*
history, 473.e1
internal organization of, 372*f*
pass transistor, A-62*b*–A-64*b*
SIMM, 473.e4, 473.e5*f*
single-transistor, A-63*f*
size, 390
speed, 23–24
synchronous (SDRAM), 371–372, A-59, A-64
two-level decoder, A-64
Dynamically linked libraries (DLLs), 130–132
defined, 130
lazy procedure linkage version, 130

E

Early restart, 384*b*
Edge-triggered clocking methodology, 241–242, A-47, A-72–A-73
advantage, A-48
clocks, A-72–A-73
drawbacks, A-73–A-74
illustrated, A-49*f*
rising edge/falling edge, A-47
EDSAC (Electronic Delay Storage Automatic Calculator), 54.e2, 473.e1, 473.e2*f*
Eispack, 227.e2–227.e3
Electrically erasable programmable read-only memory (EEPROM), 373
Elements
combinational, 240
datapath, 243, 248–249
memory, A-49–A-57
state, 240, 242, 244*f*, A-47, A-49*b*
Embedded computers, 5–6
application requirements, 6
design, 5
growth, 54.e11
Embedded Microprocessor Benchmark Consortium (EEMBC), 54.e11
Embedded RISCs. *See also* Reduced instruction set computer (RISC) architectures
addressing modes, D-6
architecture summary, D-4*f*
arithmetic/logical instructions, D-14*f*
conditional branches, D-16
constant extension summary, D-9*f*
control instructions, D-15*f*
data transfer instructions, D-13*f*
delayed branch and, D-23
DSP extensions, D-19
general purpose registers, D-5
instruction conventions, D-15*f*
instruction formats, D-8*f*
multiply-accumulate approaches, D-19*f*
Encoding
defined, C-31
RISC-V instruction, 85*f*, 119*f*
ROM control function, C-18
ROM logic function, A-15
x86 instruction, 153–154
ENIAC (Electronic Numerical Integrator and Calculator), 54.e2–54.e3, 473.e1
EPIC, 347.e4
Error correction, A-64–A-66
Error Detecting and Correcting Code (RAID 2), 458.e4
Error detection, A-65–A-66
Error detection code, 412
Ethernet, 23–24
EX stage
load instructions, 282*f*
overflow exception detection, 317, 320*f*
store instructions, 284*f*
Exabyte, 6*f*
Exception enable, 439*b*
Exceptions, 315–321
association, 321*b*
datapath with controls for handling, 318*f*
defined, 193, 315
detecting, 315
event types and, 315
imprecise, 321*b*
interrupts *versus*, 315
pipelined computer example, 318*b*–319*b*
in pipelined implementation, 317–321
precise, 321*b*
reasons for, 316–317
result due to overflow in add instruction, 320*f*
in RISC-V architecture, 316–317
saving/restoring stage on, 440
Executable files
defined, 127–129
Execute or address calculation stage, 282
Execute/address calculation
control line, 291
load instruction, 282
store instruction, 282
Execution time
CPU, 32–34
pipelining and, 276
as valid performance measure, 50–51
Explicit counters, C-23–C-24, C-26*f*
Exponents, 192

F

Failures, synchronizer, A-75–A-76
Fallacies. *See also* Pitfalls
Amdahl's law, 548
arithmetic, 222
assembly language for performance, 158*b*
commercial binary compatibility importance, 158*b*
defined, 49
GPUs, B-72, B-75
low utilization uses little power, 50*b*
peak performance, 548*b*
pipelining, 345
powerful instructions mean higher performance, 157
right shift, 222*b*
False sharing, 457
Fast carry
with first level of abstraction, A-38–A-40
with "infinite" hardware, A-38
with second level of abstraction, A-40–A-45
Fast Fourier Transforms (FFT), B-53
Fault avoidance, 411
Fault forecasting, 411
Fault tolerance, 411

Fermi architecture, 515, 544
Field programmable devices (FPDs), A-77–A-78
Field programmable gate arrays (FPGAs), A-77
Fields
  defined, 83
  format, C-31
  names, 83
  RISC-V, 83–89
Files, register, 244, 249, A-49*b*, A-53–A-55
Fine-grained multithreading, 506
Finite-state machines (FSMs), 449–454, A-66–A-71
  control, C-8–C-22
  controllers, 452*f*
  for multicycle control, C-9*f*
  for simple cache controller, 453–454
  implementation, 451, A-69
  Mealy, 452
  Moore, 452*b*–453*b*
  next-state function, 451, A-66
  output function, A-66, A-68
  state assignment, A-69
  state register implementation, A-70*f*
  style of, 452*b*–453*b*
  synchronous, A-66
  SystemVerilog, 459.e6*f*
  traffic light example, A-67
Flash memory, 373
  defined, 23
Flat address space, 469
Flip-flops
  D flip-flops, A-50–A-51, A-52
  defined, A-50–A-51
Floating point, 191–216
  assembly language, 207*f*
  backward step, 227.e3–227.e4
  binary to decimal conversion, 197*b*
  branch, 206
  challenges, 226
  diversity *versus* portability, 227.e2–227.e3
  division, 206
  first dispute, 227.e1–227.e2
  form, 192
  fused multiply add, 214*b*
  guard digits, 213*b*
  history, 227.e2
  IEEE 754 standard, 193–198
  intermediate calculations, 212–213
  operands, 207*f*
  overflow, 192
  packed format, 218
  precision, 223
  procedure with two-dimensional matrices, 80*b*
  programs, compiling, 79*b*–80*b*
  registers, 212*b*
  representation, 192–193
  RISC-V instruction frequency for, 226*f*
  RISC-V instructions, 206–212
  rounding, 212–213
  sign and magnitude, 192
  SSE2 architecture, 217, 217*f*
  subtraction, 206
  underflow, 192
  units, 213–214
  in x86, 217*f*
Floating vectors, 227.e2
Floating-point addition, 198–201
  arithmetic unit block diagram, 202*f*
  binary, 199*b*–201*b*
  illustrated, 200*f*
  instructions, 206–212
  steps, 198–199
Floating-point arithmetic (GPUs), B-41–B-46
  basic, B-42
  double precision, B-45–B-46, B-74*b*
  performance, B-44
  specialized, B-42–B-44
  supported formats, B-42
  texture operations, B-44
Floating-point control and status register (fcsr), 193
Floating-point instructions
  desktop RISC, D-12*f*
  SPARC, D-31–D-32
Floating-point multiplication, 201–206
  binary, 205*b*–206*b*
  illustrated, 204*f*
  instructions, 206
  significands, 201–205
  steps, 201–205
Flow-sensitive information, 144.e13*b*–144.e14*b*
Flushing instructions, 308–310
  exceptions and, 319*b*
For loops, 142, 144.e25
  inner, 144.e23
  SIMD and, 553.e2
Format fields, C-31
Fortran, 162.e6
Forwarding, 294–307
  ALU before, 299*f*
  control, 300
  datapath for hazard resolution, 302*f*
  defined, 268–269
  graphical representation, 269*f*
  illustrations, 345.e20
  multiple results and, 271
  multiplexors, 300*f*
  pipeline registers before, 299*f*
  with two instructions, 268*b*–269*b*
  Verilog implementation, 345.e3
Fractions, 192–193
Frame buffer, 18
Frame pointers, 104–105
Front end, 144.e2
Fully associative caches. *See also* Caches
  block replacement strategies, 445–446
  choice of, 445
  defined, 395
  memory block location, 395*f*
  misses, 398
Fully connected networks, 527
Fused-multiply-add (FMA) operation, 214*b*, B-45

G

Game consoles, B-9
Gates, A-3–A-4, A-4–A-9
  AND, A-12–A-13, C-7
  delays, A-45
  mapping ALU control function to, C-4–C-7
  NAND, A-8–A-9
  NOR, A-8–A-9, A-49*f*
Gather-scatter, 503, 544
General Purpose GPUs (GPGPUs), B-5
General-purpose registers, 147
  architectures, 162.e2*f*
  embedded RISCs, D-5
Generate
  defined, A-39
  example, A-44*b*–A-45*b*
  super, A-40
Gigabyte, 6*f*
Global common subexpression elimination, 144.e5
Global memory, B-21, B-39
Global miss rates, 408*b*

Global optimization, 144.e4–144.e10
code, 144.e6
implementing, 144.e7
Global pointers, 104*b*
GPU computing. *See also* Graphics processing units (GPUs)
defined, B-5–B-6
visual applications, B-6
GPU system architectures, B-7–B-12
graphics logical pipeline, B-10
heterogeneous, B-7–B-9
implications for, B-24–B-25
interfaces and drivers, B-9–B-10
unified, B-10–B-11
Graph coloring, 144.e11
Graphics displays
computer hardware support, 18
LCD, 18
Graphics logical pipeline, B-10
Graphics processing units (GPUs), 514–521. *See also* GPU computing
as accelerators, 514
attribute interpolation, B-43–B-44
defined, 46, 498–499, B-3
evolution, B-5
fallacies and pitfalls, B-72–B-75
floating-point arithmetic, B-16, B-41–B-46, B-74
GeForce 8-series generation, B-5
general computation, B-73*b*
General Purpose (GPGPUs), B-5
graphics mode, B-6
graphics trends, B-4
history, B-3–B-4
logical graphics pipeline, B-13–B-14
mapping applications to, B-55–B-72
memory, 514
multilevel caches and, 514
N-body applications, B-65–B-68
NVIDIA architecture, 515–517
parallel memory system, B-36–B-41
parallelism, 515, B-76
performance doubling, B-4
perspective, 519–521
programming, B-12–B-25
programming interfaces to, B-17
real-time graphics, B-13
Graphics shader programs, B-14–B-15
Gresham's Law, 227, 227.e1
Grid computing, 525*b*–526*b*
Grids, B-19
GTX 280, 540–541
Guard digits
defined, 212–213
rounding with, 213*b*

H

Half precision, B-42
Halfwords, 112
Hamming, Richard, 412
Hamming distance, 412
Hamming Error Correction Code (ECC), 412–413
calculating, 412
Hard disks
access times, 23
defined, 23
Hardware
as hierarchical layer, 13*f*
language of, 14–16
operations, 63–67
supporting procedures in, 98–108
synthesis, A-21
translating microprograms to, C-28–C-32
virtualizable, 418
Hardware description languages. *See also* Verilog
defined, A-20
using, A-20–A-26
VHDL, A-20–A-21
Hardware multithreading, 506–509
coarse-grained, 506–507
options, 507*f*
simultaneous, 507
Hardware-based speculation, 331–333
Harvard architecture, 54.e3
Hazard detection units, 303
pipeline connections for, 306–307
Hazards. *See also* Pipelining
control, 271–274, 307–315
data, 268–271, 294–307
forwarding and, 302*b*
structural, 267–268, 284
Heap
allocating space on, 104–105
defined, 105
Heterogeneous systems, B-4–B-5
architecture, B-7–B-12
defined, B-3
Hexadecimal numbers, 82
binary number conversion to, 82*f*, 83*b*
Hierarchy of memories, 12
High-level languages, 14–16
benefits, 16
computer architectures, 162.e4
importance, 16
High-level optimizations, 144.e3–144.e4
Hit rate, 368
Hit time
cache performance and, 393–394
defined, 368–369
Hit under miss, 460
Hold time, A-52–A-53
Horizontal microcode, C-32
Hot-swapping, 458.e6–458.e7
Human genome project, 4

I

I/O, 529.e1–529.e2
on system performance, 458.e1*b*–458.e2*b*
I/O benchmarks. *See* Benchmarks
IBM 360/85, 473.e5
IBM 701, 54.e4
IBM 7030, 347.e1
IBM ALOG, 227.e6
IBM Blue Gene, 553.e8–553.e9
IBM Personal Computer, 54.e7, 162.e5
IBM System/360 computers, 54.e5*f*, 227.e5, 227.e6, 347.e1
IBM z/VM, 473.e12
ID stage
branch execution in, 309–310
load instructions, 282*f*
store instruction in, 281*f*
IEEE 754 floating-point standard, 227.e7–227.e9, 193–198, 194*f*. *See also* Floating point
first chips, 227.e7–227.e9
in GPU arithmetic, B-42
implementation, 227.e9
rounding modes, 213–214
today, 227.e9
If statements, 115–116
If-then-else, 93*b*
Imagination Technologies, 145
Immediate addressing, 118
Immediate instructions, 72
Imprecise interrupts, 347.e2–347.e3, 321*b*
Index-out-of-bounds check, 96
Induction variable elimination, 144.e6
Inheritance, 144.e14
In-order commit, 330–331

Input devices, 16–17
Inputs, 253
Instances, 144.e14
Instruction count, 36, 38
Instruction decode/register file read stage
  control line, 290–294
  load instruction, 279
  store instruction, 284
Instruction execution illustrations, 345.e13–345.e20
  clock cycle 9, 345.e25*f*
  clock cycles 1 and 2, 345.e21*f*
  clock cycles 3 and 4, 345.e22*f*
  clock cycles 5 and 6, 345.e23*f*
  clock cycles 7 and 8, 345.e24*f*
  examples, 345.e15–345.e20
  forwarding, 345.e20
  no hazard, 345.e15
  pipelines with stalls and forwarding, 345.e20
Instruction fetch stage
  control line, 291
  load instruction, 279
  store instruction, 284
Instruction formats, 153
  defined, 82
  desktop/server RISC architectures, D-7*f*
  embedded RISC architectures, D-8*f*
  I-type, 84
  MIPS, 146*f*
  RISC-V, 146*f*
  R-type, 84, 253–254
  SB-type, 115
  S-type, 84–85
  UJ-type, 115
  U-type, 113–114
  x86, 153–154
Instruction latency, 346–347
Instruction mix, 39–40, 54.e9
Instruction set architecture
  branch address calculation, 246
  defined, 22, 52
  history, 162
  maintaining, 52
  protection and, 419
  thread, B-31–B-34
  virtual machine support, 418
Instruction sets, B-49
  MIPS-32, 146*f*
  RISC-V, 160
  x86 growth, 162*f*
Instruction-level parallelism (ILP), 344–345. *See also* Parallelism
  compiler exploitation, 347.e4–347.e5
  defined, 43*b*, 321–322
  exploitation, increasing, 333
  and matrix multiply, 342–345
Instructions, 60, D-25–D-27, D-40, D-40–D-43. *See also* Arithmetic instructions; MIPS; Operands
  add immediate, 72–74
  addition, 176
  Alpha, D-27–D-29
  arithmetic-logical, 243–244
  ARM, D-36–D-38
  assembly, 65
  basic block, 95*b*
  cache-aware, 472
  conditional branch, 92–93, 93*b*
  conditional move, 313*b*–314*b*
  data transfer, 68
  decision-making, 92–98
  defined, 14, 62
  desktop RISC conventions, D-12*f*
  as electronic signals, 81–82
  embedded RISC conventions, D-15*f*
  encoding, 85*f*
  fetching, 245*f*
  floating-point, 206–212
  floating-point (x86), 217*f*
  flushing, 308–310
  immediate, 72
  introduction to, 62–63
  left-to-right flow, 277
  load, 69
  logical operations, 89–92
  M32R, D-40
  memory access, B-33–B-34
  memory-reference, 237
  multiplication, 183
  nop, 304–305
  PA-RISC, D-34–D-36
  performance, 35–36
  pipeline sequence, 304*f*
  PowerPC, D-12–D-13, D-32–D-34
  PTX, B-31, B-32*f*
  representation in computer, 81–89
  restartable, 440–441
  resuming, 440*b*–441*b*
  R-type, 243–244, 248–249
  SPARC, D-29–D-32
  store, 71
  store-conditional doubleword, 122–123
  subtraction, 176
  SuperH, D-39–D-40
  thread, B-30–B-31
  Thumb, D-38–D-39
  vector, 500–502
  as words, 62
  x86, 146–155
Instructions per clock cycle (IPC), 322
Integrated circuits (ICs), 19. *See also* specific chips
  cost, 27
  defined, 25
  manufacturing process, 26
  very large-scale (VLSIs), 25
Intel Core i7, 46–49, 236, 493, 540–545
  address translation for, 460*f*
  architectural registers, 337–338
  caches in, 461*f*
  memory hierarchies of, 459–464
  microarchitecture, 337
  performance of, 462
  SPEC CPU benchmark, 46–48
  SPEC power benchmark, 48–49
  TLB hardware for, 460*f*
Intel Core i7 920, 337–340
  microarchitecture, 337
Intel Core i7 960
  benchmarking and rooflines of, 540–545
Intel Core i7 Pipelines, 334–342
  memory components, 338*f*
  performance, 340–342
  program performance, 341*b*
  specification, 335*f*
Intel IA-64 architecture, 162.e2*f*
Intel Paragon, 553.e6–553.e7
Intel Threading Building Blocks, B-60
Intel x86 microprocessors
  clock rate and power for, 40*f*
Interference graphs, 144.e10
Interleaving, 390
Interprocedural analysis, 144.e13*b*–144.e14*b*
Interrupt enable, 439*b*
Interrupt-driven I/O, 529.e3
Interrupts
  defined, 193, 315
  event types and, 315
  exceptions *versus*, 315
  imprecise, 347.e2–347.e3, 321*b*
  precise, 321*b*
  vectored, 316
Intrinsity FastMATH processor, 387–389
  caches, 388*f*
  data miss rates, 389*f*, 399*f*
  read processing, 434*f*
  TLB, 432–435
  write-through processing, 434*f*

Inverted page tables, 429
Issue packets, 324–325
I-type, 87*b*

J

Java
  bytecode, 132
  bytecode architecture, 144.e10–144.e12
  characters in, 111–113
  compiling in, 144.e18–144.e19
  goals, 132
  interpreting, 132, 144–145, 144.e14
  keywords, 144.e20
  method invocation in, 144.e20
  pointers, 144.e25–144.e26
  primitive types, 144.e25
  programs, starting, 132–133
  reference types, 144.e25
  sort algorithms, 141*f*
  strings in, 111–113
  translation hierarchy, 132*f*
  while loop compilation in, 144.e17*b*–144.e18*b*
Java Virtual Machine (JVM), 144.e15, 145
Jump-and-link register instruction (jalr), 97–99
Jump instructions, D-26
  branch instruction *versus*, 250*f*
  control and datapath for, 251
  implementing, 237–240
  instruction format, 250
Just In Time (JIT) compilers, 133, 552

K

Karnaugh maps, A-18
Kernel mode, 437
Kernels
  CUDA, B-19, B-24
  defined, B-19–B-22
Kilobyte, 6*f*

L

LAPACK, 223–224
Large-scale multiprocessors, 553.e6–553.e7
Latches
  D latch, A-50–A-51, A-51
  defined, A-50–A-51
Latency
  instruction, 346–347
  memory, B-74*b*
  pipeline, 276*b*
  use, 325–327
lb (load byte), 64*f*
lbu (load byte, unsigned), 64*f*
ld (load doubleword), 64*f*
Leaf procedures. *See also* Procedures
  defined, 102
  example, 112*f*
Least recently used (LRU)
  as block replacement strategy, 445–446
  defined, 401
  pages, 426–428
Least significant bits
  defined, 74
  SPARC, D-31
Left-to-right instruction flow, 277
Level-sensitive clocking, A-73–A-74, A-74–A-75
  defined, A-73–A-74
  two-phase, A-74
lh (load halfword), 64*f*
lhu (load halfword, unsigned), 64*f*
Link, 529.e1–529.e2
Linkers, 127–129
  defined, 127
  executable files, 127–129
  steps, 127
Linking object files, 128*b*–129*b*
Linpack, 227.e2–227.e3, 530
Liquid crystal displays (LCDs), 18
LISP, SPARC support, D-30
Live range, 144.e10
Livermore Loops, 54.e10
Load balancing, 497*b*–498*b*
Load byte, 109
Load doubleword, 69, 71–72
Load instructions. *See also* Store instructions
  access, B-41
  base register, 254
  compiling with, 71*b*
  datapath in operation for, 259*f*
  defined, 69
  EX stage, 282*f*
  halfword unsigned, 112
  ID stage, 281*f*
  IF stage, 281*f*
  load byte unsigned, 78
  load half, 112
  MEM stage, 283*f*
  pipelined datapath in, 286*f*
  signed, 78*b*
  unit for implementing, 247*f*
  unsigned, 78*b*
  WB stage, 283*f*
Loaders, 130
Load-reserved doubleword, 122–123
Load-store architectures, 162.e2
Load upper immediate, 113–114
Load-use data hazard, 269–271, 308
Load-use stalls, 308
Load word, 113*b*
Load word unsigned, 113*b*
Local area networks (LANs), 24. *See also* Networks
Local memory, B-21, B-40
Local miss rates, 408*b*
Local optimization, 144.e4. *See also* Optimization
  implementing, 144.e7
Locality
  principle, 366–367
  spatial, 366, 369*b*
  temporal, 366, 369*b*
Lock synchronization, 121
Locks, 510–513
Logic
  address select, C-24, C-25*f*
  ALU control, C-6–C-7
  combinational, 242, A-5, A-9–A-20
  components, 241
  control unit equations, C-11*f*
  design, 240–243
  equations, A-7*b*
  minimization, A-18
  programmable array (PAL), A-77
  sequential, A-4–A-5, A-55–A-57
  two-level, A-11–A-14
Logical operations, 89–92
  AND, 90
  desktop RISC, D-11*f*
  embedded RISC, D-13*f*
  NOT, 91
  OR, 91
  shifts, 90
  xor, 91
Long instruction word (LIW), 347.e4
Lookup tables (LUTs), A-77–A-78
Loop unrolling
  defined, 144.e3–144.e4, 327–328
  for multiple-issue pipelines, 327*b*–328*b*
  register renaming and, 327
Loops, 94–96
  conditional branches in, 115–116
  for, 142
  prediction and, 312*b*
  test, 142–143
  while, compiling, 94*b*–95*b*
lr.d (load reserved), 64*f*

lui (load upper immediate), 64*f*
lw (load word), 64*f*
lwu (load word, unsigned), 64*f*

M

M32R, D-15, D-40
Machine code, 82
Machine instructions, 82
Machine language, 15*f*
  branch offset in, 116*b*–117*b*
  decoding, 118–120
  defined, 14, 82
  illustrated, 15*f*
  RISC-V, 87–89
  SRAM, 19–22
  translating RISC-V assembly language into, 85*b*–86*b*
Main memory, 420. *See also* Memory
  defined, 23
  page tables, 429
  physical addresses, 420
Mapping applications, B-55–B-72
Mark computers, 54.e3
Matrix multiply, 218–222, 545–548
Mealy machine, 452, A-67, A-70–A-71, A-71*b*
Mean time to failure (MTTF), 410–411
  versus AFR of disks, 410*b*–411*b*
  improving, 411–412
Media Access Control (MAC) address, 529.e6
Megabyte, 6*f*
Memory
  addresses, 78*b*
  affinity, 538*f*
  atomic, B-21
  bandwidth, 371–372, 389*b*
  cache, 19–22, 375–410
  CAM, 400*b*–401*b*
  constant, B-40
  control, C-26
  defined, 19
  DRAM, 19, 371–373, A-62–A-64
  flash, 23
  global, B-21, B-39
  GPU, 514
  instructions, datapath for, 247
  local, B-21, B-40
  main, 23
  nonvolatile, 22–23
  operands, 68–72
  parallel system, B-36–B-41
  read-only (ROM), A-14–A-16
  SDRAM, 371–372
  secondary, 23
  shared, B-17, B-39–B-40
  spaces, B-39
  SRAM, A-57–A-59
  stalls, 392
  technologies for building, 24–28
  texture, B-40
  virtual, 419–443
  volatile, 22–23
Memory access instructions, B-33–B-34
Memory access stage
  control line, 292*f*
  load instruction, 282*f*
  store instruction, 282
Memory bandwidth, 540–541, 549*b*
Memory consistency model, 458*b*
Memory elements, A-49–A-57
  clocked, A-50
  D flip-flop, A-50–A-51, A-52
  D latch, A-51
  DRAMs, A-62–A-64
  flip-flop, A-50
  hold time, A-52–A-53
  latch, A-50
  setup time, A-52–A-53, A-53*f*
  SRAMs, A-57–A-59
  unclocked, A-50
Memory hierarchies, 537
  of ARM cortex-A53, 459–464
  block (or line), 367–368
  cache performance, 390–410
  caches, 375–390
  common framework, 443–449
  defined, 367
  design challenges, 449*b*
  development, 473.e5–473.e7
  exploiting, 364
  of Intel Core i7, 459–464
  level pairs, 368*f*
  multiple levels, 367
  overall operation of, 435*b*–436*b*
  parallelism and, 458.e1–458.e2, 454–458
  pitfalls, 468–472
  program execution time and, 409
  quantitative design parameters, 443*f*
  redundant arrays and inexpensive disks, 458
  reliance on, 369
  structure, 367*f*
  structure diagram, 370*f*
  variance, 409*b*
  virtual memory, 419–443
Memory rank, 373
Memory technologies, 370–375
  disk memory, 373–375
  DRAM technology, 370–373
  flash memory, 373
  SRAM technology, 370–371
Memory-mapped I/O, 529.e2
Memory-stall clock cycles, 391
Message passing
  defined, 521
  multiprocessors, 521–526
Metastability, A-75–A-76
Methods
  defined, 144.e14
  invoking in Java, 144.e19–144.e20
Microarchitectures, 337
  Intel Core i7 920, 337–340
Microcode
  assembler, C-30
  control unit as, C-28*f*
  defined, C-27
  dispatch ROMs, C-30, C-30*f*
  horizontal, C-32
  vertical, C-32
Microinstructions, C-31
Microprocessors
  design shift, 493
  multicore, 8, 43, 492–493
Microprograms
  as abstract control representation, C-30–C-31
  field translation, C-28–C-29
  translating to hardware, C-28–C-32
Migration, 456
Million instructions per second (MIPS), 51
Minterms
  defined, A-12–A-13, C-20
  in PLA implementation, C-20
MIP-map, B-44
MIPS and RISC-V
  common features between, 145
MIPS-16
  16-bit instruction set, D-41–D-42
  immediate fields, D-41
  instructions, D-40–D-43
  MIPS core instruction changes, D-42–D-43
  PC-relative addressing, D-41
MIPS-32 instruction set, 145
MIPS-64 instructions, 145, D-25–D-27

conditional procedure call instructions, D-27
constant shift amount, D-25
jump/call not PC-relative, D-26
move to/from control registers, D-26
nonaligned data transfers, D-25
NOR, D-25
parallel single precision floating-point operations, D-27
reciprocal and reciprocal square root, D-27
SYSCALL, D-25
TLB instructions, D-26–D-27
Mirroring, 458.e4
Miss penalty
defined, 368–369
determination, 383–384
multilevel caches, reducing, 402–405
Miss rates
block size *versus*, 383–384
data cache, 444*f*
defined, 368
global, 408*b*
improvement, 383–384
Intrinsity FastMATH processor, 389
local, 408*b*
miss sources, 448
split cache, 389*b*
Miss under miss, 460
MMX (MultiMedia eXtension), 217
Moore machines, 452, A-67, A-70–A-71, A-71*b*
Moore's law, 529.e1–529.e2, 11, 371, 514, B-72*b*
Most significant bit
1-bit ALU for, A-33*f*
defined, 74
MS-DOS, 473.e15
Multicore, 509–514
Multicore multiprocessors, 8, 43
defined, 8, 492–493
MULTICS (Multiplexed Information and Computing Service), 473.e8
Multilevel caches. *See also* Caches
complications, 408*b*
defined, 390, 408*b*
miss penalty, reducing, 402–405
performance of, 402*b*–403*b*
summary, 409–410
Multimedia extensions
desktop/server RISCs, D-16–D-18
as SIMD extensions to instruction sets, 553.e3
vector *versus*, 501*b*–502*b*
Multiple dimension arrays, 212
Multiple instruction multiple data (MIMD), 550–551
defined, 499–500
first multiprocessor, 553.e3–553.e4
Multiple instruction single data (MISD), 499–500
Multiple issue, 322
code scheduling, 326*b*–327*b*
dynamic, 322, 328–333
issue packets, 324–325
loop unrolling and, 327*b*–328*b*
processors, 322
static, 322, 324–328
throughput and, 332*b*
Multiple processors, 545–548
Multiple-clock-cycle pipeline diagrams, 286–287
five instructions, 288*f*
illustrated, 287–290
Multiplexors, A-10
controls, 451
in datapath, 255*f*
defined, 237–238
forwarding, control values, 300*f*
selector control, 251
two-input, A-10
Multiplicand, 178
Multiplication, 177–183. *See also* Arithmetic
fast, hardware, 182
faster, 182–183
first algorithm, 180*f*
floating-point, 201–206
hardware, 178–182
instructions, 183
operands, 183
product, 183
sequential version, 178–182
signed, 182
Multiplier, 178
Multiply algorithm, 178–182
Multiply-add (MAD), B-42
Multiprocessors
benchmarks, 530–540
bus-based coherent, 553.e6
defined, 492
historical perspective, 553
large-scale, 553.e6–553.e7
message-passing, 521–526
multithreaded architecture, B-26–B-27, B-36
organization, 491, 521
for performance, 549
shared memory, 492–493, 509–514
software, 493*f*
TFLOPS, 553.e5
UMA, 510
Multistage networks, 527–528
Multithreaded multiprocessor architecture, B-25–B-36
conclusion, B-36
ISA, B-31–B-34
massive multithreading, B-25–B-26
multiprocessor, B-26–B-27
multiprocessor comparison, B-35–B-36
SIMT, B-27–B-29
special function units (SFUs), B-35
streaming processor (SP), B-34
thread instructions, B-30–B-31
threads/thread blocks management, B-30
Multithreading, B-25–B-26
coarse-grained, 506–507
defined, 498–499
fine-grained, 506
hardware, 506–509
simultaneous (SMT), 507
Must-information, 144.e13*b*–144.e14*b*
Mutual exclusion, 121

N

Name dependence, 327
NAND gates, A-8–A-9
NAS (NASA Advanced Supercomputing), 532
N-body
all-pairs algorithm, B-65
GPU simulation, B-71
mathematics, B-65–B-66
multiple threads per body, B-68–B-72
optimization, B-67
performance comparison, B-69–B-70
results, B-70–B-72
shared memory use, B-67–B-68
Negation shortcut, 78–79
Nested procedures, 102–104
compiling recursive procedure showing, 102*b*–103*b*
NetFPGA 10-Gigagit Ethernet card, 529.e1*f*, 529.e2*f*
Network of Workstations, 553.e7–553.e8

Network topologies, 526–529
implementing, 528–529
multistage, 529*f*
Networking, 529.e3–529.e4
operating system in, 529.e3–529.e5
performance improvement, 529.e6–529.e9
Networks, 23–24
advantages, 23
bandwidth, 527
crossbar, 527–528
fully connected, 527
local area (LANs), 23–24
multistage, 527–528
wide area (WANs), 23–24
Newton's iteration, 212*b*
Next state
nonsequential, C-24
sequential, C-23–C-24
Next-state function, 451, A-66
defined, 451
implementing, with sequencer, C-22–C-28
Next-state outputs, C-12*b*–C-13*b*, C-27
example, C-12
implementation, C-12–C-13
logic equations, C-12*b*–C-13*b*
truth tables, C-13–C-15
No Redundancy (RAID 0), 458.e3
No write allocation, 386
Nonblocking assignment, A-24
Nonblocking caches, 334*b*, 460
Nonuniform memory access (NUMA), 510
Nonvolatile memory, 22–23
Nops, 304–305
NOR gates, A-8–A-9
cross-coupled, A-49*f*
D latch implemented with, A-51*f*
NOR operation, D-25
NOT operation, 91, A-6
Numbers
binary, 74
computer *versus* real-world, 215
decimal, 74, 77*b*
denormalized, 216
hexadecimal, 83
signed, 74–81
unsigned, 74–81
NVIDIA GeForce 8800, B-46–B-55
all-pairs N-body algorithm, B-71
dense linear algebra computations, B-51–B-53
FFT performance, B-53
instruction set, B-49
performance, B-51
rasterization, B-50
ROP, B-50–B-51
scalability, B-51
sorting performance, B-54–B-55
special function approximation statistics, B-43*f*
special function unit (SFU), B-50
streaming multiprocessor (SM), B-48–B-49
streaming processor, B-49–B-50
streaming processor array (SPA), B-46
texture/processor cluster (TPC), B-47
NVIDIA GPU architecture, 515–517
NVIDIA GTX 280, 541*f*, 542*f*
NVIDIA Tesla GPU, 540–545

O

Object files, 128*b*–129*b*
debugging information, 127
header, 126
linking, 128*b*–129*b*
relocation information, 126
static data segment, 126
symbol table, 127
text segment, 126
Object-oriented languages. *See also* Java
brief history, 162.e7
defined, 144.e14, 145
One's complement, 81, A-29
Opcodes
control line setting and, 256
defined, 83, 254
OpenGL, B-13
OpenMP (Open MultiProcessing), 512*b*–513*b*, 532
Operands, 67–74. *See also* Instructions
32-bit immediate, 113–114
adding, 175
arithmetic instructions, 67
compiling assignment when in memory, 69*b*
constant, 72–74
division, 183–191
floating-point, 207*f*
memory, 68–72
multiplication, 177–183
RISC-V, 64*f*
Operating systems
brief history, 473.e8
defined, 13
encapsulation, 22
in networking, 529.e3–529.e5
Operations
atomic, implementing, 122
hardware, 63–67
logical, 89–92
x86 integer, 151–152
Optimization
class explanation, 144.e13*f*
compiler, 141*f*
control implementation, C-27
global, 144.e4–144.e10
high-level, 144.e3–144.e4
local, 144.e4–144.e10
manual, 144
or (inclusive or), 64*f*
OR operation, 176, A-6
ori (inclusive or immediate), 64*f*
Out-of-order execution
defined, 330
performance complexity, 408*b*–409*b*
processors, 334*b*
Output devices, 16–17
Overflow
defined, 75, 192
detection, 176
exceptions, 318*f*
floating-point, 193
occurrence, 175
saturation and, 177*b*
subtraction, 175

P

P + Q redundancy (RAID 6), 458.e6
Packed floating-point format, 218
Page faults, 426. *See also* Virtual memory
for data access, 461
defined, 420–421
handling, 422, 439–441
virtual address causing, 432–435
Page tables, 445
defined, 424–425
illustrated, 427*f*
indexing, 424–425
inverted, 429
levels, 429
main memory, 429
register, 424–425
storage reduction techniques, 429
updating, 424
VMM, 441*b*

Pages. *See also* Virtual memory
defined, 420–421
dirty, 430*b*
finding, 424–425
LRU, 426–428
offset, 421
physical number, 421
placing, 424–425
size, 422*f*
virtual number, 421
Parallel bus, 529.e1–529.e2
Parallel execution, 121
Parallel memory system, B-36–B-41. *See also* Graphics processing units (GPUs)
caches, B-38
constant memory, B-40
DRAM considerations, B-37–B-38
global memory, B-39
load/store access, B-41
local memory, B-40
memory spaces, B-39
MMU, B-38–B-39
ROP, B-41
shared memory, B-39–B-40
surfaces, B-41
texture memory, B-40
Parallel processing programs, 494–499
creation difficulty, 494–499
defined, 492
for message passing, 511*b*–513*b*
great debates in, 553.e4–553.e6
for shared address space, 511*b*–513*b*
use of, 549
Parallel reduction, B-62
Parallel scan, B-60–B-63
CUDA template, B-61*f*
inclusive, B-60
tree-based, B-62*f*
Parallel software, 493
Parallelism, 12, 43*b*, 321–334
and computers arithmetic, 216–217
data-level, 226, 500
debates, 553.e4–553.e6
GPUs and, 514, B-76
instruction-level, 43, 321–322, 333
memory hierarchies and, 458.e1–458.e2, 454–458
multicore and, 509*b*
multiple issue, 322*b*
multithreading and, 507
performance benefits, 44
process-level, 492
redundant arrays and inexpensive disks, 458
subword, D-17
task, B-24
task-level, 492
thread, B-22
Paravirtualization, 472
PA-RISC, D-14, D-17
branch vectored, D-35
conditional branches, D-34, D-35*f*
debug instructions, D-36
decimal operations, D-35
extract and deposit, D-35
instructions, D-34–D-36
load and clear instructions, D-36
multiply/add and multiply/subtract, D-36
nullification, D-34
nullifying branch option, D-25
store bytes short, D-36
synthesized multiply and divide, D-34–D-35
Parity, 458.e4
bits, 412–413
code, 420, A-64–A-65
PARSEC (Princeton Application Repository for Shared Memory Computers), 532
Pass transistor, A-62*b*–A-64*b*
PCI-Express (PCIe), 529.e1–529.e2, B-7–B-8, 529
PC-relative addressing, 115–116, 118
Peak floating-point performance, 534
Pentium bug morality play, 224*f*
Performance, 28–40
assessing, 28
classic CPU equation, 36–40
components, 38*f*
CPU, 33–35
defining, 29–32
equation, using, 36–40
improving, 34*b*–35*b*
instruction, 35–36
measuring, 32–33, 54.e9
program, 9–10
ratio, 31
relative, 31*b*
response time, 30*b*
sorting, B-49–B-50
throughput, 30*b*
time measurement, 32
Personal computers (PCs), 7*f*
defined, 5
Personal mobile device (PMD)
defined, 6–7
Petabyte, 6*f*
Physical addresses, 420
mapping to, 420–421
space, 509, 511*b*–513*b*
Physically addressed caches, 436–437
Pipeline registers
before forwarding, 298–300
dependences, 297–298, 298*f*
forwarding unit selection, 302
Pipeline stalls, 270
avoiding with code reordering, 270*b*–271*b*
data hazards and, 303–307
insertion, 305*f*
load-use, 308
as solution to control hazards, 272*f*
Pipelined branches, 310*b*
Pipelined control, 290–294. *See also* Control
control lines, 290–291
overview illustration, 306*f*
specifying, 291
Pipelined datapaths, 276–294
with connected control signals, 294*f*
with control signals, 290–294
corrected, 286*f*
illustrated, 279*f*
in load instruction stages, 286*f*
Pipelined dependencies, 296*f*
Pipelines
branch instruction impact, 308*f*
effectiveness, improving, 347.e3–347.e4
execute and address calculation stage, 280, 282
five-stage, 264, 280, 288*b*–290*b*
graphic representation, 269*f*, 286–290
instruction decode and register file read stage, 278*f*, 282
instruction fetch stage, 279*f*, 282
instructions sequence, 304*f*
latency, 276*b*
memory access stage, 280, 282
multiple-clock-cycle diagrams, 286–287
performance bottlenecks, 332–333
single-clock-cycle diagrams, 286–287
stages, 264–265
static two-issue, 325*f*
write-back stage, 279, 284

Pipelining, 12, 262–276
advanced, 333–334
benefits, 262
control hazards, 271–274
data hazards, 268–271
exceptions and, 317–321
execution time and, 276*b*
fallacies, 345–346
hazards, 267–271
instruction set design for, 267
laundry analogy, 263*f*
overview, 262–276
paradox, 263–264
performance improvement, 267
pitfall, 345–346
simultaneous executing instructions, 276*b*
speed-up formula, 265
structural hazards, 267–268, 284
summary, 314–315
throughput and, 276*b*
Pitfalls. *See also* Fallacies
address space extension, 384–385
arithmetic, 222–225
associativity, 469*b*
defined, 49
GPUs, B-74
ignoring memory system behavior, 468*b*
memory hierarchies, 468–472
out-of-order processor evaluation, 469*b*
performance equation subset, 50*b*
pipelining, 345–346
pointer to automatic variables, 159*b*
sequential word addresses, 159*b*
simulating cache, 468
software development with multiprocessors, 548*b*
VMM implementation, 470–472
Pixel shader example, B-15–B-17
Pixels, 18
Pointers
arrays *versus*, 141–144
frame, 104–105
global, 104*b*
incrementing, 143
Java, 144.e25–144.e26
stack, 99, 102–104
Polling, 529.e6
Pop, 99
Power
clock rate and, 40
critical nature of, 53
efficiency, 333–334
relative, 41*b*–42*b*
PowerPC
algebraic right shift, D-33
branch registers, D-32–D-33
condition codes, D-12–D-13
instructions, D-12–D-13
instructions unique to, D-32–D-34
load multiple/store multiple, D-33
logical shifted immediate, D-33
rotate with mask, D-33
Precise interrupts, 321*b*
Prediction, 12
2-bit scheme, 312
accuracy, 312
dynamic branch, 310–314
loops and, 312*b*
steady-state, 312
Prefetching, 472, 536
Primitive types, 144.e25
Procedure calls
preservation across, 104
Procedures, 98–108
compiling, 100*b*–101*b*
compiling, showing nested procedure linking, 100*b*–101*b*
execution steps, 98
frames, 104
leaf, 102
nested, 102*b*–103*b*
recursive, 107*b*
for setting arrays to zero, 141*f*
sort, 135–140
strcpy, 110*b*–111*b*
string copy, 110*b*–111*b*
swap, 134–135
Process identifiers, 438
Process-level parallelism, 492
Processors, 234
control, 19
as cores, 43
datapath, 19
defined, 17*b*, 19
dynamic multiple-issue, 322
multiple-issue, 322
out-of-order execution, 334*b*, 408*b*–409*b*
performance growth, 44*f*
ROP, B-12, B-41
speculation, 323–324
static multiple-issue, 322, 324–328
streaming, B-34
superscalar, 328, 347.e4, 507–508
technologies for building, 24–28
two-issue, 325–327
vector, 499–500
VLIW, 324
Product, 178
Product of sums, A-11
Program counters (PCs), 243
changing with conditional branch, 313*b*–314*b*
defined, 99, 243
exception, 437, 439
incrementing, 243, 245*f*
instruction updates, 279
Program performance
elements affecting, 39*t*
understanding, 9
Programmable array logic (PAL), A-77
Programmable logic arrays (PLAs)
component dots illustration, A-16*f*
control function implementation, C-7*f*, C-20
defined, A-12–A-13
example, A-13*b*–A-14*b*
illustrated, A-13*f*
ROMs and, A-15–A-16
size, C-20
truth table implementation, A-13
Programmable logic devices (PLDs), A-77
Programmable ROMs (PROMs), A-14
Programming languages. *See also* specific languages
brief history of, 162.e6–162.e7
object-oriented, 145
variables, 67
Programs
assembly language, 125
Java, starting, 132–133
parallel processing, 492
starting, 124–133
translating, 124–133
Propagate
defined, A-39
example, A-44*b*–A-45*b*
super, A-40
Protected keywords, 144.e20
Protection
defined, 420
implementing, 437–439

mechanisms, 473.e12
VMs for, 416–417
Protection group, 458.e4
Pseudoinstructions
defined, 125
summary, 126
Pthreads (POSIX threads), 532
PTX instructions, B-31, B-32*f*
Public keywords, 144.e20
Push
defined, 99
using, 102–104

Q

Quad words, 151
Quicksort, 403*b*–405*b*, 404*f*
Quotient, 184

R

Race, A-72–A-73
Radix sort, 403*b*–405*b*, 404*f*, B-63–B-65
CUDA code, B-64*f*
implementation, B-63–B-65
RAID. *See* Redundant arrays of inexpensive disks (RAID)
RAM, 9
Raster operation (ROP) processors, B-12, B-41, B-50–B-51
fixed function, B-41
Raster refresh buffer, 18
Rasterization, B-50
Ray casting (RC), 544
Read-only memories (ROMs), A-14–A-16
control entries, C-16*b*–C-18*b*
control function encoding, C-19
dispatch, C-25*f*
implementation, C-15–C-19
logic function encoding, A-15
overhead, C-18
PLAs and, A-15–A-16, A-16
programmable (PROM), A-14
total size, C-15–C-16
Read-stall cycles, 391
Read-write head, 373
Receive message routine, 521
Recursive procedures, 107*b*. *See also* Procedures
clone invocation, 102
Reduced instruction set computer (RISC) architectures, 162.e4, 347.e3, D-3–D-5, D-5–D-9, D-9–D-16, D-16–D-18, D-19, D-20–D-25, D-25–D-27, D-27–D-29, D-29–D-32, D-32–D-34, D-34–D-36, D-36–D-38, D-38–D-39, D-39–D-40, D-40, D-40–D-43, D-43–D-45. *See also* Desktop and server RISCs; Embedded RISCs
group types, D-3–D-4
instruction set lineage, D-44*f*
Reduction, 511
Redundant arrays of inexpensive disks (RAID), 458.e1–458.e2
history, 458.e6–458.e7
RAID 0, 458.e3
RAID 1, 458.e4
RAID 2, 458.e4
RAID 3, 458.e4
RAID 4, 458.e4–458.e5
RAID 5, 458.e5–458.e6
RAID 6, 458.e6
spread of, 458.e5
summary, 458.e6–458.e7
use statistics, 458.e6*f*
Reference bit, 428*b*
References
absolute, 127
types, 144.e25
Register addressing, 118*f*
Register allocation, 144.e10–144.e12
Register files, A-49*b*, A-53–A-55
in behavioral Verilog, A-56
defined, 244, A-49*b*, A-53
single, 249
two read ports implementation, A-54*f*
with two read ports/one write port, A-54*f*
write port implementation, A-55*f*
Register-memory architecture, 162.e2
Registers, 148–151
architectural, 316, 337–338
base, 69
clock cycle time and, 67
compiling C assignment with, 67*b*–68*b*
defined, 67
destination, 254
floating-point, 212*b*
left half, 280
number specification, 244
page table, 424–425
pipeline, 297–298, 298*f*, 302
primitives, 67
renaming, 327
right half, 280
RISC-V conventions, 255*f*
spilling, 71
Status, 316
temporary, 68, 100
variables, 68
Relative performance, 31*b*
Relative power, 41*b*–42*b*
Reliability, 410–411
Remainder, defined, 184
Reorder buffers, 334*b*
Replication, 456
Requested word first, 384
Request-level parallelism, 524
Reservation stations
buffering operands in, 329
defined, 329
Response time, 30*b*
Restartable instructions, 440–441
Return address, 99
R-format
ALU operations, 245*f*
Ripple carry
adder, A-29
carry lookahead speed *versus*, A-45*b*
RISC-V, 62, 85–87
architecture, 190*f*
arithmetic instructions, 63
arithmetic/logical instructions not in, D-21*f*, D-23*f*
assembly instruction, mapping, 81*b*–82*b*
common extensions to, D-20–D-25
compiling C assignment statements into, 65*b*
compiling complex C assignment into, 66*b*
control instructions not in, D-21*f*
control registers, 439*b*
control unit, C-10
data transfer instructions not in, D-20*f*, D-22*f*
divide in, 189
exceptions in, 316–317
fields, 83–89
floating-point instructions, 206–212
floating-point instructions not in, D-22*f*

RISC-V (*Continued*)
instruction classes, 157*f*
instruction encoding, 85*f*, 119*f*
instruction formats, 120, 146*f*
instruction set, 62, 159–160, 226, 236, D-9–D-16
machine language, 87–89
memory addresses, 70*f*
memory allocation for program and data, 106*f*
multiply in, 183
Pseudo, 226*f*
register conventions, 107*f*
static multiple issue with, 324–328
Roofline model, 534–535, 536*f*, 537
with ceilings, 538*f*
computational roofline, 535, 537
illustrated, 534*f*
Opteron generations, 535
with overlapping areas shaded, 539*f*
peak floating-point performance, 538*f*
peak memory performance, 542*f*
with two kernels, 539*f*
Rotational delay. *See* Rotational latency
Rotational latency, 375
Rounding, 212–213
accurate, 212–213
bits, 214
with guard digits, 213*b*
IEEE 754 modes, 213–214
Row-major order, 211*b*–212*b*, 405
R-type, defined, 87*b*
R-type instructions, 248*b*–249*b*
datapath for, 256–259
datapath in operation for, 258*f*
RV32, 73*b*
RV64, 73*b*

S

Saturation, 177*b*
sb (store byte), 64*f*
SB-type instruction format, 115
sc.d (store conditional), 64*f*
SCALAPAK, 223–224
Scaling
strong, 497
weak, 497
Scientific notation
adding numbers in, 199
defined, 191
for reals, 191
sd (store doubleword), 64*f*
Search engines, 4
Secondary memory, 23
Sectors, 373–374
Seek, 374
Segmentation, 423*b*
Selector values, A-10
Semiconductors, 25–26
Send message routine, 521
Sensitivity list, A-23–A-24
Sequencers
explicit, C-32
implementing next-state function with, C-22–C-28
Sequential logic, A-4
Servers, 458.e6. *See also* Desktop and server RISCs
cost and capability, 5
Service accomplishment, 410–411
Service interruption, 410
Set-associative caches, 395. *See also* Caches
address portions, 399*f*
block replacement strategies, 445
choice of, 444
four-way, 396*f*, 399
memory-block location, 395*f*
misses, 397*b*–398*b*
n-way, 395
two-way, 396*f*
Set less than instruction (slt), A-31
Setup time, A-52–A-53, A-53*f*
sh (store halfword), 64*f*
Shaders
defined, B-14
floating-point arithmetic, B-14
graphics, B-14–B-15
pixel example, B-15–B-17
Shading languages, B-14
Shadowing, 458.e4
Shared memory. *See also* Memory
as low-latency memory, B-21
caching in, B-58–B-60
CUDA, B-58
N-body and, B-66*f*
per-CTA, B-39
SRAM banks, B-40
Shared memory multiprocessors (SMP), 509–514
defined, 492–493, 509–510
single physical address space, 509
synchronization, 510–513
Shift left logical immediate (slli), 90
Shift right arithmetic (srai), 90
Shift right logical immediate (srli), 90
Sign and magnitude, 192
Sign bit, 77
Sign extension, 246
defined, 78*b*
shortcut, 78–79
Signals
asserted, 242, A-4
control, 242, 255
deasserted, 242, A-4
Signed division, 187–188
Signed multiplication, 182
Signed numbers, 74–81
sign and magnitude, 75
treating as unsigned, 96
Significands, 193–194
addition, 198–199
multiplication, 201–205
Silicon, 25–26
as key hardware technology, 53
crystal ingot, 26
defined, 25–26
wafers, 26
Silicon crystal ingot, 26
SIMD (Single Instruction Multiple Data), 498–499, 550–551
computers, 553.e1–553.e3
data vector, B-35
extensions, 553.e3
for loops and, 553.e2
massively parallel multiprocessors, 553.e1
small-scale, 553.e3
vector architecture, 500–502
in x86, 500
SIMMs (single inline memory modules), 473.e4, 473.e5*f*
Simple programmable logic devices (SPLDs), A-77
Simplicity, 65–67
Simultaneous multithreading (SMT), 507
support, 507*f*
thread-level parallelism, 507
unused issue slots, 507*f*
Single error correcting/Double error correcting (SEC/DEC), 412–416
Single instruction single data (SISD), 500, 504–506
Single precision. *See also* Double precision
binary representation, 196*b*
defined, 193

Single-clock-cycle pipeline diagrams, 287–290
illustrated, 289*f*
Single-cycle datapaths. *See also* Datapaths
illustrated, 277*f*
instruction execution, 278*f*
Single-cycle implementation
control function for, 261
nonpipelined execution *versus* pipelined execution, 266*f*
non-use of, 261–262
penalty, 262
pipelined performance *versus*, 264*b*–265*b*
Single-instruction multiple-thread (SIMT), B-27–B-29
overhead, B-35
multithreaded warp scheduling, B-28*f*
processor architecture, B-28–B-29
warp execution and divergence, B-29–B-30
Single-program multiple data (SPMD), B-22
sll (shift left logical), 64*f*
slli (shift left logical immediate), 64*f*
Smalltalk-80, 162.e7
Smart phones, 7
Snooping protocol, 456–458
Snoopy cache coherence, 459.e16
Software optimization
via blocking, 405–409
Software
layers, 13*f*
multiprocessor, 492
parallel, 493
as service, 7, 524, 550
systems, 13
Sort algorithms, 141*f*
Sort procedure, 135–140. *See also* Procedures
code for body, 136–138
full procedure, 139–140
passing parameters in, 138
preserving registers in, 138–139
procedure call, 138
register allocation for, 136
Sorting performance, B-54–B-55
Space allocation
on heap, 105–108
on stack, 104–105
SPARC
annulling branch, D-23–D-25
CASA, D-31–D-32
conditional branches, D-10–D-16
fast traps, D-30
floating-point operations, D-31
instructions, D-29–D-32
least significant bits, D-31*f*
multiple precision floating-point results, D-32
nonfaulting loads, D-32
overlapping integer operations, D-31
quadruple precision floating-point arithmetic, D-36
register windows, D-29–D-30
support for LISP and Smalltalk, D-30
Sparse matrices, B-55–B-58
Sparse Matrix-Vector multiply (SpMV), B-55, B-57*f*, B-58
CUDA version, B-57*f*
serial code, B-57*f*
shared memory version, B-59*f*
Spatial locality, 366
large block exploitation of, 383
tendency, 369
SPEC, 54.e10–54.e11
CPU benchmark, 46–48
power benchmark, 48–49
SPEC89, 54.e10
SPEC92, 54.e11
SPEC95, 54.e11
SPEC2000, 54.e11
SPEC2006, 54.e11
SPECrate, 530
SPECratio, 47–48
Special function units (SFUs), B-35, B-50
defined, B-42–B-43
Speculation, 323–324
hardware-based, 331–333
implementation, 323
performance and, 323–324
problems, 323
recovery mechanism, 323
Speed-up challenge
balancing load, 497*b*–498*b*
bigger problem, 496*b*–497*b*
Spilling registers, 71*b*–72*b*, 99
Split algorithm, 544
Split caches, 389*b*
sra (shift right arithmetic), 64*f*
srai (shift right arithmetic immediate), 64*f*
srl (shift right logical), 64*f*
srli (shift right logical immediate), 64*f*
Stack architectures, 162.e3–162.e4
Stack pointers
adjustment, 102–104
defined, 99
values, 101*f*
Stacks
allocating space on, 104–105
for arguments, 99
defined, 99
pop, 99
push, 99, 102–104
Stalls, 270
avoiding with code reordering, 270*b*–271*b*
behavioral Verilog with detection, 345.e3–345.e8
data hazards and, 303–307
illustrations, 345.e20
insertion into pipeline, 305*f*
load-use, 308
memory, 391
as solution to control hazard, 271
write-back scheme, 392
write buffer, 391
Standby spares, 458.e7
State
in 2-bit prediction scheme, 312
assignment, A-69, C-27
bits, C-8–C-10
exception, saving/restoring, 440
logic components, 241
specification of, 424*b*
State elements
clock and, 241
combinational logic and, 241
defined, 240–241, A-47
inputs, 241
register file, A-49*b*
in storing/accessing instructions, 244*f*
Static branch prediction, 324
Static data
segment, 105
Static multiple-issue processors, 322, 324–328. *See also* Multiple issue
control hazards and, 324–325
instruction sets, 324
with RISC-V ISA, 324–328
Static random access memories (SRAMs), 370–371, A-57–A-66
array organization, A-61*f*
basic structure, A-60*f*
defined, 19–22, A-57
fixed access time, A-57
large, A-58
read/write initiation, A-58
synchronous (SSRAMs), A-59
three-state buffers, A-58, A-59*f*

Static variables, 104*b*
Steady-state prediction, 312
Sticky bits, 214
Store buffers, 334*b*
Store byte, 109
Store-conditional doubleword, 122–123
Store doubleword, 70–71
Store instructions. *See also* Load instructions
  access, B-41
  base register, 254
  compiling with, 71
  conditional, 122–123
  defined, 71*b*
  EX stage, 284*f*
  ID stage, 281*f*
  IF stage, 281*f*
  instruction dependency, 302*b*
  MEM stage, 283*f*
  unit for implementing, 247*f*
  WB stage, 283*f*
Store word, 113*b*
Stored program concept, 63
  as computer principle, 88*b*
  illustrated, 88*f*
  principles, 159–160
Strcpy procedure, 110*b*–111*b*. *See also* Procedures
  as leaf procedure, 111
  pointers, 111
Stream benchmark, 540*b*
Streaming multiprocessor (SM), B-13
Streaming processors, B-34, B-49–B-50
  array (SPA), B-41, B-46
Streaming SIMD Extension 2 (SSE2) floating-point architecture, 217
Streaming SIMD Extensions (SSE) and advanced vector extensions in x86, 217
Stretch computer, 347.e1, 347.e1*f*
Strings
  defined, 109–111
  in Java, 111–113
  representation, 108*f*
Strip mining, 502*b*
Striping, 458.e3
Strong scaling, 497
Structural hazards, 267, 284
sub (subtract), 64*f*
Subnormals, 216
Subtraction, 174–177. *See also* Arithmetic
  binary, 174*b*–175*b*
  floating-point, 206
  negative number, 176
  overflow, 176
Subword parallelism, 216–217, 344*f*, D-17
  and matrix multiply, 218–222
Sum of products, A-11, A-12*b*
Supercomputers, 347.e2
  defined, 5
SuperH, D-15, D-39–D-40
Superscalars
  defined, 347.e3–347.e4, 328
  dynamic pipeline scheduling, 328–329
  multithreading options, 494
Supervisor Exception Cause Register (SCAUSE), 316
Supervisor exception program counter (SEPC), 316, 364, 439
  address capture, 319–321
  defined, 317–319
  in restart determination, 316
Supervisor exception return (sret), 437
Supervisor Page Table Base Register (SPTBR), 427*f*
Supervisor Trap Vector (STVEC), 321*b*
Surfaces, B-41
sw (store word), 64*f*
Swap procedure, 134. *See also* Procedures
  body code, 134–135
  full, 135, 139–140
  register allocation, 134
Swap space, 426
Symbol tables, 126
Synchronization, 121–124, 544
  barrier, B-18, B-20, B-34
  defined, 510–513
  lock, 121
  overhead, reducing, 44–45
  unlock, 121
Synchronizers
  from D flip-flop, A-75*f*
  defined, A-75
  failure, A-75–A-76
Synchronous DRAM (SRAM), 371, A-59, A-64
Synchronous SRAM (SSRAM), A-59
Synchronous system, A-47–A-48
Syntax tree, 144.e2
System calls, defined, 364
Systems software, 13
SystemVerilog
  cache controller, 459.e1–459.e4
  cache data and tag modules, 459.e16
  FSM, 459.e6*f*
  simple cache block diagram, 459.e3*f*
  type declarations, 459.e1*f*

T

Tablets, 7*f*
Tags
  defined, 376
  in locating block, 399
  page tables and, 426
  size of, 401*b*–402*b*
Tail call, 107
Task identifiers, 438
Task parallelism, B-24
Task-level parallelism, 492
Tebibyte (TiB), 5
Telsa PTX ISA, B-31
  arithmetic instructions, B-33
  barrier synchronization, B-34
  GPU thread instructions, B-32*f*
  memory access instructions, 208
Temporal locality, 366
  tendency, 369
Temporary registers, 68, 100
Terabyte (TB), 6*f*
  defined, 5
Texture memory, B-40
Texture/processor cluster (TPC), B-47
TFLOPS multiprocessor, 553.e4–553.e5
Thrashing, 442
Thread blocks, 518*f*
  creation, B-23
  defined, B-19
  managing, B-30
  memory sharing, B-20–B-21
  synchronization, B-20–B-21
Thread parallelism, B-22
Threads
  creation, B-23
  CUDA, B-36
  ISA, B-31–B-34
  managing, B-30
  memory latencies and, B-74*b*
  multiple, per body, B-68–B-72
  warps, B-27–B-28
Three Cs model, 447*b*
Three-state buffers, A-58, A-59*f*
Throughput
  defined, 29–30
  multiple issue and, 322
  pipelining and, 264

Thumb, D-15*f*, D-38–D-39
Timing
  asynchronous inputs, A-75–A-76
  level-sensitive, A-74–A-75
  methodologies, A-71–A-77
  two-phase, A-74*f*
TLB misses, 431. *See also* Translation-lookaside buffer (TLB)
  handling, 439–441
  occurrence, 439
  problem, 442
Tomasulo's algorithm, 347.e2
Touchscreen, 19
Tournament branch predicators, 313–314
Tracks, 373–374
Transfer time, 375
Transistors, 25
Translation-lookaside buffer (TLB), 430–432, 473.e5, D-26–D-27. *See also* TLB misses
  associativities, 432
  illustrated, 431*f*
  integration, 435
  Intrinsity FastMATH, 432–435
  typical values, 432
Transmit driver and NIC hardware time *versus* receive driver and NIC hardware time, 529.e7*f*
Tree-based parallel scan, B-62*f*
Truth tables, A-5
  ALU control lines, C-5*f*
  for control bits, 253
  datapath control outputs, C-17*f*
  datapath control signals, C-14*f*
  defined, 253
  example, A-5*b*
  next-state output bits, C-15*f*
  PLA implementation, A-13
Two's complement representation, 76
  advantage, 77
  negation shortcut, 78*b*–79*b*
  rule, 80*b*
  sign extension shortcut, 79*b*–80*b*
Two-level logic, A-11–A-14
Two-phase clocking, A-74, A-74*f*
TX-2 computer, 553.e3

U

Unconditional branches, 93
Underflow, 192
Unicode
  alphabets, 111
  defined, 111
  example alphabets, 112*f*
Unified GPU architecture, B-10–B-11
  illustrated, B-11*f*
  processor array, B-11–B-12
Uniform memory access (UMA), 510, B-9
  multiprocessors, 510
Units
  commit, 329, 334*b*
  control, 239–240, 251–253, C-4–C-8, C-10*f*, C-12–C-13
  defined, 213–214
  floating point, 213–214
  hazard detection, 303, 306–307
  for load/store implementation, 247*f*
  special function (SFUs), B-35, B-42–B-43, B-50
UNIVAC I, 54.e3–54.e4, 54.e4*f*
UNIX, 162.e7, 473.e10, 473.e13, 473.e14
  AT&T, 473.e14
  Berkeley version (BSD), 473.e14
  genius, 473.e16
  history, 473.e13, 473.e14
Unlock synchronization, 121
Unsigned numbers, 74–81
Use latency
  defined, 325–327
  one-instruction, 325–327

V

Vacuum tubes, 25*f*
Valid bit, 376–378
Variables
  C language, 104*b*
  programming language, 67
  register, 67
  static, 104*b*
  storage class, 104*b*
  type, 104*b*
VAX architecture, 162.e3, 473.e6
Vector lanes, 502
Vector processors, 499–506. *See also* Processors
  conventional code comparison, 501*b*–502*b*
  instructions, 501
  multimedia extensions and, 500–502
  scalar *versus*, 502–503
Vectored interrupts, 316
Verilog
  behavioral definition of RISC-V ALU, A-25*f*
  behavioral definition with bypassing, 345.e4*f*
  behavioral definition with stalls for loads, 345.e6*f*
  behavioral specification, 345.e1–345.e3, A-21
  behavioral specification of multicycle MIPS design, 345.e12*f*
  behavioral specification with simulation, 345.e1–345.e3
  behavioral specification with stall detection, 345.e3–345.e8
  behavioral specification with synthesis, 345.e8–345.e13
  blocking assignment, A-24
  branch hazard logic implementation, 345.e8
  combinational logic, A-23–A-26
  datatypes, A-21–A-23
  defined, A-20–A-21
  forwarding implementation, 345.e3
  modules, A-23*f*
  multicycle MIPS datapath, 345.e14*f*
  nonblocking assignment, A-24
  operators, A-22–A-23
  program structure, A-23
  reg, A-21
  RISC-V ALU definition in, A-36–A-37
  sensitivity list, A-23–A-24
  sequential logic specification, A-55–A-57
  structural specification, A-21
  wire, A-21, A-22
Vertical microcode, C-32
Very large-scale integrated (VLSI) circuits, 25
Very Long Instruction Word (VLIW)
  defined, 324
  first generation computers, 347.e4
  processors, 324
VHDL, A-20–A-21
Video graphics array (VGA) controllers, B-3–B-4
Virtual addresses
  causing page faults, 440
  defined, 420–421
  mapping from, 420–421
  size, 422–423

Virtual machine monitors (VMMs)
defined, 416
implementing, 470*b*
laissez-faire attitude, 470
page tables, 441*b*
in performance improvement, 419
requirements, 418
Virtual machines (VMs), 416–419
benefits, 416–417
illusion, 441*b*
instruction set architecture support, 419
performance improvement, 419
for protection improvement, 416–417
Virtual memory, 419–443. *See also* Pages
address translation, 420–421, 430–432
integration, 435–437
for large virtual addresses, 428–429
mechanism, 442
motivations, 419–420
page faults, 420–421, 426
protection implementation, 437–439
segmentation, 423*b*
summary, 441–443
virtualization of, 441*b*
writes, 430
Virtualizable hardware, 418
Virtually addressed caches, 436
Visual computing, B-3
Volatile memory, 22

W

Wafers, 26
defects, 26–27
dies, 27–28
yield, 27
Warehouse Scale Computers (WSCs), 7, 521–526, 550
Warps, B-27–B-28
Weak scaling, 497
Wear levelling, 373
While loops, 94*b*–95*b*
Whirlwind, 473.e1
Wide area networks (WANs), 24. *See also* Networks
Wide immediate operands, 113–114
Words
accessing, 68
defined, 67
double, 151
load, 69, 71
quad, 151
store, 71*b*
Working set, 442
World Wide Web, 4
Worst-case delay, 262
Write buffers
defined, 387
stalls, 383
write-back cache, 387
Write invalidate protocols, 456
Write serialization, 455–456
Write-back caches. *See also* Caches
advantages, 446
cache coherency protocol, 459.e4
complexity, 387
defined, 386, 446
stalls, 391
write buffers, 387
Write-back stage
control line, 292*f*
load instruction, 282
store instruction, 284
Writes
complications, 386*b*–387*b*
expense, 442
handling, 385–387
memory hierarchy handling of, 333–334
schemes, 386
virtual memory, 429
write-back cache, 386–387
write-through cache, 386–387
Write-stall cycles, 391
Write-through caches. *See also* Caches
advantages, 446
defined, 385, 446
tag mismatch, 386

X

x86, 146–155
Advanced Vector Extensions in, 217–218
brief history, 162.e5–162.e6
conclusion, 154–155
data addressing modes, 149–151
evolution, 96
first address specifier encoding, 155*f*
instruction encoding, 153–154
instruction formats, 154*f*
instruction set growth, 162*f*
instruction types, 152*f*
integer operations, 151–152
registers, 149–151
SIMD in, 498–499
Streaming SIMD Extensions in, 217–218
typical instructions/functions, 154*f*
typical operations, 153*f*
unique, D-36–D-38
Xerox Alto computer, 54.e7–54.e9
XMM, 217
xor (exclusive or), 64*f*
xori (exclusive or immediate), 64*f*

Y

Yahoo! Cloud Serving Benchmark (YCSB), 532
Yield, 27
YMM, 218

Z

Zettabyte, 6*f*

B

APPENDIX

Graphics and Computing GPUs

John Nickolls
Director of Architecture
NVIDIA

David Kirk
Chief Scientist
NVIDIA

Imagination is more important than knowledge.

Albert Einstein

On Science, 1930s

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

B.1 Introduction

This appendix focuses on the **GPU**—the ubiquitous **graphics processing unit** in every PC, laptop, desktop computer, and workstation. In its most basic form, the GPU generates 2D and 3D graphics, images, and video that enable Window-based operating systems, graphical user interfaces, video games, visual imaging applications, and video. The modern GPU that we describe here is a highly parallel, highly multithreaded multiprocessor optimized for **visual computing**. To provide real-time visual interaction with computed objects via graphics, images, and video, the GPU has a unified graphics and computing architecture that serves as both a programmable graphics processor and a scalable parallel computing platform. PCs and game consoles combine a GPU with a CPU to form **heterogeneous systems**.

graphics processing unit (GPU) A processor optimized for 2D and 3D graphics, video, visual computing, and display.

visual computing A mix of graphics processing and computing that lets you visually interact with computed objects via graphics, images, and video.

heterogeneous system A system combining different processor types. A PC is a heterogeneous CPU–GPU system.

A Brief History of GPU Evolution

Fifteen years ago, there was no such thing as a GPU. Graphics on a PC were performed by a *video graphics array* (VGA) controller. A VGA controller was simply a memory controller and display generator connected to some DRAM. In the 1990s, semiconductor technology advanced sufficiently that more functions could be added to the VGA controller. By 1997, VGA controllers were beginning to incorporate some *three-dimensional* (3D) acceleration functions, including

hardware for triangle setup and rasterization (dicing triangles into individual pixels) and texture mapping and shading (applying "decals" or patterns to pixels and blending colors).

In 2000, the single chip graphics processor incorporated almost every detail of the traditional high-end workstation graphics pipeline and, therefore, deserved a new name beyond VGA controller. The term GPU was coined to denote that the graphics device had become a processor.

Over time, GPUs became more programmable, as programmable processors replaced fixed-function dedicated logic while maintaining the basic 3D graphics pipeline organization. In addition, computations became more precise over time, progressing from indexed arithmetic, to integer and fixed point, to single-precision floating-point, and recently to double-precision floating-point. GPUs have become massively parallel programmable processors with hundreds of cores and thousands of threads.

Recently, processor instructions and memory hardware were added to support general purpose programming languages, and a programming environment was created to allow GPUs to be programmed using familiar languages, including C and C++. This innovation makes a GPU a fully general-purpose, programmable, manycore processor, albeit still with some special benefits and limitations.

GPU Graphics Trends

GPUs and their associated drivers implement the OpenGL and DirectX models of graphics processing. OpenGL is an open standard for 3D graphics programming available for most computers. DirectX is a series of Microsoft multimedia programming interfaces, including Direct3D for 3D graphics. Since these **application programming interfaces (APIs)** have well-defined behavior, it is possible to build effective hardware acceleration of the graphics processing functions defined by the APIs. This is one of the reasons (in addition to increasing device density) why new GPUs are being developed every 12 to 18 months that double the performance of the previous generation on existing applications.

application programming interface (API) A set of function and data structure definitions providing an interface to a library of functions.

Frequent doubling of GPU performance enables new applications that were not previously possible. The intersection of graphics processing and parallel computing invites a new paradigm for graphics, known as visual computing. It replaces large sections of the traditional sequential hardware graphics pipeline model with programmable elements for geometry, vertex, and pixel programs. Visual computing in a modern GPU combines graphics processing and parallel computing in novel ways that permit new graphics algorithms to be implemented, and opens the door to entirely new parallel processing applications on pervasive high-performance GPUs.

Heterogeneous System

Although the GPU is arguably the most parallel and most powerful processor in a typical PC, it is certainly not the only processor. The CPU, now multicore and

soon to be manycore, is a complementary, primarily serial processor companion to the massively parallel manycore GPU. Together, these two types of processors comprise a heterogeneous multiprocessor system.

The best performance for many applications comes from using both the CPU and the GPU. This appendix will help you understand how and when to best split the work between these two increasingly parallel processors.

GPU Evolves into Scalable Parallel Processor

GPUs have evolved functionally from hardwired, limited capability VGA controllers to programmable parallel processors. This evolution has proceeded by changing the logical (API-based) graphics pipeline to incorporate programmable elements and also by making the underlying hardware pipeline stages less specialized and more programmable. Eventually, it made sense to merge disparate programmable pipeline elements into one unified array of many programmable processors.

In the GeForce 8-series generation of GPUs, the geometry, vertex, and pixel processing all run on the same type of processor. This unification allows for dramatic scalability. More programmable processor cores increase the total system throughput. Unifying the processors also delivers very effective load balancing, since any processing function can use the whole processor array. At the other end of the spectrum, a processor array can now be built with very few processors, since all of the functions can be run on the same processors.

Why CUDA and GPU Computing?

This uniform and scalable array of processors invites a new model of programming for the GPU. The large amount of floating-point processing power in the GPU processor array is very attractive for solving nongraphics problems. Given the large degree of parallelism and the range of scalability of the processor array for graphics applications, the programming model for more general computing must express the massive parallelism directly, but allow for scalable execution.

GPU computing Using a GPU for computing via a parallel programming language and API.

GPU computing is the term coined for using the GPU for computing via a parallel programming language and API, without using the traditional graphics API and graphics pipeline model. This is in contrast to the earlier **General Purpose computation on GPU (GPGPU)** approach, which involves programming the GPU using a graphics API and graphics pipeline to perform nongraphics tasks.

GPGPU Using a GPU for general-purpose computation via a traditional graphics API and graphics pipeline.

Compute Unifed Device Architecture (CUDA) is a scalable parallel programming model and software platform for the GPU and other parallel processors that allows the programmer to bypass the graphics API and graphics interfaces of the GPU and simply program in C or C++. The CUDA programming model has an SPMD (single-program multiple data) software style, in which a programmer writes a program for one thread that is instanced and executed by many threads in parallel on the multiple processors of the GPU. In fact, CUDA also provides a facility for programming multiple CPU cores as well, so CUDA is an environment for writing parallel programs for the entire heterogeneous computer system.

CUDA A scalable parallel programming model and language based on C/C++. It is a parallel programming platform for GPUs and multicore CPUs.

GPU Unifes Graphics and Computing

With the addition of CUDA and GPU computing to the capabilities of the GPU, it is now possible to use the GPU as both a graphics processor and a computing processor at the same time, and to combine these uses in visual computing applications. The underlying processor architecture of the GPU is exposed in two ways: first, as implementing the programmable graphics APIs, and second, as a massively parallel processor array programmable in C/C++ with CUDA.

Although the underlying processors of the GPU are unified, it is not necessary that all of the SPMD thread programs are the same. The GPU can run graphics shader programs for the graphics aspect of the GPU, processing geometry, vertices, and pixels, and also run thread programs in CUDA.

The GPU is truly a versatile multiprocessor architecture, supporting a variety of processing tasks. GPUs are excellent at graphics and visual computing as they were specifically designed for these applications. GPUs are also excellent at many general-purpose throughput applications that are "first cousins" of graphics, in that they perform a lot of parallel work, as well as having a lot of regular problem structure. In general, they are a good match to data-parallel problems (see Chapter 6), particularly large problems, but less so for less regular, smaller problems.

GPU Visual Computing Applications

Visual computing includes the traditional types of graphics applications plus many new applications. The original purview of a GPU was "anything with pixels," but it now includes many problems without pixels but with regular computation and/or data structure. GPUs are effective at 2D and 3D graphics, since that is the purpose for which they are designed. Failure to deliver this application performance would be fatal. 2D and 3D graphics use the GPU in its "graphics mode," accessing the processing power of the GPU through the graphics APIs, OpenGL™, and DirectX™. Games are built on the 3D graphics processing capability.

Beyond 2D and 3D graphics, image processing and video are important applications for GPUs. These can be implemented using the graphics APIs or as computational programs, using CUDA to program the GPU in computing mode. Using CUDA, image processing is simply another data-parallel array program. To the extent that the data access is regular and there is good locality, the program will be efficient. In practice, image processing is a very good application for GPUs. Video processing, especially encode and decode (compression and decompression according to some standard algorithms), is quite efficient.

The greatest opportunity for visual computing applications on GPUs is to "break the graphics pipeline." Early GPUs implemented only specific graphics APIs, albeit at very high performance. This was wonderful if the API supported the operations that you wanted to do. If not, the GPU could not accelerate your task, because early GPU functionality was immutable. Now, with the advent of GPU computing and CUDA, these GPUs can be programmed to implement a different virtual pipeline by simply writing a CUDA program to describe the computation and data flow that is desired. So, all applications are now possible, which will stimulate new visual computing approaches.

B.2 GPU System Architectures

In this section, we survey GPU system architectures in common use today. We discuss system configurations, GPU functions and services, standard programming interfaces, and a basic GPU internal architecture.

Heterogeneous CPU–GPU System Architecture

A heterogeneous computer system architecture using a GPU and a CPU can be described at a high level by two primary characteristics: first, how many functional subsystems and/or chips are used and what are their interconnection technologies and topology; and second, what memory subsystems are available to these functional subsystems. See Chapter 6 for background on the PC I/O systems and chip sets.

The Historical PC (circa 1990)

Figure B.2.1 shows a high-level block diagram of a legacy PC, circa 1990. The north bridge (see Chapter 6) contains high-bandwidth interfaces, connecting the CPU, memory, and PCI bus. The south bridge contains legacy interfaces and devices: ISA bus (audio, LAN), interrupt controller; DMA controller; time/counter. In this system, the display was driven by a simple framebuffer subsystem known

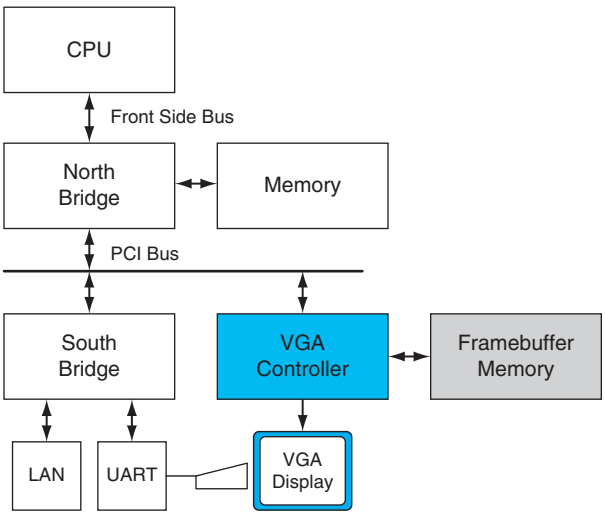


FIGURE B.2.1 Historical PC. VGA controller drives graphics display from framebuffer memory.

PCI-Express (PCIe) A standard system I/O interconnect that uses point-to-point links. Links have a configurable number of lanes and bandwidth.

as a VGA (*video graphics array*) which was attached to the PCI bus. Graphics subsystems with built-in processing elements (GPUs) did not exist in the PC landscape of 1990.

Figure B.2.2 illustrates two confgurations in common use today. These are characterized by a separate GPU (discrete GPU) and CPU with respective memory subsystems. In Figure B.2.2a, with an Intel CPU, we see the GPU attached via a 16-lane **PCI-Express** 2.0 link to provide a peak 16 GB/s transfer rate (peak of 8 GB/s in each direction). Similarly, in Figure B.2.2b, with an AMD CPU, the GPU

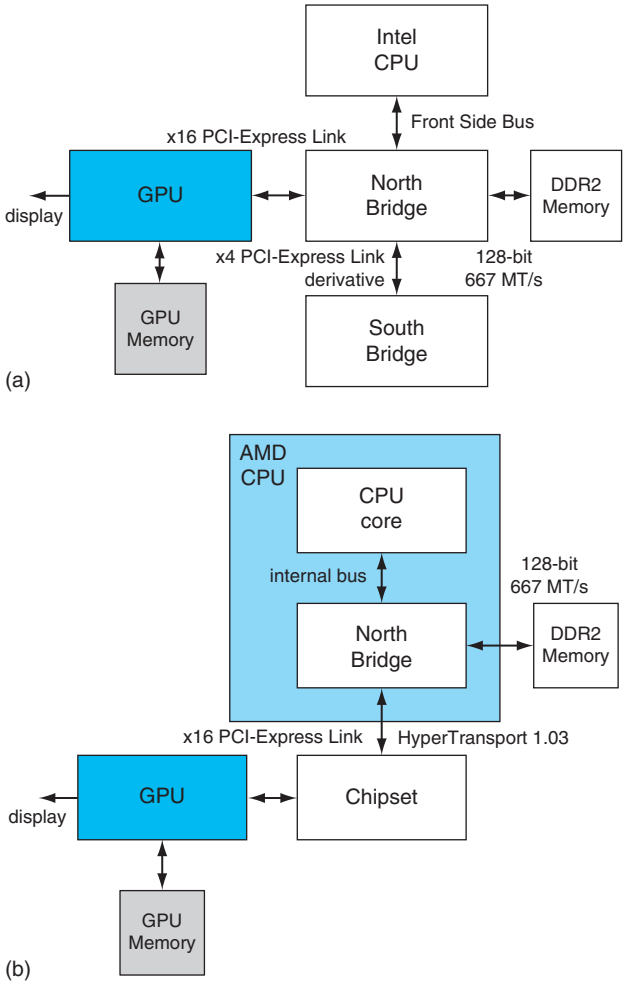


FIGURE B.2.2 Contemporary PCs with Intel and AMD CPUs. See Chapter 6 for an explanation of the components and interconnects in this figure.

is attached to the chipset, also via PCI-Express with the same available bandwidth. In both cases, the GPUs and CPUs may access each other's memory, albeit with less available bandwidth than their access to the more directly attached memories. In the case of the AMD system, the north bridge or memory controller is integrated into the same die as the CPU.

A low-cost variation on these systems, a **unified memory architecture (UMA)** system, uses only CPU system memory, omitting GPU memory from the system. These systems have relatively low-performance GPUs, since their achieved performance is limited by the available system memory bandwidth and increased latency of memory access, whereas dedicated GPU memory provides high bandwidth and low latency.

unified memory architecture (UMA) A system architecture in which the CPU and GPU share a common system memory.

A high-performance system variation uses multiple attached GPUs, typically two to four working in parallel, with their displays daisy-chained. An example is the NVIDIA SLI (scalable link interconnect) multi-GPU system, designed for high-performance gaming and workstations.

The next system category integrates the GPU with the north bridge (Intel) or chipset (AMD) with and without dedicated graphics memory.

Chapter 5 explains how caches maintain coherence in a shared address space. With CPUs and GPUs, there are multiple address spaces. GPUs can access their own physical local memory and the CPU system's physical memory using virtual addresses that are translated by an MMU on the GPU. The operating system kernel manages the GPU's page tables. A system physical page can be accessed using either coherent or noncoherent PCI-Express transactions, determined by an attribute in the GPU's page table. The CPU can access GPU's local memory through an address range (also called aperture) in the PCI-Express address space.

Game Consoles

Console systems such as the Sony PlayStation 3 and the Microsoft Xbox 360 resemble the PC system architectures previously described. Console systems are designed to be shipped with identical performance and functionality over a lifespan that can last five years or more. During this time, a system may be reimplemented many times to exploit more advanced silicon manufacturing processes and thereby to provide constant capability at ever lower costs. Console systems do not need to have their subsystems expanded and upgraded the way PC systems do, so the major internal system buses tend to be customized rather than standardized.

GPU Interfaces and Drivers

In a PC today, GPUs are attached to a CPU via PCI-Express. Earlier generations used **AGP**. Graphics applications call OpenGL [Segal and Akeley, 2006] or Direct3D [Microsoft DirectX Specifcation] API functions that use the GPU as a coprocessor. The APIs send commands, programs, and data to the GPU via a graphics device driver optimized for the particular GPU.

AGP An extended version of the original PCI I/O bus, which provided up to eight times the bandwidth of the original PCI bus to a single card slot. Its primary purpose was to connect graphics subsystems into PC systems.

Graphics Logical Pipeline

The graphics logical pipeline is described in Section B.3. Figure B.2.3 illustrates the major processing stages, and highlights the important programmable stages (vertex, geometry, and pixel shader stages).

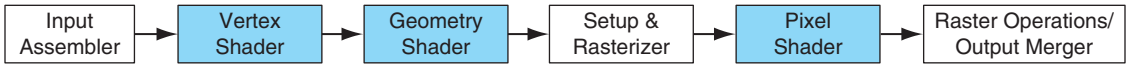


FIGURE B.2.3 Graphics logical pipeline. Programmable graphics shader stages are blue, and fixed-function blocks are white.

Mapping Graphics Pipeline to Unified GPU Processors

Figure B.2.4 shows how the logical pipeline comprising separate independent programmable stages is mapped onto a physical distributed array of processors.

Basic Unified GPU Architecture

Unified GPU architectures are based on a parallel array of many programmable processors. They unify vertex, geometry, and pixel shader processing and parallel computing on the same processors, unlike earlier GPUs which had separate processors dedicated to each processing type. The programmable processor array is tightly integrated with fixed function processors for texture filtering, rasterization, raster operations, anti-aliasing, compression, decompression, display, video decoding, and high-definition video processing. Although the fixed-function processors significantly outperform more general programmable processors in terms of absolute performance constrained by an area, cost, or power budget, we will focus on the programmable processors here.

Compared with multicore CPUs, manycore GPUs have a different architectural design point, one focused on executing many parallel threads efficiently on many

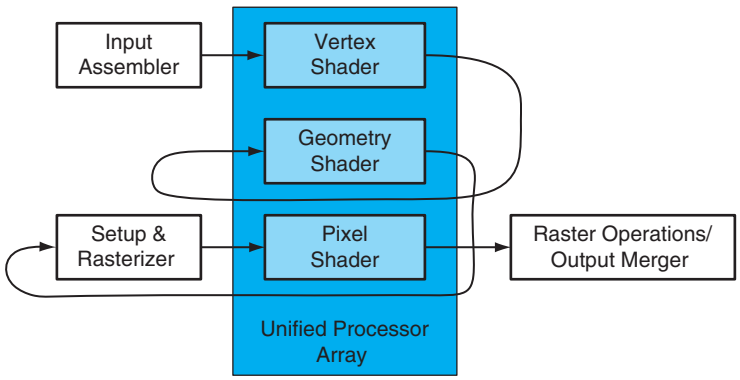


FIGURE B.2.4 Logical pipeline mapped to physical processors. The programmable shader stages execute on the array of unified processors, and the logical graphics pipeline dataflow recirculates through the processors.

processor cores. By using many simpler cores and optimizing for data-parallel behavior among groups of threads, more of the per-chip transistor budget is devoted to computation, and less to on-chip caches and overhead.

Processor Array

A unified GPU processor array contains many processor cores, typically organized into multithreaded multiprocessors. Figure B.2.5 shows a GPU with an array of 112 *streaming processor* (SP) cores, organized as 14 multithreaded *streaming multiprocessors* (SMs). Each SP core is highly multithreaded, managing 96 concurrent threads and their state in hardware. The processors connect with four 64-bit-wide DRAM partitions via an interconnection network. Each SM has eight SP cores, two *special function units* (SFUs), instruction and constant caches, a multithreaded instruction unit, and a shared memory. This is the basic Tesla architecture implemented by the NVIDIA GeForce 8800. It has a unified architecture in which the traditional graphics programs for vertex, geometry, and pixel shading run on the unified SMs and their SP cores, and computing programs run on the same processors.

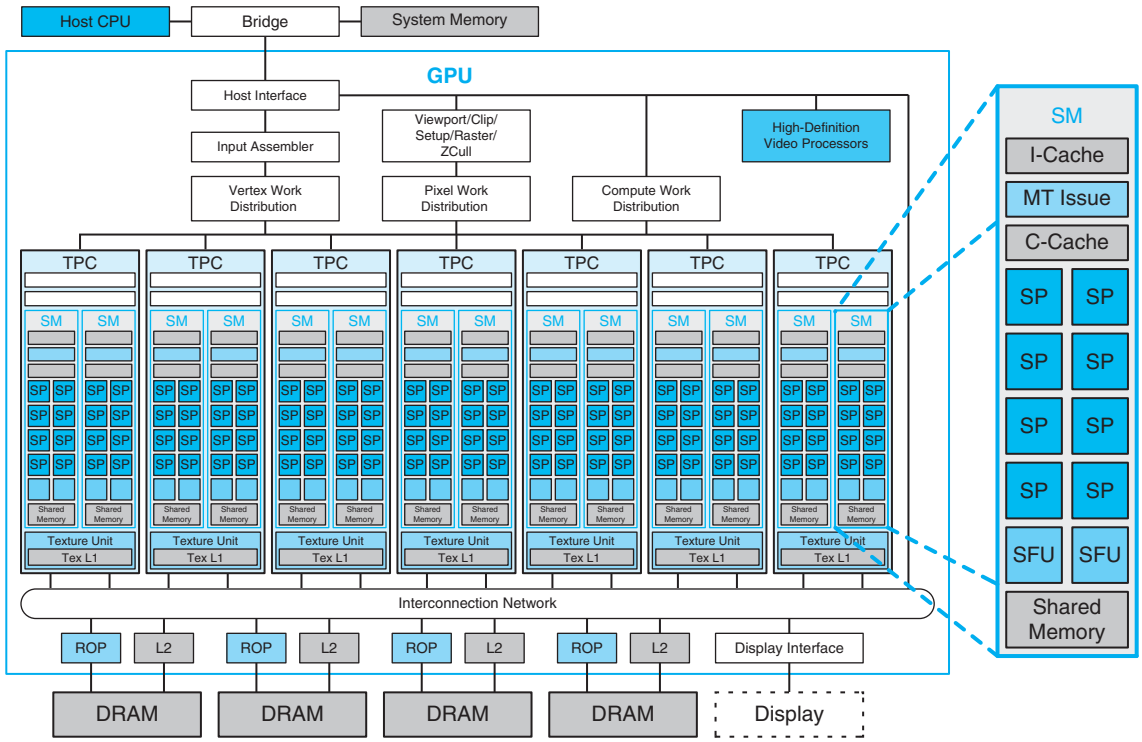


FIGURE B.2.5 Basic unified GPU architecture. Example GPU with 112 *streaming processor* (SP) cores organized in 14 *streaming multiprocessors* (SMs); the cores are highly multithreaded. It has the basic Tesla architecture of an NVIDIA GeForce 8800. The processors connect with four 64-bit-wide DRAM partitions via an interconnection network. Each SM has eight SP cores, two *special function units* (SFUs), instruction and constant caches, a multithreaded instruction unit, and a shared memory.

The processor array architecture is scalable to smaller and larger GPU configurations by scaling the number of multiprocessors and the number of memory partitions. Figure B.2.5 shows seven clusters of two SMs sharing a texture unit and a texture L1 cache. The texture unit delivers filtered results to the SM given a set of coordinates into a texture map. Because filter regions of support often overlap for successive texture requests, a small streaming L1 texture cache is effective to reduce the number of requests to the memory system. The processor array connects with *raster operation processors* (ROPs), L2 texture caches, external DRAM memories, and system memory via a GPU-wide interconnection network. The number of processors and number of memories can scale to design balanced GPU systems for different performance and market segments.

B.3 Programming GPUs

Programming multiprocessor GPUs is qualitatively different than programming other multiprocessors like multicore CPUs. GPUs provide two to three orders of magnitude more thread and data parallelism than CPUs, scaling to hundreds of processor cores and tens of thousands of concurrent threads. GPUs continue to increase their parallelism, doubling it about every 12 to 18 months, enabled by Moore's law [1965] of increasing integrated circuit density and by improving architectural efficiency. To span the wide price and performance range of different market segments, different GPU products implement widely varying numbers of processors and threads. Yet users expect games, graphics, imaging, and computing applications to work on any GPU, regardless of how many parallel threads it executes or how many parallel processor cores it has, and they expect more expensive GPUs (with more threads and cores) to run applications faster. As a result, GPU programming models and application programs are designed to scale transparently to a wide range of parallelism.

The driving force behind the large number of parallel threads and cores in a GPU is real-time graphics performance—the need to render complex 3D scenes with high resolution at interactive frame rates, at least 60 frames per second. Correspondingly, the scalable programming models of graphics shading languages such as Cg (C for graphics) and HLSL (*high-level shading language*) are designed to exploit large degrees of parallelism via many independent parallel threads and to scale to any number of processor cores. The CUDA scalable parallel programming model similarly enables general parallel computing applications to leverage large numbers of parallel threads and scale to any number of parallel processor cores, transparently to the application.

In these scalable programming models, the programmer writes code for a single thread, and the GPU runs myriad thread instances in parallel. Programs thus scale transparently over a wide range of hardware parallelism. This simple paradigm arose from graphics APIs and shading languages that describe how to shade one

vertex or one pixel. It has remained an effective paradigm as GPUs have rapidly increased their parallelism and performance since the late 1990s.

This section briefly describes programming GPUs for real-time graphics applications using graphics APIs and programming languages. It then describes programming GPUs for visual computing and general parallel computing applications using the C language and the CUDA programming model.

Programming Real-Time Graphics

APIs have played an important role in the rapid, successful development of GPUs and processors. There are two primary standard graphics APIs: **OpenGL** and **Direct3D**, one of the Microsoft DirectX multimedia programming interfaces. OpenGL, an open standard, was originally proposed and defined by Silicon Graphics Incorporated. The ongoing development and extension of the OpenGL standard [Segal and Akeley, 2006; Kessenich, 2006] is managed by Khronos, an industry consortium. Direct3D [Blythe, 2006], a de facto standard, is defined and evolved forward by Microsoft and partners. OpenGL and Direct3D are similarly structured, and continue to evolve rapidly with GPU hardware advances. They define a logical graphics processing pipeline that is mapped onto the GPU hardware and processors, along with programming models and languages for the programmable pipeline stages.

OpenGL An open-standard graphics API.

Direct3D A graphics API defined by Microsoft and partners.

Logical Graphics Pipeline

Figure B.3.1 illustrates the Direct3D 10 logical graphics pipeline. OpenGL has a similar graphics pipeline structure. The API and logical pipeline provide a streaming dataflow infrastructure and plumbing for the programmable shader stages, shown in blue. The 3D application sends the GPU a sequence of vertices grouped into geometric primitives—points, lines, triangles, and polygons. The input assembler collects vertices and primitives. The vertex shader program executes per-vertex processing,

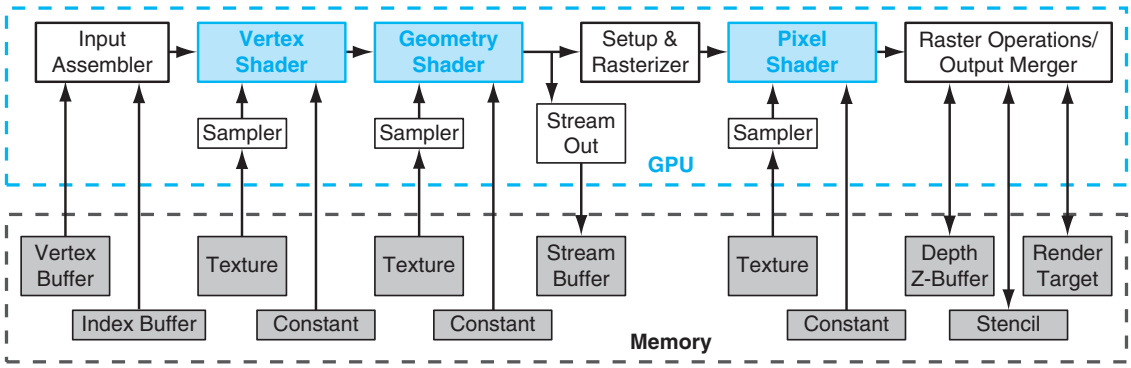


FIGURE B.3.1 Direct3D 10 graphics pipeline. Each logical pipeline stage maps to GPU hardware or to a GPU processor. Programmable shader stages are blue, fixed-function blocks are white, and memory objects are gray. Each stage processes a vertex, geometric primitive, or pixel in a streaming dataflow fashion.

including transforming the vertex 3D position into a screen position and lighting the vertex to determine its color. The geometry shader program executes per-primitive processing and can add or drop primitives. The setup and rasterizer unit generates pixel fragments (fragments are potential contributions to pixels) that are covered by a geometric primitive. The pixel shader program performs per-fragment processing, including interpolating per-fragment parameters, texturing, and coloring. Pixel shaders make extensive use of sampled and filtered lookups into large 1D, 2D, or 3D arrays called **textures**, using interpolated floating-point coordinates. Shaders use texture accesses for maps, functions, decals, images, and data. The raster operations processing (or output merger) stage performs Z-buffer depth testing and stencil testing, which may discard a hidden pixel fragment or replace the pixel's depth with the fragment's depth, and performs a color blending operation that combines the fragment color with the pixel color and writes the pixel with the blended color.

texture A 1D, 2D, or 3D array that supports sampled and filtered lookups with interpolated coordinates.

The graphics API and graphics pipeline provide input, output, memory objects, and infrastructure for the shader programs that process each vertex, primitive, and pixel fragment.

Graphics Shader Programs

Real-time graphics applications use many different **shader** programs to model how light interacts with different materials and to render complex lighting and shadows. **Shading languages** are based on a dataflow or streaming programming model that corresponds with the logical graphics pipeline. Vertex shader programs map the position of triangle vertices onto the screen, altering their position, color, or orientation. Typically a vertex shader thread inputs a floating-point (x, y, z, w) vertex position and computes a floating-point (x, y, z) screen position. Geometry shader programs operate on geometric primitives (such as lines and triangles) defined by multiple vertices, changing them or generating additional primitives. Pixel fragment shaders each "shade" one pixel, computing a floating-point *red, green, blue, alpha* (RGBA) color contribution to the rendered image at its pixel sample (x, y) image position. Shaders (and GPUs) use floating-point arithmetic for all pixel color calculations to eliminate visible artifacts while computing the extreme range of pixel contribution values encountered while rendering scenes with complex lighting, shadows, and high dynamic range. For all three types of graphics shaders, many program instances can be run in parallel, as independent parallel threads, because each works on independent data, produces independent results, and has no side effects. Independent vertices, primitives, and pixels further enable the same graphics program to run on differently sized GPUs that process different numbers of vertices, primitives, and pixels in parallel. Graphics programs thus scale transparently to GPUs with different amounts of parallelism and performance.

shader A program that operates on graphics data such as a vertex or a pixel fragment.

shading language A graphics rendering language, usually having a dataflow or streaming programming model.

Users program all three logical graphics threads with a common targeted high-level language. HLSL (high-level shading language) and Cg (C for graphics) are commonly used. They have C-like syntax and a rich set of library functions for matrix operations, trigonometry, interpolation, and texture access and filtering, but are far from general computing languages: they currently lack general memory

access, pointers, file I/O, and recursion. HLSL and Cg assume that programs live within a logical graphics pipeline, and thus I/O is implicit. For example, a pixel fragment shader may expect the geometric normal and multiple texture coordinates to have been interpolated from vertex values by upstream fixed-function stages and can simply assign a value to the COLOR output parameter to pass it downstream to be blended with a pixel at an implied (x, y) position.

The GPU hardware creates a new independent thread to execute a vertex, geometry, or pixel shader program for every vertex, every primitive, and every pixel fragment. In video games, the bulk of threads execute pixel shader programs, as there are typically 10 to 20 times more pixel fragments than vertices, and complex lighting and shadows require even larger ratios of pixel to vertex shader threads. The graphics shader programming model drove the GPU architecture to efficiently execute thousands of independent fine-grained threads on many parallel processor cores.

Pixel Shader Example

Consider the following Cg pixel shader program that implements the "environment mapping" rendering technique. For each pixel thread, this shader is passed five parameters, including 2D floating-point texture image coordinates needed to sample the surface color, and a 3D floating-point vector giving the refection of the view direction off the surface. The other three "uniform" parameters do not vary from one pixel instance (thread) to the next. The shader looks up color in two texture images: a 2D texture access for the surface color, and a 3D texture access into a cube map (six images corresponding to the faces of a cube) to obtain the external world color corresponding to the refection direction. Then the final four-component (red, green, blue, alpha) floating-point color is computed using a weighted average called a "lerp" or linear interpolation function.

```
void refection(
   float2               texCoord       : TEXCOORD0,
   float3               refection_dir  : TEXCOORD1,
   out float4           color          : COLOR,
   uniform float        shiny,
   uniform sampler2D    surfaceMap,
   uniform samplerCUBE  envMap)
{
// Fetch the surface color from a texture
   float4 surfaceColor = tex2D(surfaceMap, texCoord);

// Fetch reflected color by sampling a cube map
   float4 reflectedColor = texCUBE(environmentMap, refection_dir);

// Output is weighted average of the two colors
   color = lerp(surfaceColor, refectedColor, shiny);
}
```

Although this shader program is only three lines long, it activates a lot of GPU hardware. For each texture fetch, the GPU texture subsystem makes multiple memory accesses to sample image colors in the vicinity of the sampling coordinates, and then interpolates the final result with floating-point filtering arithmetic. The multithreaded GPU executes thousands of these lightweight Cg pixel shader threads in parallel, deeply interleaving them to hide texture fetch and memory latency.

Cg focuses the programmer's view to a single vertex or primitive or pixel, which the GPU implements as a single thread; the shader program transparently scales to exploit thread parallelism on the available processors. Being application-specific, Cg provides a rich set of useful data types, library functions, and language constructs to express diverse rendering techniques.

Figure B.3.2 shows skin rendered by a fragment pixel shader. Real skin appears quite different from flesh-color paint because light bounces around a lot before re-emerging. In this complex shader, three separate skin layers, each with unique subsurface scattering behavior, are modeled to give the skin a visual depth and translucency. Scattering can be modeled by a blurring convolution in a fattened "texture" space, with red being blurred more than green, and blue blurred less. The compiled Cg shader executes 1400 instructions to compute the color of one skin pixel.



FIGURE B.3.2 GPU-rendered image. To give the skin visual depth and translucency, the pixel shader program models three separate skin layers, each with unique subsurface scattering behavior. It executes 1400 instructions to render the red, green, blue, and alpha color components of each skin pixel fragment.

As GPUs have evolved superior floating-point performance and very high streaming memory bandwidth for real-time graphics, they have attracted highly parallel applications beyond traditional graphics. At first, access to this power was available only by couching an application as a graphics-rendering algorithm, but this GPGPU approach was often awkward and limiting. More recently, the CUDA programming model has provided a far easier way to exploit the scalable high-performance floating-point and memory bandwidth of GPUs with the C programming language.

Programming Parallel Computing Applications

CUDA, Brook, and CAL are programming interfaces for GPUs that are focused on data parallel computation rather than on graphics. CAL (*Compute Abstraction Layer*) is a low-level assembler language interface for AMD GPUs. Brook is a streaming language adapted for GPUs by Buck et al. [2004]. CUDA, developed by NVIDIA [2007], is an extension to the C and C+ + languages for scalable parallel programming of manycore GPUs and multicore CPUs. The CUDA programming model is described below, adapted from an article by Nickolls et al. [2008].

With the new model the GPU excels in data parallel and throughput computing, executing high-performance computing applications as well as graphics applications.

Data Parallel Problem Decomposition

To map large computing problems effectively to a highly parallel processing architecture, the programmer or compiler decomposes the problem into many small problems that can be solved in parallel. For example, the programmer partitions a large result data array into blocks and further partitions each block into elements, such that the result blocks can be computed independently in parallel, and the elements within each block are computed in parallel. Figure B.3.3 shows a decomposition of a result data array into a 3×2 grid of blocks, where each block is further decomposed into a 5×3 array of elements. The two-level parallel decomposition maps naturally to the GPU architecture: parallel multiprocessors compute result blocks, and parallel threads compute result elements.

The programmer writes a program that computes a sequence of result data grids, partitioning each result grid into coarse-grained result blocks that can be computed independently in parallel. T0he program computes each result block with an array of fine-grained parallel threads, partitioning the work among threads so that each computes one or more result elements.

Scalable Parallel Programming with CUDA

The CUDA scalable parallel programming model extends the C and C++ languages to exploit large degrees of parallelism for general applications on highly parallel multiprocessors, particularly GPUs. Early experience with CUDA shows that *many* sophisticated programs can be readily expressed with a few easily understood abstractions. Since NVIDIA released CUDA in 2007, developers have

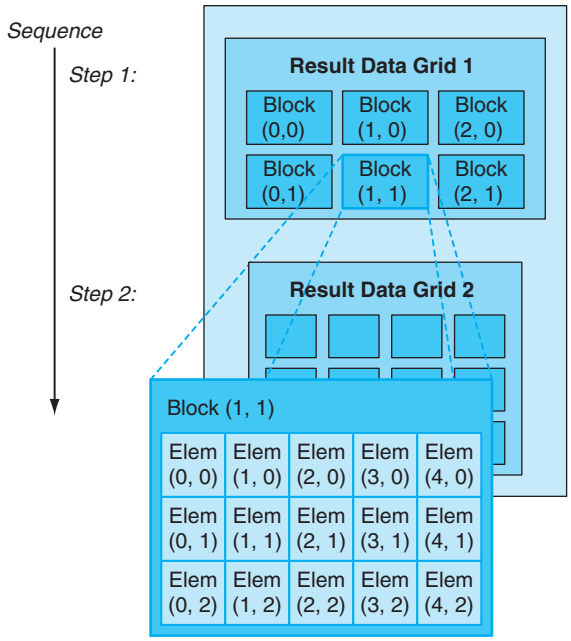


FIGURE B.3.3 Decomposing result data into a grid of blocks of elements to be computed in parallel.

rapidly developed scalable parallel programs for a wide range of applications, including seismic data processing, computational chemistry, linear algebra, sparse matrix solvers, sorting, searching, physics models, and visual computing. These applications scale transparently to hundreds of processor cores and thousands of concurrent threads. NVIDIA GPUs with the Tesla unified graphics and computing architecture (described in Sections B.4 and B.7) run CUDA C programs, and are widely available in laptops, PCs, workstations, and servers. The CUDA model is also applicable to other shared memory parallel processing architectures, including multicore CPUs.

CUDA provides three key abstractions—a *hierarchy of thread groups, shared memories*, and *barrier synchronization*—that provide a clear parallel structure to conventional C code for one thread of the hierarchy. Multiple levels of threads, memory, and synchronization provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. The abstractions guide the programmer to partition the problem into coarse subproblems that can be solved independently in parallel, and then into finer pieces that can be solved in parallel. The programming model scales transparently to large numbers of processor cores: a compiled CUDA program executes on any number of processors, and only the runtime system needs to know the physical processor count.

The CUDA Paradigm

CUDA is a minimal extension of the C and C++ programming languages. The programmer writes a serial program that calls parallel **kernels**, which may be simple functions or full programs. A kernel executes in parallel across a set of parallel threads. The programmer organizes these threads into a hierarchy of thread blocks and grids of thread blocks. A **thread block** is a set of concurrent threads that can cooperate among themselves through barrier synchronization and through shared access to a memory space private to the block. A **grid** is a set of thread blocks that may each be executed independently and thus may execute in parallel.

kernel A program or function for one thread, designed to be executed by many threads.

thread block A set of concurrent threads that execute the same thread program and may cooperate to compute a result.

grid A set of thread blocks that execute the same kernel program.

When invoking a kernel, the programmer specifies the number of threads per block and the number of blocks comprising the grid. Each thread is given a unique *thread ID* number `threadIdx` within its thread block, numbered `0, 1, 2, …, blockDim-1`, and each thread block is given a unique *block ID* number `blockIdx` within its grid. CUDA supports thread blocks containing up to 512 threads. For convenience, thread blocks and grids may have one, two, or three dimensions, accessed via `.x`, `.y`, and `.z` index fields.

As a very simple example of parallel programming, suppose that we are given two vectors x and y of n floating-point numbers each and that we wish to compute the result of $y = ax + y$ for some scalar value a. This is the so-called `SAXPY` kernel defined by the BLAS linear algebra library. Figure B.3.4 shows C code for performing this computation on both a serial processor and in parallel using CUDA.

The `__global__` declaration specifier indicates that the procedure is a kernel entry point. CUDA programs launch parallel kernels with the extended function call syntax:

```
kernel<<<dimGrid, dimBlock>>>(… parameter list …);
```

where `dimGrid` and `dimBlock` are three-element vectors of type `dim3` that specify the dimensions of the grid in blocks and the dimensions of the blocks in threads, respectively. Unspecified dimensions default to one.

In Figure B.3.4, we launch a grid of n threads that assigns one thread to each element of the vectors and puts 256 threads in each block. Each individual thread computes an element index from its thread and block IDs and then performs the desired calculation on the corresponding vector elements. Comparing the serial and parallel versions of this code, we see that they are strikingly similar. This represents a fairly common pattern. The serial code consists of a loop where each iteration is independent of all the others. Such loops can be mechanically transformed into parallel kernels: each loop iteration becomes an independent thread. By assigning a single thread to each output element, we avoid the need for any synchronization among threads when writing results to memory.

The text of a CUDA kernel is simply a C function for one sequential thread. Thus, it is generally straightforward to write and is typically simpler than writing parallel code for vector operations. Parallelism is determined clearly and explicitly by specifying the dimensions of a grid and its thread blocks when launching a kernel.

Computing y = ax + y with a serial loop:

```
void saxpy_serial(int n, float alpha, float *x, float *y)
{
     for(int i = 0; i<n; ++i)
          y[i] = alpha*x[i] + y[i];
}
// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
```

Computing y = ax + y in parallel using CUDA:

```
__global__
void saxpy_parallel(int n, float alpha, float *x, float *y)
{
     int i = blockIdx.x*blockDim.x + threadIdx.x;

     if( i<n ) y[i] = alpha*x[i] + y[i];
}

// Invoke parallel SAXPY kernel (256 threads per block)
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

FIGURE B.3.4 Sequential code (top) in C versus parallel code (bottom) in CUDA for SAXPY (see Chapter 6). CUDA parallel threads replace the C serial loop—each thread computes the same result as one loop iteration. The parallel code computes *n* results with *n* threads organized in blocks of 256 threads.

synchronization barrier Threads wait at a synchronization barrier until all threads in the thread block arrive at the barrier.

Parallel execution and thread management is automatic. All thread creation, scheduling, and termination is handled for the programmer by the underlying system. Indeed, a Tesla architecture GPU performs all thread management directly in hardware. The threads of a block execute concurrently and may synchronize at a **synchronization barrier** by calling the `__syncthreads()` intrinsic. This guarantees that no thread in the block can proceed until all threads in the block have reached the barrier. After passing the barrier, these threads are also guaranteed to see all writes to memory performed by threads in the block before the barrier. Thus, threads in a block may communicate with each other by writing and reading per-block shared memory at a synchronization barrier.

Since threads in a block may share memory and synchronize via barriers, they will reside together on the same physical processor or multiprocessor. The number of thread blocks can, however, greatly exceed the number of processors. The CUDA thread programming model virtualizes the processors and gives the programmer the flexibility to parallelize at whatever granularity is most convenient. Virtualization

into threads and thread blocks allows intuitive problem decompositions, as the number of blocks can be dictated by the size of the data being processed rather than by the number of processors in the system. It also allows the same CUDA program to scale to widely varying numbers of processor cores.

To manage this processing element virtualization and provide scalability, CUDA requires that thread blocks be able to execute independently. It must be possible to execute blocks in any order, in parallel or in series. Different blocks have no means of direct communication, although they may *coordinate* their activities using **atomic memory operations** on the global memory visible to all threads—by atomically incrementing queue pointers, for example. This independence requirement allows thread blocks to be scheduled in any order across any number of cores, making the CUDA model scalable across an arbitrary number of cores as well as across a variety of parallel architectures. It also helps to avoid the possibility of deadlock. An application may execute multiple grids either independently or dependently. Independent grids may execute concurrently, given sufficient hardware resources. Dependent grids execute sequentially, with an implicit interkernel barrier between them, thus guaranteeing that all blocks of the first grid complete before any block of the second, dependent grid begins.

atomic memory operation A memory read, modify, write operation sequence that completes without any intervening access.

Threads may access data from multiple memory spaces during their execution. Each thread has a private **local memory**. CUDA uses local memory for thread-private variables that do not fit in the thread's registers, as well as for stack frames and register spilling. Each thread block has a **shared memory**, visible to all threads of the block, which has the same lifetime as the block. Finally, all threads have access to the same **global memory**. Programs declare variables in shared and global memory with the `__shared__` and `__device__` type qualifers. On a Tesla architecture GPU, these memory spaces correspond to physically separate memories: per-block shared memory is a low-latency on-chip RAM, while global memory resides in the fast DRAM on the graphics board.

global memory Per-application memory shared by all threads.

shared memory Per-block memory shared by all threads of the block.

local memory Per-thread local memory private to the thread.

Shared memory is expected to be a low-latency memory near each processor, much like an L1 cache. It can therefore provide high-performance communication and data sharing among the threads of a thread block. Since it has the same lifetime as its corresponding thread block, kernel code will typically initialize data in shared variables, compute using shared variables, and copy shared memory results to global memory. Thread blocks of sequentially dependent grids communicate via global memory, using it to read input and write results.

Figure B.3.5 shows diagrams of the nested levels of threads, thread blocks, and grids of thread blocks. It further shows the corresponding levels of memory sharing: local, shared, and global memories for per-thread, per-thread-block, and per-application data sharing.

A program manages the global memory space visible to kernels through calls to the CUDA runtime, such as `cudaMalloc()` and `cudaFree()`. Kernels may execute on a physically separate device, as is the case when running kernels on the GPU. Consequently, the application must use `cudaMemcpy()` to copy data between the allocated space and the host system memory.

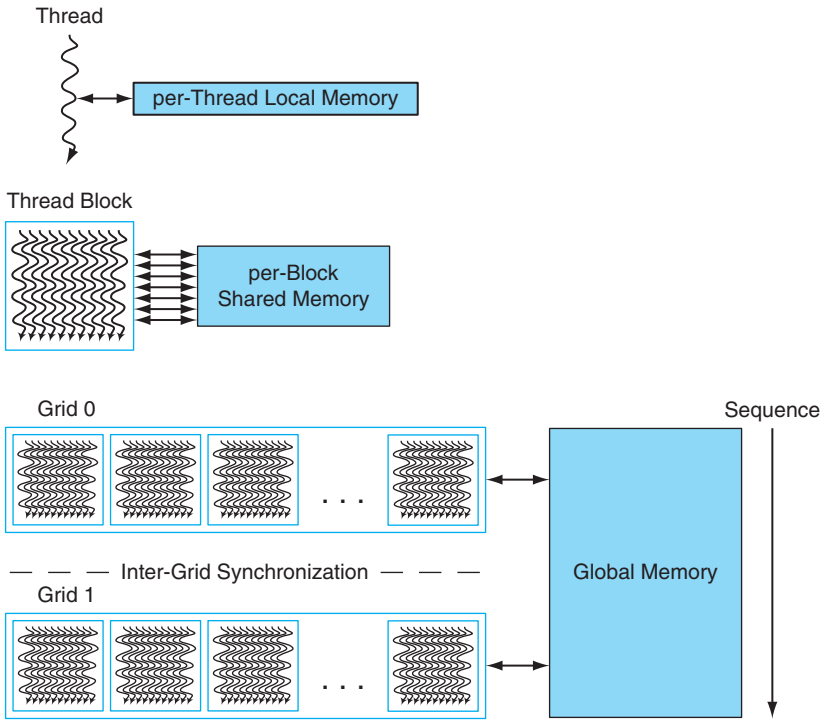


FIGURE B.3.5 Nested granularity levels—thread, thread block, and grid—have corresponding memory sharing levels—local, shared, and global. Per-thread local memory is private to the thread. Per-block shared memory is shared by all threads of the block. Per-application global memory is shared by all threads.

single-program multiple data (SPMD) A style of parallel programming model in which all threads execute the same program. SPMD threads typically coordinate with barrier synchronization.

The CUDA programming model is similar in style to the familiar **single-program multiple data (SPMD)** model—it expresses parallelism explicitly, and each kernel executes on a fixed number of threads. However, CUDA is more flexible than most realizations of SPMD, because each kernel call dynamically creates a new grid with the right number of thread blocks and threads for that application step. The programmer can use a convenient degree of parallelism for each kernel, rather than having to design all phases of the computation to use the same number of threads. Figure B.3.6 shows an example of an SPMD-like CUDA code sequence. It first instantiates `kernelF` on a 2D grid of 3 × 2 blocks where each 2D thread block consists of 5 × 3 threads. It then instantiates `kernelG` on a 1D grid of four 1D thread blocks with six threads each. Because `kernelG` depends on the results of `kernelF`, they are separated by an interkernel synchronization barrier.

The concurrent threads of a thread block express fine-grained data parallelism and thread parallelism. The independent thread blocks of a grid express coarse-grained data parallelism. Independent grids express coarse-grained task parallelism. A kernel is simply C code for one thread of the hierarchy.

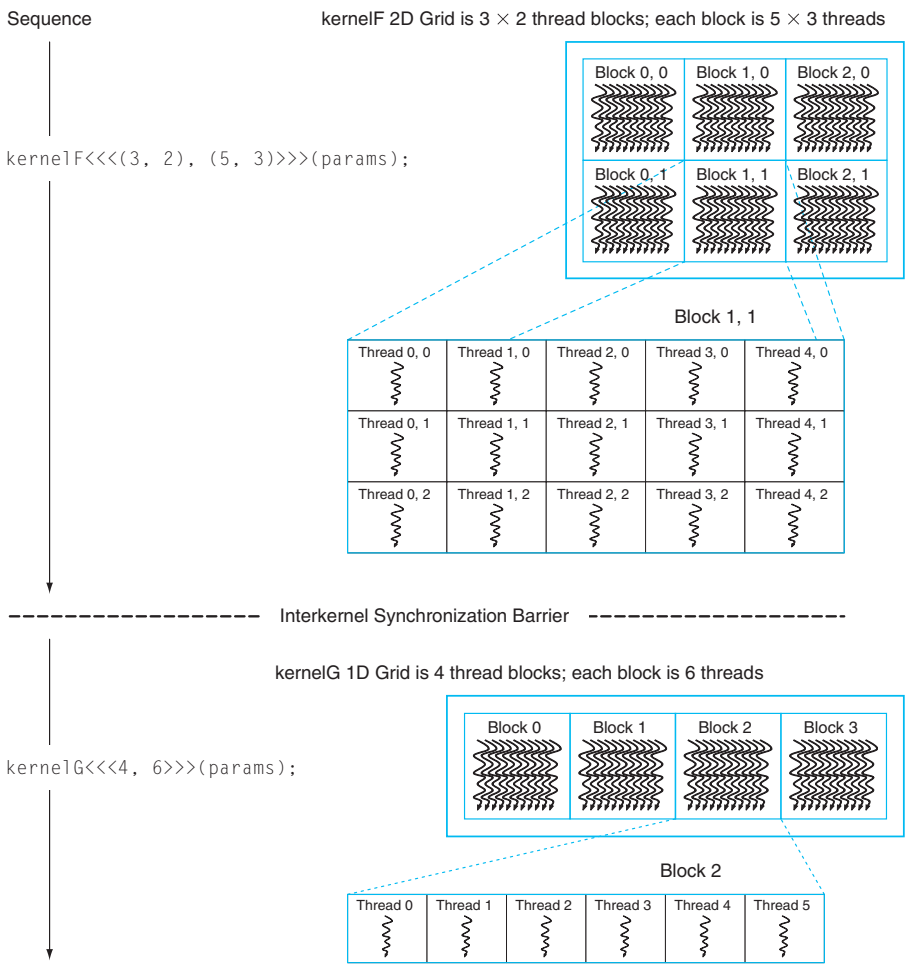


FIGURE B.3.6 Sequence of kernel *F* instantiated on a 2D grid of 2D thread blocks, an interkernel synchronization barrier, followed by kernel *G* on a 1D grid of 1D thread blocks.

Restrictions

For efficiency, and to simplify its implementation, the CUDA programming model has some restrictions. Threads and thread blocks may only be created by invoking a parallel kernel, not from within a parallel kernel. Together with the required independence of thread blocks, this makes it possible to execute CUDA programs with a simple scheduler that introduces minimal runtime overhead. In fact, the Tesla GPU architecture implements *hardware* management and scheduling of threads and thread blocks.

Task parallelism can be expressed at the thread block level but is difficult to express within a thread block because thread synchronization barriers operate on all the threads of the block. To enable CUDA programs to run on any number of processors, dependencies among thread blocks within the same kernel grid are not allowed—blocks must execute independently. Since CUDA requires that thread blocks be independent and allows blocks to be executed in any order, combining results generated by multiple blocks must in general be done by launching a second kernel on a new grid of thread blocks (although thread blocks may *coordinate* their activities using atomic memory operations on the global memory visible to all threads—by atomically incrementing queue pointers, for example).

Recursive function calls are not currently allowed in CUDA kernels. Recursion is unattractive in a massively parallel kernel, because providing stack space for the tens of thousands of threads that may be active would require substantial amounts of memory. Serial algorithms that are normally expressed using recursion, such as quicksort, are typically best implemented using nested data parallelism rather than explicit recursion.

To support a heterogeneous system architecture combining a CPU and a GPU, each with its own memory system, CUDA programs must copy data and results between host memory and device memory. The overhead of CPU–GPU interaction and data transfers is minimized by using DMA block transfer engines and fast interconnects. Compute-intensive problems large enough to need a GPU performance boost amortize the overhead better than small problems.

Implications for Architecture

The parallel programming models for graphics and computing have driven GPU architecture to be different than CPU architecture. The key aspects of GPU programs driving GPU processor architecture are:

- *Extensive use of fine-grained data parallelism:* Shader programs describe how to process a single pixel or vertex, and CUDA programs describe how to compute an individual result.
- *Highly threaded programming model:* A shader thread program processes a single pixel or vertex, and a CUDA thread program may generate a single result. A GPU must create and execute millions of such thread programs per frame, at 60 frames per second.
- *Scalability:* A program must automatically increase its performance when provided with additional processors, without recompiling.
- *Intensive floating-point (or integer) computation.*
- *Support of high-throughput computations.*

B.4 Multithreaded Multiprocessor Architecture

To address different market segments, GPUs implement scalable numbers of multiprocessors—in fact, GPUs are multiprocessors composed of multiprocessors. Furthermore, each multiprocessor is highly multithreaded to execute many fine-grained vertex and pixel shader threads efficiently. A quality basic GPU has two to four multiprocessors, while a gaming enthusiast's GPU or computing platform has dozens of them. This section looks at the architecture of one such multithreaded multiprocessor, a simplified version of the NVIDIA Tesla *streaming multiprocessor* (SM) described in Section B.7.

Why use a multiprocessor, rather than several independent processors? The parallelism within each multiprocessor provides localized high performance and supports extensive multithreading for the fine-grained parallel programming models described in Section B.3. The individual threads of a thread block execute together within a multiprocessor to share data. The multithreaded multiprocessor design we describe here has eight scalar processor cores in a tightly coupled architecture, and executes up to 512 threads (the SM described in Section B.7 executes up to 768 threads). For area and power efficiency, the multiprocessor shares large complex units among the eight processor cores, including the instruction cache, the multithreaded instruction unit, and the shared memory RAM.

Massive Multithreading

GPU processors are highly multithreaded to achieve several goals:

- Cover the latency of memory loads and texture fetches from DRAM
- Support fine-grained parallel graphics shader programming models
- Support fine-grained parallel computing programming models
- Virtualize the physical processors as threads and thread blocks to provide transparent scalability
- Simplify the parallel programming model to writing a serial program for one thread

Memory and texture fetch latency can require hundreds of processor clocks, because GPUs typically have small streaming caches rather than large working-set caches like CPUs. A fetch request generally requires a full DRAM access latency plus interconnect and buffering latency. Multithreading helps cover the latency with useful computing—while one thread is waiting for a load or texture fetch to complete, the processor can execute another thread. The fine-grained parallel programming models provide literally thousands of independent threads that can keep many processors busy despite the long memory latency seen by individual threads.

A graphics vertex or pixel shader program is a program for a single thread that processes a vertex or a pixel. Similarly, a CUDA program is a C program for a single thread that computes a result. Graphics and computing programs instantiate many parallel threads to render complex images and compute large result arrays. To dynamically balance shifting vertex and pixel shader thread workloads, each multiprocessor concurrently executes multiple different thread programs and different types of shader programs.

To support the independent vertex, primitive, and pixel programming model of graphics shading languages and the single-thread programming model of CUDA C/C++, each GPU thread has its own private registers, private per-thread memory, program counter, and thread execution state, and can execute an independent code path. To efficiently execute hundreds of concurrent lightweight threads, the GPU multiprocessor is hardware multithreaded—it manages and executes hundreds of concurrent threads in hardware without scheduling overhead. Concurrent threads within thread blocks can synchronize at a barrier with a single instruction. Lightweight thread creation, zero-overhead thread scheduling, and fast barrier synchronization efficiently support very fine-grained parallelism.

Multiprocessor Architecture

A unified graphics and computing multiprocessor executes vertex, geometry, and pixel fragment shader programs, and parallel computing programs. As Figure B.4.1 shows, the example multiprocessor consists of eight *scalar processor* (SP) cores each with a large multithreaded *register file* (RF), two *special function units* (SFUs), a multithreaded instruction unit, an instruction cache, a read-only constant cache, and a shared memory.

The 16 KB shared memory holds graphics data buffers and shared computing data. CUDA variables declared as `__shared__` reside in the shared memory. To map the logical graphics pipeline workload through the multiprocessor multiple times, as shown in Section B.2, vertex, geometry, and pixel threads have independent input and output buffers, and workloads arrive and depart independently of thread execution.

Each SP core contains scalar integer and floating-point arithmetic units that execute most instructions. The SP is hardware multithreaded, supporting up to 64 threads. Each pipelined SP core executes one scalar instruction per thread per clock, which ranges from 1.2 GHz to 1.6 GHz in different GPU products. Each SP core has a large RF of 1024 general-purpose 32-bit registers, partitioned among its assigned threads. Programs declare their register demand, typically 16 to 64 scalar 32-bit registers per thread. The SP can concurrently run many threads that use a few registers or fewer threads that use more registers. The compiler optimizes register allocation to balance the cost of spilling registers versus the cost of fewer threads. Pixel shader programs often use 16 or fewer registers, enabling each SP to run up to 64 pixel shader threads to cover long-latency texture fetches. Compiled CUDA programs often need 32 registers per thread, limiting each SP to 32 threads, which limits such a kernel program to 256 threads per thread block on this example multiprocessor, rather than its maximum of 512 threads.

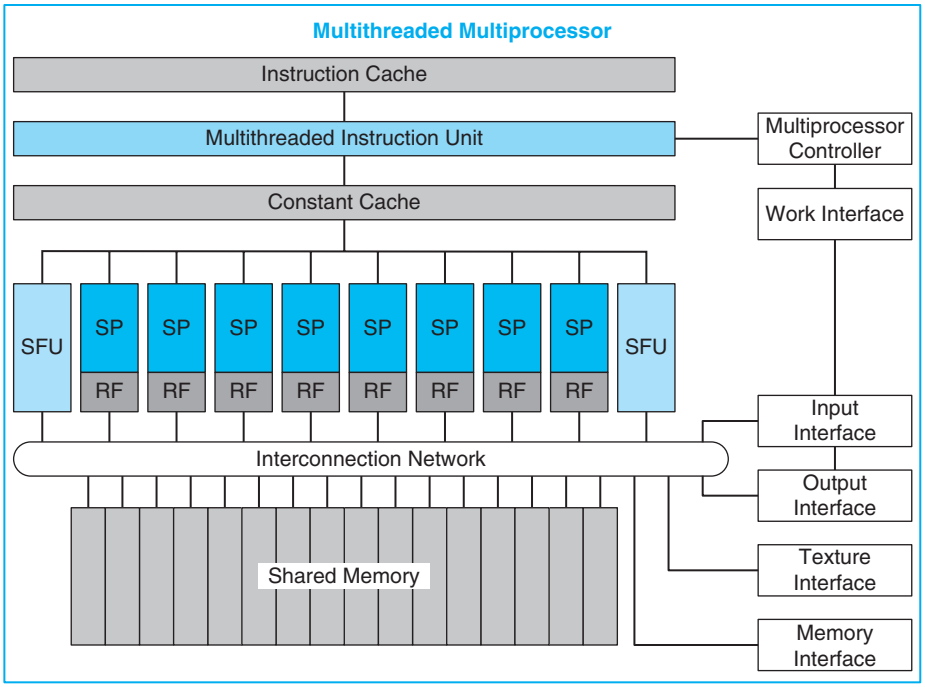


FIGURE B.4.1 Multithreaded multiprocessor with eight scalar processor (SP) cores. The eight SP cores each have a large multithreaded *register file* (RF) and share an instruction cache, multithreaded instruction issue unit, constant cache, two *special function units* (SFUs), interconnection network, and a multibank shared memory.

The pipelined SFUs execute thread instructions that compute special functions and interpolate pixel attributes from primitive vertex attributes. These instructions can execute concurrently with instructions on the SPs. The SFU is described later.

The multiprocessor executes texture fetch instructions on the texture unit via the texture interface, and uses the memory interface for external memory load, store, and atomic access instructions. These instructions can execute concurrently with instructions on the SPs. Shared memory access uses a low-latency interconnection network between the SP processors and the shared memory banks.

Single-Instruction Multiple-Thread (SIMT)

To manage and execute hundreds of threads running several different programs efficiently, the multiprocessor employs a **single-instruction multiple-thread (SIMT)** architecture. It creates, manages, schedules, and executes concurrent threads in groups of parallel threads called *warps*. The term **warp** originates from weaving, the first parallel thread technology. The photograph in Figure B.4.2 shows a warp of parallel threads emerging from a loom. This example multiprocessor uses a SIMT warp size of 32 threads, executing four threads in each of the eight SP cores over four

single-instruction multiple-thread (SIMT) A processor architecture that applies one instruction to multiple independent threads in parallel.

warp The set of parallel threads that execute the same instruction together in a SIMT architecture.

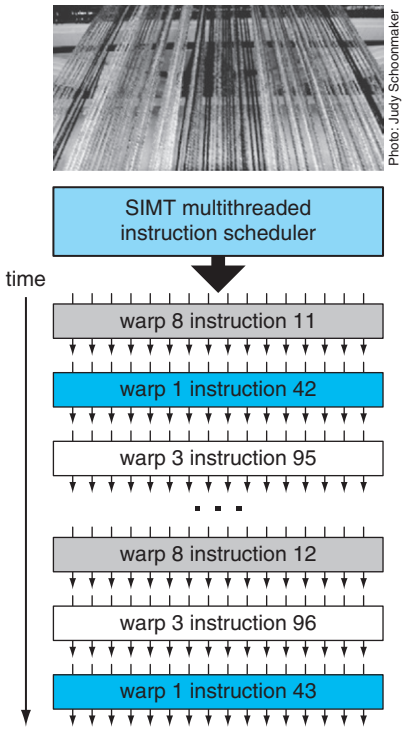


FIGURE B.4.2 SIMT multithreaded warp scheduling. The scheduler selects a ready warp and issues an instruction synchronously to the parallel threads composing the warp. Because warps are independent, the scheduler may select a different warp each time.

clocks. The Tesla SM multiprocessor described in Section B.7 also uses a warp size of 32 parallel threads, executing four threads per SP core for efficiency on plentiful pixel threads and computing threads. Thread blocks consist of one or more warps.

This example SIMT multiprocessor manages a pool of 16 warps, a total of 512 threads. Individual parallel threads composing a warp are the same type and start together at the same program address, but are otherwise free to branch and execute independently. At each instruction issue time, the SIMT multithreaded instruction unit selects a warp that is ready to execute its next instruction, and then issues that instruction to the active threads of that warp. A SIMT instruction is broadcast synchronously to the active parallel threads of a warp; individual threads may be inactive due to independent branching or predication. In this multiprocessor, each SP scalar processor core executes an instruction for four individual threads of a warp using four clocks, reflecting the 4:1 ratio of warp threads to cores.

SIMT processor architecture is akin to *single-instruction multiple data* (SIMD) design, which applies one instruction to multiple data lanes, but differs in that SIMT applies one instruction to multiple independent threads in parallel, not just

to multiple data lanes. An instruction for a SIMD processor controls a vector of multiple data lanes together, whereas an instruction for a SIMT processor controls an individual thread, and the SIMT instruction unit issues an instruction to a warp of independent parallel threads for efficiency. The SIMT processor finds data-level parallelism among threads at runtime, analogous to the way a superscalar processor finds instruction-level parallelism among instructions at runtime.

A SIMT processor realizes full efficiency and performance when all threads of a warp take the same execution path. If threads of a warp diverge via a data-dependent conditional branch, execution serializes for each branch path taken, and when all paths complete, the threads converge to the same execution path. For equal length paths, a divergent if-else code block is 50% efficient. The multiprocessor uses a branch synchronization stack to manage independent threads that diverge and converge. Different warps execute independently at full speed regardless of whether they are executing common or disjoint code paths. As a result, SIMT GPUs are dramatically more efficient and flexible on branching code than earlier GPUs, as their warps are much narrower than the SIMD width of prior GPUs.

In contrast with SIMD vector architectures, SIMT enables programmers to write thread-level parallel code for individual independent threads, as well as data-parallel code for many coordinated threads. For program correctness, the programmer can essentially ignore the SIMT execution attributes of warps; however, substantial performance improvements can be realized by taking care that the code seldom requires threads in a warp to diverge. In practice, this is analogous to the role of cache lines in traditional codes: cache line size can be safely ignored when designing for correctness but must be considered in the code structure when designing for peak performance.

SIMT Warp Execution and Divergence

The SIMT approach of scheduling independent warps is more flexible than the scheduling of previous GPU architectures. A warp comprises parallel threads of the same type: vertex, geometry, pixel, or compute. The basic unit of pixel fragment shader processing is the 2-by-2 pixel quad implemented as four pixel shader threads. The multiprocessor controller packs the pixel quads into a warp. It similarly groups vertices and primitives into warps, and packs computing threads into a warp. A thread block comprises one or more warps. The SIMT design shares the instruction fetch and issue unit efficiently across parallel threads of a warp, but requires a full warp of active threads to get full performance efficiency.

This unified multiprocessor schedules and executes multiple warp types concurrently, allowing it to concurrently execute vertex and pixel warps. Its warp scheduler operates at less than the processor clock rate, because there are four thread lanes per processor core. During each scheduling cycle, it selects a warp to execute a SIMT warp instruction, as shown in Figure B.4.2. An issued warp-instruction executes as four sets of eight threads over four processor cycles of throughput. The processor pipeline uses several clocks of latency to complete each instruction. If the number of active warps times the clocks per warp exceeds the pipeline

latency, the programmer can ignore the pipeline latency. For this multiprocessor, a round-robin schedule of eight warps has a period of 32 cycles between successive instructions for the same warp. If the program can keep 256 threads active per multiprocessor, instruction latencies up to 32 cycles can be hidden from an individual sequential thread. However, with few active warps, the processor pipeline depth becomes visible and may cause processors to stall.

A challenging design problem is implementing zero-overhead warp scheduling for a dynamic mix of different warp programs and program types. The instruction scheduler must select a warp every four clocks to issue one instruction per clock per thread, equivalent to an IPC of 1.0 per processor core. Because warps are independent, the only dependences are among sequential instructions from the same warp. The scheduler uses a register dependency scoreboard to qualify warps whose active threads are ready to execute an instruction. It prioritizes all such ready warps and selects the highest priority one for issue. Prioritization must consider warp type, instruction type, and the desire to be fair to all active warps.

Managing Threads and Thread Blocks

The multiprocessor controller and instruction unit manage threads and thread blocks. The controller accepts work requests and input data and arbitrates access to shared resources, including the texture unit, memory access path, and I/O paths. For graphics workloads, it creates and manages three types of graphics threads concurrently: vertex, geometry, and pixel. Each of the graphics work types has independent input and output paths. It accumulates and packs each of these input work types into SIMT warps of parallel threads executing the same thread program. It allocates a free warp, allocates registers for the warp threads, and starts warp execution in the multiprocessor. Every program declares its per-thread register demand; the controller starts a warp only when it can allocate the requested register count for the warp threads. When all the threads of the warp exit, the controller unpacks the results and frees the warp registers and resources.

cooperative thread array (CTA) A set of concurrent threads that executes the same thread program and may cooperate to compute a result. A GPU CTA implements a CUDA thread block.

The controller creates **cooperative thread arrays (CTAs)** which implement CUDA thread blocks as one or more warps of parallel threads. It creates a CTA when it can create all CTA warps and allocate all CTA resources. In addition to threads and registers, a CTA requires allocating shared memory and barriers. The program declares the required capacities, and the controller waits until it can allocate those amounts before launching the CTA. Then it creates CTA warps at the warp scheduling rate, so that a CTA program starts executing immediately at full multiprocessor performance. The controller monitors when all threads of a CTA have exited, and frees the CTA shared resources and its warp resources.

Thread Instructions

The SP thread processors execute scalar instructions for individual threads, unlike earlier GPU vector instruction architectures, which executed four-component vector instructions for each vertex or pixel shader program. Vertex programs

generally compute (x, y, z, w) position vectors, while pixel shader programs compute (red, green, blue, alpha) color vectors. However, shader programs are becoming longer and more scalar, and it is increasingly difficult to fully occupy even two components of a legacy GPU four-component vector architecture. In effect, the SIMT architecture parallelizes across 32 independent pixel threads, rather than parallelizing the four vector components within a pixel. CUDA C/C++ programs have predominantly scalar code per thread. Previous GPUs employed vector packing (e.g., combining subvectors of work to gain efficiency) but that complicated the scheduling hardware as well as the compiler. Scalar instructions are simpler and compiler-friendly. Texture instructions remain vector-based, taking a source coordinate vector and returning a filtered color vector.

To support multiple GPUs with different binary microinstruction formats, high-level graphics and computing language compilers generate intermediate assembler-level instructions (e.g., Direct3D vector instructions or PTX scalar instructions), which are then optimized and translated to binary GPU microinstructions. The NVIDIA PTX (parallel thread execution) instruction set definition [2007] provides a stable target ISA for compilers, and provides compatibility over several generations of GPUs with evolving binary microinstruction-set architectures. The optimizer readily expands Direct3D vector instructions to multiple scalar binary microinstructions. PTX scalar instructions translate nearly one to one with scalar binary microinstructions, although some PTX instructions expand to multiple binary microinstructions, and multiple PTX instructions may fold into one binary microinstruction. Because the intermediate assembler-level instructions use virtual registers, the optimizer analyzes data dependencies and allocates real registers. The optimizer eliminates dead code, folds instructions together when feasible, and optimizes SIMT branch diverge and converge points.

Instruction Set Architecture (ISA)

The thread ISA described here is a simplified version of the Tesla architecture PTX ISA, a register-based scalar instruction set comprising floating-point, integer, logical, conversion, special functions, flow control, memory access, and texture operations. Figure B.4.3 lists the basic PTX GPU thread instructions; see the NVIDIA PTX specification [2007] for details. The instruction format is:

```
opcode.type d, a, b, c;
```

where `d` is the destination operand, `a`, `b`, `c` are source operands, and `.type` is one of:

Type	.type Specifer
Untyped bits 8, 16, 32, and 64 bits	.b8, .b16, .b32, .b64
Unsigned integer 8, 16, 32, and 64 bits	.u8, .u16, .u32, .u64
Signed integer 8, 16, 32, and 64 bits	.s8, .s16, .s32, .s64
Floating-point 16, 32, and 64 bits	.f16, .f32, .f64

Basic PTX GPU Thread Instructions

Group	Instruction	Example	Meaning	Comments
Arithmetic	arithmetic *.type* = .s32, .u32, .f32, .s64, .u64, .f64			
	add.*type*	add.f32 d, a, b	d = a + b;	
	sub.*type*	sub.f32 d, a, b	d = a - b;	
	mul.*type*	mul.f32 d, a, b	d = a * b;	
	mad.*type*	mad.f32 d, a, b, c	d = a * b + c;	multiply-add
	div.*type*	div.f32 d, a, b	d = a / b;	multiple microinstructions
	rem.*type*	rem.u32 d, a, b	d = a % b;	integer remainder
	abs.*type*	abs.f32 d, a	d = |a|;	
	neg.*type*	neg.f32 d, a	d = 0 - a;	
	min.*type*	min.f32 d, a, b	d = (a < b)? a:b;	floating selects non-NaN
	max.*type*	max.f32 d, a, b	d = (a > b)? a:b;	floating selects non-NaN
	setp.*cmp.type*	setp.lt.f32 p, a, b	p = (a < b);	compare and set predicate
	numeric *.cmp* = eq, ne, lt, le, gt, ge; unordered *cmp* = equ, neu, ltu, leu, gtu, geu, num, nan			
	mov.*type*	mov.b32 d, a	d = a;	move
	selp.*type*	selp.f32 d, a, b, p	d = p? a: b;	select with predicate
	cvt.dtype.atype	cvt.f32.s32 d, a	d = convert(a);	convert atype to dtype
Special Function	special *.type* = .f32 (some .f64)			
	rcp.*type*	rcp.f32 d, a	d = 1/a;	reciprocal
	sqrt.*type*	sqrt.f32 d, a	d = sqrt(a);	square root
	rsqrt.*type*	rsqrt.f32 d, a	d = 1/sqrt(a);	reciprocal square root
	sin.*type*	sin.f32 d, a	d = sin(a);	sine
	cos.*type*	cos.f32 d, a	d = cos(a);	cosine
	lg2.*type*	lg2.f32 d, a	d = log(a)/log(2)	binary logarithm
	ex2.*type*	ex2.f32 d, a	d = 2 ** a;	binary exponential
Logical	logic. *type* = .pred, .b32, .b64			
	and.*type*	and.b32 d, a, b	d = a & b;	
	or.*type*	or.b32 d, a, b	d = a | b;	
	xor.*type*	xor.b32 d, a, b	d = a ^ b;	
	not.*type*	not.b32 d, a, b	d = ~a;	one's complement
	cnot.*type*	cnot.b32 d, a, b	d = (a==0)? 1:0;	C logical not
	shl.*type*	shl.b32 d, a, b	d = a << b;	shift left
	shr.*type*	shr.s32 d, a, b	d = a >> b;	shift right
Memory Access	memory *.space* = .global, .shared, .local, .const; *.type* = .b8, .u8, .s8, .b16, .b32, .b64			
	ld.*space.type*	ld.global.b32 d, [a+off]	d = *(a+off);	load from memory *space*
	st.*space.type*	st.shared.b32 [d+off], a	*(d+off) = a;	store to memory *space*
	tex.*nd.dtyp.btype*	tex.2d.v4.f32.f32 d, a, b	d = tex2d(a, b);	texture lookup
	atom.*spc.op.type*	atom.global.add.u32 d,[a], b atom.global.cas.b32 d,[a], b, c	atomic { d = *a; *a = op(*a, b); }	atomic read-modify-write operation
	atom *.op* = and, or, xor, add, min, max, exch, cas; *.spc* = .global; *.type* = .b32			
Control Flow	branch	@p bra target	if (p) goto target;	conditional branch
	call	call (ret), func, (params)	ret = func(params);	call function
	ret	ret	return;	return from function call
	bar.sync	bar.sync d	wait for threads	barrier synchronization
	exit	exit	exit;	terminate thread execution

FIGURE B.4.3 Basic PTX GPU thread instructions.

Source operands are scalar 32-bit or 64-bit values in registers, an immediate value, or a constant; predicate operands are 1-bit Boolean values. Destinations are registers, except for store to memory. Instructions are predicated by prefixing them with `@p` or `@!p`, where `p` is a predicate register. Memory and texture instructions transfer scalars or vectors of two to four components, up to 128 bits in total. PTX instructions specify the behavior of one thread.

The PTX arithmetic instructions operate on 32-bit and 64-bit floating-point, signed integer, and unsigned integer types. Recent GPUs support 64-bit double-precision floating-point; see Section B.6. On current GPUs, PTX 64-bit integer and logical instructions are translated to two or more binary microinstructions that perform 32-bit operations. The GPU special function instructions are limited to 32-bit floating-point. The thread control flow instructions are conditional `branch`, function `call` and `return`, thread `exit`, and `bar.sync` (barrier synchronization). The conditional branch instruction `@p bra target` uses a predicate register `p` (or `!p`) previously set by a compare and set predicate `setp` instruction to determine whether the thread takes the branch or not. Other instructions can also be predicated on a predicate register being true or false.

Memory Access Instructions

The `tex` instruction fetches and filters texture samples from 1D, 2D, and 3D texture arrays in memory via the texture subsystem. Texture fetches generally use interpolated floating-point coordinates to address a texture. Once a graphics pixel shader thread computes its pixel fragment color, the raster operations processor blends it with the pixel color at its assigned (x, y) pixel position and writes the final color to memory.

To support computing and C/C++ language needs, the Tesla PTX ISA implements memory load/store instructions. It uses integer byte addressing with register plus offset address arithmetic to facilitate conventional compiler code optimizations. Memory load/store instructions are common in processors, but are a significant new capability in the Tesla architecture GPUs, as prior GPUs provided only the texture and pixel accesses required by the graphics APIs.

For computing, the load/store instructions access three read/write memory spaces that implement the corresponding CUDA memory spaces in Section B.3:

- Local memory for per-thread private addressable temporary data (implemented in external DRAM)
- Shared memory for low-latency access to data shared by cooperating threads in the same CTA/thread block (implemented in on-chip SRAM)
- Global memory for large data sets shared by all threads of a computing application (implemented in external DRAM)

The memory load/store instructions `ld.global`, `st.global`, `ld.shared`, `st.shared`, `ld.local`, and `st.local` access the global, shared, and local memory spaces. Computing programs use the fast barrier synchronization instruction `bar.sync` to synchronize threads within a CTA/thread block that communicate with each other via shared and global memory.

To improve memory bandwidth and reduce overhead, the local and global load/store instructions coalesce individual parallel thread requests from the same SIMT warp together into a single memory block request when the addresses fall in the same block and meet alignment criteria. Coalescing memory requests provides a significant performance boost over separate requests from individual threads. The multiprocessor's large thread count, together with support for many outstanding load requests, helps cover load-to-use latency for local and global memory implemented in external DRAM.

The latest Tesla architecture GPUs also provide efficient atomic memory operations on memory with the `atom.op.u32` instructions, including integer operations `add`, `min`, `max`, `and`, `or`, `xor`, `exchange`, and cas (compare-and-swap) operations, facilitating parallel reductions and parallel data structure management.

Barrier Synchronization for Thread Communication

Fast barrier synchronization permits CUDA programs to communicate frequently via shared memory and global memory by simply calling `__syncthreads();` as part of each interthread communication step. The synchronization intrinsic function generates a single `bar.sync` instruction. However, implementing fast barrier synchronization among up to 512 threads per CUDA thread block is a challenge.

Grouping threads into SIMT warps of 32 threads reduces the synchronization difficulty by a factor of 32. Threads wait at a barrier in the SIMT thread scheduler so they do not consume any processor cycles while waiting. When a thread executes a `bar.sync` instruction, it increments the barrier's thread arrival counter and the scheduler marks the thread as waiting at the barrier. Once all the CTA threads arrive, the barrier counter matches the expected terminal count, and the scheduler releases all the threads waiting at the barrier and resumes executing threads.

Streaming Processor (SP)

The multithreaded *streaming processor* (SP) core is the primary thread instruction processor in the multiprocessor. Its *register file* (RF) provides 1024 scalar 32-bit registers for up to 64 threads. It executes all the fundamental floating-point operations, including `add.f32`, `mul.f32`, `mad.f32` (floating multiply-add), `min.f32`, `max.f32`, and `setp.f32` (floating compare and set predicate). The floating-point add and multiply operations are compatible with the IEEE 754 standard for single-precision FP numbers, including *not-a-number* (NaN) and infinity values. The SP core also implements all of the 32-bit and 64-bit integer arithmetic, comparison, conversion, and logical PTX instructions shown in Figure B.4.3.

The floating-point `add` and `mul` operations employ IEEE round-to-nearest-even as the default rounding mode. The `mad.f32` floating-point multiply-add operation performs a multiplication with truncation, followed by an addition with round-to-nearest-even. The SP flushes input denormal operands to sign-preserved-zero. Results that underflow the target output exponent range are flushed to sign-preserved-zero after rounding.

Special Function Unit (SFU)

Certain thread instructions can execute on the SFUs, concurrently with other thread instructions executing on the SPs. The SFU implements the special function instructions of Figure B.4.3, which compute 32-bit floating-point approximations to reciprocal, reciprocal square root, and key transcendental functions. It also implements 32-bit floating-point planar attribute interpolation for pixel shaders, providing accurate interpolation of attributes such as color, depth, and texture coordinates.

Each pipelined SFU generates one 32-bit floating-point special function result per cycle; the two SFUs per multiprocessor execute special function instructions at a quarter the simple instruction rate of the eight SPs. The SFUs also execute the `mul.f32` multiply instruction concurrently with the eight SPs, increasing the peak computation rate up to 50% for threads with a suitable instruction mixture.

For functional evaluation, the Tesla architecture SFU employs quadratic interpolation based on enhanced minimax approximations for approximating the reciprocal, reciprocal square-root, $\log_2 x$, $2x$, and sin/cos functions. The accuracy of the function estimates ranges from 22 to 24 mantissa bits. See Section B.6 for more details on SFU arithmetic.

Comparing with Other Multiprocessors

Compared with SIMD vector architectures such as x86 SSE, the SIMT multiprocessor can execute individual threads independently, rather than always executing them together in synchronous groups. SIMT hardware finds data parallelism among independent threads, whereas SIMD hardware requires the software to express data parallelism explicitly in each vector instruction. A SIMT machine executes a warp of 32 threads synchronously when the threads take the same execution path, yet can execute each thread independently when they diverge. The advantage is significant because SIMT programs and instructions simply describe the behavior of a single independent thread, rather than a SIMD data vector of four or more data lanes. Yet the SIMT multiprocessor has SIMD-like efficiency, spreading the area and cost of one instruction unit across the 32 threads of a warp and across the eight streaming processor cores. SIMT provides the performance of SIMD together with the productivity of multithreading, avoiding the need to explicitly code SIMD vectors for edge conditions and partial divergence.

The SIMT multiprocessor imposes little overhead because it is hardware multithreaded with hardware barrier synchronization. That allows graphics shaders and CUDA threads to express very fine-grained parallelism. Graphics and CUDA programs use threads to express fine-grained data parallelism in a per-thread program, rather than forcing the programmer to express it as SIMD vector instructions. It is simpler and more productive to develop scalar single-thread code than vector code, and the SIMT multiprocessor executes the code with SIMD-like efficiency.

Coupling eight streaming processor cores together closely into a multiprocessor and then implementing a scalable number of such multiprocessors makes a two-level multiprocessor composed of multiprocessors. The CUDA programming model exploits the two-level hierarchy by providing individual threads for fine-grained parallel computations, and by providing grids of thread blocks for coarse-grained parallel operations. The same thread program can provide both fine-grained and coarse-grained operations. In contrast, CPUs with SIMD vector instructions must use two different programming models to provide fine-grained and coarse-grained operations: coarse-grained parallel threads on different cores, and SIMD vector instructions for fine-grained data parallelism.

Multithreaded Multiprocessor Conclusion

The example GPU multiprocessor based on the Tesla architecture is highly multithreaded, executing a total of up to 512 lightweight threads concurrently to support fine-grained pixel shaders and CUDA threads. It uses a variation on SIMD architecture and multithreading called SIMT (*single-instruction multiple-thread*) to efficiently broadcast one instruction to a warp of 32 parallel threads, while permitting each thread to branch and execute independently. Each thread executes its instruction stream on one of the eight *streaming processor* (SP) cores, which are multithreaded up to 64 threads.

The PTX ISA is a register-based load/store scalar ISA that describes the execution of a single thread. Because PTX instructions are optimized and translated to binary microinstructions for a specific GPU, the hardware instructions can evolve rapidly without disrupting compilers and software tools that generate PTX instructions.

B.5 Parallel Memory System

Outside of the GPU itself, the memory subsystem is the most important determiner of the performance of a graphics system. Graphics workloads demand very high transfer rates to and from memory. Pixel write and blend (read-modify-write) operations, depth buffer reads and writes, and texture map reads, as well as command and object vertex and attribute data reads, comprise the majority of memory traffic.

Modern GPUs are highly parallel, as shown in Figure B.2.5. For example, the GeForce 8800 can process 32 pixels per clock, at 600 MHz. Each pixel typically requires a color read and write and a depth read and write of a 4-byte pixel. Usually an average of two or three texels of four bytes each are read to generate the pixel's color. So for a typical case, there is a demand of 28 bytes times 32 pixels = 896 bytes per clock. Clearly the bandwidth demand on the memory system is enormous.

To supply these requirements, GPU memory systems have the following characteristics:

- They are wide, meaning there are a large number of pins to convey data between the GPU and its memory devices, and the memory array itself comprises many DRAM chips to provide the full total data bus width.
- They are fast, meaning aggressive signaling techniques are used to maximize the data rate (bits/second) per pin.
- GPUs seek to use every available cycle to transfer data to or from the memory array. To achieve this, GPUs specifically do not aim to minimize latency to the memory system. High throughput (utilization efficiency) and short latency are fundamentally in conflict.
- Compression techniques are used, both lossy, of which the programmer must be aware, and lossless, which is invisible to the application and opportunistic.
- Caches and work coalescing structures are used to reduce the amount of off-chip traffic needed and to ensure that cycles spent moving data are used as fully as possible.

DRAM Considerations

GPUs must take into account the unique characteristics of DRAM. DRAM chips are internally arranged as multiple (typically four to eight) banks, where each bank includes a power-of-2 number of rows (typically around 16,384), and each row contains a power-of-2 number of bits (typically 8192). DRAMs impose a variety of timing requirements on their controlling processor. For example, dozens of cycles are required to activate one row, but once activated, the bits within that row are randomly accessible with a new column address every four clocks. *Double-data rate* (DDR) synchronous DRAMs transfer data on both rising and falling edges of the interface clock (see Chapter 5). So a 1 GHz clocked DDR DRAM transfers data at 2 gigabits per second per data pin. Graphics DDR DRAMs usually have 32 bidirectional data pins, so eight bytes can be read or written from the DRAM per clock.

GPUs internally have a large number of generators of memory traffic. Different stages of the logical graphics pipeline each have their own request streams: command and vertex attribute fetch, shader texture fetch and load/store, and pixel depth and color read-write. At each logical stage, there are often multiple independent units to deliver the parallel throughput. These are each independent memory requestors. When viewed at the memory system, there is an enormous number of uncorrelated requests in flight. This is a natural mismatch to the reference pattern preferred by the DRAMs. A solution is for the GPU's memory controller to maintain separate heaps of traffic bound for different DRAM banks, and wait until enough traffic for

a particular DRAM row is pending before activating that row and transferring all the traffic at once. Note that accumulating pending requests, while good for DRAM row locality and thus efficient use of the data bus, leads to longer average latency as seen by the requestors whose requests spend time waiting for others. The design must take care that no particular request waits too long, otherwise some processing units can starve waiting for data and ultimately cause neighboring processors to become idle.

GPU memory subsystems are arranged as multiple *memory partitions*, each of which comprises a fully independent memory controller and one or two DRAM devices that are fully and exclusively owned by that partition. To achieve the best load balance and therefore approach the theoretical performance of n partitions, addresses are finely interleaved evenly across all memory partitions. The partition interleaving stride is typically a block of a few hundred bytes. The number of memory partitions is designed to balance the number of processors and other memory requesters.

Caches

GPU workloads typically have very large working sets—on the order of hundreds of megabytes to generate a single graphics frame. Unlike with CPUs, it is not practical to construct caches on chips large enough to hold anything close to the full working set of a graphics application. Whereas CPUs can assume very high cache hit rates (99.9% or more), GPUs experience hit rates closer to 90% and must therefore cope with many misses in flight. While a CPU can reasonably be designed to halt while waiting for a rare cache miss, a GPU needs to proceed with misses and hits intermingled. We call this a *streaming cache architecture*.

GPU caches must deliver very high-bandwidth to their clients. Consider the case of a texture cache. A typical texture unit may evaluate two bilinear interpolations for each of four pixels per clock cycle, and a GPU may have many such texture units all operating independently. Each bilinear interpolation requires four separate texels, and each texel might be a 64-bit value. Four 16-bit components are typical. Thus, total bandwidth is $2 \times 4 \times 4 \times 64 = 2048$ bits per clock. Each separate 64-bit texel is independently addressed, so the cache needs to handle 32 unique addresses per clock. This naturally favors a multibank and/or multiport arrangement of SRAM arrays.

MMU

Modern GPUs are capable of translating virtual addresses to physical addresses. On the GeForce 8800, all processing units generate memory addresses in a 40-bit virtual address space. For computing, load and store thread instructions use 32-bit byte addresses, which are extended to a 40-bit virtual address by adding a 40-bit offset. A memory management unit performs virtual to physical address

translation; hardware reads the page tables from local memory to respond to misses on behalf of a hierarchy of translation lookaside buffers spread out among the processors and rendering engines. In addition to physical page bits, GPU page table entries specify the compression algorithm for each page. Page sizes range from 4 to 128 kilobytes.

Memory Spaces

As introduced in Section B.3, CUDA exposes different memory spaces to allow the programmer to store data values in the most performance-optimal way. For the following discussion, NVIDIA Tesla architecture GPUs are assumed.

Global memory

Global memory is stored in external DRAM; it is not local to any one physical *streaming multiprocessor* (SM) because it is meant for communication among different CTAs (thread blocks) in different grids. In fact, the many CTAs that reference a location in global memory may not be executing in the GPU at the same time; by design, in CUDA a programmer does not know the relative order in which CTAs are executed. Because the address space is evenly distributed among all memory partitions, there must be a read/write path from any streaming multiprocessor to any DRAM partition.

Access to global memory by different threads (and different processors) is not guaranteed to have sequential consistency. Thread programs see a relaxed memory ordering model. Within a thread, the order of memory reads and writes to the same address is preserved, but the order of accesses to different addresses may not be preserved. Memory reads and writes requested by different threads are unordered. Within a CTA, the barrier synchronization instruction `bar.sync` can be used to obtain strict memory ordering among the threads of the CTA. The `membar` thread instruction provides a memory barrier/fence operation that commits prior memory accesses and makes them visible to other threads before proceeding. Threads can also use the atomic memory operations described in Section B.4 to coordinate work on memory they share.

Shared memory

Per-CTA shared memory is only visible to the threads that belong to that CTA, and shared memory only occupies storage from the time a CTA is created to the time it terminates. Shared memory can therefore reside on-chip. This approach has many benefits. First, shared memory traffc does not need to compete with limited off-chip bandwidth needed for global memory references. Second, it is practical to build very high-bandwidth memory structures on-chip to support the read/write demands of each streaming multiprocessor. In fact, the shared memory is closely coupled to the streaming multiprocessor.

Each streaming multiprocessor contains eight physical thread processors. During one shared memory clock cycle, each thread processor can process two threads' worth of instructions, so 16 threads' worth of shared memory requests must be handled in each clock. Because each thread can generate its own addresses, and the addresses are typically unique, the shared memory is built using 16 independently addressable SRAM banks. For common access patterns, 16 banks are sufficient to maintain throughput, but pathological cases are possible; for example, all 16 threads might happen to access a different address on one SRAM bank. It must be possible to route a request from any thread lane to any bank of SRAM, so a 16-by-16 interconnection network is required.

Local Memory

Per-thread local memory is private memory visible only to a single thread. Local memory is architecturally larger than the thread's register file, and a program can compute addresses into local memory. To support large allocations of local memory (recall the total allocation is the per-thread allocation times the number of active threads), local memory is allocated in external DRAM.

Although global and per-thread local memory reside off-chip, they are well-suited to being cached on-chip.

Constant Memory

Constant memory is read-only to a program running on the SM (it can be written via commands to the GPU). It is stored in external DRAM and cached in the SM. Because commonly most or all threads in a SIMT warp read from the same address in constant memory, a single address lookup per clock is sufficient. The constant cache is designed to broadcast scalar values to threads in each warp.

Texture Memory

Texture memory holds large read-only arrays of data. Textures for computing have the same attributes and capabilities as textures used with 3D graphics. Although textures are commonly two-dimensional images (2D arrays of pixel values), 1D (linear) and 3D (volume) textures are also available.

A compute program references a texture using a `tex` instruction. Operands include an identifier to name the texture, and one, two, or three coordinates based on the texture dimensionality. The floating-point coordinates include a fractional portion that specifies a sample location, often in-between texel locations. Noninteger coordinates invoke a bilinear weighted interpolation of the four closest values (for a 2D texture) before the result is returned to the program.

Texture fetches are cached in a streaming cache hierarchy designed to optimize throughput of texture fetches from thousands of concurrent threads. Some programs use texture fetches as a way to cache global memory.

Surfaces

Surface is a generic term for a one-dimensional, two-dimensional, or three-dimensional array of pixel values and an associated format. A variety of formats are defined; for example, a pixel may be defined as four 8-bit RGBA integer components, or four 16-bit floating-point components. A program kernel does not need to know the surface type. A `tex` instruction recasts its result values as floating-point, depending on the surface format.

Load/Store Access

Load/store instructions with integer byte addressing enable the writing and compiling of programs in conventional languages like C and C++. CUDA programs use load/store instructions to access memory.

To improve memory bandwidth and reduce overhead, the local and global load/store instructions coalesce individual parallel thread requests from the same warp together into a single memory block request when the addresses fall in the same block and meet alignment criteria. Coalescing individual small memory requests into large block requests provides a significant performance boost over separate requests. The large thread count, together with support for many outstanding load requests, helps cover load-to-use latency for local and global memory implemented in external DRAM.

ROP

As shown in Figure B.2.5, NVIDIA Tesla architecture GPUs comprise a scalable *streaming processor array* (SPA), which performs all of the GPU's programmable calculations, and a scalable memory system, which comprises external DRAM control and fixed function *Raster Operation Processors* (ROPs) that perform color and depth framebuffer operations directly on memory. Each ROP unit is paired with a specific memory partition. ROP partitions are fed from the SMs via an interconnection network. Each ROP is responsible for depth and stencil tests and updates, as well as color blending. The ROP and memory controllers cooperate to implement lossless color and depth compression (up to 8:1) to reduce external bandwidth demand. ROP units also perform atomic operations on memory.

B.6 Floating-point Arithmetic

GPUs today perform most arithmetic operations in the programmable processor cores using IEEE 754-compatible single precision 32-bit floating-point operations (see Chapter 3). The fixed-point arithmetic of early GPUs was succeeded by 16-bit, 24-bit, and 32-bit floating-point, then IEEE 754-compatible 32-bit floating-point.

Some fixed-function logic within a GPU, such as texture-filtering hardware, continues to use proprietary numeric formats. Recent GPUs also provide IEEE 754-compatible double-precision 64-bit floating-point instructions.

Supported Formats

The IEEE 754 standard for floating-point arithmetic specifies basic and storage formats. GPUs use two of the basic formats for computation, 32-bit and 64-bit binary floating-point, commonly called single precision and double precision. The standard also specifies a 16-bit binary storage floating-point format, **half precision**. GPUs and the Cg shading language employ the narrow 16-bit half data format for efficient data storage and movement, while maintaining high dynamic range. GPUs perform many texture filtering and pixel blending computations at half precision within the texture filtering unit and the raster operations unit. The OpenEXR high dynamic-range image file format developed by Industrial Light and Magic [2003] uses the identical half format for color component values in computer imaging and motion picture applications.

half precision A 16-bit binary floating-point format, with 1 sign bit, 5-bit exponent, 10-bit fraction, and an implied integer bit.

Basic Arithmetic

Common single-precision floating-point operations in GPU programmable cores include addition, multiplication, **multiply-add**, minimum, maximum, compare, set predicate, and conversions between integer and floating-point numbers. Floating-point instructions often provide source operand modifiers for negation and absolute value.

The floating-point addition and multiplication operations of most GPUs today are compatible with the IEEE 754 standard for single precision FP numbers, including *not-a-number* (NaN) and infinity values. The FP addition and multiplication operations use IEEE round-to-nearest-even as the default rounding mode. To increase floating-point instruction throughput, GPUs often use a compound multiply-add instruction (`mad`). The multiply-add operation performs FP multiplication with truncation, followed by FP addition with round-to-nearest-even. It provides two floating-point operations in one issuing cycle, without requiring the instruction scheduler to dispatch two separate instructions, but the computation is not fused and truncates the product before the addition. This makes it different from the fused multiply-add instruction discussed in Chapter 3 and later in this section. GPUs typically flush denormalized source operands to sign-preserved zero, and they flush results that underflow the target output exponent range to sign-preserved zero after rounding.

multiply-add (MAD) A single floating-point instruction that performs a compound operation: multiplication followed by addition.

Specialized Arithmetic

GPUs provide hardware to accelerate special function computation, attribute interpolation, and texture filtering. Special function instructions include cosine,

sine, binary exponential, binary logarithm, reciprocal, and reciprocal square root. Attribute interpolation instructions provide efficient generation of pixel attributes, derived from plane equation evaluation. The **special function unit (SFU)** introduced in Section B.4 computes special functions and interpolates planar attributes [Oberman and Siu, 2005].

special function unit (SFU) A hardware unit that computes special functions and interpolates planar attributes.

Several methods exist for evaluating special functions in hardware. It has been shown that quadratic interpolation based on Enhanced Minimax Approximations is a very efficient method for approximating functions in hardware, including reciprocal, reciprocal square-root, $\log_2 x$, 2^x, sin, and cos.

We can summarize the method of SFU quadratic interpolation. For a binary input operand X with n-bit significand, the significand is divided into two parts: X_u is the upper part containing m bits, and X_l is the lower part containing n-m bits. The upper m bits X_u are used to consult a set of three lookup tables to return three finite-word coefficients C_0, C_1, and C_2. Each function to be approximated requires a unique set of tables. These coefficients are used to approximate a given function f(X) in the range $X_u \le X < X_u + 2^{-m}$ by evaluating the expression:

$$f(X) = C_0 + C_1 X_1 + C_2 X_1^2$$

The accuracy of each of the function estimates ranges from 22 to 24 significand bits. Example function statistics are shown in Figure B.6.1.

The IEEE 754 standard specifies exact-rounding requirements for division and square root; however, for many GPU applications, exact compliance is not required. Rather, for those applications, higher computational throughput is more important than last-bit accuracy. For the SFU special functions, the CUDA math library provides both a full accuracy function and a fast function with the SFU instruction accuracy.

Another specialized arithmetic operation in a GPU is attribute interpolation. Key *attributes* are usually specified for vertices of primitives that make up a scene to be rendered. Example attributes are color, depth, and texture coordinates. These attributes must be interpolated in the (x,y) screen space as needed to determine the

Function	Input interval	Accuracy (good bits)	ULP* error	% exactly rounded	Monotonic
$1/x$	[1, 2)	24.02	0.98	87	Yes
1/sqrt(x)	[1, 4)	23.40	1.52	78	Yes
2^x	[0, 1)	22.51	1.41	74	Yes
$\log_2 x$	[1, 2)	22.57	N/A**	N/A	Yes
sin/cos	[0, π/2)	22.47	N/A	N/A	No

*ULP: unit in the last place.**N/A: not applicable.

FIGURE B.6.1 Special function approximation statistics. For the NVIDIA GeForce 8800 *special function unit* (SFU).

values of the attributes at each pixel location. The value of a given attribute *U* in an (x, y) plane can be expressed using plane equations of the form:

$$U(x,y) = A_u x + B_u Y + C_u$$

where *A*, *B*, and *C* are interpolation parameters associated with each attribute *U*. The interpolation parameters *A*, *B*, and *C* are all represented as single-precision floating-point numbers.

Given the need for both a function evaluator and an attribute interpolator in a pixel shader processor, a single SFU that performs both functions for efficiency can be designed. Both functions use a sum of products operation to interpolate results, and the number of terms to be summed in both functions is very similar.

Texture Operations

Texture mapping and filtering is another key set of specialized floating-point arithmetic operations in a GPU. The operations used for texture mapping include:

1. Receive texture address (s, t) for the current screen pixel (x, y), where s and t are single-precision floating-point numbers.
2. Compute the level of detail to identify the correct texture **MIP-map** level.
3. Compute the trilinear interpolation fraction.
4. Scale texture address (s, t) for the selected MIP-map level.
5. Access memory and retrieve desired texels (texture elements).
6. Perform filtering operation on texels.

MIP-map A Latin phrase *multum in parvo*, or much in a small space. A MIP-map contains precalculated images of different resolutions, used to increase rendering speed and reduce artifacts.

Texture mapping requires a significant amount of floating-point computation for full-speed operation, much of which is done at 16-bit half precision. As an example, the GeForce 8800 Ultra delivers about 500 GFLOPS of proprietary format floating-point computation for texture mapping instructions, in addition to its conventional IEEE single-precision floating-point instructions. For more details on texture mapping and filtering, see Foley and van Dam [1995].

Performance

The floating-point addition and multiplication arithmetic hardware is fully pipelined, and latency is optimized to balance delay and area. While pipelined, the throughput of the special functions is less than the floating-point addition and multiplication operations. Quarter-speed throughput for the special functions is typical performance in modern GPUs, with one SFU shared by four SP cores. In contrast, CPUs typically have significantly lower throughput for similar functions, such as division and square root, albeit with more accurate results. The attribute interpolation hardware is typically fully pipelined to enable full-speed pixel shaders.

Double precision

Newer GPUs such as the Tesla T10P also support IEEE 754 64-bit double-precision operations in hardware. Standard floating-point arithmetic operations in double precision include addition, multiplication, and conversions between different floating-point and integer formats. The 2008 IEEE 754 floating-point standard includes specification for the *fused-multiply-add* (FMA) operation, as discussed in Chapter 3. The FMA operation performs a floating-point multiplication followed by an addition, with a single rounding. The fused multiplication and addition operations retain full accuracy in intermediate calculations. This behavior enables more accurate floating-point computations involving the accumulation of products, including dot products, matrix multiplication, and polynomial evaluation. The FMA instruction also enables efficient software implementations of exactly rounded division and square root, removing the need for a hardware division or square root unit.

A double-precision hardware FMA unit implements 64-bit addition, multiplication, conversions, and the FMA operation itself. The architecture of a

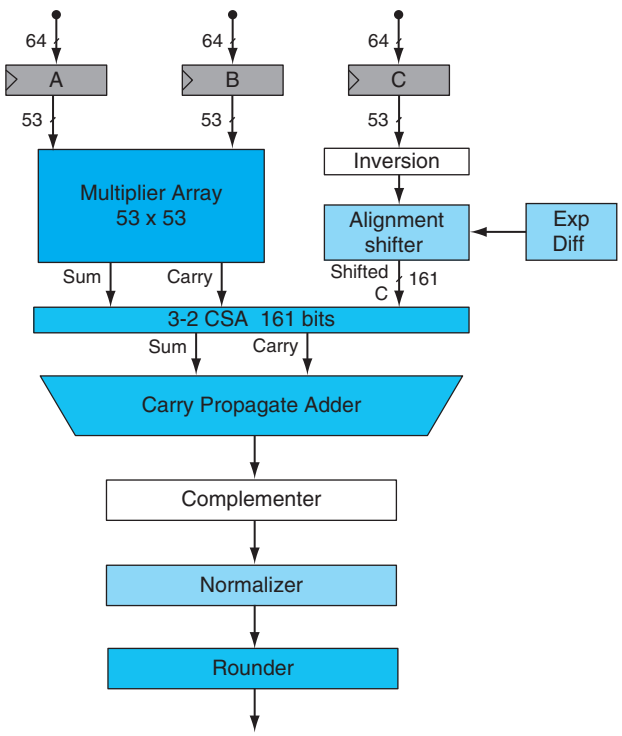


FIGURE B.6.2 Double-precision fused-multiply-add (FMA) unit. Hardware to implement floating-point A× B+ C for double precision.

double-precision FMA unit enables full-speed denormalized number support on both inputs and outputs. Figure B.6.2 shows a block diagram of an FMA unit.

As shown in Figure B.6.2, the significands of A and B are multiplied to form a 106-bit product, with the results left in carry-save form. In parallel, the 53-bit addend C is conditionally inverted and aligned to the 106-bit product. The sum and carry results of the 106-bit product are summed with the aligned addend through a 161-bit-wide *carry-save adder* (CSA). The carry-save output is then summed together in a carry-propagate adder to produce an unrounded result in nonredundant, two's complement form. The result is conditionally recomplemented, so as to return a result in sign-magnitude form. The complemented result is normalized, and then it is rounded to fit within the target format.

B.7 Real Stuff: The NVIDIA GeForce 8800

The NVIDIA GeForce 8800 GPU, introduced in November 2006, is a unified vertex and pixel processor design that also supports parallel computing applications written in C using the CUDA parallel programming model. It is the first implementation of the Tesla unified graphics and computing architecture described in Section B.4 and in Lindholm et al. [2008]. A family of Tesla architecture GPUs addresses the different needs of laptops, desktops, workstations, and servers.

Streaming Processor Array (SPA)

The GeForce 8800 GPU shown in Figure B.7.1 contains 128 *streaming processor* (SP) cores organized as 16 *streaming multiprocessors* (SMs). Two SMs share a texture unit in each *texture/processor cluster* (TPC). An array of eight TPCs makes up the *streaming processor array* (SPA), which executes all graphics shader programs and computing programs.

The host interface unit communicates with the host CPU via the PCI-Express bus, checks command consistency, and performs context switching. The input assembler collects geometric primitives (points, lines, triangles). The work distribution blocks dispatch vertices, pixels, and compute thread arrays to the TPCs in the SPA. The TPCs execute vertex and geometry shader programs and computing programs. Output geometric data are sent to the viewport/clip/setup/raster/zcull block to be rasterized into pixel fragments that are then redistributed back into the SPA to execute pixel shader programs. Shaded pixels are sent across the interconnection network for processing by the ROP units. The network also routes texture memory read requests from the SPA to DRAM and reads data from DRAM through a level-2 cache back to the SPA.

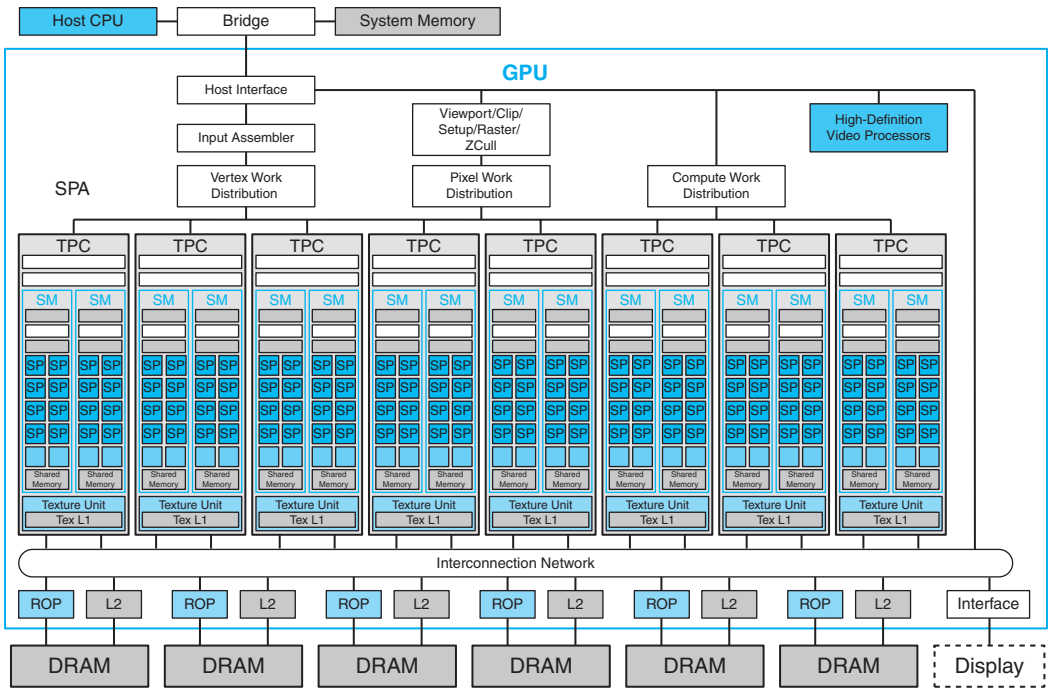


FIGURE B.7.1 NVIDIA Tesla unified graphics and computing GPU architecture. This GeForce 8800 has 128 *streaming processor* (SP) cores in 16 *streaming multiprocessors* (SMs), arranged in eight *texture/processor clusters* (TPCs). The processors connect with six 64-bit-wide DRAM partitions via an interconnection network. Other GPUs implementing the Tesla architecture vary the number of SP cores, SMs, DRAM partitions, and other units.

Texture/Processor Cluster (TPC)

Each TPC contains a geometry controller, an SMC, two SMs, and a texture unit as shown in Figure B.7.2.

The geometry controller maps the logical graphics vertex pipeline into recirculation on the physical SMs by directing all primitive and vertex attribute and topology flow in the TPC.

The SMC controls multiple SMs, arbitrating the shared texture unit, load/store path, and I/O path. The SMC serves three graphics workloads simultaneously: vertex, geometry, and pixel.

The texture unit processes a texture instruction for one vertex, geometry, or pixel quad, or four compute threads per cycle. Texture instruction sources are texture coordinates, and the outputs are weighted samples, typically a four-component (RGBA) floating-point color. The texture unit is deeply pipelined. Although it contains a streaming cache to capture filtering locality, it streams hits mixed with misses without stalling.

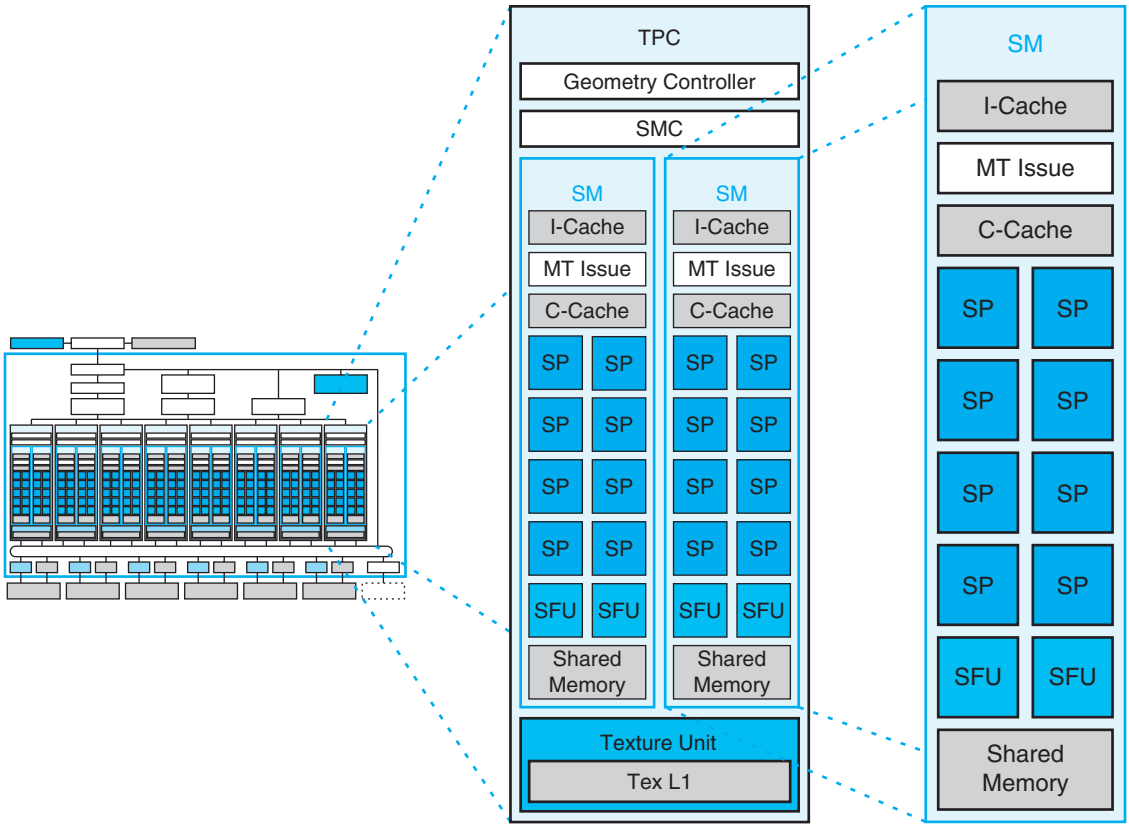


FIGURE B.7.2 Texture/processor cluster (TPC) and a streaming multiprocessor (SM). Each SM has eight *streaming processor* (SP) cores, two SFUs, and a shared memory.

Streaming Multiprocessor (SM)

The SM is a unified graphics and computing multiprocessor that executes vertex, geometry, and pixel-fragment shader programs and parallel computing programs. The SM consists of eight SP thread processor cores, two SFUs, a multithreaded instruction fetch and issue unit (MT issue), an instruction cache, a read-only constant cache, and a 16 KB read/write shared memory. It executes scalar instructions for individual threads.

The GeForce 8800 Ultra clocks the SP cores and SFUs at 1.5 GHz, for a peak of 36 GFLOPS per SM. To optimize power and area efficiency, some SM nondatapath units operate at half the SP clock rate.

To efficiently execute hundreds of parallel threads while running several different programs, the SM is hardware multithreaded. It manages and executes up to 768 concurrent threads in hardware with zero scheduling overhead. Each thread has its own thread execution state and can execute an independent code path.

A warp consists of up to 32 threads of the same type—vertex, geometry, pixel, or compute. The SIMT design, previously described in Section B.4, shares the SM instruction fetch and issue unit efficiently across 32 threads but requires a full warp of active threads for full performance efficiency.

The SM schedules and executes multiple warp types concurrently. Each issue cycle, the scheduler selects one of the 24 warps to execute a SIMT warp instruction. An issued warp instruction executes as four sets of eight threads over four processor cycles. The SP and SFU units execute instructions independently, and by issuing instructions between them on alternate cycles, the scheduler can keep both fully occupied. A scoreboard qualifies each warp for issue each cycle. The instruction scheduler prioritizes all ready warps and selects the one with highest priority for issue. Prioritization considers warp type, instruction type, and "fairness" to all warps executing in the SM.

The SM executes *cooperative thread arrays* (CTAs) as multiple concurrent warps which access a shared memory region allocated dynamically for the CTA.

Instruction Set

Threads execute scalar instructions, unlike previous GPU vector instruction architectures. Scalar instructions are simpler and compiler-friendly. Texture instructions remain vector-based, taking a source coordinate vector and returning a filtered color vector.

The register-based instruction set includes all the floating-point and integer arithmetic, transcendental, logical, flow control, memory load/store, and texture instructions listed in the PTX instruction table of Figure B.4.3. Memory load/store instructions use integer byte addressing with register-plus-offset address arithmetic. For computing, the load/store instructions access three read-write memory spaces: local memory for per-thread, private, temporary data; shared memory for low-latency per-CTA data shared by the threads of the CTA; and global memory for data shared by all threads. Computing programs use the fast barrier synchronization `bar.sync` instruction to synchronize threads within a CTA that communicate with each other via shared and global memory. The latest Tesla architecture GPUs implement PTX atomic memory operations, which facilitate parallel reductions and parallel data structure management.

Streaming Processor (SP)

The multithreaded SP core is the primary thread processor, as introduced in Section B.4. Its register file provides 1024 scalar 32-bit registers for up to 96 threads (more threads than in the example SP of Section B.4). Its floating-point add and

multiply operations are compatible with the IEEE 754 standard for single-precision FP numbers, including *not-a-number* (NaN) and infinity. The add and multiply operations use IEEE round-to-nearest-even as the default rounding mode. The SP core also implements all of the 32-bit and 64-bit integer arithmetic, comparison, conversion, and logical PTX instructions in Figure B.4.3. The processor is fully pipelined, and latency is optimized to balance delay and area.

Special Function Unit (SFU)

The SFU supports computation of both transcendental functions and planar attribute interpolation. As described in Section B.6, it uses quadratic interpolation based on enhanced minimax approximations to approximate the reciprocal, reciprocal square root, $\log_2 x$, 2^x, and sin/cos functions at one result per cycle. The SFU also supports pixel attribute interpolation such as color, depth, and texture coordinates at four samples per cycle.

Rasterization

Geometry primitives from the SMs go in their original round-robin input order to the viewport/clip/setup/raster/zcull block. The viewport and clip units clip the primitives to the view frustum and to any enabled user clip planes, and then transform the vertices into screen (pixel) space.

Surviving primitives then go to the setup unit, which generates edge equations for the rasterizer. A coarse-rasterization stage generates all pixel tiles that are at least partially inside the primitive. The zcull unit maintains a hierarchical z surface, rejecting pixel tiles if they are conservatively known to be occluded by previously drawn pixels. The rejection rate is up to 256 pixels per clock. Pixels that survive zcull then go to a fine-rasterization stage that generates detailed coverage information and depth values.

The depth test and update can be performed ahead of the fragment shader, or after, depending on current state. The SMC assembles surviving pixels into warps to be processed by an SM running the current pixel shader. The SMC then sends surviving pixel and associated data to the ROP.

Raster Operations Processor (ROP) and Memory System

Each ROP is paired with a specific memory partition. For each pixel fragment emitted by a pixel shader program, ROPs perform depth and stencil testing and updates, and in parallel, color blending and updates. Lossless color compression (up to 8:1) and depth compression (up to 8:1) are used to reduce DRAM bandwidth. Each ROP has a peak rate of four pixels per clock and supports 16-bit floating-point and 32-bit floating-point HDR formats. ROPs support double-rate-depth processing when color writes are disabled.

Antialiasing support includes up to 16× multisampling and supersampling. The *coverage-sampling antialiasing* (CSAA) algorithm computes and stores Boolean coverage at up to 16 samples and compresses redundant color, depth, and stencil information into the memory footprint and a bandwidth of four or eight samples for improved performance.

The DRAM memory data bus width is 384 pins, arranged in six independent partitions of 64 pins each. Each partition supports double-data-rate DDR2 and graphics-oriented GDDR3 protocols at up to 1.0 GHz, yielding a bandwidth of about 16 GB/s per partition, or 96 GB/s.

The memory controllers support a wide range of DRAM clock rates, protocols, device densities, and data bus widths. Texture and load/store requests can occur between any TPC and any memory partition, so an interconnection network routes requests and responses.

Scalability

The Tesla unified architecture is designed for scalability. Varying the number of SMs, TPCs, ROPs, caches, and memory partitions provides the right balance for different performance and cost targets in GPU market segments. *Scalable link interconnect* (SLI) connects multiple GPUs, providing further scalability.

Performance

The GeForce 8800 Ultra clocks the SP thread processor cores and SFUs at 1.5 GHz, for a theoretical operation peak of 576 GFLOPS. The GeForce 8800 GTX has a 1.35 GHz processor clock and a corresponding peak of 518 GFLOPS.

The following three sections compare the performance of a GeForce 8800 GPU with a multicore CPU on three different applications—dense linear algebra, fast Fourier transforms, and sorting. The GPU programs and libraries are compiled CUDA C code. The CPU code uses the single-precision multithreaded Intel MKL 10.0 library to leverage SSE instructions and multiple cores.

Dense Linear Algebra Performance

Dense linear algebra computations are fundamental in many applications. Volkov and Demmel [2008] present GPU and CPU performance results for single-precision dense matrix-matrix multiplication (the SGEMM routine) and LU, QR, and Cholesky matrix factorizations. Figure B.7.3 compares GFLOPS rates on SGEMM dense matrix-matrix multiplication for a GeForce 8800 GTX GPU with a quad-core CPU. Figure B.7.4 compares GFLOPS rates on matrix factorization for a GPU with a quad-core CPU.

Because SGEMM matrix-matrix multiply and similar BLAS3 routines are the bulk of the work in matrix factorization, their performance sets an upper bound on factorization rate. As the matrix order increases beyond 200 to 400, the factorization

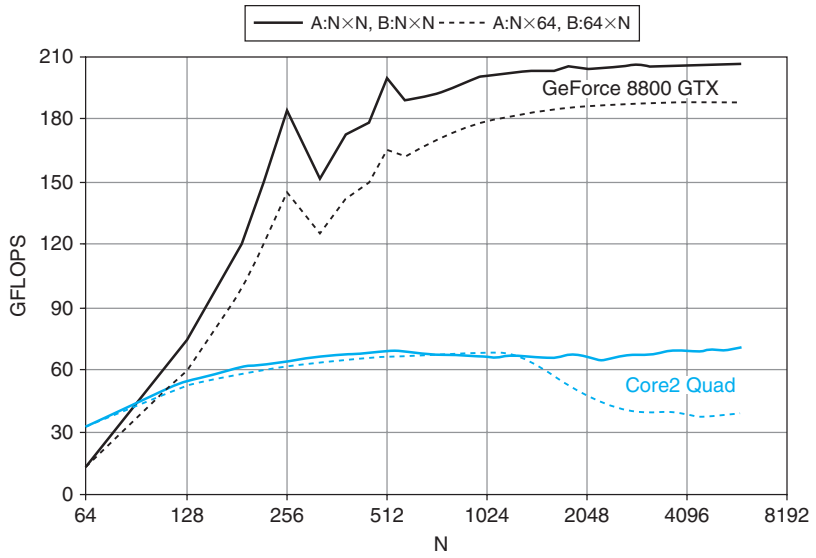


FIGURE B.7.3 SGEMM dense matrix-matrix multiplication performance rates. The graph shows single-precision GFLOPS rates achieved in multiplying square N×N matrices (solid lines) and thin N ×64 and 64×N matrices (dashed lines). Adapted from Figure 6 of Volkov and Demmel [2008]. The black lines are a 1.35 GHz GeForce 8800 GTX using Volkov's SGEMM code (now in NVIDIA CUBLAS 2.0) on matrices in GPU memory. The blue lines are a quad-core 2.4 GHz Intel Core2 Quad Q6600, 64-bit Linux, Intel MKL 10.0 on matrices in CPU memory.

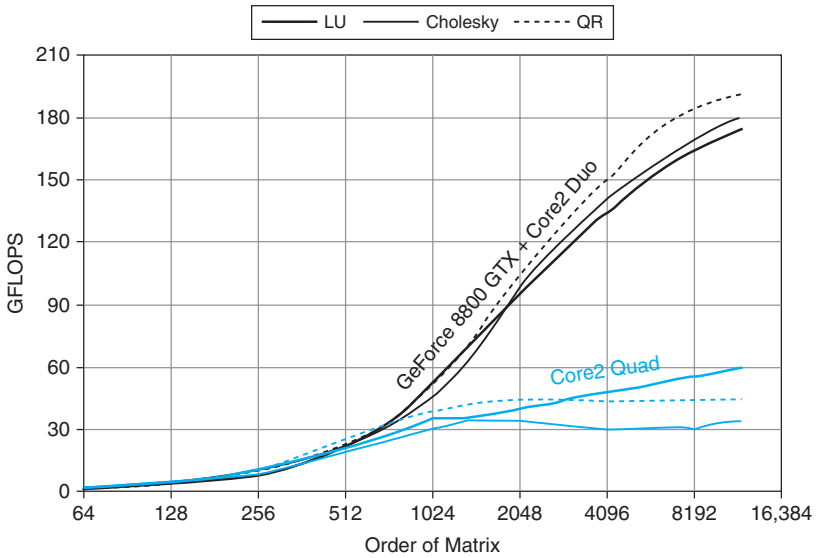


FIGURE B.7.4 Dense matrix factorization performance rates. The graph shows GFLOPS rates achieved in matrix factorizations using the GPU and using the CPU alone. Adapted from Figure 7 of Volkov and Demmel [2008]. The black lines are for a 1.35 GHz NVIDIA GeForce 8800 GTX, CUDA 1.1, Windows XP attached to a 2.67 GHz Intel Core2 Duo E6700 Windows XP, including all CPU–GPU data transfer times. The blue lines are for a quad-core 2.4 GHz Intel Core2 Quad Q6600, 64-bit Linux, Intel MKL 10.0.

problem becomes large enough that SGEMM can leverage the GPU parallelism and overcome the CPU–GPU system and copy overhead. Volkov's SGEMM matrix-matrix multiply achieves 206 GFLOPS, about 60% of the GeForce 8800 GTX peak multiply-add rate, while the QR factorization reached 192 GFLOPS, about 4.3 times the quad-core CPU.

FFT Performance

Fast Fourier Transforms (FFTs) are used in many applications. Large transforms and multidimensional transforms are partitioned into batches of smaller 1D transforms.

Figure B.7.5 compares the in-place 1D complex single-precision FFT performance of a 1.35 GHz GeForce 8800 GTX (dating from late 2006) with a 2.8 GHz quad-Core Intel Xeon E5462 series (code named "Harpertown," dating from late 2007). CPU performance was measured using the Intel *Math Kernel Library* (MKL) 10.0 FFT with four threads. GPU performance was measured using the NVIDIA CUFFT 2.1 library and batched 1D radix-16 decimation-in-frequency FFTs. Both CPU and GPU throughput performance was measured using batched FFTs; batch size was $2^{24}/n$, where n is the transform size. Thus, the workload for every transform size was 128 MB. To determine GFLOPS rate, the number of operations per transform was taken as $5n \log_2 n$.

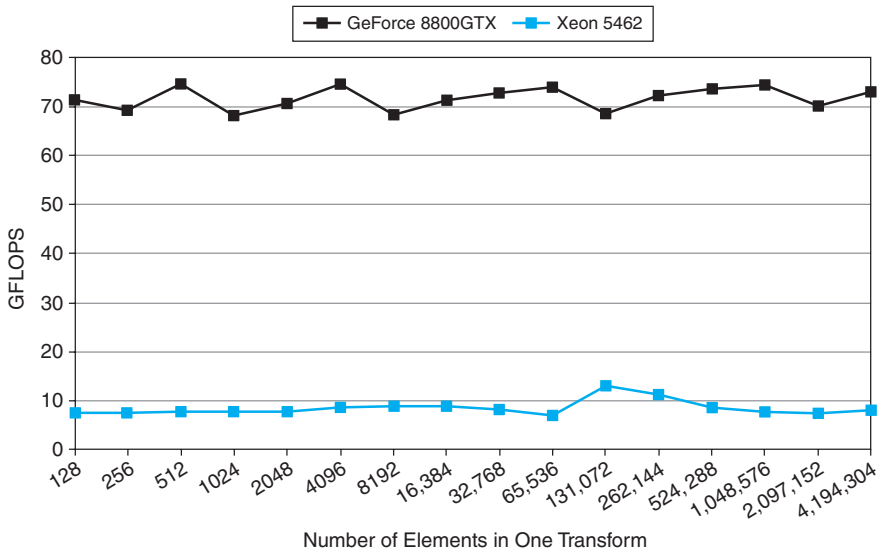


FIGURE B.7.5 Fast Fourier transform throughput performance. The graph compares the performance of batched one-dimensional in-place complex FFTs on a 1.35 GHz GeForce 8800 GTX with a quad-core 2.8 GHz Intel Xeon E5462 series (code named "Harpertown"), 6MB L2 Cache, 4GB Memory, 1600 FSB, Red Hat Linux, Intel MKL 10.0.

Sorting Performance

In contrast to the applications just discussed, sort requires far more substantial coordination among parallel threads, and parallel scaling is correspondingly harder to obtain. Nevertheless, a variety of well-known sorting algorithms can be efficiently parallelized to run well on the GPU. Satish et al. [2008] detail the design of sorting algorithms in CUDA, and the results they report for radix sort are summarized below.

Figure B.7.6 compares the parallel sorting performance of a GeForce 8800 Ultra with an 8-core Intel Clovertown system, both of which date to early 2007. The CPU cores are distributed between two physical sockets. Each socket contains a multichip module with twin Core2 chips, and each chip has a 4MB L2 cache. All sorting routines were designed to sort key-value pairs where both keys and values are 32-bit integers. The primary algorithm being studied is radix sort, although the quicksort-based `parallel_sort()` procedure provided by Intel's Threading Building Blocks is also included for comparison. Of the two CPU-based radix sort codes, one was implemented using only the scalar instruction set and the other utilizes carefully hand-tuned assembly language routines that take advantage of the SSE2 SIMD vector instructions.

The graph itself shows the achieved sorting rate—defined as the number of elements sorted divided by the time to sort—for a range of sequence sizes. It is

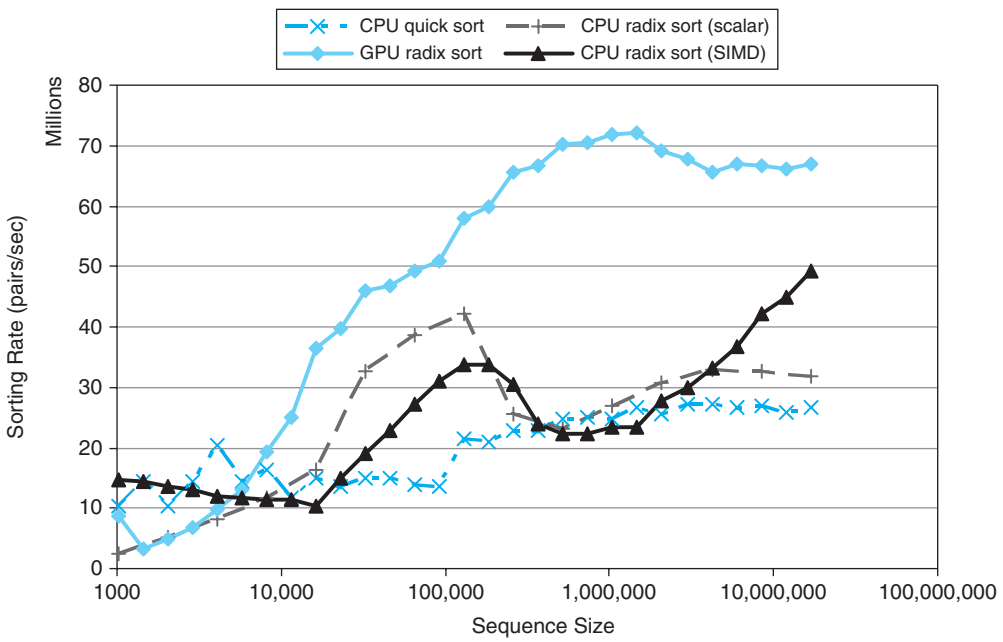


FIGURE B.7.6 Parallel sorting performance. This graph compares sorting rates for parallel radix sort implementations on a 1.5 GHz GeForce 8800 Ultra and an 8-core 2.33 GHz Intel Core2 Xeon E5345 system.

apparent from this graph that the GPU radix sort achieved the highest sorting rate for all sequences of 8K-elements and larger. In this range, it is on average 2.6 times faster than the quicksort-based routine and roughly two times faster than the radix sort routines, all of which were using the eight available CPU cores. The CPU radix sort performance varies widely, likely due to poor cache locality of its global permutations.

B.8 Real Stuff: Mapping Applications to GPUs

The advent of multicore CPUs and manycore GPUs means that mainstream processor chips are now parallel systems. Furthermore, their parallelism continues to scale with Moore's law. The challenge is to develop mainstream visual computing and high-performance computing applications that transparently scale their parallelism to leverage the increasing number of processor cores, much as 3D graphics applications transparently scale their parallelism to GPUs with widely varying numbers of cores.

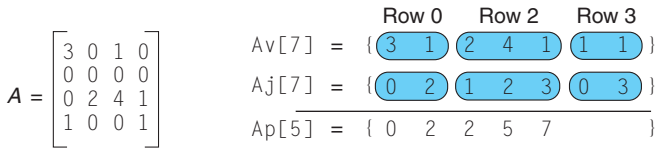
This section presents examples of mapping scalable parallel computing applications to the GPU using CUDA.

Sparse Matrices

A wide variety of parallel algorithms can be written in CUDA in a fairly straightforward manner, even when the data structures involved are not simple regular grids. *Sparse matrix-vector multiplication* (SpMV) is a good example of an important numerical building block that can be parallelized quite directly using the abstractions provided by CUDA. The kernels we discuss below, when combined with the provided CUBLAS vector routines, make writing iterative solvers such as the conjugate gradient method straightforward.

A sparse $n \times n$ matrix is one in which the number of nonzero entries m is only a small fraction of the total. Sparse matrix representations seek to store only the nonzero elements of a matrix. Since it is fairly typical that a sparse $n \times n$ matrix will contain only $m = O(n)$ nonzero elements, this represents a substantial saving in storage space and processing time.

One of the most common representations for general unstructured sparse matrices is the *compressed sparse row* (CSR) representation. The m nonzero elements of the matrix A are stored in row-major order in an array `Av`. A second array `Aj` records the corresponding column index for each entry of `Av`. Finally, an array `Ap` of $n+1$ elements records the extent of each row in the previous arrays; the entries for row i in `Aj` and `Av` extend from index `Ap[i]` up to, but not including, index `Ap[i + 1]`. This implies that `Ap[0]` will always be 0 and `Ap[n]` will always be the number of nonzero elements in the matrix. Figure B.8.1 shows an example of the CSR representation of a simple matrix.



a. Sample matrix *A* b. CSR representation of matrix

FIGURE B.8.1 Compressed sparse row (CSR) matrix.

```
float multiply_row(unsigned int rowsize,
                   unsigned int *Aj, // column indices for row
                   float *Av,        // nonzero entries for row
                   float *x)         // the RHS vector
{
    float sum = 0;

    for(unsigned int column=0; column<rowsize; ++column)
        sum += Av[column] * x[Aj[column]];

    return sum;
}
```

FIGURE B.8.2 Serial C code for a single row of sparse matrix-vector multiply.

Given a matrix A in CSR form and a vector x, we can compute a single row of the product $y = Ax$ using the `multiply_row()` procedure shown in Figure B.8.2. Computing the full product is then simply a matter of looping over all rows and computing the result for that row using `multiply_row()`, as in the serial C code shown in Figure B.8.3.

This algorithm can be translated into a parallel CUDA kernel quite easily. We simply spread the loop in `csrmul_serial()` over many parallel threads. Each thread will compute exactly one row of the output vector y. The code for this kernel is shown in Figure B.8.4. Note that it looks extremely similar to the serial loop used in the `csrmul_serial()` procedure. There are really only two points of difference. First, the row index for each thread is computed from the block and thread indices assigned to each thread, eliminating the for-loop. Second, we have a conditional that only evaluates a row product if the row index is within the bounds of the matrix (this is necessary since the number of rows n need not be a multiple of the block size used in launching the kernel).

```
void csrmul_serial(unsigned int *Ap, unsigned int *Aj,
                   float *Av, unsigned int num_rows,
                   float *x, float *y)
{
    for(unsigned int row=0; row<num_rows; ++row)
    {
        unsigned int row_begin = Ap[row];
        unsigned int row_end   = Ap[row+1];

        y[row] = multiply_row(row_end-row_begin, Aj+row_begin,
                              Av+row_begin, x);
    }
}
```

FIGURE B.8.3 Serial code for sparse matrix-vector multiply.

```
__global__
void csrmul_kernel(unsigned int *Ap, unsigned int *Aj,
                   float *Av, unsigned int num_rows,
                   float *x, float *y)
{
    unsigned int row = blockIdx.x*blockDim.x + threadIdx.x;

    if( row<num_rows )
    {
        unsigned int row_begin = Ap[row];
        unsigned int row_end   = Ap[row+1];

        y[row] = multiply_row(row_end-row_begin, Aj+row_begin,
                              Av+row_begin, x);
    }
}
```

FIGURE B.8.4 CUDA version of sparse matrix-vector multiply.

Assuming that the matrix data structures have already been copied to the GPU device memory, launching this kernel will look like:

```
unsigned int blocksize = 128;  // or any size up to 512
unsigned int nblocks = (num_rows + blocksize - 1) / blocksize;
csrmul_kernel<<<nblocks,blocksize>>>(Ap, Aj, Av, num_rows, x, y);
```

The pattern that we see here is a very common one. The original serial algorithm is a loop whose iterations are independent of each other. Such loops can be parallelized quite easily by simply assigning one or more iterations of the loop to each parallel thread. The programming model provided by CUDA makes expressing this type of parallelism particularly straightforward.

This general strategy of decomposing computations into blocks of independent work, and more specifically breaking up independent loop iterations, is not unique to CUDA. This is a common approach used in one form or another by various parallel programming systems, including OpenMP and Intel's Threading Building Blocks.

Caching in Shared Memory

The SpMV algorithms outlined above are fairly simplistic. There are a number of optimizations that can be made in both the CPU and GPU codes that can improve performance, including loop unrolling, matrix reordering, and register blocking. The parallel kernels can also be reimplemented in terms of data parallel *scan* operations presented by Sengupta et al. [2007].

One of the important architectural features exposed by CUDA is the presence of the per-block shared memory, a small on-chip memory with very low latency. Taking advantage of this memory can deliver substantial performance improvements. One common way of doing this is to use shared memory as a software-managed cache to hold frequently reused data. Modifcations using shared memory are shown in Figure B.8.5.

In the context of sparse matrix multiplication, we observe that several rows of *A* may use a particular array element `x[i]`. In many common cases, and particularly when the matrix has been reordered, the rows using `x[i]` will be rows near row *i*. We can therefore implement a simple caching scheme and expect to achieve some performance benefit. The block of threads processing rows *i* through *j* will load `x[i]` through `x[j]` into its shared memory. We will unroll the `multiply_row()` loop and fetch elements of `x` from the cache whenever possible. The resulting code is shown in Figure B.8.5. Shared memory can also be used to make other optimizations, such as fetching `Ap[row+1]` from an adjacent thread rather than refetching it from memory.

Because the Tesla architecture provides an explicitly managed on-chip shared memory, rather than an implicitly active hardware cache, it is fairly common to add this sort of optimization. Although this can impose some additional development burden on the programmer, it is relatively minor, and the potential performance benefits can be substantial. In the example shown above, even this fairly simple use of shared memory returns a roughly 20% performance improvement on representative matrices derived from 3D surface meshes. The availability of an explicitly managed memory in lieu of an implicit cache also has the advantage that caching and prefetching policies can be specifically tailored to the application needs.

```
__global__
void csrmul_cached(unsigned int *Ap, unsigned int *Aj,
                   float *Av, unsigned int num_rows,
                   const float *x, float *y)
{
    // Cache the rows of x[] corresponding to this block.
    __shared__ float cache[blocksize];

    unsigned int block_begin = blockIdx.x * blockDim.x;
    unsigned int block_end   = block_begin + blockDim.x;
    unsigned int row         = block_begin + threadIdx.x;

    // Fetch and cache our window of x[].
    if( row<num_rows)  cache[threadIdx.x] = x[row];
    __syncthreads();

    if( row<num_rows )
    {
        unsigned int row_begin = Ap[row];
        unsigned int row_end   = Ap[row+1];
        float sum = 0, x_j;

        for(unsigned int col=row_begin; col<row_end; ++col)
        {
            unsigned int j = Aj[col];

            // Fetch x_j from our cache when possible
            if( j>=block_begin && j<block_end )
                x_j = cache[j-block_begin];
            else
                x_j = x[j];

            sum += Av[col] * x_j;
        }

        y[row] = sum;
    }
}
```

FIGURE B.8.5 Shared memory version of sparse matrix-vector multiply.

These are fairly simple kernels whose purpose is to illustrate basic techniques in writing CUDA programs, rather than how to achieve maximal performance. Numerous possible avenues for optimization are available, several of which are explored by Williams et al. [2007] on a handful of different multicore architectures. Nevertheless, it is still instructive to examine the comparative performance of even these simplistic kernels. On a 2 GHz Intel Core2 Xeon E5335 processor, the `csrmul_serial()` kernel runs at roughly 202 million nonzeros processed per second, for a collection of Laplacian matrices derived from 3D triangulated surface meshes. Parallelizing this kernel with the `parallel_for` construct provided by Intel's Threading Building Blocks produces parallel speed-ups of 2.0, 2.1, and 2.3 running on two, four, and eight cores of the machine, respectively. On a GeForce 8800 Ultra, the `csrmul_kernel()` and `csrmul_cached()` kernels achieve processing rates of roughly 772 and 920 million nonzeros per second, corresponding to parallel speed-ups of 3.8 and 4.6 times over the serial performance of a single CPU core.

Scan and Reduction

Parallel *scan*, also known as parallel *prefix sum*, is one of the most important building blocks for data-parallel algorithms [Blelloch, 1990]. Given a sequence a of n elements:

$$[a_0, a_1, \ldots, a_{n-1}]$$

and a binary associative operator $\oplus$, the `scan` function computes the sequence:

$$\text{scan}(a, \oplus) = [a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1})]$$

As an example, if we take $\oplus$ to be the usual addition operator, then applying scan to the input array

$$a = [3\,1\,7\,0\,4\,1\,6\,3]$$

will produce the sequence of partial sums:

$$\text{scan}(a, +) = [3\,4\,11\,11\,15\,16\,22\,25]$$

This scan operator is an *inclusive* scan, in the sense that element i of the output sequence incorporates element a_i of the input. Incorporating only previous elements would yield an *exclusive* scan operator, also known as a *prefix-sum* operation.

The serial implementation of this operation is extremely simple. It is simply a loop that iterates once over the entire sequence, as shown in Figure B.8.6.

At first glance, it might appear that this operation is inherently serial. However, it can actually be implemented in parallel efficiently. The key observation is that

```
template<class T>
__host__ T plus_scan(T *x, unsigned int n)
{
    for(unsigned int i=1; i<n; ++i)
        x[i] = x[i-1] + x[i];
}
```

FIGURE B.8.6 Template for serial plus-scan.

```
template<class T>
__device__ T plus_scan(T *x)
{
    unsigned int i = threadIdx.x;
    unsigned int n = blockDim.x;

    for(unsigned int offset=1; offset<n; offset *= 2)
    {
        T t;

        if(i>=offset)  t = x[i-offset];
        __syncthreads();

        if(i>=offset)  x[i] = t + x[i];
        __syncthreads();
    }
    return x[i];
}
```

FIGURE B.8.7 CUDA template for parallel plus-scan.

because addition is associative, we are free to change the order in which elements are added together. For instance, we can imagine adding pairs of consecutive elements in parallel, and then adding these partial sums, and so on.

One simple scheme for doing this is from Hillis and Steele [1989]. An implementation of their algorithm in CUDA is shown in Figure B.8.7. It assumes that the input array `x[ ]` contains exactly one element per thread of the thread block. It performs $\log_2 n$ iterations of a loop collecting partial sums together.

To understand the action of this loop, consider Figure B.8.8, which illustrates the simple case for n=8 threads and elements. Each level of the diagram represents one step of the loop. The lines indicate the location from which the data are being fetched. For each element of the output (i.e., the final row of the diagram) we are building a summation tree over the input elements. The edges highlighted in blue show the form of this summation tree for the final element. The leaves of this tree are all the initial elements. Tracing back from any output element shows that it incorporates all input values up to and including itself.

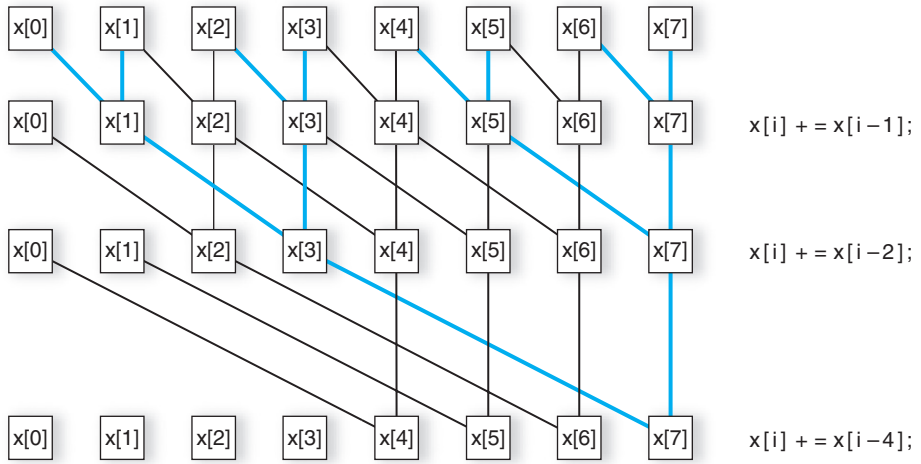


FIGURE B.8.8 Tree-based parallel scan data references.

While simple, this algorithm is not as efficient as we would like. Examining the serial implementation, we see that it performs *O(n)* additions. The parallel implementation, in contrast, performs *O(n* log *n)* additions. For this reason, it is not *work efficient*, since it does more work than the serial implementation to compute the same result. Fortunately, there are other techniques for implementing scan that are work-efficient. Details on more efficient implementation techniques and the extension of this per-block procedure to multiblock arrays are provided by Sengupta et al. [2007].

In some instances, we may only be interested in computing the sum of all elements in an array, rather than the sequence of all prefix sums returned by `scan`. This is the *parallel reduction* problem. We could simply use a scan algorithm to perform this computation, but reduction can generally be implemented more efficiently than scan.

Figure B.8.9 shows the code for computing a reduction using addition. In this example, each thread simply loads one element of the input sequence (i.e., it initially sums a subsequence of length 1). At the end of the reduction, we want thread 0 to hold the sum of all elements initially loaded by the threads of its block. The loop in this kernel implicitly builds a summation tree over the input elements, much like the scan algorithm above.

At the end of this loop, thread 0 holds the sum of all the values loaded by this block. If we want the final value of the location pointed to by `total` to contain the total of all elements in the array, we must combine the partial sums of all the blocks in the grid. One strategy to do this would be to have each block write its partial sum into a second array and then launch the reduction kernel again, repeating the process until we had reduced the sequence to a single value. A more attractive alternative supported by the Tesla GPU architecture is to use the `atomicAdd()` primitive, an efficient atomic

```
__global__
void plus_reduce(int *input, unsigned int N, int *total)
{
    unsigned int tid = threadIdx.x;
    unsigned int i   = blockIdx.x*blockDim.x + threadIdx.x;

    // Each block loads its elements into shared memory, padding
    // with 0 if N is not a multiple of blocksize
    __shared__ int x[blocksize];
    x[tid] = (i<N) ? input[i] : 0;
    __syncthreads();

    // Every thread now holds 1 input value in x[]
    //
    // Build summation tree over elements.
    for(int s=blockDim.x/2; s>0; s=s/2)
    {
        if(tid < s)  x[tid] += x[tid + s];
        __syncthreads();
    }

    // Thread 0 now holds the sum of all input values
    // to this block. Have it add that sum to the running total
    if( tid == 0 )  atomicAdd(total, x[tid]);
}
```

FIGURE B.8.9 CUDA implementation of plus-reduction.

read-modify-write primitive supported by the memory subsystem. This eliminates the need for additional temporary arrays and repeated kernel launches.

Parallel reduction is an essential primitive for parallel programming and highlights the importance of per-block shared memory and low-cost barriers in making cooperation among threads efficient. This degree of data shuffling among threads would be prohibitively expensive if done in off-chip global memory.

Radix Sort

One important application of scan primitives is in the implementation of sorting routines. The code in Figure B.8.10 implements a radix sort of integers across a single thread block. It accepts as input an array `values` containing one 32-bit integer for each thread of the block. For efficiency, this array should be stored in per-block shared memory, but this is not required for the sort to behave correctly.

This is a fairly simple implementation of radix sort. It assumes the availability of a procedure `partition_by_bit()` that will partition the given array such that

```
__device__ void radix_sort(unsigned int *values)
{
    for(int bit=0; bit<32; ++bit)
    {
        partition_by_bit(values, bit);
        __syncthreads();
    }
}
```

FIGURE B.8.10 CUDA code for radix sort.

```
__device__ void partition_by_bit(unsigned int *values,
                                 unsigned int bit)
{
    unsigned int i    = threadIdx.x;
    unsigned int size = blockDim.x;
    unsigned int x_i  = values[i];
    unsigned int p_i  = (x_i >> bit) & 1;

    values[i] = p_i;
    __syncthreads();

    // Compute number of T bits up to and including p_i.
    // Record the total number of F bits as well.
    unsigned int T_before = plus_scan(values);
    unsigned int T_total  = values[size-1];
    unsigned int F_total  = size - T_total;
    __syncthreads();

    // Write every x_i to its proper place
    if( p_i )
        values[T_before-1 + F_total] = x_i;
    else
        values[i - T_before] = x_i;
}
```

FIGURE B.8.11 CUDA code to partition data on a bit-by-bit basis, as part of radix sort.

all values with a 0 in the designated bit will come before all values with a 1 in that bit. To produce the correct output, this partitioning must be stable.

Implementing the partitioning procedure is a simple application of scan. Thread i holds the value x_i and must calculate the correct output index at which to write this value. To do so, it needs to calculate (1) the number of threads $j < i$ for which the designated bit is 1 and (2) the total number of bits for which the designated bit is 0. The CUDA code for `partition_by_bit()` is shown in Figure B.8.11.

A similar strategy can be applied for implementing a radix sort kernel that sorts an array of large length, rather than just a one-block array. The fundamental step remains the scan procedure, although when the computation is partitioned across multiple kernels, we must double-buffer the array of values rather than doing the partitioning in place. Details on performing radix sorts on large arrays efficiently are provided by Satish et al. [2008].

N-Body Applications on a GPU[1]

Nyland et al. [2007] describe a simple yet useful computational kernel with excellent GPU performance—the *all-pairs N-body* algorithm. It is a time-consuming component of many scientific applications. N-body simulations calculate the evolution of a system of bodies in which each body continuously interacts with every other body. One example is an astrophysical simulation in which each body represents an individual star, and the bodies gravitationally attract each other. Other examples are protein folding, where N-body simulation is used to calculate electrostatic and van der Waals forces; turbulent fluid flow simulation; and global illumination in computer graphics.

The all-pairs N-body algorithm calculates the total force on each body in the system by computing each pair-wise force in the system, summing for each body. Many scientists consider this method to be the most accurate, with the only loss of precision coming from the floating-point hardware operations. The drawback is its $O(n^2)$ computational complexity, which is far too large for systems with more than 10 bodies. To overcome this high cost, several simplifications have been proposed to yield $O(n \log n)$ and $O(n)$ algorithms; examples are the Barnes-Hut algorithm, the Fast Multipole Method and Particle-Mesh-Ewald summation. All of the *fast* methods still rely on the all-pairs method as a kernel for accurate computation of short-range forces; thus it continues to be important.

N-Body Mathematics

For gravitational simulation, calculate the body-body force using elementary physics. Between two bodies indexed by i and j, the 3D force vector is:

$$\mathbf{f}_{ij} = G\frac{m_i m_j}{||\ \mathbf{r}_{ij}\ ||^2} \times \frac{\mathbf{r}_{ij}}{||\ \mathbf{r}_{ij}\ ||}$$

The force magnitude is calculated in the left term, while the direction is computed in the right (unit vector pointing from one body to the other).

Given a list of interacting bodies (an entire system or a subset), the calculation is simple: for all pairs of interactions, compute the force and sum for each body. Once the total forces are calculated, they are used to update each body's position and velocity, based on the previous position and velocity. The calculation of the forces has complexity $O(n^2)$, while the update is $O(n)$.

[1] Adapted from Nyland et al. [2007], "Fast N-Body Simulation with CUDA," Chapter 31 of *GPU Gems 3*.

The serial force-calculation code uses two nested for-loops iterating over pairs of bodies. The outer loop selects the body for which the total force is being calculated, and the inner loop iterates over all the bodies. The inner loop calls a function that computes the pair-wise force, then adds the force into a running sum.

To compute the forces in parallel, we assign one thread to each body, since the calculation of force on each body is independent of the calculation on all other bodies. Once all of the forces are computed, the positions and velocities of the bodies can be updated.

The code for the serial and parallel versions is shown in Figure B.8.12 and Figure B.8.13. The serial version has two nested for-loops. The conversion to CUDA, like many other examples, converts the serial outer loop to a per-thread kernel where each thread computes the total force on a single body. The CUDA kernel computes a global thread ID for each thread, replacing the iterator variable of the serial outer loop. Both kernels finish by storing the total acceleration in a global array used to compute the new position and velocity values in a subsequent step. The outer loop is replaced by a CUDA kernel grid that launches *N* threads, one for each body.

```
void accel_on_all_bodies()
{
    int i, j;
    float3 acc(0.0f, 0.0f, 0.0f);

    for (i = 0; i < N; i++) {
       for (j = 0; j < N; j++) {
          acc = body_body_interaction(acc, body[i], body[j]);
       }
       accel[i] = acc;
    }
}
```

FIGURE B.8.12 Serial code to compute all pair-wise forces on N bodies.

```
__global__ void accel_on_one_body()
{
      int i = threadIdx.x + blockDim.x * blockIdx.x;
      int j;
      float3 acc(0.0f, 0.0f, 0.0f);

      for (j = 0; j < N; j++) {
          acc = body_body_interaction(acc, body[i], body[j]);
      }
      accel[i] = acc;
}
```

FIGURE B.8.13 CUDA thread code to compute the total force on a single body.

Optimization for GPU Execution

The CUDA code shown is functionally correct, but is not efficient, as it ignores key architectural features. Better performance can be achieved with three main optimizations. First, shared memory can be used to avoid identical memory reads between threads. Second, using multiple threads per body improves performance for small values of *N*. Third, loop unrolling reduces loop overhead.

Using Shared Memory

Shared memory can hold a subset of body positions, much like a cache, eliminating redundant global memory requests between threads. We optimize the code shown above to have each of *p* threads in a thread-block load *one* position into shared memory (for a total of *p* positions). Once all the threads have loaded a value into shared memory, ensured by `__syncthreads()`, each thread can then perform *p* interactions (using the data in shared memory). This is repeated *N*/*p* times to complete the force calculation for each body, which reduces the number of requests to memory by a factor of *p* (typically in the range 32–128).

The function called `accel_on_one_body()` requires a few changes to support this optimization. The modified code is shown in Figure B.8.14.

```
__shared__ float4 shPosition[256];
…
__global__ void accel_on_one_body()
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int j, k;
    int p = blockDim.x;
    float3 acc(0.0f, 0.0f, 0.0f);
    float4 myBody = body[i];

    for (j = 0; j < N; j += p) {  // Outer loops jumps by p each time
        shPosition[threadIdx.x] = body[j+threadIdx.x];
        __syncthreads();
        for (k = 0; k < p; k++) { // Inner loop accesses p positions
            acc = body_body_interaction(acc, myBody, shPosition[k]);
        }
        __syncthreads();
    }
    accel[i] = acc;
}
```

FIGURE B.8.14 CUDA code to compute the total force on each body, using shared memory to improve performance.

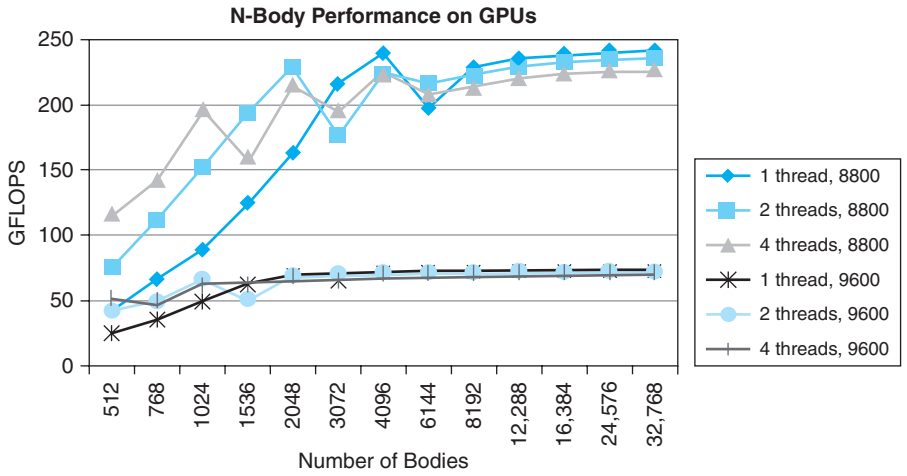


FIGURE B.8.15 Performance measurements of the N-body application on a GeForce 8800 GTX and a GeForce 9600. The 8800 has 128 stream processors at 1.35 GHz, while the 9600 has 64 at 0.80 GHz (about 30% of the 8800). The peak performance is 242 GFLOPS. For a GPU with more processors, the problem needs to be bigger to achieve full performance (the 9600 peak is around 2048 bodies, while the 8800 doesn't reach its peak until 16,384 bodies). For small N, more than one thread per body can significantly improve performance, but eventually incurs a performance penalty as N grows.

The loop that formerly iterated over all bodies now jumps by the block dimension p. Each iteration of the outer loop loads p successive positions into shared memory (one position per thread). The threads synchronize, and then p force calculations are computed by each thread. A second synchronization is required to ensure that new values are not loaded into shared memory prior to all threads completing the force calculations with the current data.

Using shared memory reduces the memory bandwidth required to less than 10% of the total bandwidth that the GPU can sustain (using less than 5 GB/s). This optimization keeps the application busy performing computation rather than waiting on memory accesses, as it would have done without the use of shared memory. The performance for varying values of N is shown in Figure B.8.15.

Using Multiple Threads per Body

Figure B.8.15 shows performance degradation for problems with small values of N (N< 4096) on the GeForce 8800 GTX. Many research efforts that rely on N-body calculations focus on small N (for long simulation times), making it a target of our optimization efforts. Our presumption to explain the lower performance was that there was simply not enough work to keep the GPU busy when N is small. The solution is to allocate more threads per body. We change the thread-block dimensions from $(p, 1, 1)$ to $(p, q, 1)$, where q threads divide the work of a single body into equal parts. By allocating the additional threads within the same thread block, partial results can be stored in shared memory. When all the force calculations are

done, the q partial results can be collected and summed to compute the final result. Using two or four threads per body leads to large improvements for small N.

As an example, the performance on the 8800 GTX jumps by 110% when $N = 1024$ (one thread achieves 90 GFLOPS, where four achieve 190 GFLOPS). Performance degrades slightly on large N, so we only use this optimization for N smaller than 4096. The performance increases are shown in Figure B.8.15 for a GPU with 128 processors and a smaller GPU with 64 processors clocked at two-thirds the speed.

Performance Comparison

The performance of the N-body code is shown in Figure B.8.15 and Figure B.8.16. In Figure B.8.15, performance of high- and medium-performance GPUs is shown, along with the performance improvements achieved by using multiple threads per body. The performance on the faster GPU ranges from 90 to just under 250 GFLOPS.

Figure B.8.16 shows nearly identical code (C++ versus CUDA) running on Intel Core2 CPUs. The CPU performance is about 1% of the GPU, in the range of 0.2 to 2 GFLOPS, remaining nearly constant over the wide range of problem sizes.

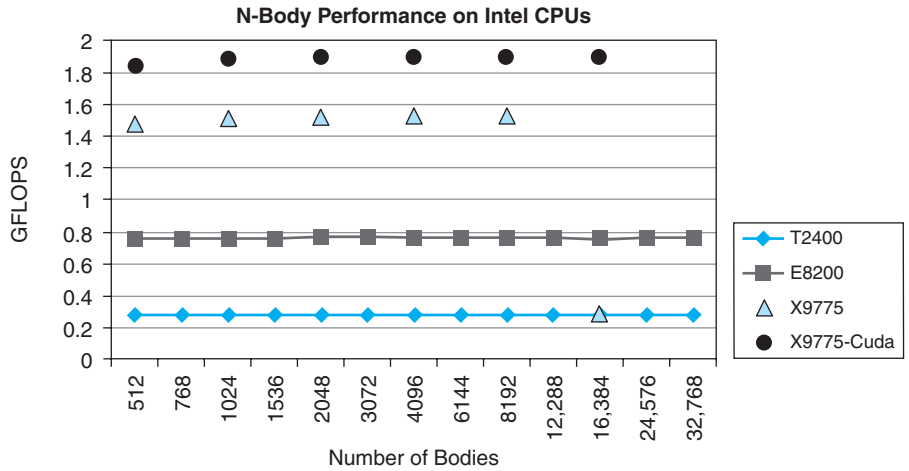


FIGURE B.8.16 Performance measurements on the N-body code on a CPU. The graph shows single precision N-body performance using Intel Core2 CPUs, denoted by their CPU model number. Note the dramatic reduction in GFLOPS performance (shown in GFLOPS on the y-axis), demonstrating how much faster the GPU is compared to the CPU. The performance on the CPU is generally independent of problem size, except for an anomalously low performance when N = 16,384 on the X9775 CPU. The graph also shows the results of running the CUDA version of the code (using the CUDA-for-CPU compiler) on a single CPU core, where it outperforms the C++ code by 24%. As a programming language, CUDA exposes parallelism and locality that a compiler can exploit. The Intel CPUs are a 3.2 GHz Extreme X9775 (code named "Penryn"), a 2.66 GHz E8200 (code named "Wolfdale"), a desktop, pre-Penryn CPU, and a 1.83 GHz T2400 (code named "Yonah"), a 2007 laptop CPU. The Penryn version of the Core 2 architecture is particularly interesting for N-body calculations with its 4-bit divider, allowing division and square root operations to execute four times faster than previous Intel CPUs.

The graph also shows the results of compiling the CUDA version of the code for a CPU, where the performance improves by 24%. CUDA, as a programming language, exposes parallelism, allowing the compiler to make better use of the SSE vector unit on a single core. The CUDA version of the N-body code naturally maps to multicore CPUs as well (with grids of blocks), where it achieves nearly perfect scaling on an eight-core system with N = 4096 (ratios of 2.0, 3.97, and 7.94 on two, four, and eight cores, respectively).

Results

With a modest effort, we developed a computational kernel that improves GPU performance over multicore CPUs by a factor of up to 157. Execution time for the N-body code running on a recent CPU from Intel (Penryn X9775 at 3.2 GHz, single core) took more than 3 seconds per frame to run the same code that runs at a 44 Hz frame rate on a GeForce 8800 GPU. On pre-Penryn CPUs, the code requires 6–16 seconds, and on older Core2 processors and Pentium IV processor, the time is about 25 seconds. We must divide the apparent increase in performance in half, as the CPU requires only half as many calculations to compute the same result (using the optimization that the forces on a pair of bodies are equal in strength and opposite in direction).

How can the GPU speed up the code by such a large amount? The answer requires inspecting architectural details. The pair-wise force calculation requires 20 floating-point operations, comprised mostly of addition and multiplication instructions (some of which can be combined using a multiply-add instruction), but there are also division and square root instructions for vector normalization. Intel CPUs take many cycles for single-precision division and square root instructions,[2] although this has improved in the latest Penryn CPU family with its faster 4-bit divider.[3] Additionally, the limitations in register capacity lead to many MOV instructions in the x86 code (presumably to/from L1 cache). In contrast, the GeForce 8800 executes a reciprocal square-root thread instruction in four clocks; see Section B.6 for special function accuracy. It has a larger register file (per thread) and shared memory that can be accessed as an instruction operand. Finally, the CUDA compiler emits 15 instructions for one iteration of the loop, compared with more than 40 instructions from a variety of x86 CPU compilers. Greater parallelism, faster execution of complex instructions, more register space, and an efficient compiler all combine to explain the dramatic performance improvement of the N-body code between the CPU and the GPU.

---

[2] The x86 SSE instructions reciprocal-square-root (RSQRT*) and reciprocal (RCP*) were not considered, as their accuracy is too low to be comparable.

[3] Intel Corporation, *Intel 64 and IA-32 Architectures Optimization Reference Manual*. November 2007. Order Number: 248966-016. Also available at www.intel.com/design/processor/manuals/248966.pdf.

On a GeForce 8800, the all-pairs N-body algorithm delivers more than 240 GFLOPS of performance, compared to less than 2 GFLOPS on recent sequential processors. Compiling and executing the CUDA version of the code on a CPU demonstrates that the problem scales well to multicore CPUs, but is still significantly slower than a single GPU.

We coupled the GPU N-body simulation with a graphical display of the motion, and can interactively display 16K bodies interacting at 44 frames per second. This allows astrophysical and biophysical events to be displayed and navigated at interactive rates. Additionally, we can parameterize many settings, such as noise reduction, damping, and integration techniques, immediately displaying their effects on the dynamics of the system. This provides scientists with stunning visual imagery, boosting their insights on otherwise invisible systems (too large or small, too fast or too slow), allowing them to create better models of physical phenomena.

Figure B.8.17 shows a time-series display of an astrophysical simulation of 16K bodies, with each body acting as a galaxy. The initial configuration is a spherical shell

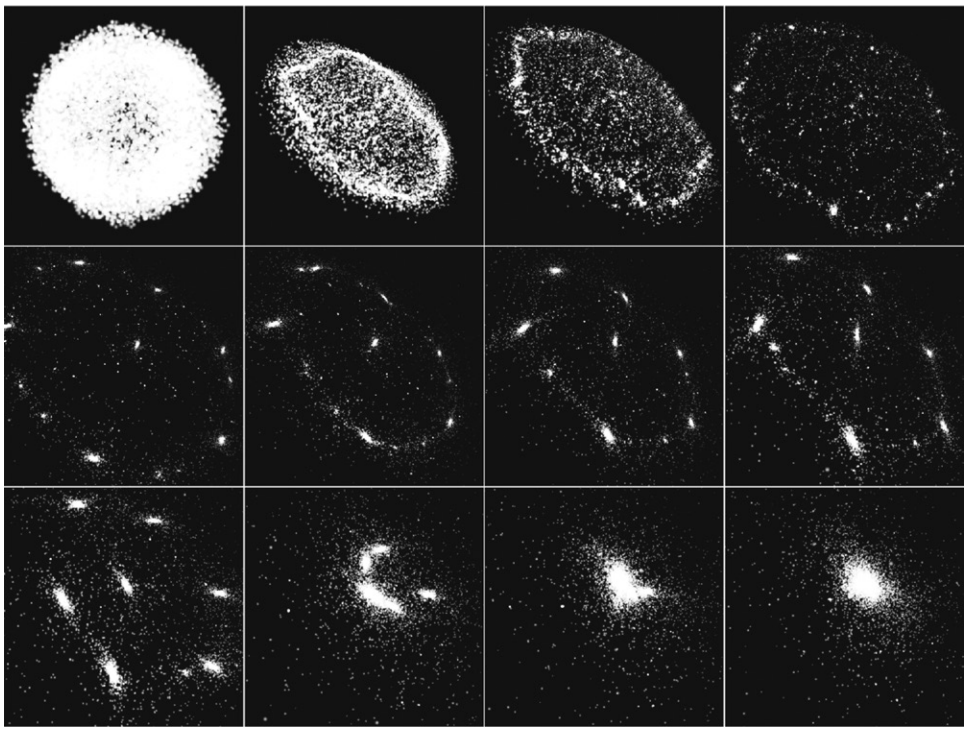


FIGURE B.8.17 Twelve images captured during the evolution of an N-body system with 16,384 bodies.

of bodies rotating about the *z*-axis. One phenomenon of interest to astrophysicists is the clustering that occurs, along with the merging of galaxies over time. For the interested reader, the CUDA code for this application is available in the CUDA SDK from www.nvidia.com/CUDA.

B.9 Fallacies and Pitfalls

GPUs have evolved and changed so rapidly that many fallacies and pitfalls have arisen. We cover a few here.

Fallacy GPUs are just SIMD vector multiprocessors.

It is easy to draw the false conclusion that GPUs are simply SIMD vector multiprocessors. GPUs do have a SPMD-style programming model, in that a programmer can write a single program that is executed in multiple thread instances with multiple data. The execution of these threads is not purely SIMD or vector, however; it is *single-instruction multiple-thread* (SIMT), described in Section B.4. Each GPU thread has its own scalar registers, thread private memory, thread execution state, thread ID, independent execution and branch path, and effective program counter, and can address memory independently. Although a group of threads (e.g., a warp of 32 threads) executes more efficiently when the PCs for the threads are the same, this is not necessary. So, the multiprocessors are not purely SIMD. The thread execution model is MIMD with barrier synchronization and SIMT optimizations. Execution is more efficient if individual thread load/store memory accesses can be coalesced into block accesses, as well. However, this is not strictly necessary. In a purely SIMD vector architecture, memory/register accesses for different threads must be aligned in a regular vector pattern. A GPU has no such restriction for register or memory accesses; however, execution is more efficient if warps of threads access local blocks of data.

In a further departure from a pure SIMD model, an SIMT GPU can execute more than one warp of threads concurrently. In graphics applications, there may be multiple groups of vertex programs, pixel programs, and geometry programs running in the multiprocessor array concurrently. Computing programs may also execute different programs concurrently in different warps.

Fallacy GPU performance cannot grow faster than Moore's law.

Moore's law is simply a rate. It is not a "speed of light" limit for any other rate. Moore's law describes an expectation that, over time, as semiconductor technology advances and transistors become smaller, the manufacturing cost per transistor will decline exponentially. Put another way, given a constant manufacturing cost, the

number of transistors will increase exponentially. Gordon Moore [1965] predicted that this progression would provide roughly two times the number of transistors for the same manufacturing cost every year, and later revised it to doubling every 2 years. Although Moore made the initial prediction in 1965 when there were just 50 components per integrated circuit, it has proved remarkably consistent. The reduction of transistor size has historically had other benefits, such as lower power per transistor and faster clock speeds at constant power.

This increasing bounty of transistors is used by chip architects to build processors, memory, and other components. For some time, CPU designers have used the extra transistors to increase processor performance at a rate similar to Moore's law, so much so that many people think that processor performance growth of two times every 18–24 months is Moore's law. In fact, it is not.

Microprocessor designers spend some of the new transistors on processor cores, improving the architecture and design, and pipelining for more clock speed. The rest of the new transistors are used for providing more cache, to make memory access faster. In contrast, GPU designers use almost none of the new transistors to provide more cache; most of the transistors are used for improving the processor cores and adding more processor cores.

GPUs get faster by four mechanisms. First, GPU designers reap the Moore's law bounty directly by applying exponentially more transistors to building more parallel, and thus faster, processors. Second, GPU designers can improve on the architecture over time, increasing the efficiency of the processing. Third, Moore's law assumes constant cost, so the Moore's law rate can clearly be exceeded by spending more for larger chips with more transistors. Fourth, GPU memory systems have increased their effective bandwidth at a pace nearly comparable to the processing rate, by using faster memories, wider memories, data compression, and better caches. The combination of these four approaches has historically allowed GPU performance to double regularly, roughly every 12 to 18 months. This rate, exceeding the rate of Moore's law, has been demonstrated on graphics applications for approximately 10 years and shows no sign of significant slowdown. The most challenging rate limiter appears to be the memory system, but competitive innovation is advancing that rapidly too.

Fallacy GPUs only render 3D graphics; they can't do general computation.

GPUs are built to render 3D graphics as well as 2D graphics and video. To meet the demands of graphics software developers as expressed in the interfaces and performance/feature requirements of the graphics APIs, GPUs have become massively parallel programmable floating-point processors. In the graphics domain, these processors are programmed through the graphics APIs and with arcane graphics programming languages (GLSL, Cg, and HLSL, in OpenGL and Direct3D). However, there is nothing preventing GPU architects from exposing

the parallel processor cores to programmers without the graphics API or the arcane graphics languages.

In fact, the Tesla architecture family of GPUs exposes the processors through a software environment known as CUDA, which allows programmers to develop general application programs using the C language and soon C++. GPUs are Turing-complete processors, so they can run any program that a CPU can run, although perhaps less well. And perhaps faster.

Fallacy GPUs cannot run double-precision floating-point programs fast.

In the past, GPUs could not run double-precision floating-point programs at all, except through software emulation. And that's not very fast at all. GPUs have made the progression from indexed arithmetic representation (lookup tables for colors) to 8-bit integers per color component, to fixed-point arithmetic, to single-precision floating-point, and recently added double precision. Modern GPUs perform virtually all calculations in single-precision IEEE floating-point arithmetic, and are beginning to use double precision in addition.

For a small additional cost, a GPU can support double-precision floating-point as well as single-precision floating-point. Today, double-precision runs more slowly than the single-precision speed, about five to ten times slower. For incremental additional cost, double-precision performance can be increased relative to single precision in stages, as more applications demand it.

Fallacy GPUs don't do floating-point correctly.

GPUs, at least in the Tesla architecture family of processors, perform single-precision floating-point processing at a level prescribed by the IEEE 754 floating-point standard. So, in terms of accuracy, GPUs are the equal of any other IEEE 754-compliant processors.

Today, GPUs do not implement some of the specific features described in the standard, such as handling denormalized numbers and providing precise floating-point exceptions. However, the recently introduced Tesla T10P GPU provides full IEEE rounding, fused-multiply-add, and denormalized number support for double precision.

Pitfall Just use more threads to cover longer memory latencies.

CPU cores are typically designed to run a single thread at full speed. To run at full speed, every instruction and its data need to be available when it is time for that instruction to run. If the next instruction is not ready or the data required for that instruction is not available, the instruction cannot run and the processor stalls. External memory is distant from the processor, so it takes many cycles of wasted execution to fetch data from memory. Consequently, CPUs require large local

caches to keep running without stalling. Memory latency is long, so it is avoided by striving to run in the cache. At some point, program working set demands may be larger than any cache. Some CPUs have used multithreading to tolerate latency, but the number of threads per core has generally been limited to a small number.

The GPU strategy is different. GPU cores are designed to run many threads concurrently, but only one instruction from any thread at a time. Another way to say this is that a GPU runs each thread slowly, but in aggregate runs the threads efficiently. Each thread can tolerate some amount of memory latency, because other threads can run.

The downside of this is that multiple—many multiple threads—are required to cover the memory latency. In addition, if memory accesses are scattered or not correlated among threads, the memory system will get progressively slower in responding to each individual request. Eventually, even the multiple threads will not be able to cover the latency. So, the pitfall is that for the "just use more threads" strategy to work for covering latency, you have to have enough threads, and the threads have to be well-behaved in terms of locality of memory access.

Fallacy O(n) algorithms are difficult to speed up.

No matter how fast the GPU is at processing data, the steps of transferring data to and from the device may limit the performance of algorithms with O(n) complexity (with a small amount of work per datum). The highest transfer rate over the PCIe bus is approximately 48 GB/second when DMA transfers are used, and slightly less for nonDMA transfers. The CPU, in contrast, has typical access speeds of 8–12 GB/second to system memory. Example problems, such as vector addition, will be limited by the transfer of the inputs to the GPU and the returning output from the computation.

There are three ways to overcome the cost of transferring data. First, try to leave the data on the GPU for as long as possible, instead of moving the data back and forth for different steps of a complicated algorithm. CUDA deliberately leaves data alone in the GPU between launches to support this.

Second, the GPU supports the concurrent operations of copy-in, copy-out and computation, so data can be streamed in and out of the device while it is computing. This model is useful for any data stream that can be processed as it arrives. Examples are video processing, network routing, data compression/decompression, and even simpler computations such as large vector mathematics.

The third suggestion is to use the CPU and GPU together, improving performance by assigning a subset of the work to each, treating the system as a heterogeneous computing platform. The CUDA programming model supports allocation of work to one or more GPUs along with continued use of the CPU without the use of threads (via asynchronous GPU functions), so it is relatively simple to keep all GPUs and a CPU working concurrently to solve problems even faster.

B.10 Concluding Remarks

GPUs are massively parallel processors and have become widely used, not only for 3D graphics, but also for many other applications. This wide application was made possible by the evolution of graphics devices into programmable processors. The graphics application programming model for GPUs is usually an API such as DirectX™ or OpenGL™. For more general-purpose computing, the CUDA programming model uses an SPMD (*single-program multiple data*) style, executing a program with many parallel threads.

GPU parallelism will continue to scale with Moore's law, mainly by increasing the number of processors. Only the parallel programming models that can readily scale to hundreds of processor cores and thousands of threads will be successful in supporting manycore GPUs and CPUs. Also, only those applications that have many largely independent parallel tasks will be accelerated by massively parallel manycore architectures.

Parallel programming models for GPUs are becoming more flexible, for both graphics and parallel computing. For example, CUDA is evolving rapidly in the direction of full C/C++ functionality. Graphics APIs and programming models will likely adapt parallel computing capabilities and models from CUDA. Its SPMD-style threading model is scalable, and is a convenient, succinct, and easily learned model for expressing large amounts of parallelism.

Driven by these changes in the programming models, GPU architecture is in turn becoming more flexible and more programmable. GPU fixed-function units are becoming accessible from general programs, along the lines of how CUDA programs already use texture intrinsic functions to perform texture lookups using the GPU texture instruction and texture unit.

GPU architecture will continue to adapt to the usage patterns of both graphics and other application programmers. GPUs will continue to expand to include more processing power through additional processor cores, as well as increasing the thread and memory bandwidth available for programs. In addition, the programming models must evolve to include programming heterogeneous manycore systems including both GPUs and CPUs.

Acknowledgments

This appendix is the work of several authors at NVIDIA. We gratefully acknowledge the significant contributions of Michael Garland, John Montrym, Doug Voorhies, Lars Nyland, Erik Lindholm, Paulius Micikevicius, Massimiliano Fatica, Stuart Oberman, and Vasily Volkov.

B.11 Historical Perspective and Further Reading

Graphics Pipeline Evolution

3D graphics pipeline hardware evolved from the large expensive systems of the early 1980s to small workstations and then to PC accelerators in the mid- to late-1990s. During this period, three major transitions occurred:

- Performance-leading graphics subsystems declined in price from $50,000 to $200.
- Performance increased from 50 million pixels per second to 1 billion pixels per second and from 100,000 vertices per second to 10 million vertices per second.
- Native hardware capabilities evolved from wireframe (polygon outlines) to flat shaded (constant color) filled polygons, to smooth shaded (interpolated color) filled polygons, to full-scene anti-aliasing with texture mapping and rudimentary multitexturing.

Fixed-Function Graphics Pipelines

Throughout this period, graphics hardware was configurable, but not programmable by the application developer. With each generation, incremental improvements were offered. But developers were growing more sophisticated and asking for more new features than could be reasonably offered as built-in fixed functions. The NVIDIA GeForce 3, described by Lindholm et al. [2001], took the first step toward true general shader programmability. It exposed to the application developer what had been the private internal instruction set of the floating-point vertex engine. This coincided with the release of Microsoft's DirectX 8 and OpenGL's vertex shader extensions. Later GPUs, at the time of DirectX 9, extended general programmability and floating point capability to the pixel fragment stage, and made texture available at the vertex stage. The ATI Radeon 9700, introduced in 2002, featured a programmable 24-bit floating-point pixel fragment processor programmed with DirectX 9 and OpenGL. The GeForce FX added 32-bit floating-point pixel processors. This was part of a general trend toward unifying the functionality of the different stages, at least as far as the application programmer was concerned. NVIDIA's GeForce 6800 and 7800 series were built with separate processor designs and separate hardware dedicated to the vertex and to the fragment processing. The XBox 360 introduced an early unified processor GPU in 2005, allowing vertex and pixel shaders to execute on the same processor.

Evolution of Programmable Real-Time Graphics

During the last 30 years, graphics architecture has evolved from a simple pipeline for drawing wireframe diagrams to a highly parallel design consisting of several deep parallel pipelines capable of rendering complex interactive imagery that appears three-dimensional. Concurrently, many of the calculations involved became far more sophisticated and user-programmable.

In these graphics pipelines, certain stages do a great deal of floating-point arithmetic on completely independent data, such as transforming the position of triangle vertexes or generating pixel colors. This data independence is a key difference between GPUs and CPUs. A single frame, rendered in 1/60th of a second, might have 1 million triangles and 6 million pixels. The opportunity to use hardware parallelism to exploit this data independence is tremendous.

The specific functions executed at a few graphics pipeline stages vary with rendering algorithms and have evolved to be programmable. Vertex programs map the position of triangle vertices on to the screen, altering their position, color, or orientation. Typically a vertex shader thread inputs a floating-point (x, y, z, w) vertex position and computes a floating-point (x, y, z) screen position. Geometry programs operate on primitives defined by multiple vertices, changing them or generating additional primitives. Pixel fragment shaders each "shade" one pixel, computing a floating-point *red, green, blue, alpha* (RGBA) color contribution to the rendered image at its pixel sample (x, y) image position. For all three types of graphics shaders, program instances can be run in parallel, because each works on independent data, produces independent results, and has no side effects.

Between these programmable graphics pipeline stages are dozens of fixed-function stages which perform well-defined tasks far more efficiently than a programmable processor could and which would benefit far less from programmability. For example, between the geometry processing stage and the pixel processing stage is a "rasterizer," a complex state machine that determines exactly which pixels (and portions thereof) lie within each geometric primitive's boundaries. Together, the mix of programmable and fixed-function stages is engineered to balance extreme performance with user control over the rendering algorithms.

Common rendering algorithms perform a single pass over input primitives and access other memory resources in a highly coherent manner; these algorithms provide excellent bandwidth utilization and are largely insensitive to memory latency. Combined with a pixel shader workload that is usually compute-limited, these characteristics have guided GPUs along a different evolutionary path than CPUs. Whereas CPU die area is dominated by cache memory, GPUs are dominated by floating-point datapath and fixed-function logic. GPU memory interfaces emphasize bandwidth over latency (since latency can be readily hidden by a high thread count); indeed, bandwidth is typically many times higher than a CPU, exceeding 100 GB/second in some cases. The far-higher number of fine-grained lightweight threads effectively exploits the rich parallelism available.

Beginning with NVIDIA's GeForce 8800 GPU in 2006, the three programmable graphics stages are mapped to an array of unified processors; the logical graphics pipeline is physically a recirculating path that visits these processors three times, with much fixed-function graphics logic between visits. Since different rendering algorithms present wildly different loads among the three programmable stages, this unification provides processor load balancing.

Unified Graphics and Computing Processors

By the DirectX 10 generation, the functionality of vertex and pixel fragment shaders was to be made identical to the programmer, and in fact a new logical stage was introduced, the geometry shader, to process all the vertices of a primitive rather than vertices in isolation. The GeForce 8800 was designed with DirectX 10 in mind. Developers were coming up with more sophisticated shading algorithms, and this motivated a sharp increase in the available shader operation rate, particularly floating-point operations. NVIDIA chose to pursue a processor design with higher operating frequency than standard-cell methodologies had allowed, to deliver the desired operation throughput as area-efficiently as possible. High-clock-speed design requires substantially more engineering effort, and this favored designing one processor, rather than two (or three, given the new geometry stage). It became worthwhile to take on the engineering challenges of a unified processor (load balancing and recirculation of a logical pipeline onto threads of the processor array) to get the benefits of one processor design.

GPGPU: an Intermediate Step

As DirectX 9-capable GPUs became available, some researchers took notice of the raw performance growth path of GPUs and began to explore the use of GPUs to solve complex parallel problems. DirectX 9 GPUs had been designed only to match the features required by the graphics API. To access the computational resources, a programmer had to cast their problem into native graphics operations. For example, to run many simultaneous instances of a pixel shader, a triangle had to be issued to the GPU (with clipping to a rectangle shape if that's what was desired). Shaders did not have the means to perform arbitrary scatter operations to memory. The only way to write a result to memory was to emit it as a pixel color value, and configure the framebuffer operation stage to write (or blend, if desired) the result to a two-dimensional framebuffer. Furthermore, the only way to get a result from one pass of computation to the next was to write all parallel results to a pixel framebuffer, then use that framebuffer as a texture map as input to the pixel fragment shader of the next stage of the computation. Mapping general computations to a GPU in this era was quite awkward. Nevertheless, intrepid researchers demonstrated a handful of useful applications with painstaking efforts. This field was called "GPGPU" for general purpose computing on GPUs.

GPU Computing

While developing the Tesla architecture for the GeForce 8800, NVIDIA realized its potential usefulness would be much greater if programmers could think of the GPU as a processor. NVIDIA selected a programming approach in which programmers would explicitly declare the data-parallel aspects of their workload.

For the DirectX 10 generation, NVIDIA had already begun work on a high-efficiency floating-point and integer processor that could run a variety of simultaneous workloads to support the logical graphics pipeline. This processor was designed to take advantage of the common case of groups of threads executing the same code path. NVIDIA added memory load and store instructions with integer byte addressing to support the requirements of compiled C programs. It introduced the thread block (cooperative thread array), grid of thread blocks, and barrier synchronization to dispatch and manage highly parallel computing work. Atomic memory operations were added. NVIDIA developed the CUDA C/C++ compiler, libraries, and runtime software to enable programmers to readily access the new data-parallel computation model and develop applications.

Scalable GPUs

Scalability has been an attractive feature of graphics systems from the beginning. Workstation graphics systems gave customers a choice in pixel horsepower by varying the number of pixel processor circuit boards installed. Prior to the mid-1990s PC graphics scaling was almost nonexistent. There was one option—the VGA controller. As 3D-capable accelerators appeared, the market had room for a range of offerings. 3dfx introduced multiboard scaling with the original SLI (*Scan Line Interleave*) on their Voodoo2, which held the performance crown for its time (1998). Also in 1998, NVIDIA introduced distinct products as variants on a single architecture with Riva TNT Ultra (high-performance) and Vanta (low-cost), first by speed binning and packaging, then with separate chip designs (GeForce 2 GTS & GeForce 2 MX). At present, for a given architecture generation, four or five separate GPU chip designs are needed to cover the range of desktop PC performance and price points. In addition, there are separate segments in notebook and workstation systems. After acquiring 3dfx, NVIDIA continued the multi-GPU SLI concept in 2004, starting with GeForce 6800—providing multi-GPU scalability transparently to the programmer and to the user. Functional behavior is identical across the scaling range; one application will run unchanged on any implementation of an architectural family.

CPUs are scaling to higher transistor counts by increasing the number of constant-performance cores on a die, rather than increasing the performance of a single core. At this writing the industry is transitioning from dual-core to quad-core, with eight-core not far behind. Programmers are forced to find fourfold to eightfold task parallelism to fully utilize these processors, and applications using task parallelism must be rewritten frequently to target each successive doubling of

core count. In contrast, the highly multithreaded GPU encourages the use of many-fold data parallelism and thread parallelism, which readily scales to thousands of parallel threads on many processors. The GPU scalable parallel programming model for graphics and parallel computing is designed for transparent and portable scalability. A graphics program or CUDA program is written once and runs on a GPU with any number of processors. As shown in Section B.1, a CUDA programmer explicitly states both fine-grained and coarse-grained parallelism in a thread program by decomposing the problem into grids of thread blocks—the same program will run efficiently on GPUs or CPUs of any size in current and future generations as well.

Recent Developments

Academic and industrial work on applications using CUDA has produced hundreds of examples of successful CUDA programs. Many of these programs run the application tens or hundreds of times faster than multicore CPUs are capable of running them. Examples include n-body simulation, molecular modeling, computational finance, and oil and gas exploration data processing. Although many of these use single-precision floating-point arithmetic, some problems require double precision. The recent arrival of double-precision floating-point in GPUs enables an even broader range of applications to benefit from GPU acceleration.

For a comprehensive list and examples of current developments in applications that are accelerated by GPUs, visit CUDAZone: www.nvidia.com/CUDA.

Future Trends

Naturally, the number of processor cores will continue to increase in proportion to increases in available transistors as silicon processes improve. In addition, GPUs will continue to enjoy vigorous architectural evolution. Despite their demonstrated high performance on data-parallel applications, GPU core processors are still of relatively simple design. More aggressive techniques will be introduced with each successive architecture to increase the actual utilization of the calculating units. Because scalable parallel computing on GPUs is a new field, novel applications are rapidly being created. By studying them, GPU designers will discover and implement new machine optimizations.

Further Reading

Akeley, K. and T. Jermoluk [1988]. "High-Performance Polygon Rendering," *Proc. SIGGRAPH 1988* (August), 239–46.

Akeley, K. [1993]. "RealityEngine Graphics." *Proc. SIGGRAPH 1993* (August), 109–16.

Blelloch, G. B. [1990]. "Prefix Sums and Their Applications". In John H. Reif (Ed.), *Synthesis of Parallel Algorithms*, Morgan Kaufmann Publishers, San Francisco.

Blythe, D. [2006]. "The Direct3D 10 System", *ACM Trans. Graphics* Vol. 25, no. 3 (July), 724–34.

Buck, I., T. Foley, D. Horn, J. Sugerman, K. Fatahlian, M. Houston, and P. Hanrahan [2004]. "Brook for GPUs: Stream Computing on Graphics Hardware." *Proc. SIGGRAPH 2004*, 777–86, August. http://doi.acm.org/10.1145/1186562.1015800.

Elder, G. [2002] "Radeon 9700." Eurographics/SIGGRAPH Workshop on Graphics Hardware, Hot3D Session, www.graphicshardware.org/previous/www_2002/presentations/Hot3D-RADEON9700.ppt.

Fernando, R. and M. J. Kilgard [2003]. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*, Addison-Wesley, Reading, MA.

Fernando, R. (Ed.), [2004]. *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*, Addison-Wesley, Reading, MA. https://developer.nvidia.com/gpugems/GPUGems/gpugems_pref01.html.

Foley, J., A. van Dam, S. Feiner, and J. Hughes [1995]. *Computer Graphics: Principles and Practice, second edition in C*, Addison-Wesley, Reading, MA.

Hillis, W. D. and G. L. Steele [1986]. "Data parallel algorithms." *Commun. ACM* 29, 12 (Dec.), 1170–83. http://doi.acm.org/10.1145/7902.7903.

IEEE Std 754-2008 [2008]. *IEEE Standard for Floating-Point Arithmetic*. ISBN 978-0-7381-5752-8, STD95802, http://ieeexplore.ieee.org/servlet/opac?punumber=4610933 (Aug. 29).

Industrial Light and Magic [2003]. *OpenEXR*, www.openexr.com.

Intel Corporation [2007]. *Intel 64 and IA-32 Architectures Optimization Reference Manual.* November. Order Number: 248966-016. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

Kessenich, J. [2006]. *The OpenGL Shading Language, Language Version 1.20, Sept. 2006.* www.opengl.org/documentation/specs/.

Kirk, D. and D. Voorhies [1990]. "The Rendering Architecture of the DN10000VS." *Proc. SIGGRAPH 1990* (August), 299–307.

Lindholm E., M.J. Kilgard, and H. Moreton [2001]. "A User- Programmable Vertex Engine." *Proc. SIGGRAPH 2001* (August), 149–58.

Lindholm, E., J. Nickolls, S. Oberman, and J. Montrym [2008]. "NVIDIA Tesla: A Unified Graphics and Computing Architecture", *IEEE Micro* Vol. 28, no. 2 (March–April), 39–55.

Microsoft Corporation. Microsoft DirectX Specification, https://msdn.microsoft.com/en-us/library/windows/apps/hh452744.aspx.

Microsoft Corporation [2003]. Microsoft DirectX 9 Programmable Graphics Pipeline, Microsoft Press, Redmond, WA.

Montrym, J., D. Baum, D. Dignam, and C. Migdal [1997]. "InfiniteReality: A Real-Time Graphics System." *Proc. SIGGRAPH 1997* (August), 293–301.

Montrym, J. and H. Moreton [2005]. "The GeForce 6800", *IEEE Micro*, Vol. 25, no. 2 (March–April), 41–51.

Moore, G. E. [1965]. "Cramming more components onto integrated circuits", *Electronics*, Vol. 38, no. 8 (April 19).

Nguyen, H. (Ed.), [2008]. *GPU Gems 3*, Addison-Wesley, Reading, MA.

Nickolls, J., I. Buck, M. Garland, and K. Skadron [2008]. "Scalable Parallel Programming with CUDA", *ACM Queue* Vol. 6, no. 2 (March–April) 40–53.

NVIDIA [2007]. CUDA Zone. http://www.nvidia.com/object/cuda_home_new.html.

NVIDIA [2007]. *CUDA Programming Guide 1.1*. https://developer.nvidia.com/nvidia-gpu-programming-guide.

NVIDIA [2007]. *PTX: Parallel Thread Execution ISA version 1.1*. www.nvidia.com/object/io_1195170102263.html.

Nyland, L., M. Harris, and J. Prins [2007]. "Fast N-Body Simulation with CUDA." In H. Nguyen (Ed.), *GPU Gems 3,* Addison-Wesley, Reading, MA.

Oberman, S. F. and M. Y. Siu [2005]. "A High-Performance Area- Efficient Multifunction Interpolator," *Proc. Seventeenth IEEE Symp. Computer Arithmetic*, 272–79.

Patterson, D. A. and J. L. Hennessy [2004]. *Computer Organization and Design: The Hardware/Software Interface*, third edition, Morgan Kaufmann Publishers, San Francisco.

Pharr, M. ed. [2005]. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Addison-Wesley, Reading, MA.

Satish, N., M. Harris, and M. Garland [2008]. "Designing Efficient Sorting Algorithms for Manycore GPUs," NVIDIA Technical Report NVR-2008-001.

Segal, M. and K. Akeley [2006]. *The OpenGL Graphics System: A Specification, Version 2.1, Dec. 1, 2006*. www.opengl.org/documentation/specs/.

Sengupta, S., M. Harris, Y. Zhang, and J. D. Owens [2007]. "Scan Primitives for GPU Computing." In *Proc. of Graphics Hardware 2007* (August), 97–106.

Volkov, V. and J. Demmel [2008]. "LU, QR and Cholesky Factorizations using Vector Capabilities of GPUs," Technical Report No. UCB/EECS-2008-49, 1–11. http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-49.pdf.

Williams, S., L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel [2007]. "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," In *Proc. Supercomputing* 2007, November.

C

APPENDIX

Mapping Control to Hardware

A custom format such as this is slave to the architecture of the hardware and the instruction set it serves. The format must strike a proper compromise between ROM size, ROM-output decoding, circuitry size, and machine execution rate.

Jim McKevit, et al.
8086 design report, 1997

C.1 **Introduction** C-3
C.2 **Implementing Combinational Control Units** C-4
C.3 **Implementing Finite-State Machine Control** C-8
C.4 **Implementing the Next-State Function with a Sequencer** C-22

C.5 Translating a Microprogram to Hardware C-28
C.6 Concluding Remarks C-32
C.7 Exercises C-33

C.1 Introduction

Control typically has two parts: a combinational part that lacks state and a sequential control unit that handles sequencing and the main control in a multicycle design. Combinational control units are often used to handle part of the decode and control process. The ALU control in Chapter 4 is such an example. A single-cycle implementation like that in Chapter 4 can also use a combinational controller, since it does not require multiple states. Section C.2 examines the implementation of these two combinational units from the truth tables of Chapter 4.

Since sequential control units are larger and often more complex, there are a wider variety of techniques for implementing a sequential control unit. The usefulness of these techniques depends on the complexity of the control, characteristics such as the average number of next states for any given state, and the implementation technology.

The most straightforward way to implement a sequential control function is with a block of logic that takes as inputs the current state and the opcode field of the Instruction register and produces as outputs the datapath control signals and the value of the next state. The initial representation may be either a finite-state diagram or a microprogram. In the latter case, each microinstruction represents a state.

In an implementation using a finite-state controller, the next-state function will be computed with logic. Section C.3 constructs such an implementation both for a ROM and a PLA.

An alternative method of implementation computes the next-state function by using a counter that increments the current state to determine the next state. When the next state doesn't follow sequentially, other logic is used to determine the state. Section C.4 explores this type of implementation and shows how it can be used to implement finite-state control.

In Section C.5, we show how a microprogram representation of sequential control is translated to control logic.

C.2 Implementing Combinational Control Units

In this section, we show how the ALU control unit and main control unit for the single clock design are mapped down to the gate level. With modern *computer-aided design* (CAD) systems, this process is completely mechanical. The examples illustrate how a CAD system takes advantage of the structure of the control function, including the presence of don't-care terms.

Mapping the ALU Control Function to Gates

Figure C.2.1 shows the truth table for the ALU control function that was developed in Chapter 4, Section 4.4. A logic block that implements this ALU control function will have four distinct outputs (called Operation3, Operation2, Operation1, and Operation0), each corresponding to one of the four bits of the ALU control in the last column of Figure C.2.1. The logic function for each output is constructed by combining all the truth table entries that set that particular output. For example, the low-order bit of the ALU control (Operation0) is set by the last two entries of the truth table in Figure C.2.1. Thus, the truth table for Operation0 will have these two entries.

Figure C.2.2 shows the truth tables for each of the four ALU control bits. We have taken advantage of the common structure in each truth table to incorporate additional don't cares. For example, the five lines in the truth table of Figure C.2.1 that set Operation1 are reduced to just two entries in Figure C.2.2. A logic minimization program will use the don't-care terms to reduce the number of gates and the number of inputs to each gate in a logic gate realization of these truth tables.

A confusing aspect of Figure C.2.2 is that there is no logic function for Operation3. That is because this control line is only used for the NOR operation, which is not needed for the RISC-V subset in Figure 4.12.

From the simplified truth table in Figure C.2.2, we can generate the logic shown in Figure C.2.3, which we call the *ALU control block*. This process is straightforward

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 C.2.1 The truth table for the four ALU control bits (called Operation) as a function of the ALUOp and function code field. This table is the same as that shown in Figure 4.13.

ALUOp		Function code fields					
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0
0	1	X	X	X	X	X	X
1	X	X	X	X	X	1	X

a. The truth table for Operation2 = 1 (this table corresponds to the second to left bit of the Operation field in Figure C.2.1)

ALUOp		Function code fields					
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0
0	X	X	X	X	X	X	X
X	X	X	X	X	0	X	X

b. The truth table for Operation1 = 1

ALUOp		Function code fields					
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0
1	X	X	X	X	X	X	1
1	X	X	X	1	X	X	X

c. The truth table for Operation0 = 1

FIGURE C.2.2 The truth tables for three ALU control lines. Only the entries for which the output is 1 are shown. The bits in each field are numbered from right to left starting with 0; thus F5 is the most significant bit of the function field, and F0 is the least significant bit. Similarly, the names of the signals corresponding to the 4-bit operation code supplied to the ALU are Operation3, Operation2, Operation1, and Operation0 (with the last being the least significant bit). Thus the truth table above shows the input combinations for which the ALU control should be 0010, 0001, 0110, or 0111 (the other combinations are not used). The ALUOp bits are named ALUOp1 and ALUOp0. The three output values depend on the 2-bit ALUOp field and, when that field is equal to 10, the 6-bit function code in the instruction. Accordingly, when the ALUOp field is not equal to 10, we don't care about the function code value (it is represented by an X). There is no truth table for when Operation3=1 because it is always set to 0 in Figure C.2.1. See Appendix A for more background on don't cares.

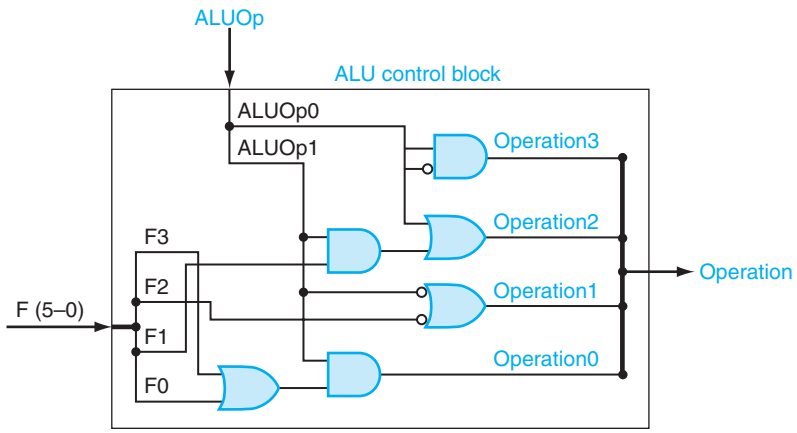


FIGURE C.2.3 The ALU control block generates the four ALU control bits, based on the function code and ALUOp bits. This logic is generated directly from the truth table in Figure C.2.2. Only 4 of the 6 bits in the function code are actually needed as inputs, since the upper 2 bits are always don't cares. Let's examine how this logic relates to the truth table of Figure C.2.2. Consider the Operation2 output, which is generated by two lines in the truth table for Operation2. The second line is the AND of two terms (F1 = 1 and ALUOp1 = 1); the top two-input AND gate corresponds to this term. The other term that causes Operation2 to be asserted is simply ALUOp0. These two terms are combined with an OR gate whose output is Operation2. The outputs Operation0 and Operation1 are derived in similar fashion from the truth table. Since Operation3 is always 0, we connect a signal and its complement as inputs to an AND gate to generate 0.

and can be done with a CAD program. An example of how the logic gates can be derived from the truth tables is given in the legend to Figure C.2.3.

This ALU control logic is simple because there are only three outputs, and only a few of the possible input combinations need to be recognized. If a large number of possible ALU function codes had to be transformed into ALU control signals, this simple method would not be efficient. Instead, you could use a decoder, a memory, or a structured array of logic gates. These techniques are described in Appendix A, and we will see examples when we examine the implementation of the multicycle controller in Section C.3.

Elaboration: In general, a logic equation and truth table representation of a logic function are equivalent. (We discuss this in further detail in Appendix A). However, when a truth table only specifies the entries that result in nonzero outputs, it may not completely describe the logic function. A full truth table completely indicates all don't-care entries. For example, the encoding 11 for ALUOp always generates a don't care in the output. Thus a complete truth table would have XXX in the output portion for all entries with 11 in the ALUOp field. These don't-care entries allow us to replace the ALUOp field 10 and 01 with 1X and X1, respectively. Incorporating the don't-care terms and minimizing the logic is both complex and error-prone and, thus, is better left to a program.

Mapping the Main Control Function to Gates

Implementing the main control function with an unstructured collection of gates, as we did for the ALU control, is reasonable because the control function is neither complex nor large, as we can see from the truth table shown in Figure C.2.4. However, if most of the 64 possible opcodes were used and there were many more control lines, the number of gates would be much larger and each gate could have many more inputs.

Since any function can be computed in two levels of logic, another way to implement a logic function is with a structured two-level logic array. Figure C.2.5 shows such an implementation. It uses an array of AND gates followed by an array of OR gates. This structure is called a *programmable logic array* (PLA). A PLA is one of the most common ways to implement a control function. We will return to the topic of using structured logic elements to implement control when we implement the finite-state controller in the next section.

Control	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 C.2.4 The control function for the simple one-clock implementation is completely specified by this truth table. This table is the same as that shown in Figure 4.22.

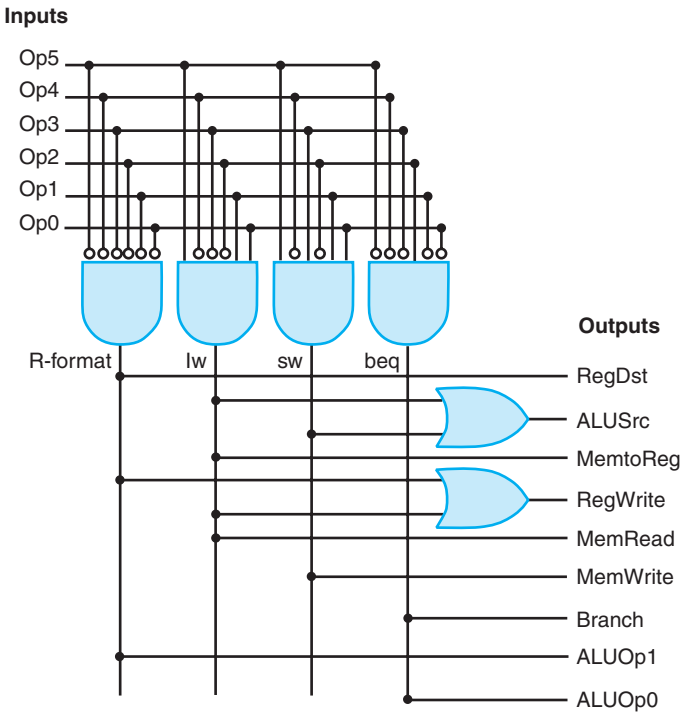


FIGURE C.2.5 The structured implementation of the control function as described by the truth table in Figure C.2.4. The structure, called a *programmable logic array* (PLA), uses an array of AND gates followed by an array of OR gates. The inputs to the AND gates are the function inputs and their inverses (bubbles indicate inversion of a signal). The inputs to the OR gates are the outputs of the AND gates (or, as a degenerate case, the function inputs and inverses). The output of the OR gates is the function outputs.

C.3 Implementing Finite-State Machine Control

To implement the control as a finite-state machine, we must first assign a number to each of the 10 states; any state could use any number, but we will use the sequential numbering for simplicity. Figure C.3.1 shows the finite-state diagram. With 10 states, we will need 4 bits to encode the state number, and we call these state bits S3, S2, S1, and S0. The current-state number will be stored in a state register, as shown in Figure C.3.2. If the states are assigned sequentially, state *i* is encoded using the

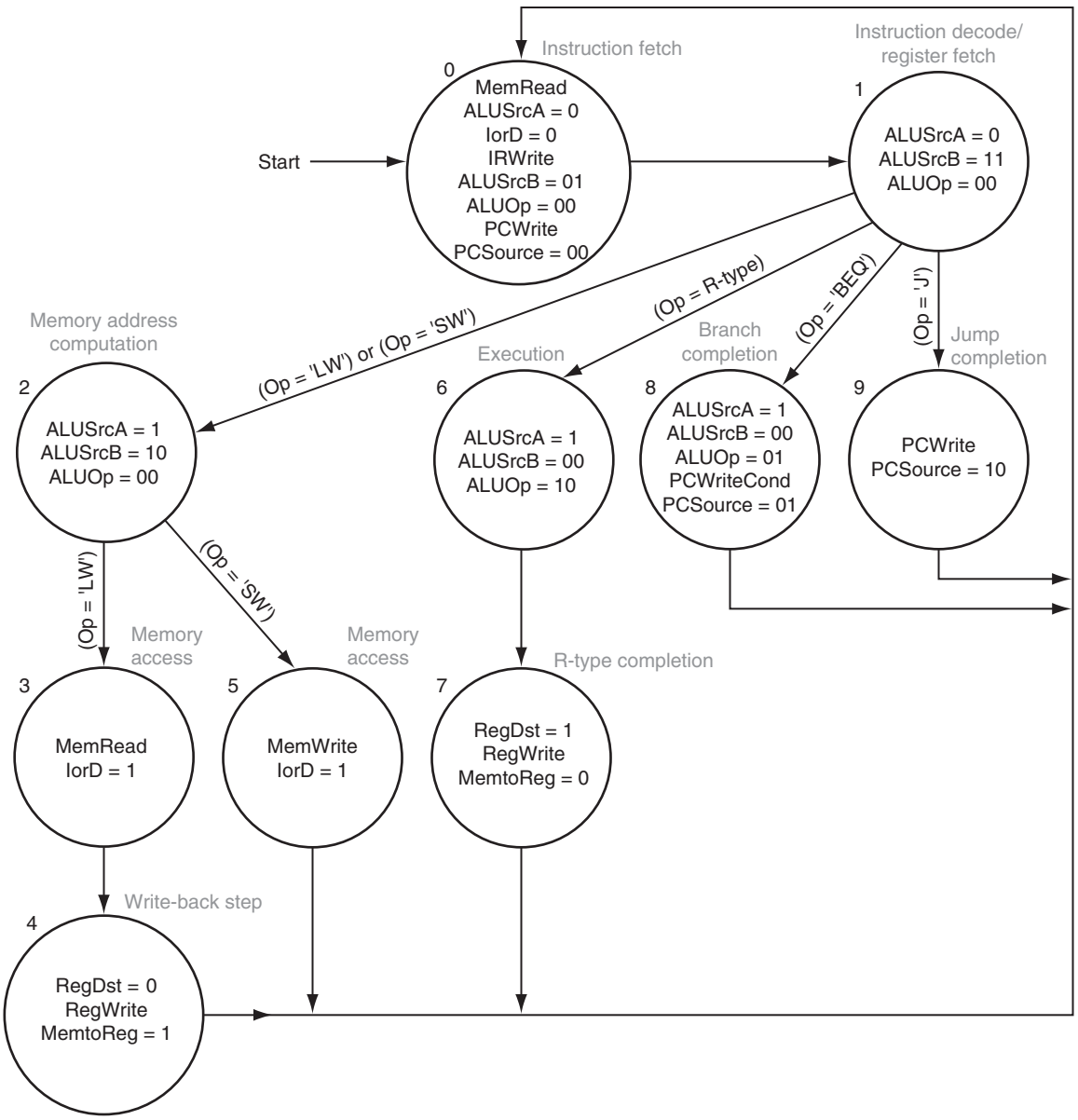


FIGURE C.3.1 The finite-state diagram for multicycle control.

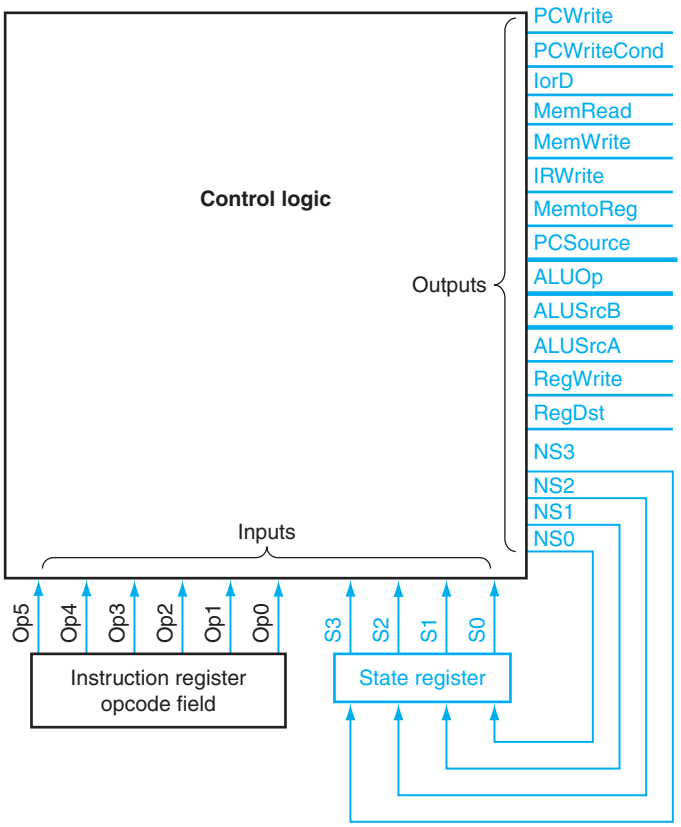


FIGURE C.3.2 The control unit for RISC-V will consist of some control logic and a register to hold the state. The state register is written at the active clock edge and is stable during the clock cycle.

state bits as the binary number *i*. For example, state 6 is encoded as 0110_{two} or S3 = 0, S2 = 1, S1 = 1, S0 = 0, which can also be written as

$$\overline{S3} \cdot S2 \cdot S1 \cdot \overline{S0}$$

The control unit has outputs that specify the next state. These are written into the state register on the clock edge and become the new state at the beginning of the next clock cycle following the active clock edge. We name these outputs NS3, NS2, NS1, and NS0. Once we have determined the number of inputs, states, and outputs, we know what the basic outline of the control unit will look like, as we show in Figure C.3.2.

The block labeled "control logic" in Figure C.3.2 is combinational logic. We can think of it as a big table giving the value of the outputs in terms of the inputs. The logic in this block implements the two different parts of the finite-state machine. One part is the logic that determines the setting of the datapath control outputs, which depend only on the state bits. The other part of the control logic implements the next-state function; these equations determine the values of the next-state bits based on the current-state bits and the other inputs (the 6-bit opcode).

Figure C.3.3 shows the logic equations: the top portion shows the outputs, and the bottom portion shows the next-state function. The values in this table were

Output	Current states	Op
PCWrite	state0 + state9	
PCWriteCond	state8	
IorD	state3 + state5	
MemRead	state0 + state3	
MemWrite	state5	
IRWrite	state0	
MemtoReg	state4	
PCSource1	state9	
PCSource0	state8	
ALUOp1	state6	
ALUOp0	state8	
ALUSrcB1	state1 +state2	
ALUSrcB0	state0 + state1	
ALUSrcA	state2 + state6 + state8	
RegWrite	state4 + state7	
RegDst	state7	
NextState0	state4 + state5 + state7 + state8 + state9	
NextState1	state0	
NextState2	state1	(Op = 'lw') + (Op = 'sw')
NextState3	state2	(Op = 'lw')
NextState4	state3	
NextState5	state2	(Op = 'sw')
NextState6	state1	(Op = 'R-type')
NextState7	state6	
NextState8	state1	(Op = 'beq')
NextState9	state1	(Op = 'jmp')

FIGURE C.3.3 The logic equations for the control unit shown in a shorthand form. Remember that "+" stands for OR in logic equations. The state inputs and NextState outputs must be expanded by using the state encoding. Any blank entry is a don't care.

determined from the state diagram in Figure C.3.1. Whenever a control line is active in a state, that state is entered in the second column of the table. Likewise, the next-state entries are made whenever one state is a successor to another.

In Figure C.3.3, we use the abbreviation state*N* to stand for current state *N*. Thus, state*N* is replaced by the term that encodes the state number *N*. We use NextState*N* to stand for the setting of the next-state outputs to *N*. This output is implemented using the next-state outputs (NS). When NextState*N* is active, the bits NS[3–0] are set corresponding to the binary version of the value *N*. Of course, since a given next-state bit is activated in multiple next states, the equation for each state bit will be the OR of the terms that activate that signal. Likewise, when we use a term such as (Op = '`lw`'), this corresponds to an AND of the opcode inputs that specifies the encoding of the opcode `lw` in 6 bits, just as we did for the simple control unit in the previous section of this chapter. Translating the entries in Figure C.3.3 into logic equations for the outputs is straightforward.

EXAMPLE

Logic Equations for Next-State Outputs

Give the logic equation for the low-order next-state bit, NS0.

ANSWER

The next-state bit NS0 should be active whenever the next state has NS0 = 1 in the state encoding. This is true for NextState1, NextState3, NextState5, NextState7, and NextState9. The entries for these states in Figure C.3.3 supply the conditions when these next-state values should be active. The equation for each of these next states is given below. The first equation states that the next state is 1 if the current state is 0; the current state is 0 if each of the state input bits is 0, which is what the rightmost product term indicates.

$$\text{NextState1} = \text{State0} = \overline{S3} \cdot \overline{S2} \cdot \overline{S1} \cdot \overline{S0}$$

$$\text{NextState3} = \text{State2} \cdot (\text{Op}[5\text{-}0] = \text{lw})$$

$$= \overline{S3} \cdot \overline{S2} \cdot S1 \cdot \overline{S0} \cdot Op5 \cdot \overline{Op4} \cdot \overline{Op3} \cdot \overline{Op2} \cdot Op1 \cdot Op0$$

NextState5 = State2 · (Op[5-0]=sw)
= $\overline{S3} \cdot \overline{S2} \cdot \overline{S1} \cdot \overline{S0} \cdot Op5 \cdot \overline{Op4} \cdot Op3 \cdot \overline{Op2} \cdot Op1 \cdot Op0$
NextState7 = State6 = S3 · S2 · S1 · S0
NextState9 = State1 · (Op[5-0] jmp)
= $\overline{S3} \cdot \overline{S2} \cdot \overline{S1} \cdot S0 \cdot \overline{Op5} \cdot \overline{Op4} \cdot \overline{Op3} \cdot \overline{Op2} \cdot Op1 \cdot Op0$
NS0 is the logical sum of all these terms.

As we have seen, the control function can be expressed as a logic equation for each output. This set of logic equations can be implemented in two ways: corresponding to a complete truth table, or corresponding to a two-level logic structure that allows a sparse encoding of the truth table. Before we look at these implementations, let's look at the truth table for the complete control function.

It is simplest if we break the control function defined in Figure C.3.3 into two parts: the next-state outputs, which may depend on all the inputs, and the control signal outputs, which depend only on the current-state bits. Figure C.3.4 shows the truth tables for all the datapath control signals. Because these signals actually depend only on the state bits (and not the opcode), each of the entries in a table in Figure C.3.4 actually represents 64 (= 2^6) entries, with the 6 bits named Op having all possible values; that is, the Op bits are don't-care bits in determining the data path control outputs. Figure C.3.5 shows the truth table for the next-state bits NS[3–0], which depend on the state input bits and the instruction bits, which supply the opcode.

Elaboration: There are many opportunities to simplify the control function by observing similarities among two or more control signals and by using the semantics of the implementation. For example, the signals PCWriteCond, PCSource0, and ALUOp0 are all asserted in exactly one state, state 8. These three control signals can be replaced by a single signal.

s3	s2	s1	s0
0	0	0	0
1	0	0	1

a. Truth table for PCWrite

s3	s2	s1	s0
1	0	0	0

b. Truth table for PCWriteCond

s3	s2	s1	s0
0	0	1	1
0	1	0	1

c. Truth table for IorD

s3	s2	s1	s0
0	0	0	0
0	0	1	1

d. Truth table for MemRead

s3	s2	s1	s0
0	1	0	1

e. Truth table for MemWrite

s3	s2	s1	s0
0	0	0	0

f. Truth table for IRWrite

s3	s2	s1	s0
0	1	0	0

g. Truth table for MemtoReg

s3	s2	s1	s0
1	0	0	1

h. Truth table for PCSource1

s3	s2	s1	s0
1	0	0	0

i. Truth table for PCSource0

s3	s2	s1	s0
0	1	1	0

j. Truth table for ALUOp1

s3	s2	s1	s0
1	0	0	0

k. Truth table for ALUOp0

s3	s2	s1	s0
0	0	0	1
0	0	1	0

l. Truth table for ALUSrcB1

s3	s2	s1	s0
0	0	0	0
0	0	0	1

m. Truth table for ALUSrcB0

s3	s2	s1	s0
0	0	1	0
0	1	1	0
1	0	0	0

n. Truth table for ALUSrcA

s3	s2	s1	s0
0	1	0	0
0	1	1	1

o. Truth table for RegWrite

s3	s2	s1	s0
0	1	1	1

p. Truth table for RegDst

FIGURE C.3.4 The truth tables are shown for the 16 datapath control signals that depend only on the current-state input bits, which are shown for each table. Each truth table row corresponds to 64 entries: one for each possible value of the six Op bits. Notice that some of the outputs are active under nearly the same circumstances. For example, in the case of PCWriteCond, PCSource0, and ALUOp0, these signals are active only in state 8 (see b, i, and k). These three signals could be replaced by one signal. There are other opportunities for reducing the logic needed to implement the control function by taking advantage of further similarities in the truth tables.

Op5	Op4	Op3	Op2	Op1	Op0	S3	S2	S1	S0
0	0	0	0	1	0	0	0	0	1
0	0	0	1	0	0	0	0	0	1

a. The truth table for the NS3 output, active when the next state is 8 or 9. This signal is activated when the current state is 1.

Op5	Op4	Op3	Op2	Op1	Op0	S3	S2	S1	S0
0	0	0	0	0	0	0	0	0	1
1	0	1	0	1	1	0	0	1	0
X	X	X	X	X	X	0	0	1	1
X	X	X	X	X	X	0	1	1	0

b. The truth table for the NS2 output, which is active when the next state is 4, 5, 6, or 7. This situation occurs when the current state is one of 1, 2, 3, or 6.

Op5	Op4	Op3	Op2	Op1	Op0	S3	S2	S1	S0
0	0	0	0	0	0	0	0	0	1
1	0	0	0	1	1	0	0	0	1
1	0	1	0	1	1	0	0	0	1
1	0	0	0	1	1	0	0	1	0
X	X	X	X	X	X	0	1	1	0

c. The truth table for the NS1 output, which is active when the next state is 2, 3, 6, or 7. The next state is one of 2, 3, 6, or 7 only if the current state is one of 1, 2, or 6.

Op5	Op4	Op3	Op2	Op1	Op0	S3	S2	S1	S0
X	X	X	X	X	X	0	0	0	0
1	0	0	0	1	1	0	0	1	0
1	0	1	0	1	1	0	0	1	0
X	X	X	X	X	X	0	1	1	0
0	0	0	0	1	0	0	0	0	1

d. The truth table for the NS0 output, which is active when the next state is 1, 3, 5, 7, or 9. This happens only if the current state is one of 0, 1, 2, or 6.

FIGURE C.3.5 The four truth tables for the four next-state output bits (NS[3–0]). The next-state outputs depend on the value of Op[5-0], which is the opcode field, and the current state, given by S[3–0]. The entries with X are don't-care terms. Each entry with a don't-care term corresponds to two entries, one with that input at 0 and one with that input at 1. Thus an entry with n don't-care terms actually corresponds to 2^n truth table entries.

A ROM Implementation

Probably the simplest way to implement the control function is to encode the truth tables in a read-only memory (ROM). The number of entries in the memory for the truth tables of Figures C.3.4 and C.3.5 is equal to all possible values of the inputs (the 6 opcode bits plus the 4 state bits), which is $2^{\#\text{ inputs}} = 2^{10} = 1024$. The inputs

to the control unit become the address lines for the ROM, which implements the control logic block that was shown in Figure C.3.2. The width of each entry (or word in the memory) is 20 bits, since there are 16 datapath control outputs and 4 next-state bits. This means the total size of the ROM is $2^{10} \times 20 = 20$ Kbits.

The setting of the bits in a word in the ROM depends on which outputs are active in that word. Before we look at the control words, we need to order the bits within the control input (the address) and output words (the contents), respectively. We will number the bits using the order in Figure C.3.2, with the next-state bits being the low-order bits of the control *word* and the current-state input bits being the low-order bits of the *address*. This means that the PCWrite output will be the high-order bit (bit 19) of each memory word, and NS0 will be the low-order bit. The high-order address bit will be given by Op5, which is the high-order bit of the instruction, and the low-order address bit will be given by S0.

We can construct the ROM contents by building the entire truth table in a form where each row corresponds to one of the 2^n unique input combinations, and a set of columns indicates which outputs are active for that input combination. We don't have the space here to show all 1024 entries in the truth table. However, by separating the datapath control and next-state outputs, we do, since the datapath control outputs depend only on the current state. The truth table for the datapath control outputs is shown in Figure C.3.6. We include only the encodings of the state inputs that are in use (that is, values 0 through 9 corresponding to the 10 states of the state machine).

The truth table in Figure C.3.6 directly gives the contents of the upper 16 bits of each word in the ROM. The 4-bit input field gives the low-order 4 address bits of each word, and the column gives the contents of the word at that address.

If we did show a full truth table for the datapath control bits with both the state number and the opcode bits as inputs, the opcode inputs would all be don't cares. When we construct the ROM, we cannot have any don't cares, since the addresses into the ROM must be complete. Thus, the same datapath control outputs will occur many times in the ROM, since this part of the ROM is the same whenever the state bits are identical, independent of the value of the opcode inputs.

EXAMPLE

Control ROM Entries

For what ROM addresses will the bit corresponding to PCWrite, the high bit of the control word, be 1?

Outputs	Input values (S[3–0])									
	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001
PCWrite	1	0	0	0	0	0	0	0	0	1
PCWriteCond	0	0	0	0	0	0	0	0	1	0
IorD	0	0	0	1	0	1	0	0	0	0
MemRead	1	0	0	1	0	0	0	0	0	0
MemWrite	0	0	0	0	0	1	0	0	0	0
IRWrite	1	0	0	0	0	0	0	0	0	0
MemtoReg	0	0	0	0	1	0	0	0	0	0
PCSource1	0	0	0	0	0	0	0	0	0	1
PCSource0	0	0	0	0	0	0	0	0	1	0
ALUOp1	0	0	0	0	0	0	1	0	0	0
ALUOp0	0	0	0	0	0	0	0	0	1	0
ALUSrcB1	0	1	1	0	0	0	0	0	0	0
ALUSrcB0	1	1	0	0	0	0	0	0	0	0
ALUSrcA	0	0	1	0	0	0	1	0	1	0
RegWrite	0	0	0	0	1	0	0	1	0	0
RegDst	0	0	0	0	0	0	0	1	0	0

FIGURE C.3.6 The truth table for the 16 datapath control outputs, which depend only on the state inputs. The values are determined from Figure C.3.4. Although there are 16 possible values for the 4-bit state field, only 10 of these are used and are shown here. The 10 possible values are shown at the top; each column shows the setting of the datapath control outputs for the state input value that appears at the top of the column. For example, when the state inputs are 0011 (state 3), the active datapath control outputs are IorD or MemRead.

ANSWER

PCWrite is high in states 0 and 9; this corresponds to addresses with the 4 low-order bits being either 0000 or 1001. The bit will be high in the memory word independent of the inputs Op[5–0], so the addresses with the bit high are 000000000, 0000001001, 0000010000, 0000011001, . . . , 1111110000, 1111111001. The general form of this is XXXXXX0000 or XXXXXX1001, where XXXXXX is any combination of bits, and corresponds to the 6-bit opcode on which this output does not depend.

We will show the entire contents of the ROM in two parts to make it easier to show. Figure C.3.7 shows the upper 16 bits of the control word; this comes directly from Figure C.3.6. These datapath control outputs depend only on the state inputs, and this set of words would be duplicated 64 times in the full ROM, as we discussed above. The entries corresponding to input values 1010 through 1111 are not used, so we do not care what they contain.

Figure C.3.8 shows the lower four bits of the control word corresponding to the next-state outputs. The last column of the table in Figure C.3.8 corresponds to all the possible values of the opcode that do not match the specified opcodes. In state 0, the next state is always state 1, since the instruction was still being fetched. After state 1, the opcode field must be valid. The table indicates this by the entries marked illegal; we discuss how to deal with these exceptions and interrupts opcodes in Section 4.9.

Not only is this representation as two separate tables a more compact way to show the ROM contents; it is also a more efficient way to implement the ROM. The majority of the outputs (16 of 20 bits) depends only on four of the 10 inputs. The number of bits in total when the control is implemented as two separate ROMs is $2^4 \times 16 + 2^{10} \times 4 = 256 + 4096 = 4.3$ Kbits, which is about one-fifth of the size of a single ROM, which requires $2^{10} \times 20 = 20$ Kbits. There is some overhead associated with any structured-logic block, but in this case the additional overhead of an extra ROM would be much smaller than the savings from splitting the single ROM.

Lower 4 bits of the address	Bits 19–4 of the word
0000	1001010000001000
0001	0000000000011000
0010	0000000000010100
0011	0011000000000000
0100	0000001000000010
0101	0010100000000000
0110	0000000001000100
0111	0000000000000011
1000	0100000010100100
1001	1000000100000000

FIGURE C.3.7 The contents of the upper 16 bits of the ROM depend only on the state inputs. These values are the same as those in Figure C.3.6, simply rotated 90°. This set of control words would be duplicated 64 times for every possible value of the upper six bits of the address.

Although this ROM encoding of the control function is simple, it is wasteful, even when divided into two pieces. For example, the values of the Instruction register inputs are often not needed to determine the next state. Thus, the next-state ROM has many entries that are either duplicated or are don't care. Consider the case when the machine is in state 0: there are 2^6 entries in the ROM (since the opcode field can have any value), and these entries will all have the same contents (namely, the control word 0001). The reason that so much of the ROM is wasted is that the ROM implements the complete truth table, providing the opportunity to have a different output for every combination of the inputs. But most combinations of the inputs either never happen or are redundant!

	Op [5–0]					
Current state S[3–0]	**000000 (R-format)**	**000010 (jmp)**	**000100 (beq)**	**100011 (lw)**	**101011 (sw)**	**Any other value**
0000	0001	0001	0001	0001	0001	0001
0001	0110	1001	1000	0010	0010	Illegal
0010	XXXX	XXXX	XXXX	0011	0101	Illegal
0011	0100	0100	0100	0100	0100	Illegal
0100	0000	0000	0000	0000	0000	Illegal
0101	0000	0000	0000	0000	0000	Illegal
0110	0111	0111	0111	0111	0111	Illegal
0111	0000	0000	0000	0000	0000	Illegal
1000	0000	0000	0000	0000	0000	Illegal
1001	0000	0000	0000	0000	0000	Illegal

FIGURE C.3.8 This table contains the lower 4 bits of the control word (the NS outputs), which depend on both the state inputs, S[3–0], and the opcode, Op[5–0], which correspond to the instruction opcode. These values can be determined from Figure C.3.5. The opcode name is shown under the encoding in the heading. The four bits of the control word whose address is given by the current-state bits and Op bits are shown in each entry. For example, when the state input bits are 0000, the output is always 0001, independent of the other inputs; when the state is two, the next state is don't care for three of the inputs, three for `lw`, and five for `sw`. Together with the entries in Figure C.3.7, this table specifies the contents of the control unit ROM. For example, the word at address 1000110001 is obtained by finding the upper 16 bits in the table in Figure C.3.7 using only the state input bits (0001) and concatenating the lower four bits found by using the entire address (0001 to find the row and 100011 to find the column). The entry from Figure C.3.7 yields 0000000000011000, while the appropriate entry in the table immediately above is 0010. Thus the control word at address 1000110001 is 00000000000110000010. The column labeled "Any other value" applies only when the Op bits do not match one of the specified opcodes.

A PLA Implementation

We can reduce the amount of control storage required at the cost of using more complex address decoding for the control inputs, which will encode only the input combinations that are needed. The logic structure most often used to do this is a *programmed logic array* (PLA), which we mentioned earlier and illustrated in Figure C.2.5. In a PLA, each output is the logical OR of one or more minterms. A *minterm*, also called a *product term*, is simply a logical AND of one or more inputs. The inputs can be thought of as the address for indexing the PLA, while the minterms select which of all possible address combinations are interesting. A minterm corresponds to a single entry in a truth table, such as those in Figure C.3.4, including possible don't-care terms. Each output consists of an OR of these minterms, which exactly corresponds to a complete truth table. However, unlike a ROM, only those truth table entries that produce an active output are needed, and only one copy of each minterm is required, even if the minterm contains don't cares. Figure C.3.9 shows the PLA that implements this control function.

As we can see from the PLA in Figure C.3.9, there are 17 unique minterms—10 that depend only on the current state and seven others that depend on a combination of the Op field and the current-state bits. The total size of the PLA is proportional to (#inputs × #product terms) + (#outputs × #product terms), as we can see symbolically from the figure. This means the total size of the PLA in Figure C.3.9 is proportional to (10 × 17) + (20 × 17) = 510. By comparison, the size of a single ROM is proportional to 20 Kb, and even the two-part ROM has a total of 4.3 Kb. Because the size of a PLA cell will be only slightly larger than the size of a bit in a ROM, a PLA will be a much more efficient implementation for this control unit.

Of course, just as we split the ROM in two, we could split the PLA into two PLAs: one with four inputs and 10 minterms that generates the 16 control outputs, and one with 10 inputs and seven minterms that generates the four next-state outputs. The first PLA would have a size proportional to (4 × 10) + (10 × 16) = 200, and the second PLA would have a size proportional to (10 × 7) + (4 × 7) = 98. This would yield a total size proportional to 298 PLA cells, about 55% of the size of a single PLA. These two PLAs will be considerably smaller than an implementation using two ROMs. For more details on PLAs and their implementation, as well as the references for books on logic design, see Appendix A.

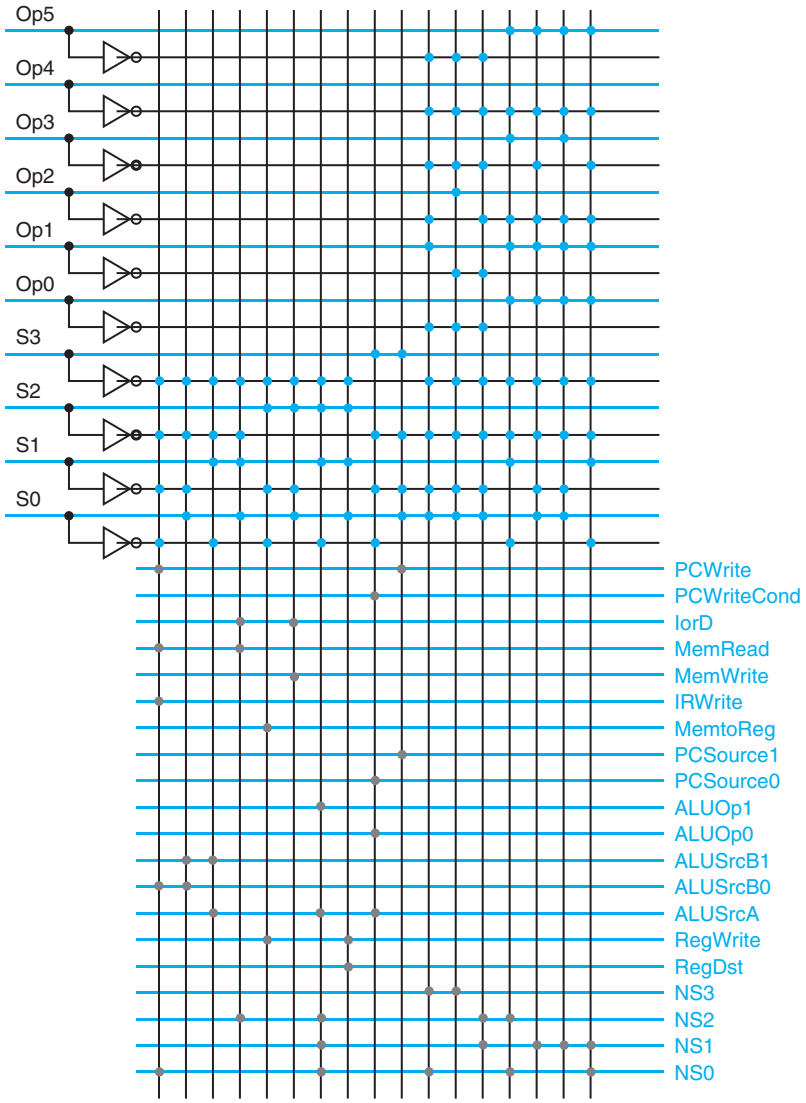


FIGURE C.3.9 This PLA implements the control function logic for the multicycle implementation. The inputs to the control appear on the left and the outputs on the right. The top half of the figure is the AND plane that computes all the minterms. The minterms are carried to the OR plane on the vertical lines. Each colored dot corresponds to a signal that makes up the minterm carried on that line. The sum terms are computed from these minterms, with each gray dot representing the presence of the intersecting minterm in that sum term. Each output consists of a single sum term.

C.4 Implementing the Next-State Function with a Sequencer

Let's look carefully at the control unit we built in the last section. If you examine the ROMs that implement the control in Figures C.3.7 and C.3.8, you can see that much of the logic is used to specify the next-state function. In fact, for the implementation using two separate ROMs, 4096 out of the 4368 bits (94%) correspond to the next-state function! Furthermore, imagine what the control logic would look like if the instruction set had many more different instruction types, some of which required many clocks to implement. There would be many more states in the finite-state machine. In some states, we might be branching to a large number of different states depending on the instruction type (as we did in state 1 of the finite-state machine in Figure C.3.1). However, many of the states would proceed in a sequential fashion, just as states 3 and 4 do in Figure C.3.1.

For example, if we included floating point, we would see a sequence of many states in a row that implement a multicycle floating-point instruction. Alternatively, consider how the control might look for a machine that can have multiple memory operands per instruction. It would require many more states to fetch multiple memory operands. The result of this would be that the control logic will be dominated by the encoding of the next-state function. Furthermore, much of the logic will be devoted to sequences of states with only one path through them that look like states 2 through 4 in Figure C.3.1. With more instructions, these sequences will consist of many more sequentially numbered states than for our simple subset.

To encode these more complex control functions efficiently, we can use a control unit that has a counter to supply the sequential next state. This counter often eliminates the need to encode the next-state function explicitly in the control unit. As shown in Figure C.4.1, an adder is used to increment the state, essentially turning it into a counter. The incremented state is always the state that follows in numerical order. However, the finite-state machine sometimes "branches." For example, in state 1 of the finite-state machine (see Figure C.3.1), there are four possible next states, only one of which is the sequential next state. Thus, we need to be able to choose between the incremented state and a new state based on the inputs from the Instruction register and the current state. Each control word will include control lines that will determine how the next state is chosen.

It is easy to implement the control output signal portion of the control word, since, if we use the same state numbers, this portion of the control word will look exactly like the ROM contents shown in Figure C.3.7. However, the method for selecting the next state differs from the next-state function in the finite-state machine.

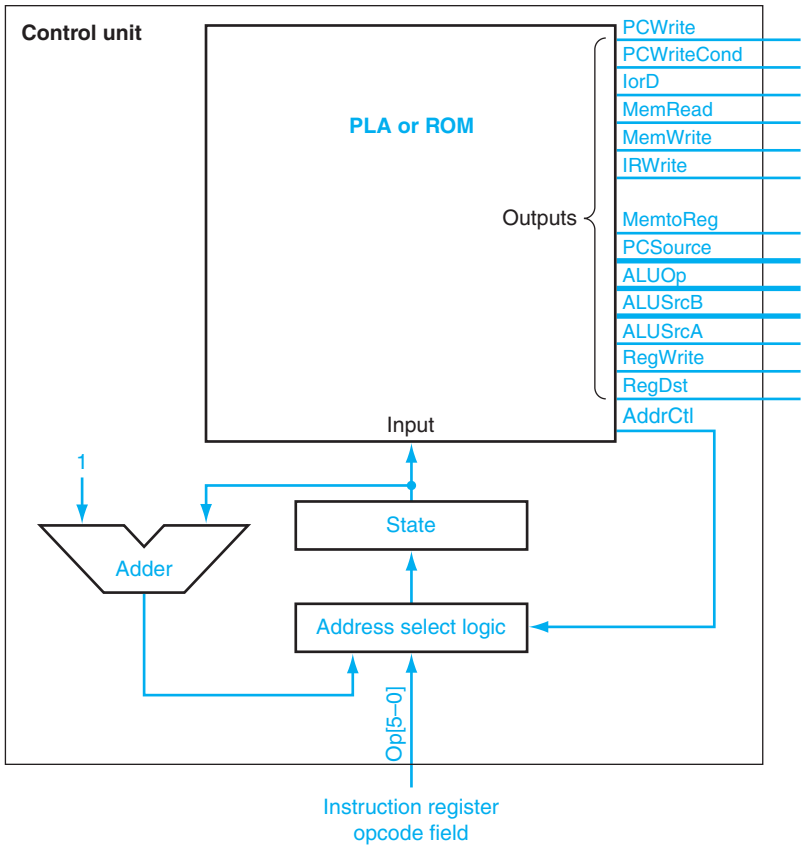


FIGURE C.4.1 The control unit using an explicit counter to compute the next state. In this control unit, the next state is computed using a counter (at least in some states). By comparison, Figure C.3.2 encodes the next state in the control logic for every state. In this control unit, the signals labeled *AddrCtl* control how the next state is determined.

With an explicit counter providing the sequential next state, the control unit logic need only specify how to choose the state when it is not the sequentially following state. There are two methods for doing this. The first is a method we have already seen: namely, the control unit explicitly encodes the next-state function. The difference is that the control unit need only set the next-state lines when the designated next state is not the state that the counter indicates. If the number of

states is large and the next-state function that we need to encode is mostly empty, this may not be a good choice, since the resulting control unit will have lots of empty or redundant space. An alternative approach is to use separate external logic to specify the next state when the counter does not specify the state. Many control units, especially those that implement large instruction sets, use this approach, and we will focus on specifying the control externally.

Although the nonsequential next state will come from an external table, the control unit needs to specify when this should occur and how to find that next state. There are two kinds of "branching" that we must implement in the address select logic. First, we must be able to jump to one of a number of states based on the opcode portion of the Instruction register. This operation, called a *dispatch*, is usually implemented by using a set of special ROMs or PLAs included as part of the address selection logic. An additional set of control outputs, which we call AddrCtl, indicates when a dispatch should be done. Looking at the finite-state diagram (Figure C.3.1), we see that there are two states in which we do a branch based on a portion of the opcode. Thus we will need two small dispatch tables. (Alternatively, we could also use a single dispatch table and use the control bits that select the table as address bits that choose from which portion of the dispatch table to select the address.)

The second type of branching that we must implement consists of branching back to state 0, which initiates the execution of the next RISC-V instruction. Thus there are four possible ways to choose the next state (three types of branches, plus incrementing the current-state number), which can be encoded in 2 bits. Let's assume that the encoding is as follows:

AddrCtl value	Action
0	Set state to 0
1	Dispatch with ROM 1
2	Dispatch with ROM 2
3	Use the incremented state

If we use this encoding, the address select logic for this control unit can be implemented as shown in Figure C.4.2.

To complete the control unit, we need only specify the contents of the dispatch ROMs and the values of the address-control lines for each state. We have already specified the datapath control portion of the control word using the ROM contents of Figure C.3.7 (or the corresponding portions of the PLA in Figure C.3.9). The next-state counter and dispatch ROMs take the place of the portion of the control unit that was computing the next state, which was shown in Figure C.3.8. We are only implementing a portion of the instruction set, so the dispatch ROMs will be largely empty. Figure C.4.3 shows the entries that must be assigned for this subset.

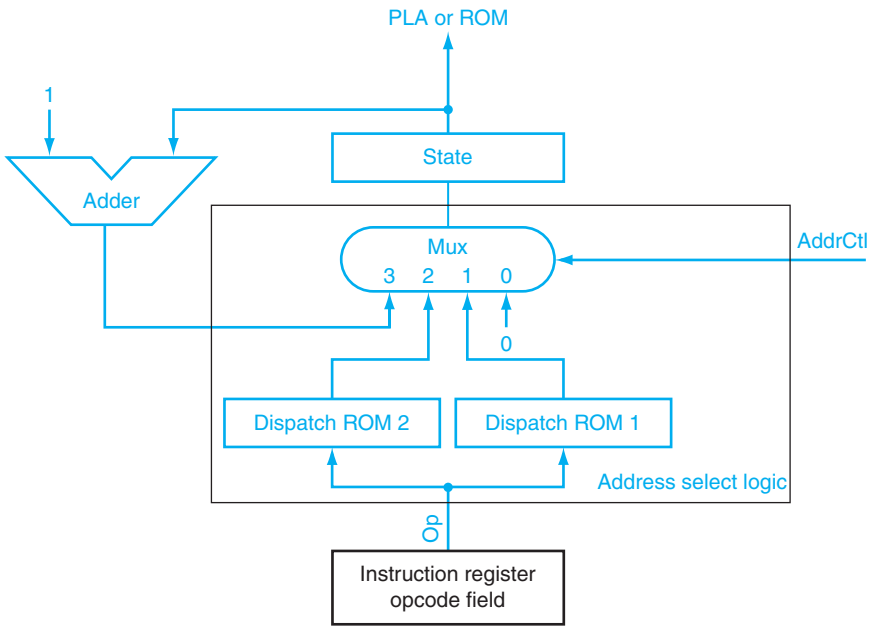


FIGURE C.4.2 This is the address select logic for the control unit of Figure C.4.1.

Dispatch ROM 1		
Op	**Opcode name**	**Value**
000000	R-format	0110
000010	jmp	1001
000100	beq	1000
100011	lw	0010
101011	sw	0010

Dispatch ROM 2		
Op	**Opcode name**	**Value**
100011	lw	0011
101011	sw	0101

FIGURE C.4.3 The dispatch ROMs each have $2^6 = 64$ entries that are 4 bits wide, since that is the number of bits in the state encoding. This figure only shows the entries in the ROM that are of interest for this subset. The first column in each table indicates the value of Op, which is the address used to access the dispatch ROM. The second column shows the symbolic name of the opcode. The third column indicates the value at that address in the ROM.

State number	Address-control action	Value of AddrCtl
0	Use incremented state	3
1	Use dispatch ROM 1	1
2	Use dispatch ROM 2	2
3	Use incremented state	3
4	Replace state number by 0	0
5	Replace state number by 0	0
6	Use incremented state	3
7	Replace state number by 0	0
8	Replace state number by 0	0
9	Replace state number by 0	0

FIGURE C.4.4 The values of the address-control lines are set in the control word that corresponds to each state.

Now we can determine the setting of the address selection lines (AddrCtl) in each control word. The table in Figure C.4.4 shows how the address control must be set for every state. This information will be used to specify the setting of the AddrCtl field in the control word associated with that state.

The contents of the entire control ROM are shown in Figure C.4.5. The total storage required for the control is quite small. There are 10 control words, each 18 bits wide, for a total of 180 bits. In addition, the two dispatch tables are 4 bits wide and each has 64 entries, for a total of 512 additional bits. This total of 692 bits beats the implementation that uses two ROMs with the next-state function encoded in the ROMs (which requires 4.3 Kbits).

Of course, the dispatch tables are sparse and could be more efficiently implemented with two small PLAs. The control ROM could also be replaced with a PLA.

State number	Control word bits 17–2	Control word bits 1–0
0	1001010000001000	11
1	0000000000011000	01
2	0000000000010100	10
3	0011000000000000	11
4	0000001000000010	00
5	0010100000000000	00
6	0000000001000100	11
7	0000000000000011	00
8	0100000010100100	00
9	1000000100000000	00

FIGURE C.4.5 The contents of the control memory for an implementation using an explicit counter. The first column shows the state, while the second shows the datapath control bits, and the last column shows the address-control bits in each control word. Bits 17–2 are identical to those in Figure C.3.7.

Optimizing the Control Implementation

We can further reduce the amount of logic in the control unit by two different techniques. The first is *logic minimization*, which uses the structure of the logic equations, including the don't-care terms, to reduce the amount of hardware required. The success of this process depends on how many entries exist in the truth table, and how those entries are related. For example, in this subset, only the `lw` and `sw` opcodes have an active value for the signal Op5, so we can replace the two truth table entries that test whether the input is `lw` or `sw` by a single test on this bit; similarly, we can eliminate several bits used to index the dispatch ROM because this single bit can be used to find `lw` and `sw` in the first dispatch ROM. Of course, if the opcode space were less sparse, opportunities for this optimization would be more difficult to locate. However, in choosing the opcodes, the architect can provide additional opportunities by choosing related opcodes for instructions that are likely to share states in the control.

A different sort of optimization can be done by assigning the state numbers in a finite-state or microcode implementation to minimize the logic. This optimization, called *state assignment*, tries to choose the state numbers such that the resulting logic equations contain more redundancy and can thus be simplified. Let's consider the case of a finite-state machine with an encoded next-state control first, since it allows states to be assigned arbitrarily. For example, notice that in the finite-state machine, the signal RegWrite is active only in states 4 and 7. If we encoded those states as 8 and 9, rather than 4 and 7, we could rewrite the equation for RegWrite as simply a test on bit S3 (which is only on for states 8 and 9). This renumbering allows us to combine the two truth table entries in part (o) of Figure C.3.4 and replace them with a single entry, eliminating one term in the control unit. Of course, we would have to renumber the existing states 8 and 9, perhaps as 4 and 7.

The same optimization can be applied in an implementation that uses an explicit program counter, though we are more restricted. Because the next-state number is often computed by incrementing the current-state number, we cannot arbitrarily assign the states. However, if we keep the states where the incremented state is used as the next state in the same order, we can reassign the consecutive states as a block. In an implementation with an explicit next-state counter, state assignment may allow us to simplify the contents of the dispatch ROMs.

If we look again at the control unit in Figure C.4.1, it looks remarkably like a computer in its own right. The ROM or PLA can be thought of as memory supplying instructions for the datapath. The state can be thought of as an instruction address. Hence the origin of the name *microcode* or *microprogrammed control*. The control words are thought of as *microinstructions* that control the datapath, and the State register is called the *microprogram counter*. Figure C.4.6 shows a view of the control unit as *microcode*. The next section describes how we map from a microprogram to microcode.

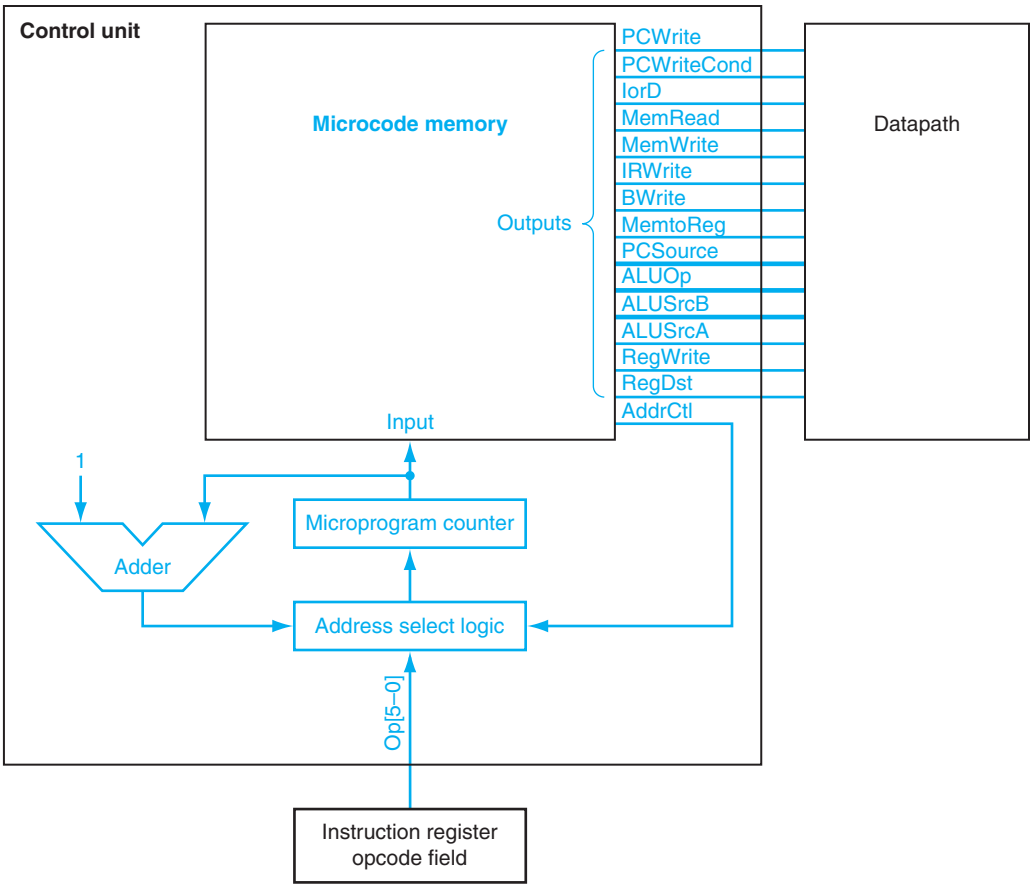


FIGURE C.4.6 The control unit as a microcode. The use of the word "micro" serves to distinguish between the program counter in the datapath and the microprogram counter, and between the microcode memory and the instruction memory.

C.5 Translating a Microprogram to Hardware

To translate a microprogram into actual hardware, we need to specify how each field translates into control signals. We can implement a microprogram with either finite-state control or a microcode implementation with an explicit sequencer. If we choose a finite-state machine, we need to construct the next-state function from

the microprogram. Once this function is known, we can map a set of truth table entries for the next-state outputs. In this section, we will show how to translate the microprogram, assuming that the next state is specified by a sequencer. From the truth tables we will construct, it would be straightforward to build the next-state function for a finite-state machine.

Field name	Value	Signals active	Comment
ALU control	Add	ALUOp = 00	Cause the ALU to add.
	Subt	ALUOp = 01	Cause the ALU to subtract; this implements the compare for branches.
	Func code	ALUOp = 10	Use the instruction's function code to determine ALU control.
SRC1	PC	ALUSrcA = 0	Use the PC as the first ALU input.
	A	ALUSrcA = 1	Register A is the first ALU input.
SRC2	B	ALUSrcB = 00	Register B is the second ALU input.
	4	ALUSrcB = 01	Use 4 as the second ALU input.
	Extend	ALUSrcB = 10	Use output of the sign extension unit as the second ALU input.
	Extshft	ALUSrcB = 11	Use the output of the shift-by-two unit as the second ALU input.
Register control	Read		Read two registers using the rs and rt fields of the IR as the register numbers and putting the data into registers A and B.
	Write ALU	RegWrite, RegDst = 1, MemtoReg = 0	Write a register using the rd field of the IR as the register number and the contents of ALUOut as the data.
	Write MDR	RegWrite, RegDst = 0, MemtoReg = 1	Write a register using the rt field of the IR as the register number and the contents of the MDR as the data.
Memory	Read PC	MemRead, IorD = 0, IRWrite	Read memory using the PC as address; write result into IR (and the MDR).
	Read ALU	MemRead, IorD = 1	Read memory using ALUOut as address; write result into MDR.
	Write ALU	MemWrite, IorD = 1	Write memory using the ALUOut as address, contents of B as the data.
PC write control	ALU	PCSource = 00, PCWrite	Write the output of the ALU into the PC.
	ALUOut-cond	PCSource = 01, PCWriteCond	If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.
	Jump address	PCSource = 10, PCWrite	Write the PC with the jump address from the instruction.
Sequencing	Seq	AddrCtl = 11	Choose the next microinstruction sequentially.
	Fetch	AddrCtl = 00	Go to the first microinstruction to begin a new instruction.
	Dispatch 1	AddrCtl = 01	Dispatch using the ROM 1.
	Dispatch 2	AddrCtl = 10	Dispatch using the ROM 2.

FIGURE C.5.1 Each microcode field translates to a set of control signals to be set. These 22 different values of the fields specify all the required combinations of the 18 control lines. Control lines that are not set, which correspond to actions, are 0 by default. Multiplexor control lines are set to 0 if the output matters. If a multiplexor control line is not explicitly set, its output is a don't care and is not used.

Assuming an explicit sequencer, we need to do two additional tasks to translate the microprogram: assign addresses to the microinstructions and fill in the contents of the dispatch ROMs. This process is essentially the same as the process of translating an assembly language program into machine instructions: the fields of the assembly language or microprogram instruction are translated, and labels on the instructions must be resolved to addresses.

Figure C.5.1 shows the various values for each microinstruction field that controls the datapath and how these fields are encoded as control signals. If the field corresponding to a signal that affects a unit with state (i.e., Memory, Memory register, ALU destination, or PCWriteControl) is blank, then no control signal should be active. If a field corresponding to a multiplexor control signal or the ALU operation control (i.e., ALUOp, SRC1, or SRC2) is blank, the output is unused, so the associated signals may be set as don't care.

The sequencing field can have four values: Fetch (meaning go to the Fetch state), Dispatch 1, Dispatch 2, and Seq. These four values are encoded to set the 2-bit address control just as they were in Figure C.4.4: Fetch = 0, Dispatch 1 = 1, Dispatch 2 = 2, Seq = 3. Finally, we need to specify the contents of the dispatch tables to relate the dispatch entries of the sequence field to the symbolic labels in the microprogram. We use the same dispatch tables as we did earlier in Figure C.4.3.

A microcode assembler would use the encoding of the sequencing field, the contents of the symbolic dispatch tables in Figure C.5.2, the specification in Figure C.5.1, and the actual microprogram to generate the microinstructions.

Since the microprogram is an abstract representation of the control, there is a great deal of flexibility in how the microprogram is translated. For example, the address assigned to many of the microinstructions can be chosen arbitrarily; the only restrictions are those imposed by the fact that certain microinstructions must

dispatch table 1		
Opcode field	**Opcode name**	**Value**
000000	R-format	Rformat1
000010	`jmp`	JUMP1
000100	`beq`	BEQ1
100011	`lw`	Mem1
101011	`sw`	Mem1

Microcode dispatch table 2		
Opcode field	**Opcode name**	**Value**
100011	`lw`	LW2
101011	`sw`	SW2

FIGURE C.5.2 The two microcode dispatch ROMs showing the contents in symbolic form and using the labels in the microprogram.

occur in sequential order (so that incrementing the State register generates the address of the next instruction). Thus the microcode assembler may reduce the complexity of the control by assigning the microinstructions cleverly.

Organizing the Control to Reduce the Logic

For a machine with complex control, there may be a great deal of logic in the control unit. The control ROM or PLA may be very costly. Although our simple implementation had only an 18-bit microinstruction (assuming an explicit sequencer), there have been machines with microinstructions that are hundreds of bits wide. Clearly, a designer would like to reduce the number of microinstructions and the width.

The ideal approach to reducing control store is to first write the complete microprogram in a symbolic notation and then measure how control lines are set in each microinstruction. By taking measurements we are able to recognize control bits that can be encoded into a smaller field. For example, if no more than one of eight lines is set simultaneously in the same microinstruction, then this subset of control lines can be encoded into a 3-bit field ($\log_2 8 = 3$). This change saves five bits in every microinstruction and does not hurt CPI, though it does mean the extra hardware cost of a 3-to-8 decoder needed to generate the eight control lines when they are required at the datapath. It may also have some small clock cycle impact, since the decoder is in the signal path. However, shaving five bits off control store width will usually overcome the cost of the decoder, and the cycle time impact will probably be small or nonexistent. For example, this technique can be applied to bits 13–6 of the microinstructions in this machine, since only one of the seven bits of the control word is ever active (see Figure C.4.5).

This technique of reducing field width is called *encoding*. To further save space, control lines may be encoded together if they are only occasionally set in the same microinstruction; two microinstructions instead of one are then required when both must be set. As long as this doesn't happen in critical routines, the narrower microinstruction may justify a few extra words of control store.

Microinstructions can be made narrower still if they are broken into different formats and given an opcode or *format field* to distinguish them. The format field gives all the unspecified control lines their default values, so as not to change anything else in the machine, and is similar to the opcode of an instruction in a more powerful instruction set. For example, we could use a different format for microinstructions that did memory accesses from those that did register-register ALU operations, taking advantage of the fact that the memory access control lines are not needed in microinstructions controlling ALU operations.

Reducing hardware costs by using format fields usually has an additional performance cost beyond the requirement for more decoders. A microprogram using a single microinstruction format can specify any combination of operations in a datapath and can take fewer clock cycles than a microprogram made up of restricted microinstructions that cannot perform any combination of operations in

a single microinstruction. However, if the full capability of the wider microprogram word is not heavily used, then much of the control store will be wasted, and the machine could be made smaller and faster by restricting the microinstruction capability.

The narrow, but usually longer, approach is often called *vertical microcode*, while the wide but short approach is called *horizontal microcode*. It should be noted that the terms "vertical microcode" and "horizontal microcode" have no universal definition—the designers of the 8086 considered its 21-bit microinstruction to be more horizontal than in other single-chip computers of the time. The related terms *maximally encoded* and *minimally encoded* are probably better than vertical and horizontal.

C.6 Concluding Remarks

We began this appendix by looking at how to translate a finite-state diagram to an implementation using a finite-state machine. We then looked at explicit sequencers that use a different technique for realizing the next-state function. Although large microprograms are often targeted at implementations using this explicit next-state approach, we can also implement a microprogram with a finite-state machine. As we saw, both ROM and PLA implementations of the logic functions are possible. The advantages of explicit versus encoded next state and ROM versus PLA implementation are summarized below.

The BIG Picture

Independent of whether the control is represented as a finite-state diagram or as a microprogram, translation to a hardware control implementation is similar. Each state or microinstruction asserts a set of control outputs and specifies how to choose the next state.

The next-state function may be implemented by either encoding it in a finite-state machine or using an explicit sequencer. The explicit sequencer is more efficient if the number of states is large and there are many sequences of consecutive states without branching.

The control logic may be implemented with either ROMs or PLAs (or even a mix). PLAs are more efficient unless the control function is very dense. ROMs may be appropriate if the control is stored in a separate memory, as opposed to within the same chip as the datapath.

C.7 Exercises

C.1 [10] <§C.2> Instead of using four state bits to implement the finite-state machine in Figure C.3.1, use nine state bits, each of which is a 1 only if the finite-state machine is in that particular state (e.g., S1 is 1 in state 1, S2 is 1 in state 2, etc.). Redraw the PLA (Figure C.3.9).

C.2 [5] <§C.3> We wish to add the instruction `jal` (jump and link). Make any necessary changes to the datapath or to the control signals if needed. You can photocopy figures to make it faster to show the additions. How many product terms are required in a PLA that implements the control for the single-cycle datapath for `jal`?

C.3 [5] <§C.3> Now we wish to add the instruction `addi` (add immediate). Add any necessary changes to the datapath and to the control signals. How many product terms are required in a PLA that implements the control for the single-cycle datapath for `addiu`?

C.4 [10] <§C.3> Determine the number of product terms in a PLA that implements the finite-state machine for `addi`. The easiest way to do this is to construct the additions to the truth tables for `addi`.

C.5 [20] <§C.4> Implement the finite-state machine of using an explicit counter to determine the next state. Fill in the new entries for the additions to Figure C.4.5. Also, add any entries needed to the dispatch ROMs of Figure C.5.2.

C.6 [15] <§§C.3–C.6> Determine the size of the PLAs needed to implement the multicycle machine, assuming that the next-state function is implemented with a counter. Implement the dispatch tables of Figure C.5.2 using two PLAs and the contents of the main control unit in Figure C.4.5 using another PLA. How does the total size of this solution compare to the single PLA solution with the next state encoded? What if the main PLAs for both approaches are split into two separate PLAs by factoring out the next-state or address select signals?

D

APPENDIX

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

RISC: any computer announced after 1985.

Steven Przybylskic
A Designer of the Stanford MIPS

D.1 Introduction D-3
D.2 Addressing Modes and Instruction Formats D-5
D.3 Instructions: The MIPS Core Subset D-9
D.4 Instructions: Multimedia Extensions of the Desktop/Server RISCs D-16
D.5 Instructions: Digital Signal-Processing Extensions of the Embedded RISCs D-19
D.6 Instructions: Common Extensions to MIPS Core D-20
D.7 Instructions Unique to MIPS-64 D-25
D.8 Instructions Unique to Alpha D-27
D.9 Instructions Unique to SPARC v9 D-29
D.10 Instructions Unique to PowerPC D-32
D.11 Instructions Unique to PA-RISC 2.0 D-34
D.12 Instructions Unique to ARM D-36
D.13 Instructions Unique to Thumb D-38
D.14 Instructions Unique to SuperH D-39
D.15 Instructions Unique to M32R D-40
D.16 Instructions Unique to MIPS-16 D-40
D.17 Concluding Remarks D-43

D.1 Introduction

We cover two groups of *reduced instruction set computer* (RISC) architectures in this appendix. The first group is the desktop and server RISCs:

- Digital Alpha
- Hewlett-Packard PA-RISC
- IBM and Motorola PowerPC
- MIPS INC MIPS-64
- Sun Microsystems SPARC

The second group is the embedded RISCs:

- Advanced RISC Machines ARM
- Advanced RISC Machines Thumb
- Hitachi SuperH
- Mitsubishi M32R
- MIPS INC MIPS-16

	Alpha	MIPS I	PA-RISC 1.1	PowerPC	SPARCv8
Date announced	1992	1986	1986	1993	1987
Instruction size (bits)	32	32	32	32	32
Address space (size, model)	64 bits, flat	32 bits, flat	48 bits, segmented	32 bits, flat	32 bits, flat
Data alignment	Aligned	Aligned	Aligned	Unaligned	Aligned
Data addressing modes	1	1	5	4	2
Protection	Page	Page	Page	Page	Page
Minimum page size	8 KB	4 KB	4 KB	4 KB	8 KB
I/O	Memory mapped	Memory mapped	Memory mapped	Memory mapped	Memory mapped
Integer registers (number, model, size)	31 GPR × 64 bits	31 GPR × 32 bits	31 GPR × 32 bits	32 GPR × 32 bits	31 GPR × 32 bits
Separate floating-point registers	31 × 32 or 31 × 64 bits	16 × 32 or 16 × 64 bits	56 × 32 or 28 × 64 bits	32 × 32 or 32 × 64 bits	32 × 32 or 32 × 64 bits
Floating-point format	IEEE 754 single, double	IEEE 754 single, double	IEEE 754 single, double	IEEE 754 single, double	IEEE 754 single, double

FIGURE D.1.1 Summary of the first version of five architectures for desktops and servers. Except for the number of data address modes and some instruction set details, the integer instruction sets of these architectures are very similar. Contrast this with Figure D.17.1. Later versions of these architectures all support a flat, 64-bit address space.

	ARM	Thumb	SuperH	M32R	MIPS-16
Date announced	1985	1995	1992	1997	1996
Instruction size (bits)	32	16	16	16/32	16/32
Address space (size, model)	32 bits, flat	32 bits, flat	32 bits, flat	32 bits, flat	32/64 bits, flat
Data alignment	Aligned	Aligned	Aligned	Aligned	Aligned
Data addressing modes	6	6	4	3	2
Integer registers (number, model, size)	15 GPR x 32 bits	8 GPR + SP, LR x 32 bits	16 GPR x 32 bits	16 GPR x 32 bits	8 GPR + SP, RA x 32/64 bits
I/O	Memory mapped	Memory mapped	Memory mapped	Memory mapped	Memory mapped

FIGURE D.1.2 Summary of five architectures for embedded applications. Except for number of data address modes and some instruction set details, the integer instruction sets of these architectures are similar. Contrast this with Figure D.17.1.

There has never been another class of computers so similar. This similarity allows the presentation of 10 architectures in about 50 pages. Characteristics of the desktop and server RISCs are found in Figure D.1.1 and the embedded RISCs in Figure D.1.2.

Notice that the embedded RISCs tend to have eight to 16 general-purpose registers while the desktop/server RISCs have 32, and that the length of instructions is 16 to 32 bits in embedded RISCs but always 32 bits in desktop/server RISCs.

Although shown as separate embedded instruction set architectures, Thumb and MIPS-16 are really optional modes of ARM and MIPS invoked by call instructions. When in this mode, they execute a subset of the native architecture using 16-bit-long instructions. These 16-bit instruction sets are not intended to be full architectures, but they are enough to encode most procedures. Both machines expect procedures to be homogeneous, with all instructions in either 16-bit mode or 32-bit mode. Programs will consist of procedures in 16-bit mode for density or in 32-bit mode for performance.

One complication of this description is that some of the older RISCs have been extended over the years. We have decided to describe the latest versions of the architectures: MIPS-64, Alpha version 3, PA-RISC 2.0, and SPARC version 9 for the desktop/server; ARM version 4, Thumb version 1, Hitachi SuperH SH-3, M32R version 1, and MIPS-16 version 1 for the embedded ones.

The remaining sections proceed as follows: after discussing the addressing modes and instruction formats of our RISC architectures, we present the survey of the instructions in five steps:

- Instructions found in the MIPS core, which is defined in Chapters 2 and 3 of the main text
- Multimedia extensions of the desktop/server RISCs
- Digital signal-processing extensions of the embedded RISCs
- Instructions not found in the MIPS core but found in two or more architectures
- The unique instructions and characteristics of each of the 10 architectures

We give the evolution of the instruction sets in the final section and conclude with speculation about future directions for RISCs.

D.2 Addressing Modes and Instruction Formats

Figure D.2.1 shows the data addressing modes supported by the desktop architectures. Since all have one register that always has the value 0 when used in address modes, the absolute address mode with limited range can be synthesized using zero as the base in displacement addressing. (This register can be changed

by ALU operations in PowerPC; it is always 0 in the other machines.) Similarly, register indirect addressing is synthesized by using displacement addressing with an offset of 0. Simplified addressing modes is one distinguishing feature of RISC architectures.

Figure D.2.2 shows the data addressing modes supported by the embedded architectures. Unlike the desktop RISCs, these embedded machines do not reserve a register to contain 0. Although most have two to three simple addressing modes, ARM and SuperH have several, including fairly complex calculations. ARM has an addressing mode that can shift one register by any amount, add it to the other registers to form the address, and then update one register with this new address.

References to code are normally PC-relative, although jump register indirect is supported for returning from procedures, for *case* statements, and for pointer function calls. One variation is that PC-relative branch addresses are shifted left 2 bits before being added to the PC for the desktop RISCs, thereby increasing the branch distance. This works because the length of all instructions for the desktop RISCs is 32 bits, and instructions must be aligned on 32-bit words in memory. Embedded architectures with 16-bit-long instructions usually shift the PC-relative address by one for similar reasons.

Addressing mode	Alpha	MIPS-64	PA-RISC 2.0	PowerPC	SPARCv9
Register + offset (displacement or based)	X	X	X	X	X
Register + register (indexed)		X (FP)	X (Loads)	X	X
Register + scaled register (scaled)			X		
Register + offset and update register			X	X	
Register + register and update register			X	X	

FIGURE D.2.1 Summary of data addressing modes supported by the desktop architectures. PA-RISC also has short address versions of the offset addressing modes. MIPS-64 has indexed addressing for floating-point loads and stores. (These addressing modes are described in Figure 2.18.)

Addressing mode	ARMv4	Thumb	SuperH	M32R	MIPS-16
Register + offset (displacement or based)	X	X	X	X	X
Register + register (indexed)	X	X	X		
Register + scaled register (scaled)	X				
Register + offset and update register	X				
Register + register and update register	X				
Register indirect			X	X	
Autoincrement, autodecrement	X	X	X	X	
PC-relative data	X	X (loads)	X		X (loads)

FIGURE D.2.2 Summary of data addressing modes supported by the embedded architectures. SuperH and M32R have separate register indirect and register + offset addressing modes rather than just putting 0 in the offset of the latter mode. This increases the use of 16-bit instructions in the M32R, and it gives a wider set of address modes to different data transfer instructions in SuperH. To get greater addressing range, ARM and Thumb shift the offset left one or two bits if the data size is halfword or word. (These addressing modes are described in Figure 2.18.)

Figure D.2.3 shows the format of the desktop RISC instructions, which include the size of the address. Each instruction set architecture uses these four primary instruction formats. Figure D.2.4 shows the six formats for the embedded RISC machines. The desire to have smaller code size via 16-bit instructions leads to more instruction formats.

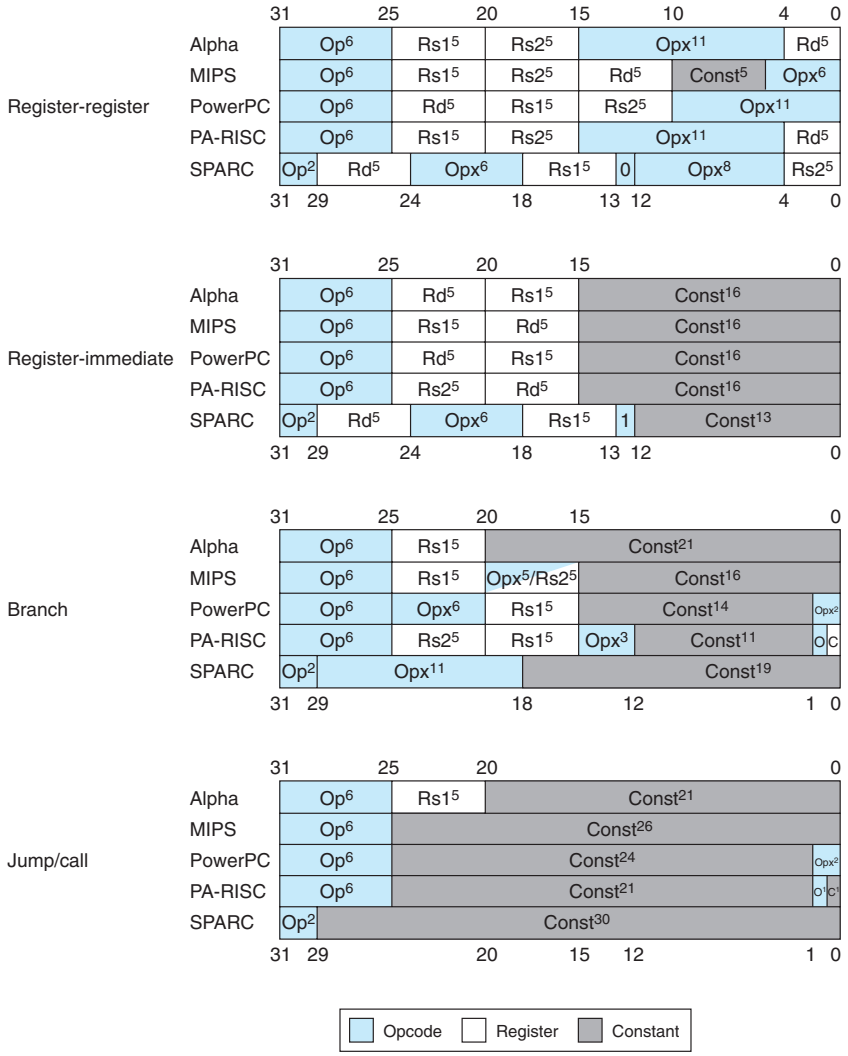


FIGURE D.2.3 Instruction formats for desktop/server RISC architectures. These four formats are found in all five architectures. (The superscript notation in this figure means the width of a field in bits.) Although the register fields are located in similar pieces of the instruction, be aware that the destination and two source fields are scrambled. Op = the main opcode, Opx = an opcode extension, Rd = the destination register, Rs1 = source register 1, Rs2 = source register 2, and Const = a constant (used as an immediate or as an address). Unlike the other RISCs, Alpha has a format for immediates in arithmetic and logical operations that is different from the data transfer format shown here. It provides an 8-bit immediate in bits 20 to 13 of the RR format, with bits 12 to 5 remaining as an opcode extension.

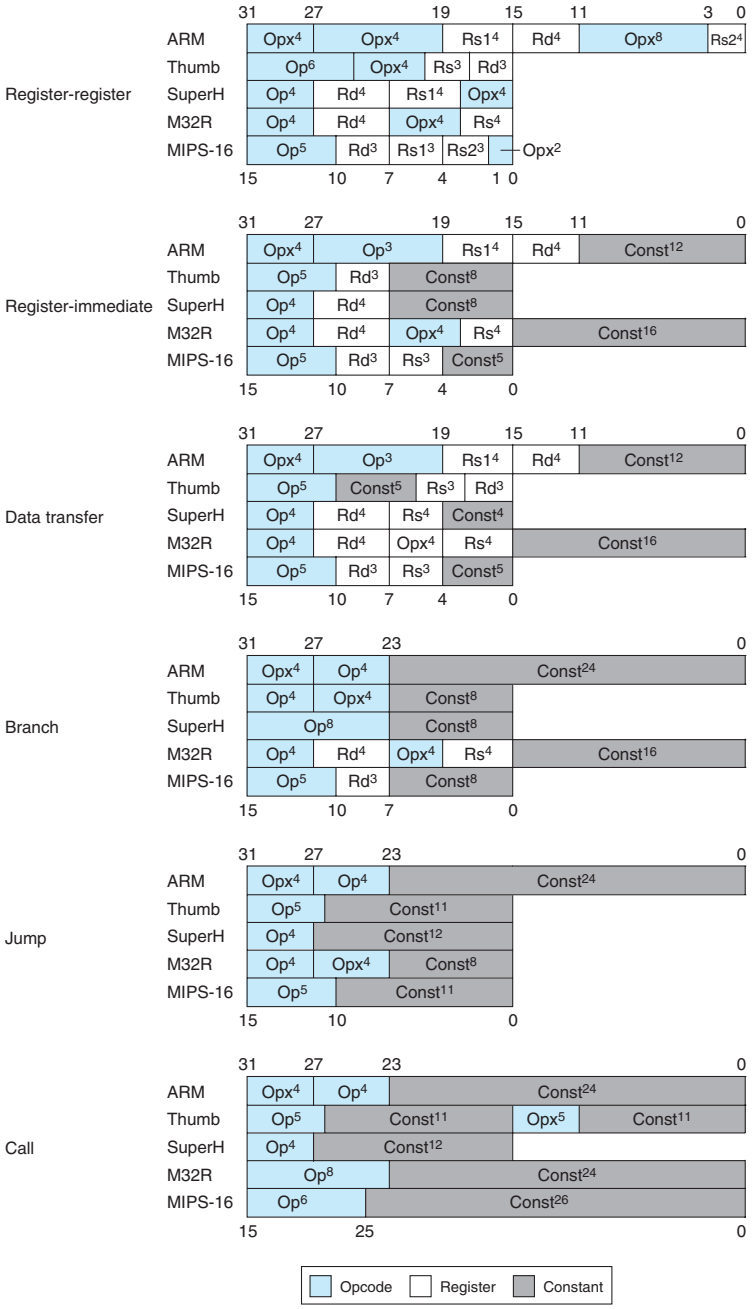


FIGURE D.2.4 Instruction formats for embedded RISC architectures. These six formats are found in all five architectures. The notation is the same as in Figure D.2.3. Note the similarities in branch, jump, and call formats, and the diversity in register-register, register-immediate, and data transfer formats. The differences result from whether the architecture has eight or 16 registers, whether it is a two- or three-operand format, and whether the instruction length is 16 or 32 bits.

Format: instruction category	Alpha	MIPS-64	PA-RISC 2.0	PowerPC	SPARCv9
Branch: all	Sign	Sign	Sign	Sign	Sign
Jump/call: all	Sign	—	Sign	Sign	Sign
Register-immediate: data transfer	Sign	Sign	Sign	Sign	Sign
Register-immediate: arithmetic	Zero	Sign	Sign	Sign	Sign
Register-immediate: logical	Zero	Zero	—	Zero	Sign

FIGURE D.2.5 Summary of constant extension for desktop RISCs. The constants in the jump and call instructions of MIPS are not sign-extended, since they only replace the lower 28 bits of PC, leaving the upper 4 bits unchanged. PA-RISC has no logical immediate instructions.

Format: instruction category	Armv4	Thumb	SuperH	M32R	MIPS-16
Branch: all	Sign	Sign	Sign	Sign	Sign
Jump/call: all	Sign	Sign/Zero	Sign	Sign	—
Register-immediate: data transfer	Zero	Zero	Zero	Sign	Zero
Register-immediate: arithmetic	Zero	Zero	Sign	Sign	Zero/Sign
Register-immediate: logical	Zero	—	Zero	Zero	—

FIGURE D.2.6 Summary of constant extension for embedded RISCs. The 16-bit-length instructions have much shorter immediates than those of the desktop RISCs, typically only 5 to 8 bits. Most embedded RISCs, however, have a way to get a long address for procedure calls from two sequencial halfwords. The constants in the jump and call instructions of MIPS are not sign-extended, since they only replace the lower 28 bits of the PC, leaving the upper 4 bits unchanged. The 8-bit immediates in ARM can be rotated right an even number of bits between 2 and 30, yielding a large range of immediate values. For example, all powers of two are immediates in ARM.

Figures D.2.5 and D.2.6 show the variations in extending constant fields to the full width of the registers. In this subtle point, the RISCs are similar but not identical.

D.3 Instructions: The MIPS Core Subset

The similarities of each architecture allow simultaneous descriptions, starting with the operations equivalent to the MIPS core.

MIPS Core Instructions

Almost every instruction found in the MIPS core is found in the other architectures, as Figures D.3.1 through D.3.5 show. (For reference, definitions of the MIPS instructions are found in the MIPS Reference Data Card at the beginning of the book.) Instructions are listed under four categories: data transfer (Figure D.3.1); arithmetic/logical (Figure D.3.2); control (Figure D.3.3); and floating point (Figure D.3.4). A fifth category (Figure D.3.5) shows conventions for register

Data transfer (instruction formats)	R-I	R-I	R-I, R-R	R-I, R-R	R-I, R-R
Instruction name	**Alpha**	**MIPS-64**	**PA-RISC 2.0**	**PowerPC**	**SPARCv9**
Load byte signed	LDBU; SEXTB	LB	LDB; EXTRW,S 31,8	LBZ; EXTSB	LDSB
Load byte unsigned	LDBU	LBU	LDB, LDBX, LDBS	LBZ	LDUB
Load halfword signed	LDWU; SEXTW	LH	LDH; EXTRW,S 31,16	LHA	LDSH
Load halfword unsigned	LDWU	LHU	LDH, LDHX, LDHS	LHZ	LDUH
Load word	LDLS	LW	LDW, LDWX, LDWS	LW	LD
Load SP float	LDS*	LWC1	FLDWX, FLDWS	LFS	LDF
Load DP float	LDT	LDC1	FLDDX, FLDDS	LFD	LDDF
Store byte	STB	SB	STB, STBX, STBS	STB	STB
Store halfword	STW	SH	STH, STHX, STHS	STH	STH
Store word	STL	SW	STW, STWX, STWS	STW	ST
Store SP float	STS	SWC1	FSTWX, FSTWS	STFS	STF
Store DP float	STT	SDC1	FSTDX, FSTDS	STFD	STDF
Read, write special registers	MF_, MT_	MF, MT_	MFCTL, MTCTL	MFSPR, MF_, MTSPR, MT_	RD, WR, RDPR, WRPR, LDXFSR, STXFSR
Move integer to FP register	ITOFS	MFC1/DMFC1	STW; FLDWX	STW; LDFS	ST; LDF
Move FP to integer register	FTTOIS	MTC1/DMTC1	FSTWX; LDW	STFS; LW	STF; LD

FIGURE D.3.1 Desktop RISC data transfer instructions equivalent to MIPS core. A sequence of instructions to synthesize a MIPS instruction is shown separated by semicolons. If there are several choices of instructions equivalent to MIPS core, they are separated by commas. For this figure, halfword is 16 bits and word is 32 bits. Note that in Alpha, LDS converts single-precision floating point to double precision and loads the entire 64-bit register.

usage and pseudoinstructions on each architecture. If a MIPS core instruction requires a short sequence of instructions in other architectures, these instructions are separated by semicolons in Figures D.3.1 through D.3.5. (To avoid confusion, the destination register will always be the leftmost operand in this appendix, independent of the notation normally used with each architecture.) Figures D.3.6 through D.3.9 show the equivalent listing for embedded RISCs. Note that floating point is generally not defined for the embedded RISCs.

Every architecture must have a scheme for compare and conditional branch, but despite all the similarities, each of these architectures has found a different way to perform the operation.

Compare and Conditional Branch

SPARC 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 pipelined implementation. Although condition codes can be set as a side effect of an operation, explicit compares are synthesized with a subtract using r0 as the destination. SPARC conditional branches

Arithmetic/logical (instruction formats)	R-R, R-I	R-R, R-I	R-R, R-I	R-R, R-I	R-R, R-I
Instruction name	**Alpha**	**MIPS-64**	**PA-RISC 2.0**	**PowerPC**	**SPARCv9**
Add	ADDL	ADDU, ADDU	ADDL, LDO, ADDI, UADDCM	ADD, ADDI	ADD
Add (trap if overflow)	ADDLV	ADD, ADDI	ADDO, ADDIO	ADDO; MCRXR; BC	ADDcc; TVS
Sub	SUBL	SUBU	SUB, SUBI	SUBF	SUB
Sub (trap if overflow)	SUBLV	SUB	SUBTO, SUBIO	SUBF/oe	SUBcc; TVS
Multiply	MULL	MULT, MULTU	SHiADD;...; (i=1,2,3)	MULLW, MULLI	MULX
Multiply (trap if overflow)	MULLV	—	SHiADDO;...;	—	—
Divide	—	DIV, DIVU	DS;...; DS	DIVW	DIVX
Divide (trap if overflow)	—	—	—	—	—
And	AND	AND, ANDI	AND	AND, ANDI	AND
Or	BIS	OR, ORI	OR	OR, ORI	OR
Xor	XOR	XOR, XORI	XOR	XOR, XORI	XOR
Load high part register	LDAH	LUI	LDIL	ADDIS	SETHI (B fmt.)
Shift left logical	SLL	SLLV, SLL	DEPW, Z 31-i,32-i	RLWINM	SLL
Shift right logical	SRL	SRLV, SRL	EXTRW, U 31, 32-i	RLWINM 32-i	SRL
Shift right arithmetic	SRA	SRAV, SRA	EXTRW, S 31, 32-i	SRAW	SRA
Compare	CMPEQ, CMPLT, CMPLE	SLT/U, SLTI/U	COMB	CMP(I)CLR	SUBcc r0,...

FIGURE D.3.2 Desktop RISC arithmetic/logical instructions equivalent to MIPS core. Dashes mean the operation is not available in that architecture, or not synthesized in a few instructions. Such a sequence of instructions is shown separated by semicolons. If there are several choices of instructions equivalent to MIPS core, they are separated by commas. Note that in the "Arithmetic/logical" category, all machines but SPARC use separate instruction mnemonics to indicate an immediate operand; SPARC offers immediate versions of these instructions but uses a single mnemonic. (Of course these are separate opcodes!)

Control (instruction formats)	B, J/C	B, J/C	B, J/C	B, J/C	B, J/C
Instruction name	**Alpha**	**MIPS-64**	**PA-RISC 2.0**	**PowerPC**	**SPARCv9**
Branch on integer compare	B_ (<, >, <=, >=, =, not=)	BEQ, BNE, B_Z (<, >, <=, >=)	COMB, COMIB	BC	BR_Z, BPcc (<, >, <=, >=, =, not=)
Branch on floating-point compare	FB_(<, >, <=, >=, =, not=)	BC1T, BC1F	FSTWX f0; LDW t; BB t	BC	FBPfcc (<, >, <=, >=, =,...)
Jump, jump register	BR, JMP	J, JR	BL r0, BLR r0	B, BCLR, BCCTR	BA, JMPL r0,...
Call, call register	BSR	JAL, JALR	BL, BLE	BL, BLA, BCLRL, BCCTRL	CALL, JMPL
Trap	CALL_PAL GENTRAP	BREAK	BREAK	TW, TWI	Ticc, SIR
Return from interrupt	CALL_PAL REI	JR; ERET	RFI, RFIR	RFI	DONE, RETRY, RETURN

FIGURE D.3.3 Desktop RISC control instructions equivalent to MIPS core. If there are several choices of instructions equivalent to MIPS core, they are separated by commas.

Floating point (instruction formats)	R-R	R-R	R-R	R-R	R-R
Instruction name	**Alpha**	**MIPS-64**	**PA-RISC 2.0**	**PowerPC**	**SPARCv9**
Add single, double	ADDS, ADDT	ADD.S, ADD.D	FADD FADD/dbl	FADDS, FADD	FADDS, FADDD
Subtract single, double	SUBS, SUBT	SUB.S, SUB.D	FSUB FSUB/dbl	FSUBS, FSUB	FSUBS, FSUBD
Multiply single, double	MULS, MULT	MUL.S, MUL.D	FMPY FMPY/dbl	FMULS, FMUL	FMULS, FMULD
Divide single, double	DIVS, DIVT	DIV.S, DIV.D	FDIV, FDIV/dbl	FDIVS, FDIV	FDIVS, FDIVD
Compare	CMPT_ (=, <, <=, UN)	C_.S, C_.D (<, >, <=, >=, =, . . .)	FCMP, FCMP/dbl (<, =, >)	FCMP	FCMPS, FCMPD
Move R-R	ADDT Fd, F31, Fs	MOV.S, MOV.D	FCPY	FMV	FMOVS/D/Q
Convert (single, double, integer) to (single, double, integer)	CVTST, CVTTS, CVTTQ, CVTQS, CVTQT	CVT.S.D, CVT.D.S, CVT.S.W, CVT.D.W, CVT.W.S, CVT.W.D	FCNVFF,s,d FCNVFF,d,s FCNVXF,s,s FCNVXF,d,d FCNVFX,s,s FCNVFX,d,s	—, FRSP, —, FCTIW,—, —	FSTOD, FDTOS, FSTOI, FDTOI, FITOS, FITOD

FIGURE D.3.4 Desktop RISC floating-point 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 MIPS core, they are separated by commas.

Conventions	Alpha	MIPS-64	PA-RISC 2.0	PowerPC	SPARCv9
Register with value 0	r31 (source)	r0	r0	r0 (addressing)	r0
Return address register	(any)	r31	r2, r31	link (special)	r31
No-op	LDQ_U r31, . . .	SLL r0, r0, r0	OR r0, r0, r0	ORI r0, r0, #0	SETHI r0, 0
Move R-R integer	BIS. . . , r31, . . .	ADD. . . , r0, . . .	OR. . . , r0, . . .	OR rx, ry, ry	OR. . . , r0, . . .
Operand order	OP Rs1, Rs2, Rd	OP Rd, Rs1, Rs2	OP Rs1, Rs2, Rd	OP Rd, Rs1, Rs2	OP Rs1, Rs2, Rd

FIGURE D.3.5 Conventions of desktop RISC architectures equivalent to MIPS core.

test condition codes to determine all possible unsigned and signed relations. Floating point uses separate condition codes to encode the IEEE 754 conditions, requiring a floating-point compare instruction. Version 9 expanded SPARC branches in four ways: a separate set of condition codes for 64-bit operations; a branch that tests the contents of a register and branches if the value is =, not =, <, <=, >=, or <= 0 (see MIPS below); three more sets of floating-point condition codes; and branch instructions that encode static branch prediction.

PowerPC also uses four condition codes—*less than, greater than, equal*, and *summary overflow*—but it has eight copies of them. This redundancy allows the PowerPC instructions to use different condition codes without conflict, essentially giving PowerPC eight extra 4-bit registers. Any of these eight condition codes can be the target of a compare instruction, and any can be the source of a conditional branch. The integer instructions have an option bit that behaves as if the integer op

Instruction name	ARMv4	Thumb	SuperH	M32R	MIPS-16
Data transfer (instruction formats)	**DT**	**DT**	**DT**	**DT**	**DT**
Load byte signed	LDRSB	LDRSB	MOV.B	LDB	LB
Load byte unsigned	LDRB	LDRB	MOV.B; EXTU.B	LDUB	LBU
Load halfword signed	LDRSH	LDRSH	MOV.W	LDH	LH
Load halfword unsigned	LDRH	LDRH	MOV.W; EXTU.W	LDUH	LHU
Load word	LDR	LDR	MOV.L	LD	LW
Store byte	STRB	STRB	MOV.B	STB	SB
Store halfword	STRH	STRH	MOV.W	STH	SH
Store word	STR	STR	MOV.L	ST	SW
Read, write special registers	MRS, MSR	—[1]	LDC, STC	MVFC, MVTC	MOVE

FIGURE D.3.6 Embedded RISC data transfer instructions equivalent to MIPS core. A sequence of instructions to synthesize a MIPS instruction is shown separated by semicolons. Note that floating point is generally not defined for the embedded RISCs. Thumb and MIPS-16 are just 16-bit instruction subsets of the ARM and MIPS architectures, so machines can switch modes and execute the full instruction set. We use —[1] to show sequences that are available in 32-bit mode but not 16-bit mode in Thumb or MIPS-16.

is followed by a compare to zero that sets the first condition "register." PowerPC also lets the second "register" be optionally set by floating-point instructions. PowerPC provides logical operations among these eight 4-bit condition code registers (CRAND, CROR, CRXOR, CRNAND, CRNOR, CREQV), allowing more complex conditions to be tested by a single branch.

MIPS uses the contents of registers to evaluate conditional branches. Any two registers can be compared for equality (BEQ) or inequality (BNE), and then the branch is taken if the condition holds. The set on less than instructions (SLT, SLTI, SLTU, SLTIU) compare two operands and then set the destination register to 1 if less and to 0 otherwise. These instructions are enough to synthesize the full set of relations. Because of the popularity of comparisons to 0, MIPS includes special compare and branch instructions for all such comparisons: greater than or equal to zero (BGEZ), greater than zero (BGTZ), less than or equal to zero (BLEZ), and less than zero (BLTZ). Of course, equal and not equal to zero can be synthesized using r0 with BEQ and BNE. Like SPARC, MIPS I uses a condition code for floating point with separate floating-point compare and branch instructions; MIPS IV expanded this to eight floating-point condition codes, with the floating point comparisons and branch instructions specifying the condition to set or test.

Alpha compares (CMPEQ, CMPLT, CMPLE, CMPULT, CMPULE) test two registers and set a third to 1 if the condition is true and to 0 otherwise. Floating-point compares (CMTEQ, CMTLT, CMTLE, CMTUN) set the result to 2.0 if the condition holds and to 0 otherwise. The branch instructions compare one register to 0 (BEQ, BGE, BGT, BLE, BLT, BNE) or its least significant bit to 0 (BLBC, BLBS) and then branch if the condition holds.

Arithmetic/logical (instruction formats)	R-R, R-I	R-R, R-I	R-R, R-I	R-R, R-I	R-R, R-I
Instruction name	**ARMv4**	**Thumb**	**SuperH**	**M32R**	**MIPS-16**
Add	ADD	ADD	ADD	ADD, ADDI, ADD3	ADDU, ADDIU
Add (trap if overflow)	ADDS; SWIVS	ADD; BVC .+4; SWI	ADDV	ADDV, ADDV3	—[1]
Subtract	SUB	SUB	SUB	SUB	SUBU
Subtract (trap if overflow)	SUBS; SWIVS	SUB; BVC .+1; SWI	SUBV	SUBV	—[1]
Multiply	MUL	MUL	MUL	MUL	MULT, MULTU
Multiply (trap if overflow)					—
Divide	—	—	DIV1, DIVoS, DIVoU	DIV, DIVU	DIV, DIVU
Divide (trap if overflow)	—	—			—
And	AND	AND	AND	AND, AND3	AND
Or	ORR	ORR	OR	OR, OR3	OR
Xor	EOR	EOR	XOR	XOR, XOR3	XOR
Load high part register	—	—		SETH	—[1]
Shift left logical	LSL[3]	LSL[2]	SHLL, SHLLn	SLL, SLLI, SLL3	SLLV, SLL
Shift right logical	LSR[3]	LSR[2]	SHRL, SHRLn	SRL, SRLI, SRL3	SRLV, SRL
Shift right arithmetic	ASR[3]	ASR[2]	SHRA, SHAD	SRA, SRAI, SRA3	SRAV, SRA
Compare	CMP,CMN, TST,TEQ	CMP, CMN, TST	CMP/cond, TST	CMP/I, CMPU/I	CMP/I[2], SLT/I, SLT/IU

FIGURE D.3.7 Embedded RISC arithmetic/logical instructions equivalent to MIPS core. Dashes mean the operation is not available in that architecture, or not synthesized in a few instructions. Such a sequence of instructions is shown separated by semicolons. If there are several choices of instructions equivalent to MIPS core, they are separated by commas. Thumb and MIPS-16 are just 16-bit instruction subsets of the ARM and MIPS architectures, so machines can switch modes and execute the full instruction set. We use —[1] to show sequences that are available in 32-bit mode but not 16-bit mode in Thumb or MIPS-16. The superscript 2 shows new instructions found only in 16-bit mode of Thumb or MIPS-16, such as CMP/I[2]. ARM includes shifts as part of every data operation instruction, so the shifts with superscript 3 are just a variation of a move instruction, such as LSR[3].

PA-RISC has many branch options, which we'll see in Section D.11. The most straightforward is a compare and branch instruction (COMB), which compares two registers, branches depending on the standard relations, and then tests the least significant bit of the result of the comparison.

ARM is similar to SPARC, in that it provides four traditional condition codes that are optionally set. 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. Like SPARC, the conditional version of the ARM branch instruction tests condition codes to determine all possible unsigned and signed relations.

Control (instruction formats)	B, J, C	B, J, C	B, J, C	B, J, C	B, J, C
Instruction name	**ARMv4**	**Thumb**	**SuperH**	**M32R**	**MIPS-16**
Branch on integer compare	B/cond	B/cond	BF, BT	BEQ, BNE, BC, BNC, B__Z	BEQZ[2], BNEZ[2], BTEQZ[2], BTNEZ[2]
Jump, jump register	MOV pc, ri	MOV pc, ri	BRA, JMP	BRA, JMP	B[2], JR
Call, call register	BL	BL	BSR, JSR	BL, JL	JAL, JALR, JALX[2]
Trap	SWI	SWI	TRAPA	TRAP	BREAK
Return from interrupt	MOVS pc, r14	—[1]	RTS	RTE	—[1]

FIGURE D.3.8 Embedded RISC control instructions equivalent to MIPS core. Thumb and MIPS-16 are just 16-bit instruction subsets of the ARM and MIPS architectures, so machines can switch modes and execute the full instruction set. We use —[1] to show sequences that are available in 32-bit mode but not 16-bit mode in Thumb or MIPS-16. The superscript 2 shows new instructions found only in 16-bit mode of Thumb or MIPS-16, such as BTEQZ[2].

Conventions	ARMv4	Thumb	SuperH	M32R	MIPS-16
Return address reg.	R14	R14	PR (special)	R14	RA (special)
No-op	MOV r0, r0	MOV r0, r0	NOP	NOP	SLL r0, r0
Operands, order	OP Rd, Rs1, Rs2	OP Rd, Rs1	OP Rs1, Rd	OP Rd, Rs1	OP Rd, Rs1, Rs2

FIGURE D.3.9 Conventions of embedded RISC instructions equivalent to MIPS core.

As we shall see in Section D.12, one unusual feature of ARM is that every instruction has the option of executing conditionally depending on the condition codes. (This bears similarities to the annulling option of PA-RISC, seen in Section D.11.)

Not surprisingly, Thumb follows ARM. The differences are that setting condition codes are not optional, the `TEQ` instruction is dropped, and there is no conditional execution of instructions.

The Hitachi SuperH uses a single T-bit condition that is set by compare instructions. Two branch instructions decide to branch if either the T bit is 1 (BT) or the T bit is 0 (BF). The two flavors of branches allow fewer comparison instructions.

Mitsubishi M32R also offers a single condition code bit (C) used for signed and unsigned comparisons (`CMP`, `CMPI`, `CMPU`, `CMPUI`) to see if one register is less than the other or not, similar to the MIPS set on less than instructions. Two branch instructions test to see if the C bit is 1 or 0: BC and BNC. The M32R also includes instructions to branch on equality or inequality of registers (`BEQ` and `BNE`) and all relations of a register to 0 (`BGEZ`, `BGTZ`, `BLEZ`, `BLTZ`, `BEQZ`, `BNEZ`). Unlike BC and BNC, these last instructions are all 32 bits wide.

MIPS-16 keeps set on less than instructions (`SLT`, `SLTI`, `SLTU`, `SLTIU`), but instead of putting the result in one of the eight registers, it is placed in a special register named T. MIPS-16 is always implemented in machines that also have the full 32-bit MIPS instructions and registers; hence, register T is really register 24 in the full MIPS architecture. The MIPS-16 branch instructions test to see if a register is or is not equal to zero (`BEQZ` and `BNEZ`). There are also instructions that branch

	Alpha	MIPS-64	PA-RISC 2.0	PowerPC	SPARCv9
Number of condition code bits (integer and FP)	0	8 FP	8 FP	8 × 4 both	2 × 4 integer, 4 × 2 FP
Basic compare instructions (integer and FP)	1 integer, 1 FP	1 integer, 1 FP	4 integer, 2 FP	4 integer, 2 FP	1 FP
Basic branch instructions (integer and FP)	1	2 integer, 1 FP	7 integer	1 both	3 integer, 1 FP
Compare register with register/const and branch	—	=, not=	=, not=, <, <=, >, >=, even, odd	—	—
Compare register to zero and branch	=, not=, <, <=, >, >=, even, odd	=, not=, <, <=, >, >=	=, not=, <, <=, >, >=, even, odd	—	=, not=, <, <=, >, >=

FIGURE D.3.10 Summary of five desktop RISC approaches to conditional branches. Floating-point branch on PA-RISC is accomplished by copying the FP status register into an integer register and then using the branch on bit instruction to test the FP comparison bit. Integer compare on SPARC is synthesized with an arithmetic instruction that sets the condition codes using r0 as the destination.

	ARMv4	Thumb	SuperH	M32R	MIPS-16
Number of condition code bits	4	4	1	1	1
Basic compare instructions	4	3	2	2	2
Basic branch instructions	1	1	2	3	2
Compare register with register/const and branch	—	—	=, >, >=	=, not=	—
Compare register to zero and branch	—	—	=, >, >=	=, not=, <, <=, >, >=	=, not=

FIGURE D.3.11 Summary of five embedded RISC approaches to conditional branches.

if register T is or is not equal to zero (`BTEQZ` and `BTNEZ`). To test if two registers are equal, MIPS added compare instructions (`CMP`, `CMPI`) that compute the exclusive OR of two registers and place the result in register T. Compare was added since MIPS-16 left out instructions to compare and branch if registers are equal or not (`BEQ` and `BNE`).

Figures D.3.10 and D.3.11 summarize the schemes used for conditional branches.

D.4 Instructions: Multimedia Extensions of the Desktop/Server RISCs

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 use 8 bits to represent each of the three primary colors plus 8 bits for the 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, but due to the infrequency of arithmetic operations on these data sizes in typical integer programs, there is little support beyond data transfers. The architects of the Intel i860, which was justified as a graphical accelerator within the company, recognized that many graphics and audio applications would perform the same operation on vectors of these data. Although a vector unit was beyond the transistor budget of the i860 in 1989, by partitioning the carry chains within a 64-bit ALU, it could perform simultaneous operations on short vectors of eight 8-bit operands, four 16-bit operands, or two 32-bit operands. The cost of such partitioned ALUs was small. Applications that lend themselves to such support include MPEG (video), games like DOOM (3-D graphics), Adobe Photoshop (digital photography), and teleconferencing (audio and image processing).

Like a virus, over time such multimedia support has spread to nearly every desktop microprocessor. HP was the first successful desktop RISC to include such support. As we shall see, this virus spread unevenly. The PowerPC is the only holdout, and rumors are that it is "running a fever."

These extensions have been called subword parallelism, vector, or SIMD (single-instruction, multiple data) (see Chapter 6). Since Intel marketing uses SIMD to describe the MMX extension of the 8086, that has become the popular name. Figure D.4.1 summarizes the support by architecture.

From Figure D.4.1, you can see that, in general, MIPS MDMX works on eight bytes or four halfwords per instruction, HP PA-RISC MAX2 works on four halfwords, SPARC VIS works on four halfwords or two words, and Alpha doesn't do much. The Alpha MAX operations are just byte versions of compare, min, max, and absolute difference, leaving it up to software to isolate fields and perform parallel adds, subtracts, and multiplies on bytes and halfwords. MIPS also added operations to work on two 32-bit floating-point operands per cycle, but they are considered part of MIPS V and not simply multimedia extensions (see Section D.7).

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. Commonly found in digital signal processors (see the next section), these saturating operations are helpful in routines for filtering.

These machines largely used existing register sets to hold operands: integer registers for Alpha and HP PA-RISC and floating-point registers for MIPS and Sun. Hence data transfers are accomplished with standard load and store instructions. MIPS also added a 192-bit (3*64) wide register to act as an accumulator for some operations. By having three times the native data width, it can be partitioned to accumulate either eight bytes with 24 bits per field or four halfwords with 48 bits

Instruction category	Alpha MAX	MIPS MDMX	PA-RISC MAX2	PowerPC	SPARC VIS
Add/subtract		8B, 4H	4H		4H, 2W
Saturating add/sub		8B, 4H	4H		
Multiply		8B, 4H			4B/H
Compare	8B (>=)	8B, 4H (=,<,<=)			4H, 2W (=, not=, >, <=)
Shift right/left		8B, 4H	4H		
Shift right arithmetic		4H	4H		
Multiply and add		8B, 4H			
Shift and add (saturating)			4H		
And/or/xor	8B, 4H, 2W	8B, 4H, 2W	8B, 4H, 2W		8B, 4H, 2W
Absolute difference	8B				8B
Max/min	8B, 4W	8B, 4H			
Pack (2*n* bits –> *n* bits)	2W->2B, 4H->4B	2*2W->4H, 2*4H->8B	2*4H->8B		2W->2H, 2W->2B, 4H->4B
Unpack/merge	2B->2W, 4B->4H	2*4B->8B, 2*2H->4H			4B->4H, 2*4B->8B
Permute/shuffle		8B, 4H	4H		
Register sets	Integer	Fl. Pt. + 192b Acc.	Integer		Fl. Pt.

FIGURE D.4.1 Summary of multimedia support for desktop RISCs. B stands for byte (8 bits), H for half word (16 bits), and W for word (32 bits). Thus 8B means an operation on eight bytes in a single instruction. Pack and unpack use the notation 2*2W to mean two operands each with two words. Note that MDMX has vector/scalar operations, where the scalar is specified as an element of one of the vector registers. This table is a simplification of the full multimedia architectures, leaving out many details. For example, MIPS MDMX includes instructions to multiplex between two operands, HP MAX2 includes an instruction to calculate averages, and SPARC VIS includes instructions to set registers to constants. Also, this table does not include the memory alignment operation of MDMX, MAX, and VIS.

per field. This wide accumulator can be used for add, subtract, and multiply/ add instructions. MIPS claims performance advantages of two to four times for the accumulator.

Perhaps the surprising conclusion of this table is the lack of consistency. The only operations found on all four are the logical operations (AND, OR, XOR), which do not need a partitioned ALU. If we leave out the frugal Alpha, then the only other common operations are parallel adds and subtracts on four halfwords.

Each manufacturer states that these are instructions intended to be used in hand-optimized subroutine libraries, an intention likely to be followed, as a compiler that works well with multimedia extensions of all desktop RISCs would be challenging.

D.5 Instructions: Digital Signal-Processing Extensions of the Embedded RISCs

One feature found in every digital signal processor (DSP) architecture is support for integer multiply-accumulate. The multiplies tend to be on shorter words than regular integers, such as 16 bits, and the accumulator tends to be on longer words, such as 64 bits. The reason for multiply-accumulate is to efficiently implement digital filters, common in DSP applications. Since Thumb and MIPS-16 are subset architectures, they do not provide such support. Instead, programmers should use the DSP or multimedia extensions found in the 32-bit mode instructions of ARM and MIPS-64.

Figure D.5.1 shows the size of the multiply, the size of the accumulator, and the operations and instruction names for the embedded RISCs. Machines with accumulator sizes greater than 32 and less than 64 bits will force the upper bits to remain as the sign bits, thereby "saturating" the add to set to maximum and minimum fixed-point values if the operations overflow.

	ARMv4	Thumb	SuperH	M32R	MIPS-16
Size of multiply	32B × 32B	—	32B × 32B, 16B × 16B	32B × 16B, 16B × 16B	—
Size of accumulator	32B/64B	—	32B/42B, 48B/64B	56B	—
Accumulator name	Any GPR or pairs of GPRs	—	MACH, MACL	ACC	—
Operations	32B/64B product + 64B accumulate signed/ unsigned	—	32B product + 42B/32B accumulate (operands in memory); 64B product + 64B/48B accumulate (operands in memory); clear MAC	32B/48B product + 64B accumulate, round, move	—
Corresponding instruction names	MLA, SMLAL, UMLAL	—	MAC, MACS, MAC.L, MAC.LS, CLRMAC	MACHI/MACLO, MACWHI/MACWLO, RAC, RACH, MVFACHI/ MVFACLO, MVTACHI/ MVTACLO	—

FIGURE D.5.1 Summary of five embedded RISC approaches to multiply-accumulate.

D.6 Instructions: Common Extensions to MIPS Core

Figures D.6.1 through D.6.7 list instructions not found in Figures D.3.5 through D.3.11 in the same four categories. Instructions are put in these lists if they appear in more than one of the standard architectures. The instructions are defined using the hardware description language defined in Figure D.6.8.

Although most of the categories are self-explanatory, a few bear comment:

- The "atomic swap" row means a primitive that can exchange a register with memory without interruption. This is useful for operating system semaphores in a uniprocessor as well as for multiprocessor synchronization (see Section 2.11 in Chapter 2).
- The 64-bit data transfer and operation rows show how MIPS, PowerPC, and SPARC define 64-bit addressing and integer operations. SPARC simply defines all register and addressing operations to be 64 bits, adding only

Name	Definition	Alpha	MIPS-64	PA-RISC 2.0	PowerPC	SPARCv9
Atomic swap R/M (for locks and semaphores)	Temp<–Rd; Rd<–Mem[x]; Mem[x]<–Temp	LDL/Q_L; STL/Q_C	LL; SC	— (see D.8)	LWARX; STWCX	CASA, CASX
Load 64-bit integer	$Rd <-_{64}$ Mem[x]	LDQ	LD	LDD	LD	LDX
Store 64-bit integer	Mem[x] $<-_{64}$ Rd	STQ	SD	STD	STD	STX
Load 32-bit integer unsigned	$Rd_{32..63} <-_{32}$ Mem[x]; $Rd_{0..31} <-_{32} 0$	LDL; EXTLL	LWU	LDW	LWZ	LDUW
Load 32-bit integer signed	$Rd_{32..63} <-_{32}$ Mem[x]; $Rd_{0..31} <-_{32} Mem[x]_0$	LDL	LW	LDW; EXTRD,S 63, 8	LWA	LDSW
Prefetch	Cache[x]<–*hint*	FETCH, FETCH_M*	PREF, PREFX	LDD, r0 LDW, r0	DCBT, DCBTST	PRE-FETCH
Load coprocessor	Coprocessor<– Mem[x]	—	LWCi	CLDWX, CLDWS	—	—
Store coprocessor	Mem[x]<– Coprocessor	—	SWCi	CSTWX, CSTWS	—	—
Endian	(Big/little endian?)	Either	Either	Either	Either	Either
Cache flush	(Flush cache block at this address)	ECB	CPOop	FDC, FIC	DCBF	FLUSH
Shared memory synchronization	(All prior data transfers complete before next data transfer may start)	WMB	SYNC	SYNC	SYNC	MEMBAR

FIGURE D.6.1 Data transfer instructions not found in MIPS core but found in two or more of the five desktop architectures. The load linked/store conditional pair of instructions gives Alpha and MIPS atomic operations for semaphores, allowing data to be read from memory, modified, and stored without fear of interrupts or other machines accessing the data in a multiprocessor (see Chapter 2). Prefetching in the Alpha to external caches is accomplished with FETCH and FETCH_M; on-chip cache prefetches use LD_Q A, R31, and LD_Y A. F31 is used in the Alpha 21164 (see Bhandarkar [1995], p. 190).

Name	Definition	Alpha	MIPS-64	PA-RISC 2.0	PowerPC	SPARCv9
64-bit integer arithmetic ops	$Rd <-_{64} Rs_1\ op_{64}\ Rs_2$	ADD, SUB, MUL	DADD, DSUB DMULT, DDIV	ADD, SUB, SHLADD, DS	ADD, SUBF, MULLD, DIVD	ADD, SUB, MULX, S/UDIVX
64-bit integer logical ops	$Rd <-_{64} Rs_1\ op_{64}\ Rs_2$	AND, OR, XOR	AND, OR, XOR	AND, OR, XOR	AND, OR, XOR	AND, OR, XOR
64-bit shifts	$Rd <-_{64} Rs_1\ op_{64}\ Rs_2$	SLL, SRA, SRL	DSLL/V, DSRA/V, DSRL/V	DEPD,Z EXTRD,S EXTRD,U	SLD, SRAD, SRLD	SLLX, SRAX, SRLX
Conditional move	if (cond) Rd<–Rs	CMOV_	MOVN/Z	SUBc, n; ADD	—	MOVcc, MOVr
Support for multiword integer add	CarryOut, Rd <– Rs1 + Rs2 + OldCarryOut	—	ADU; SLTU; ADDU, DADU; SLTU; DADDU	ADDC	ADDC, ADDE	ADDcc
Support for multiword integer sub	CarryOut, Rd <– Rs1 Rs2 + OldCarryOut	—	SUBU; SLTU; SUBU, DSUBU; SLTU; DSUBU	SUBB	SUBFC, SUBFE	SUBcc
And not	Rd <– Rs1 & ~(Rs2)	BIC	—	ANDCM	ANDC	ANDN
Or not	Rd <– Rs1 | ~(Rs2)	ORNOT	—	—	ORC	ORN
Add high immediate	$Rd_{0..15} <- Rs1_{0..15}$ + (Const<<16);	—	—	ADDIL (R-I)	ADDIS (R-I)	—
Coprocessor operations	(Defined by coprocessor)	—	COPi	COPR,i	—	IMPDEPi

FIGURE D.6.2 Arithmetic/logical instructions not found in MIPS core but found in two or more of the five desktop architectures.

Name	Definition	Alpha	MIPS-64	PA-RISC 2.0	PowerPC	SPARCv9
Optimized delayed branches	(Branch not always delayed)	—	BEQL, BNEL, B_ZL (<, >, <=, >=)	COMBT, n, COMBF, n	—	BPcc, A, FPBcc, A
Conditional trap	if (COND) {R31<–PC; PC <–0..0#i}	—	T_,,T_I (=, not=, <, >, <=, >=)	SUBc, n; BREAK	TW, TD, TWI, TDI	Tcc
No. control registers	Misc. regs (virtual memory, interrupts, . . .)	6	equiv. 12	32	33	29

FIGURE D.6.3 Control instructions not found in MIPS core but found in two or more of the five desktop architectures.

special instructions for 64-bit shifts, data transfers, and branches. MIPS includes the same extensions, plus it adds separate 64-bit signed arithmetic instructions. PowerPC adds 64-bit right shift, load, store, divide, and compare and has a separate mode determining whether instructions are interpreted as 32- or 64-bit operations; 64-bit operations will not work in a machine that

Name	Definition	Alpha	MIPS-64	PA-RISC 2.0	PowerPC	SPARCv9
Multiply and add	Fd <– (Fs1 × Fs2) + Fs3	—	MADD.S/D	FMPYFADD sgl/dbl	FMADD/S	
Multiply and sub	Fd <– (Fs1 × Fs2) – Fs3	—	MSUB.S/D		FMSUB/S	
Neg mult and add	Fd <– -((Fs1 × Fs2) + Fs3)	—	NMADD.S/D	FMPYFNEG sgl/dbl	FNMADD/S	
Neg mult and sub	Fd <– -((Fs1 × Fs2) – Fs3)	—	NMSUB.S/D		FNMSUB/S	
Square root	Fd <– SQRT(Fs)	SQRT_	SQRT.S/D	FSQRT sgl/dbl	FSQRT/S	FSQRTS/D
Conditional move	if (cond) Fd<–Fs	FCMOV_	MOVF/T, MOVF/T.S/D	FTESTFCPY	—	FMOVcc
Negate	Fd <– Fs ^ x80000000	CPYSN	NEG.S/D	FNEG sgl/dbl	FNEG	FNEGS/D/Q
Absolute value	Fd <– Fs & x7FFFFFFF	—	ABS.S/D	FABS/dbl	FABS	FABSS/D/Q

FIGURE D.6.4 Floating-point instructions not found in MIPS core but found in two or more of the five desktop architectures.

Name	Definition	ARMv4	Thumb	SuperH	M32R	MIPS-16
Atomic swap R/M (for semaphores)	Temp<–Rd; Rd<–Mem[x]; Mem[x]<–Temp	SWP, SWPB	—[1]	(see TAS)	LOCK; UNLOCK	—[1]
Memory management unit	Paged address translation	Via coprocessor instructions	—[1]	LDTLB		—[1]
Endian	(Big/little endian?)	Either	Either	Either	Big	Either

FIGURE D.6.5 Data transfer instructions not found in MIPS core but found in two or more of the five embedded architectures. We use —[1] to show sequences that are available in 32-bit mode but not 16-bit mode in Thumb or MIPS-16.

only supports 32-bit mode. PA-RISC is expanded to 64-bit addressing and operations in version 2.0.

- The "prefetch" instruction supplies an address and hint to the implementation about the data. Hints include whether the data are likely to be read or written soon, likely to be read or written only once, or likely to be read or written many times. Prefetch does not cause exceptions. MIPS has a version that adds two registers to get the address for floating-point programs, unlike nonfloating-point MIPS programs.
- In the "Endian" row, "Big/little" means there is a bit in the program status register that allows the processor to act either as big endian or little endian (see Appendix A). This can be accomplished by simply complementing some of the least significant bits of the address in data transfer instructions.

- The "shared memory synchronization" helps with cache-coherent multiprocessors: all loads and stores executed before the instruction must complete before loads and stores after it can start. (See Chapter 2.)
- The "coprocessor operations" row lists several categories that allow for the processor to be extended with special-purpose hardware.

Name	Definition	ARMv4	Thumb	SuperH	M32R	MIPS-16
Load immediate	Rd<—Imm	MOV	MOV	MOV, MOVA	LDI, LD24	LI
Support for multiword integer add	CarryOut, Rd <— Rd + Rs1 + OldCarryOut	ADCS	ADC	ADDC	ADDX	—[1]
Support for multiword integer sub	CarryOut, Rd <— Rd – Rs1 + OldCarryOut	SBCS	SBC	SUBC	SUBX	—[1]
Negate	Rd <— 0 – Rs1		NEG[2]	NEG	NEG	NEG
Not	Rd <— ~(Rs1)	MVN	MVN	NOT	NOT	NOT
Move	Rd <— Rs1	MOV	MOV	MOV	MV	MOVE
Rotate right	Rd <— Rs i, >> $Rd_{0...i-1}$ <— $Rs_{31-i...31}$	ROR	ROR	ROTC		
And not	Rd <— Rs1 & ~(Rs2)	BIC	BIC			

FIGURE D.6.6 Arithmetic/logical instructions not found in MIPS core but found in two or more of the five embedded architectures. We use —[1] to show sequences that are available in 32-bit mode but not in 16-bit mode in Thumb or MIPS-16. The superscript 2 shows new instructions found only in 16-bit mode of Thumb or MIPS-16, such as NEG[2].

Name	Definition	ARMv4	Thumb	SuperH	M32R	MIPS-16
No. control registers	Misc. registers	21	29	9	5	36

FIGURE D.6.7 Control information in the five embedded architectures.

One difference that needs a longer explanation is the optimized branches. Figure D.6.9 shows the options. The Alpha and PowerPC offer branches that take effect immediately, like branches on earlier architectures. To accelerate branches, these machines use branch prediction (see Chapter 4). All the rest of the desktop RISCs offer delayed branches. The embedded RISCs generally do not support delayed branch, with the exception of SuperH, which has it as an option.

The other three desktop RISCs provide a version of delayed branch that makes it easier to fill the delay slot. The SPARC "annulling" branch executes the instruction in the delay slot only if the branch is taken; otherwise the instruction is annulled. This means the instruction at the target of the branch can safely be copied into the delay slot, since it will only be executed if the branch is taken. The restrictions are that the target is not another branch and that the target is known at compile time. (SPARC also offers a nondelayed jump because an unconditional branch with the annul bit set does not execute the following instruction.) Later versions of the MIPS

Notation	Meaning	Example	Meaning
<--	Data transfer. Length of transfer is given by the destination's length; the length is specified when not clear.	`Regs[R1]<--Regs[R2];`	Transfer contents of R2 to R1. Registers have a fixed length, so transfers shorter than the register size must indicate which bits are used.
M	Array of memory accessed in bytes. The starting address for a transfer is indicated as the index to the memory array.	`Regs[R1]<--M[x];`	Place contents of memory location x into `R1`. If a transfer starts at `M[i]` and requires 4 bytes, the transferred bytes are `M[i]`, `M[i+1]`, `M[i+2]`, and `M[i+3]`.
<--n	Transfer an *n*-bit field, used whenever length of transfer is not clear.	$M[y] \leftarrow\!\!-_{16} M[x]$;	Transfer 16 bits starting at memory location x to memory location y. The length of the two sides should match.
X_n	Subscript selects a bit.	$Regs[R1]_0 \leftarrow\!\!- 0$;	Change sign bit of `R1` to `0`. (Bits are numbered from MSB starting at 0.)
$X_{m..n}$	Subscript selects a field.	$Regs[R3]_{24..31} \leftarrow\!\!- M[x]$;	Moves contents of memory location x into low-order byte of R3.
X^n	Superscript replicates a bit field.	$Regs[R3]_{0..23} \leftarrow\!\!- 024$;	Sets high-order three bytes of `R3` to 0.
##	Concatenates two fields.	$Regs[R3] \leftarrow\!\!-^{24} 0 \#\# M[x]$; $F2\#\#F3 \leftarrow\!\!-_{64} M[x]$;	Moves contents of location x into low byte of `R3`; clears upper three bytes. Moves 64 bits from memory starting at location x; 1st 32 bits go into `F2`, 2nd 32 into `F3`.
*, &	Dereference a pointer; get the address of a variable.	`p*<--&x;`	Assign to object pointed to by p the address of the variable x.
<<, >>	C logical shifts (left, right).	`Regs[R1] << 5`	Shift `R1` left 5 bits.
==, !=, >, <, >=, <=	C relational operators; equal, not equal, greater, less, greater or equal, less or equal.	`(Regs[R1]== Regs[R2]) & (Regs[R3]!=Regs[R4])`	True if contents of `R1` equal the contents of `R2` and contents of `R3` do not equal the contents of `R4`.
&, |, ^, !	C bitwise logical operations: AND, OR, exclusive OR, and complement.	`(Regs[R1] & (Regs[R2]| Regs[R3]))`	Bitwise AND of `R1` and bitwise OR of `R2` and `R3`.

FIGURE D.6.8 Hardware description notation (and some standard C operators).

	(Plain) branch	Delayed branch	Annulling delayed branch	
Found in architectures	Alpha, PowerPC, ARM, Thumb, SuperH, M32R, MIPS-16	MIPS-64, PA-RISC, SPARC, SuperH	MIPS-64, SPARC	PA-RISC
Execute following instruction	Only if branch *not* taken	Always	Only if branch taken	If forward branch *not* taken or backward branch taken

FIGURE D.6.9 When the instruction following the branch is executed for three types of branches.

architecture have added a branch likely instruction that also annuls the following instruction if the branch is not taken. PA-RISC allows almost any instruction to annul the next instruction, including branches. Its "nullifying" branch option will execute the next instruction depending on the direction of the branch and whether it is taken (i.e., if a forward branch is not taken or a backward branch is taken). Presumably this choice was made to optimize loops, allowing the instructions following the exit branch and the looping branch to execute in the common case.

Now that we have covered the similarities, we will focus on the unique features of each architecture. We first cover the desktop/server RISCs, ordering them by length of description of the unique features from shortest to longest, and then the embedded RISCs.

D.7 Instructions Unique to MIPS-64

MIPS has gone through five generations of instruction sets, and this evolution has generally added features found in other architectures. Here are the salient unique features of MIPS, the first several of which were found in the original instruction set.

Nonaligned Data Transfers

MIPS has special instructions to handle misaligned words in memory. A rare event in most programs, it is included for supporting 16-bit minicomputer applications and for doing `memcpy` and `strcpy` faster. Although most RISCs trap if you try to load a word or store a word to a misaligned address, on all architectures misaligned words can be accessed without traps by using four load byte instructions and then assembling the result using shifts and logical ORs. The MIPS load and store word left and right instructions (`LWL, LWR, SWL, SWR`) allow this to be done in just two instructions: LWL loads the left portion of the register and LWR loads the right portion of the register. SWL and SWR do the corresponding stores. Figure D.7.1 shows how they work. There are also 64-bit versions of these instructions.

Remaining Instructions

Below is a list of the remaining unique details of the MIPS-64 architecture:

- *NOR*—This logical instruction calculates ~(Rs1 | Rs2).
- *Constant shift amount*—Nonvariable shifts use the 5-bit constant field shown in the register-register format in Figure D.2.3.
- *SYSCALL*—This special trap instruction is used to invoke the operating system.

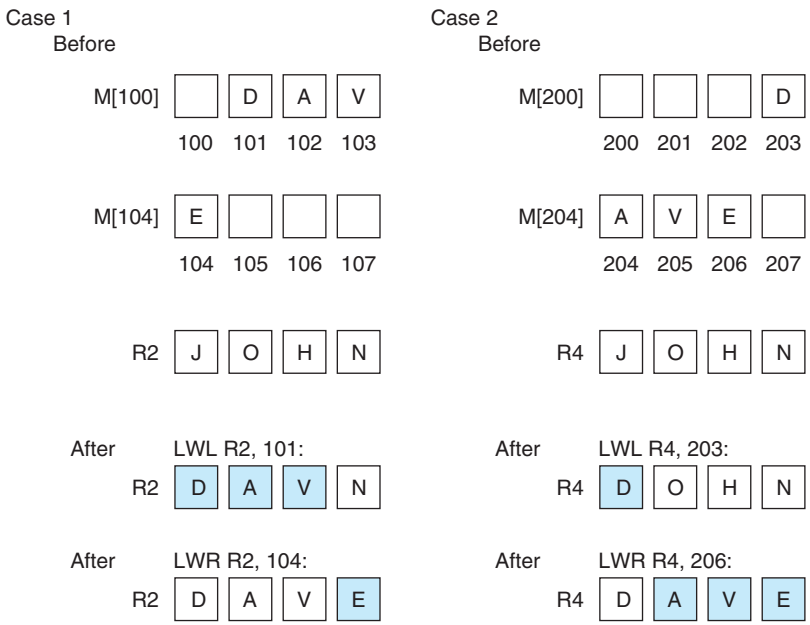


FIGURE D.7.1 MIPS instructions for unaligned word reads. This figure assumes operation in big-endian mode. Case 1 first loads the three bytes 101, 102, and 103 into the left of R2, leaving the least significant byte undisturbed. The following LWR simply loads byte 104 into the least significant byte of R2, leaving the other bytes of the register unchanged using LWL. Case 2 first loads byte 203 into the most significant byte of R4, and the following LWR loads the other three bytes of R4 from memory bytes 204, 205, and 206. LWL reads the word with the first byte from memory, shifts to the left to discard the unneeded byte(s), and changes only those bytes in Rd. The byte(s) transferred are from the first byte to the lowest-order byte of the word. The following LWR addresses the last byte, right-shifts to discard the unneeded byte(s), and finally changes only those bytes of Rd. The byte(s) transferred are from the last byte up to the highest-order byte of the word. Store word left (SWL) is simply the inverse of LWL, and store word right (SWR) is the inverse of LWR. Changing to little-endian mode flips which bytes are selected and discarded. (If big-little, left-right, load-store seem confusing, don't worry; they work!)

- *Move to/from control registers*—`CTCi` and `CFCi` move between the integer registers and control registers.
- *Jump/call not PC-relative*—The 26-bit address of jumps and calls is not added to the PC. It is shifted left two bits and replaces the lower 28 bits of the PC. This would only make a difference if the program were located near a 256 MB boundary.
- *TLB instructions—Translation-lookaside buffer* (TLB) misses were handled in software in MIPS I, so the instruction set also had instructions for manipulating the registers of the TLB (see Chapter 5 for more on TLBs). These registers are considered part of the "system coprocessor." Since MIPS I

the instructions differ among versions of the architecture; they are more part of the implementations than part of the instruction set architecture.

- *Reciprocal and reciprocal square root*—These instructions, which do not follow IEEE 754 guidelines of proper rounding, are included apparently for applications that value speed of divide and square root more than they value accuracy.
- *Conditional procedure call instructions*—`BGEZAL` saves the return address and branches if the content of Rs1 is greater than or equal to zero, and `BLTZAL` does the same for less than zero. The purpose of these instructions is to get a PC-relative call. (There are "likely" versions of these instructions as well.)
- *Parallel single-precision floating-point operations*—As well as extending the architecture with parallel integer operations in MDMX, MIPS-64 also supports two parallel 32-bit floating-point operations on 64-bit registers in a single instruction. "Paired single" operations include add (`ADD.PS`), subtract (`SUB.PS`), compare (`C.__.PS`), convert (`CVT.PS.S, CVT.S.PL, CVT.S.PU`), negate (`NEG.PS`), absolute value (`ABS.PS`), move (`MOV.PS, MOVF.PS, MOVT.PS`), multiply (`MUL.PS`), multiply-add (`MADD.PS`), and multiply-subtract (`MSUB.PS`).

There is no specific provision in the MIPS architecture for floating-point execution to proceed in parallel with integer execution, but the MIPS implementations of floating point allow this to happen by checking to see if arithmetic interrupts are possible early in the cycle. Normally, exception detection would force serialization of execution of integer and floating-point operations.

D.8 Instructions Unique to Alpha

The Alpha was intended to be an architecture that made it easy to build high-performance implementations. Toward that goal, the architects originally made two controversial decisions: imprecise floating-point exceptions and no byte or halfword data transfers.

To simplify pipelined execution, Alpha does not require that an exception should act as if no instructions past a certain point are executed and that all before that point have been executed. It supplies the `TRAPB` instruction, which stalls until all prior arithmetic instructions are guaranteed to complete without incurring arithmetic exceptions. In the most conservative mode, placing one TRAPB per exception-causing instruction slows execution by roughly five times but provides precise exceptions (see Darcy and Gay [1996]).

Code that does not include TRAPB does not obey the IEEE 754 floating-point standard. The reason is that parts of the standard (NaNs, infinities, and denormals) are implemented in software on Alpha, as they are on many other microprocessors. To implement these operations in software, however, programs must find the offending instruction and operand values, which cannot be done with imprecise interrupts!

When the architecture was developed, it was believed by the architects that byte loads and stores would slow down data transfers. Byte loads require an extra shifter in the data transfer path, and byte stores require that the memory system perform a read-modify-write for memory systems with error correction codes, since the new ECC value must be recalculated. This omission meant that byte stores required the sequence load word, replaced the desired byte, and then stored the word. (Inconsistently, floating-point loads go through considerable byte swapping to convert the obtuse VAX floating-point formats into a canonical form.)

To reduce the number of instructions to get the desired data, Alpha includes an elaborate set of byte manipulation instructions: extract field and zero rest of a register (EXTxx), insert field (INSxx), mask rest of a register (MSKxx), zero fields of a register (ZAP), and compare multiple bytes (CMPGE).

Apparently, the implementors were not as bothered by load and store byte as were the original architects. Beginning with the shrink of the second version of the Alpha chip (21164A), the architecture does include loads and stores for bytes and halfwords.

Remaining Instructions

Below is a list of the remaining unique instructions of the Alpha architecture:

- *PAL code*—To provide the operations that the VAX performed in microcode, Alpha provides a mode that runs with all privileges enabled, interrupts disabled, and virtual memory mapping turned off for instructions. PAL (privileged architecture library) code is used for TLB management, atomic memory operations, and some operating system primitives. PAL code is called via the CALL_PAL instruction.
- *No divide*—Integer divide is not supported in hardware.
- "*Unaligned*" *load-store*—LDQ_U and STQ_U load and store 64-bit data using addresses that ignore the least significant three bits. Extract instructions then select the desired unaligned word using the lower address bits. These instructions are similar to LWL/R, SWL/R in MIPS.
- *Floating-point single precision represented as double precision*—Single-precision data are kept as conventional 32-bit formats in memory but are converted to 64-bit double-precision format in registers.
- *Floating-point register F31 is fixed at zero*—To simplify comparisons to zero.

- *VAX floating-point formats*—To maintain compatibility with the VAX architecture, in addition to the IEEE 754 single- and double-precision formats called S and T, Alpha supports the VAX single- and double-precision formats called F and G, but not VAX format D. (D had too narrow an exponent field to be useful for double precision and was replaced by G in VAX code.)
- *Bit count instructions*—Version 3 of the architecture added instructions to count the number of leading zeros (`CTLZ`), count the number of trailing zeros (`CTTZ`), and count the number of ones in a word (`CTPOP`). Originally found on Cray computers, these instructions help with decryption.

D.9 Instructions Unique to SPARC v9

Several features are unique to SPARC.

Register Windows

The primary unique feature of SPARC is register windows, an optimization for reducing register traffic on procedure calls. Several banks of registers are used, with a new one allocated on each procedure call. Although this could limit the depth of procedure calls, the limitation is avoided by operating the banks as a circular buffer, providing unlimited depth. The knee of the cost/performance curve seems to be six to eight banks.

SPARC can have between two and 32 windows, typically using eight registers each for the globals, locals, incoming parameters, and outgoing parameters. (Given that each window has 16 unique registers, an implementation of SPARC can have as few as 40 physical registers and as many as 520, although most have 128 to 136, so far.) Rather than tie window changes with call and return instructions, SPARC has the separate instructions `SAVE` and `RESTORE`. `SAVE` is used to "save" the caller's window by pointing to the next window of registers in addition to performing an add instruction. The trick is that the source registers are from the caller's window of the addition operation, while the destination register is in the callee's window. SPARC compilers typically use this instruction for changing the stack pointer to allocate local variables in a new stack frame. `RESTORE` is the inverse of SAVE, bringing back the caller's window while acting as an add instruction, with the source registers from the callee's window and the destination register in the caller's window. This automatically deallocates the stack frame. Compilers can also make use of it for generating the callee's final return value.

The danger of register windows is that the larger number of registers could slow down the clock rate. This was not the case for early implementations. The SPARC architecture (with register windows) and the MIPS R2000 architecture (without)

have been built in several technologies since 1987. For several generations, the SPARC clock rate has not been slower than the MIPS clock rate for implementations in similar technologies, probably because cache access times dominate register access times in these implementations. The current-generation machines took different implementation strategies—in order versus out of order—and it's unlikely that the number of registers by themselves determined the clock rate in either machine. Recently, other architectures have included register windows: Tensilica and IA-64.

Another data transfer feature is alternate space option for loads and stores. This simply allows the memory system to identify memory accesses to input/output devices, or to control registers for devices such as the cache and memory management unit.

Fast Traps

Version 9 SPARC includes support to make traps fast. It expands the single level of traps to at least four levels, allowing the window overflow and underflow trap handlers to be interrupted. The extra levels mean the handler does not need to check for page faults or misaligned stack pointers explicitly in the code, thereby making the handler faster. Two new instructions were added to return from this multilevel handler: `RETRY` (which retries the interrupted instruction) and DONE (which does not). To support user-level traps, the instruction RETURN will return from the trap in nonprivileged mode.

Support for LISP and Smalltalk

The primary remaining arithmetic feature is tagged addition and subtraction. The designers of SPARC spent some time thinking about languages like LISP and Smalltalk, and this influenced some of the features of SPARC already discussed: register windows, conditional trap instructions, calls with 32-bit instruction addresses, and multiword arithmetic (see Taylor et al. [1986] and Ungar et al. [1984]). A small amount of support is offered for tagged data types with operations for addition, subtraction, and, hence, comparison. The two least significant bits indicate whether the operand is an integer (coded as 00), so `TADDcc` and `TSUBcc` set the overflow bit if either operand is not tagged as an integer or if the result is too large. A subsequent conditional branch or trap instruction can decide what to do. (If the operands are not integers, software recovers the operands, checks the types of the operands, and invokes the correct operation based on those types.) It turns out that the misaligned memory access trap can also be put to use for tagged data, since loading from a pointer with the wrong tag can be an invalid access. Figure D.9.1 shows both types of tag support.

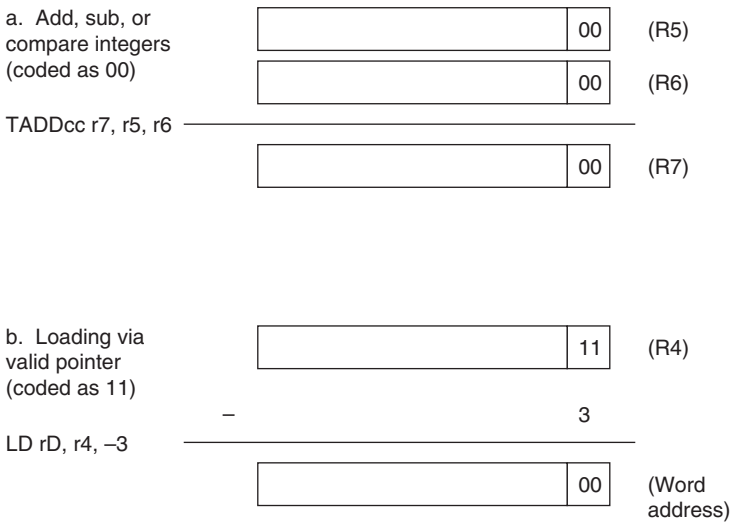


FIGURE D.9.1 SPARC uses the two least significant bits to encode different data types for the tagged arithmetic instructions. a. Integer arithmetic takes a single cycle as long as the operands and the result are integers. b. The misaligned trap can be used to catch invalid memory accesses, such as trying to use an integer as a pointer. For languages with paired data like LISP, an offset of − 3 can be used to access the even word of a pair (CAR) and + 1 can be used for the odd word of a pair (CDR).

Overlapped Integer and Floating-Point Operations

SPARC allows floating-point instructions to overlap execution with integer instructions. To recover from an interrupt during such a situation, `SPARC` has a queue of pending floating-point instructions and their addresses. `RDPR` allows the processor to empty the queue. The second floating-point feature is the inclusion of floating-point square root instructions `FSQRTS`, `FSQRTD`, and `FSQRTQ`.

Remaining Instructions

The remaining unique features of SPARC are as follows:

- `JMPL` uses `Rd` to specify the return address register, so specifying `r31` makes it similar to `JALR` in `MIPS` and specifying `r0` makes it like `JR`.
- `LDSTUB` loads the value of the byte into `Rd` and then stores `FF16` into the addressed byte. This version 8 instruction can be used to implement synchronization (see Chapter 2).
- CASA (*CASXA*) atomically compares a value in a processor register to a 32-bit (64-bit) value in memory; if and only if they are equal, it swaps the value in memory with the value in a second processor register. This version 9

instruction can be used to construct wait-free synchronization algorithms that do not require the use of locks.

- `XNOR` calculates the exclusive OR with the complement of the second operand.
- `BPcc`, `BPr`, and `FBPcc` include a branch prediction bit so that the compiler can give hints to the machine about whether a branch is likely to be taken or not.
- `ILLTRAP` causes an illegal instruction trap. Muchnick [1988] explains how this is used for proper execution of aggregate returning procedures in C.
- `POPC` counts the number of bits set to one in an operand, also found in the third version of the Alpha architecture.
- *Nonfaulting loads* allow compilers to move load instructions ahead of conditional control structures that control their use. Hence, nonfaulting loads will be executed speculatively.
- *Quadruple-precision floating-point arithmetic and data transfer* allow the floating-point registers to act as eight 128-bit registers for floating-point operations and data transfers.
- *Multiple-precision floating-point results for multiply* mean that two single-precision operands can result in a double-precision product and two double-precision operands can result in a quadruple-precision product. These instructions can be useful in complex arithmetic and some models of floating-point calculations.

D.10 Instructions Unique to PowerPC

PowerPC is the result of several generations of IBM commercial RISC machines—IBM RT/PC, IBM Power1, and IBM Power2—plus the Motorola 8800.

Branch Registers: Link and Counter

Rather than dedicate one of the 32 general-purpose registers to save the return address on procedure call, PowerPC puts the address into a special register called the *link register*. Since many procedures will return without calling another procedure, the link doesn't always have to be saved. Making the return address a special register makes the return jump faster, since the hardware need not go through the register read pipeline stage for return jumps.

In a similar vein, PowerPC has a *count register* to be used in *for* loops where the program iterates a fixed number of times. By using a special register, the branch

hardware can determine quickly whether a branch based on the count register is likely to branch, since the value of the register is known early in the execution cycle. Tests of the value of the count register in a branch instruction will automatically decrement the count register.

Given that the count register and link register are already located with the hardware that controls branches, and that one of the problems in branch prediction is getting the target address early in the pipeline, the PowerPC architects decided to make a second use of these registers. Either register can hold a target address of a conditional branch. Thus, PowerPC supplements its basic conditional branch with two instructions that get the target address from these registers (`BCLR`, `BCCTR`).

Remaining Instructions

Unlike most other RISC machines, register 0 is not hardwired to the value 0. It cannot be used as a base register—that is, it generates a 0 in this case—but in base + index addressing it can be used as the index. The other unique features of the PowerPC are as follows:

- *Load multiple and store multiple* save or restore up to 32 registers in a single instruction.
- `LSW` and `STSW` permit fetching and storing of fixed- and variable-length strings that have arbitrary alignment.
- *Rotate with mask* instructions support bit field extraction and insertion. One version rotates the data and then performs logical `AND` with a mask of ones, thereby extracting a field. The other version rotates the data but only places the bits into the destination register where there is a corresponding 1 bit in the mask, thereby inserting a field.
- *Algebraic right shift* sets the carry bit (`CA`) if the operand is negative and any 1 bits are shifted out. Thus, a signed divide by any constant power of two that rounds toward 0 can be accomplished with an `SRAWI` followed by `ADDZE`, which adds `CA` to the register.
- `CBTLZ` will count leading zeros.
- `SUBFIC` computes (immediate - RA), which can be used to develop a one's or two's complement.
- *Logical shifted immediate* instructions shift the 16-bit immediate to the left 16 bits before performing AND, OR, or XOR.

D.11 Instructions Unique to PA-RISC 2.0

PA-RISC was expanded slightly in 1990 with version 1.1 and changed significantly in 2.0 with 64-bit extensions in 1996. PA-RISC perhaps has the most unusual features of any desktop RISC machine. For example, it has the most addressing modes and instruction formats, and, as we shall see, several instructions that are really the combination of two simpler instructions.

Nullification

As shown in Figure D.6.9, several RISC machines can choose not to execute the instruction following a delayed branch to improve utilization of the branch slot. This is called *nullification* in PA-RISC, and it has been generalized to apply to any arithmetic/logical instruction as well as to all branches. Thus, an `add` instruction can add two operands, store the sum, and cause the following instruction to be skipped if the sum is zero. Like conditional move instructions, nullification allows PA-RISC to avoid branches in cases where there is just one instruction in the *then* part of an *if* statement.

A Cornucopia of Conditional Branches

Given nullification, PA-RISC did not need to have separate conditional branch instructions. The inventors could have recommended that nullifying instructions precede unconditional branches, thereby simplifying the instruction set. Instead, PA-RISC has the largest number of conditional branches of any RISC machine. Figure D.11.1 shows the conditional branches of PA-RISC. As you can see, several are really combinations of two instructions.

Synthesized Multiply and Divide

PA-RISC provides several primitives so that multiply and divide can be synthesized in software. Instructions that shift one operand 1, 2, or 3 bits and then add, trapping or not on overflow, are useful in multiplies. (Alpha also includes instructions that multiply the second operand of adds and subtracts by 4 or by 8: `S4ADD`, `S8ADD`, `S4SUB`, and `S8SUB`.) The divide step performs the critical step of nonrestoring divide, adding or subtracting depending on the sign of the prior result. Magenheimer et al. [1988] measured the size of operands in multiplies and divides to show how well the multiply step would work. Using these data for C programs, Muchnick [1988] found that by making special cases, the average multiply by a constant takes six clock cycles and the multiply of variables takes 24 clock cycles. PA- RISC has ten instructions for these operations.

Name	Instruction	Notation	
COMB	Compare and branch	`if (cond(Rs1,Rs2))`	`{PC <-- PC + offset12}`
COMIB	Compare immediate and branch	`if (cond(imm5,Rs2))`	`{PC <-- PC + offset12}`
MOVB	Move and branch	`Rs2 <-- Rs1, if (cond(Rs1,0))`	`{PC <-- PC + offset12}`
MOVIB	Move immediate and branch	`Rs2 <-- imm5, if (cond(imm5,0))`	`{PC <-- PC + offset12}`
ADDB	Add and branch	`Rs2 <-- Rs1 + Rs2, if (cond(Rs1 + Rs2,0))`	`{PC <-- PC + offset12}`
ADDIB	Add immediate and branch	`Rs2 <-- imm5 + Rs2, if (cond(imm5 + Rs2,0))`	`{PC <-- PC + offset12}`
BB	Branch on bit	`if (cond(Rsp,0))`	`{PC <-- PC + offset12}`
BVB	Branch on variable bit	`if (cond(Rssar,0))`	`{PC <-- PC + offset12}`

FIGURE D.11.1 The PA-RISC conditional branch instructions. The 12-bit offset is called `offset12` in this table, and the 5-bit immediate is called `imm5`. The 16 conditions are =, <, <=, odd, signed overflow, unsigned no overflow, zero or no overflow unsigned, never, and their respective complements. The BB instruction selects one of the 32 bits of the register and branches depending on whether its value is 0 or 1. The BVB selects the bit to branch using the shift amount register, a special-purpose register. The subscript notation specifies a bit field.

The original SPARC architecture used similar optimizations, but with increasing numbers of transistors the instruction set was expanded to include full multiply and divide operations. PA-RISC gives some support along these lines by putting a full 32-bit integer multiply in the floating-point unit; however, the integer data must first be moved to floating-point registers.

Decimal Operations

COBOL programs will compute on decimal values, stored as 4 bits per digit, rather than converting back and forth between binary and decimal. PA-RISC has instructions that will convert the sum from a normal 32-bit add into proper decimal digits. It also provides logical and arithmetic operations that set the condition codes to test for carries of digits, bytes, or halfwords. These operations also test whether bytes or halfwords are zero. These operations would be useful in arithmetic on 8-bit ASCII characters. Five PA-RISC instructions provide decimal support.

Remaining Instructions

Here are some remaining PA-RISC instructions:

- *Branch vectored* shifts an index register left 3 bits, adds it to a base register, and then branches to the calculated address. It is used for *case* statements.
- *Extract* and *deposit* instructions allow arbitrary bit fields to be selected from or inserted into registers. Variations include whether the extracted field is sign-extended, whether the bit field is specified directly in the instruction or indirectly in another register, and whether the rest of the register is set to zero or left unchanged. PA-RISC has 12 such instructions.

- To simplify use of 32-bit address constants, PA-RISC includes `ADDIL`, which adds a left-adjusted 21-bit constant to a register and places the result in register 1. The following data transfer instruction uses offset addressing to add the lower 11 bits of the address to register 1. This pair of instructions allows PA-RISC to add a 32-bit constant to a base register, at the cost of changing register 1.
- PA-RISC has nine debug instructions that can set breakpoints on instruction or data addresses and return the trapped addresses.
- *Load* and *clear* instructions provide a semaphore or lock that reads a value from memory and then writes zero.
- *Store bytes short* optimizes unaligned data moves, moving either the leftmost or the rightmost bytes in a word to the effective address, depending on the instruction options and condition code bits.
- Loads and stores work well with caches by having options that give hints about whether to load data into the cache if it's not already in the cache. For example, a load with a destination of register 0 is defined to be a software-controlled cache prefetch.
- PA-RISC 2.0 extended cache hints to stores to indicate block copies, recommending that the processor not load data into the cache if it's not already in the cache. It also can suggest that on loads and stores, there is spatial locality to prepare the cache for subsequent sequential accesses.
- PA-RISC 2.0 also provides an optional branch target stack to predict indirect jumps used on subroutine returns. Software can suggest which addresses get placed on and removed from the branch target stack, but hardware controls whether or not these are valid.
- *Multiply/add* and *multiply/subtract* are floating-point operations that can launch two independent floating-point operations in a single instruction in addition to the fused multiply/add and fused multiply/negate/add introduced in version 2.0 of PA-RISC.

D.12 Instructions Unique to ARM

It's hard to pick the most unusual feature of ARM, but perhaps it is the conditional execution of instructions. Every instruction starts with a 4-bit field that determines whether it will act as a 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.

The 12-bit immediate field has a novel interpretation. The 8 least significant bits are zero-extended to a 32-bit value, then rotated right the number of bits specified in the first 4 bits of the field multiplied by two. Whether this split actually catches more immediates than a simple 12-bit field would be an interesting study. One advantage is that this scheme can represent all powers of two in a 32-bit word.

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. Once again, it would be interesting to see how often operations like rotate-and-add, shift-right-and-test, and so on occur in ARM programs.

Remaining Instructions

Below is a list of the remaining unique instructions of the ARM architecture:

- *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—offering up to four times the bandwidth of a single register load-store—and today, block copies are the most important use.
- *Reverse subtract*—RSB allows the first register to be subtracted from the immediate or shifted register. RSC does the same thing, but includes the carry when calculating the difference.
- *Long multiplies*—Similarly to MIPS, Hi and Lo registers get the 64-bit signed product (SMULL) or the 64-bit unsigned product (UMULL).
- *No divide*—Like the Alpha, integer divide is not supported in hardware.
- *Conditional trap*—A common extension to the MIPS core found in desktop RISCs (Figures D.6.1 through D.6.4), it comes for free in the conditional execution of all ARM instructions, including SWI.
- *Coprocessor interface*—Like many of the desktop RISCs, ARM defines a full set of coprocessor instructions: data transfer, moves between general-purpose and coprocessor registers, and coprocessor operations.
- *Floating-point architecture*—Using the coprocessor interface, a floating-point architecture has been defined for ARM. It was implemented as the FPA10 coprocessor.
- *Branch and exchange instruction sets*—The BX instruction is the transition between ARM and Thumb, using the lower 31 bits of the register to set the PC and the most significant bit to determine if the mode is ARM (1) or Thumb (0).

D.13 Instructions Unique to Thumb

In the ARM version 4 model, frequently executed procedures will use ARM instructions to get maximum performance, with the less frequently executed ones using Thumb to reduce the overall code size of the program. Since typically only a few procedures dominate execution time, the hope is that this hybrid gets the best of both worlds.

Although Thumb instructions are translated by the hardware into conventional ARM instructions for execution, there are several restrictions. First, conditional execution is dropped from almost all instructions. Second, only the first eight registers are easily available in all instructions, with the stack pointer, link register, and program counter used implicitly in some instructions. Third, Thumb uses a two-operand format to save space. Fourth, the unique shifted immediates and shifted second operands have disappeared and are replaced by separate shift instructions. Fifth, the addressing modes are simplified. Finally, putting all instructions into 16 bits forces many more instruction formats.

In many ways, the simplified Thumb architecture is more conventional than ARM. Here are additional changes made from ARM in going to Thumb:

- *Drop of immediate logical instructions*—Logical immediates are gone.
- *Condition codes implicit*—Rather than have condition codes set optionally, they are defined by the opcode. All ALU instructions and none of the data transfers set the condition codes.
- *Hi/Lo register access*—The 16 ARM registers are halved into Lo registers and Hi registers, with the eight Hi registers including the stack pointer (SP), link register, and PC. The Lo registers are available in all ALU operations. Variations of ADD, BX, CMP, and MOV also work with all combinations of Lo and Hi registers. SP and PC registers are also available in variations of data transfers and add immediates. Any other operations on the Hi registers require one MOV to put the value into a Lo register, perform the operation there, and then transfer the data back to the Hi register.
- *Branch/call distance*—Since instructions are 16 bits wide, the 8-bit conditional branch address is shifted by one instead of by two. Branch with link is specified in two instructions, concatenating 11 bits from each instruction and shifting them left to form a 23-bit address to load into PC.
- *Distance for data transfer offsets*—The offset is now 5 bits for the general-purpose registers and 8 bits for SP and PC.

D.14 Instructions Unique to SuperH

Register 0 plays a special role in SuperH address modes. It can be added to another register to form an address in indirect indexed addressing and PC-relative addressing. R0 is used to load constants to give a larger addressing range than can easily be fit into the 16-bit instructions of the SuperH. R0 is also the only register that can be an operand for immediate versions of AND, CMP, OR, and XOR. Below is a list of the remaining unique details of the SuperH architecture:

- *Decrement and test*—`DT` decrements a register and sets the T bit to 1 if the result is 0.
- *Optional delayed branch*—Although the other embedded RISC machines generally do not use delayed branches (see Appendix A), SuperH offers optional delayed branch execution for `BT` and `BF`.
- *Many multiplies*—Depending on whether the operation is signed or unsigned, if the operands are 16 bits or 32 bits, or if the product is 32 bits or 64 bits, the proper multiply instruction is `MULS`, `MULU`, `DMULS`, `DMULU`, or `MUL`. The product is found in the MACL and MACH registers.
- *Zero and sign extension*—Byte or halfwords are either zero-extended (`EXTU`) or sign-extended (`EXTS`) within a 32-bit register.
- *One-bit shift amounts*—Perhaps in an attempt to make them fit within the 16-bit instructions, shift instructions only shift a single bit at a time.
- *Dynamic shift amount*—These variable shifts test the sign of the amount in a register to determine whether they shift left (positive) or shift right (negative). Both logical (`SHLD`) and arithmetic (`SHAD`) instructions are supported. These instructions help offset the 1-bit constant shift amounts of standard shifts.
- *Rotate*—SuperH offers rotations by 1 bit left (`ROTL`) and right (`ROTR`), which set the T bit with the value rotated, and also have variations that include the T bit in the rotations (`ROTCL` and `ROTCR`).
- *SWAP*—This instruction swaps either the high and low bytes of a 32-bit word or the two bytes of the rightmost 16 bits.
- *Extract word* (`XTRCT`)—The middle 32 bits from a pair of 32-bit registers are placed in another register.
- *Negate with carry*—Like `SUBC` (Figure D.6.6), except the first operand is 0.
- *Cache prefetch*—Like many of the desktop RISCs (Figures D.6.1 through D.6.4), SuperH has an instruction (`PREF`) to prefetch data into the cache.

- *Test-and-set*—SuperH uses the older test-and-set (`TAS`) instruction to perform atomic locks or semaphores (see Chapter 2). `TAS` first loads a byte from memory. It then sets the T bit to 1 if the byte is 0 or to 0 if the byte is not 0. Finally, it sets the most significant bit of the byte to 1 and writes the result back to memory.

D.15 Instructions Unique to M32R

The most unusual feature of the M32R is a slight VLIW approach to the pairs of 16-bit instructions. A bit is reserved in the first instruction of the pair to say whether this instruction can be executed in parallel with the next instruction— that is, the two instructions are independent—or if these two must be executed sequentially. (An earlier machine that offered a similar option was the Intel i860.) This feature is included for future implementations of the architecture.

One surprise is that all branch displacements are shifted left 2 bits before being added to the PC, and the lower 2 bits of the PC are set to 0. Since some instructions are only 16 bits long, this shift means that a branch cannot go to any instruction in the program: it can only branch to instructions on word boundaries. A similar restriction is placed on the return address for the branch-and-link and jump-and-link instructions: they can only return to a word boundary. Thus, for a slightly larger branch distance, software must ensure that all branch addresses and all return addresses are aligned to a word boundary. The M32R code space is probably slightly larger, and it probably executes more `nop` instructions than it would if the branch address was only shifted left 1 bit.

However, the VLIW feature above means that a `nop` can execute in parallel with another 16-bit instruction so that the padding doesn't take more clock cycles. The code size expansion depends on the ability of the compiler to schedule code and to pair successive 16-bit instructions; Mitsubishi claims that code size overall is only 7% larger than that for the Motorola 6800 architecture.

The last remaining novel feature is that the result of the divide operation is the remainder instead of the quotient.

D.16 Instructions Unique to MIPS-16

MIPS-16 is not really a separate instruction set but a 16-bit extension of the full 32-bit MIPS architecture. It is compatible with any of the 32-bit address MIPS architectures (MIPS I, MIPS II) or 64-bit architectures (MIPS III, IV, V). The ISA mode bit determines the width of instructions: 0 means 32-bit-wide instructions

and 1 means 16-bit-wide instructions. The new JALX instruction toggles the ISA mode bit to switch to the other ISA. JR and JALR have been redefined to set the ISA mode bit from the most significant bit of the register containing the branch address, and this bit is not considered part of the address. All jump-and-link instructions save the current mode bit as the most significant bit of the return address.

Hence, MIPS supports whole procedures containing either 16-bit or 32-bit instructions, but it does not support mixing the two lengths together in a single procedure. The one exception is the JAL and JALX: these two instructions need 32 bits even in the 16-bit mode, presumably to get a large enough address to branch to far procedures.

In picking this subset, MIPS decided to include opcodes for some three-operand instructions and to keep 16 opcodes for 64-bit operations. The combination of this many opcodes and operands in 16 bits led the architects to provide only eight easy-to-use registers—just like Thumb—whereas the other embedded RISCs offer about 16 registers. Since the hardware must include the full 32 registers of the 32-bit ISA mode, MIPS-16 includes move instructions to copy values between the eight MIPS-16 registers and the remaining 24 registers of the full MIPS architecture. To reduce pressure on the eight visible registers, the stack pointer is considered a separate register. MIPS-16 includes a variety of separate opcodes to do data transfers using SP as a base register and to increment SP: LWSP, LDSP, SWSP, SDSP, ADJSP, DADJSP, ADDIUSPD, and DADDIUSP.

To fit within the 16-bit limit, immediate fields have generally been shortened to 5 to 8 bits. MIPS-16 provides a way to extend its shorter immediates into the full width of immediates in the 32-bit mode. Borrowing a trick from the Intel 8086, the EXTEND instruction is really a 16-bit prefix that can be prepended to any MIPS-16 instruction with an address or immediate field. The prefix supplies enough bits to turn the 5-bit field of data transfers and 5- to 8-bit fields of arithmetic immediates into 16-bit constants. Alas, there are two exceptions. ADDIU and DADDIU start with 4-bit immediate fields, but since EXTEND can only supply 11 more bits, the wider immediate is limited to 15 bits. EXTEND also extends the 3-bit shift fields into 5-bit fields for shifts. (In case you were wondering, the EXTEND prefix does *not* need to start on a 32-bit boundary.)

To further address the supply of constants, MIPS-16 added a new addressing mode! PC-relative addressing for load word (LWPC) and load double (LDPC) shifts an 8-bit immediate field by 2 or 3 bits, respectively, adding it to the PC with the lower 2 or 3 bits cleared. The constant word or doubleword is then loaded into a register. Thus 32-bit or 64-bit constants can be included with MIPS-16 code, despite the loss of LIU to set the upper register bits. Given the new addressing mode, there is also an instruction (ADDIUPC) to calculate a PC-relative address and place it in a register.

MIPS-16 differs from the other embedded RISCs in that it can subset a 64-bit address architecture. As a result it has 16-bit instruction-length versions of 64-bit

data operations: data transfer (LD, SD, LWU), arithmetic operations (DADDU/IU, DSUBU, DMULT/U, DDIV/U), and shifts (DSLL/V, DSRA/V, DSRL/V).

Since MIPS plays such a prominent role in this book, we show all the additional changes made from the MIPS core instructions in going to MIPS-16:

- *Drop of signed arithmetic instructions*—Arithmetic instructions that can trap were dropped to save opcode space: ADD, ADDI, SUB, DADD, DADDI, DSUB.
- *Drop of immediate logical instructions*—Logical immediates are gone too: ANDI, ORI, XORI.
- *Branch instructions pared down*—Comparing two registers and then branching did not fit, nor did all the other comparisons of a register to zero. Hence these instructions didn't make it either: BEQ, BNE, BGEZ, BGTZ, BLEZ, and BLTZ. As mentioned in Section D.3, to help compensate MIPS-16 includes compare instructions to test if two registers are equal. Since compare and set on less than set the new T register, branches were added to test the T register.
- *Branch distance*—Since instructions are 16 bits wide, the branch address is shifted by one instead of by two.
- *Delayed branches disappear*—The branches take effect before the next instruction. Jumps still have a one-slot delay.
- *Extension and distance for data transfer offsets*—The 5-bit and 8-bit fields are zero-extended instead of sign-extended in 32-bit mode. To get greater range, the immediate fields are shifted left 1, 2, or 3 bits depending on whether the data are halfword, word, or doubleword. If the EXTEND prefix is prepended to these instructions, they use the conventional signed 16-bit immediate of the 32-bit mode.
- *Extension of arithmetic immediates*—The 5-bit and 8-bit fields are zero-extended for set on less than and compare instructions, for forming a PC-relative address, and for adding to SP and placing the result in a register (ADDIUSP, DADDIUSP). Once again, if the EXTEND prefix is prepended to these instructions, they use the conventional signed 16-bit immediate of the 32-bit mode. They are still sign-extended for general adds and for adding to SP and placing the result back in SP (ADJSP, DADJSP). Alas, code density and orthogonality are strange bedfellows in MIPS-16!
- *Redefining shift amount of 0*—MIPS-16 defines the value 0 in the 3-bit shift field to mean a shift of 8 bits.
- *New instructions added due to loss of register 0 as zero*—Load immediate, negate, and not were added, since these operations could no longer be synthesized from other instructions using r0 as a source.

D.17 Concluding Remarks

This appendix covers the addressing modes, instruction formats, and all instructions found in 10 RISC architectures. Although the later sections of the appendix concentrate on the differences, it would not be possible to cover 10 architectures in these few pages if there were not so many similarities. In fact, we would guess that more than 90% of the instructions executed for any of these architectures would be found in Figures D.3.5 through D.3.11. To contrast this homogeneity, Figure D.17.1 gives a summary for four architectures from the 1970s in a format similar to that shown in Figure D.1.1. (Imagine trying to write a single chapter in this style for those architectures!) In the history of computing, there has never been such widespread agreement on computer architecture.

	IBM 360/370	Intel 8086	Motorola 68000	DEC VAX
Date announced	1964/1970	1978	1980	1977
Instruction size(s) (bits)	16, 32, 48	8, 16, 24, 32, 40, 48	16, 32, 48, 64, 80	8, 16, 24, 32, . . . , 432
Addressing (size, model)	24 bits, flat/31 bits, flat	4 + 16 bits, segmented	24 bits, flat	32 bits, flat
Data aligned?	Yes 360/No 370	No	16-bit aligned	No
Data addressing modes	2/3	5	9	= 14
Protection	Page	None	Optional	Page
Page size	2 KB & 4 KB	—	0.25 to 32 KB	0.5 KB
I/O	Opcode	Opcode	Memory mapped	Memory mapped
Integer registers (size, model, number)	16 GPR × 32 bits	8 dedicated data × 16 bits	8 data and 8 address × 32 bits	15 GPR × 32 bits
Separate floating-point registers	4 × 64 bits	Optional: 8 × 80 bits	Optional: 8 × 80 bits	0
Floating-point format	IBM (floating hexadecimal)	IEEE 754 single, double, extended	IEEE 754 single, double, extended	DEC

FIGURE D.17.1 Summary of four 1970s architectures. Unlike the architectures in Figure D.1.1, there is little agreement between these architectures in any category.

This style of architecture cannot remain static, however. Like people, instruction sets tend to get bigger as they get older. Figure D.17.2 shows the genealogy of these instruction sets, and Figure D.17.3 shows which features were added to or deleted from generations of desktop RISCs over time.

As you can see, all the desktop RISC machines have evolved to 64-bit address architectures, and they have done so fairly painlessly.

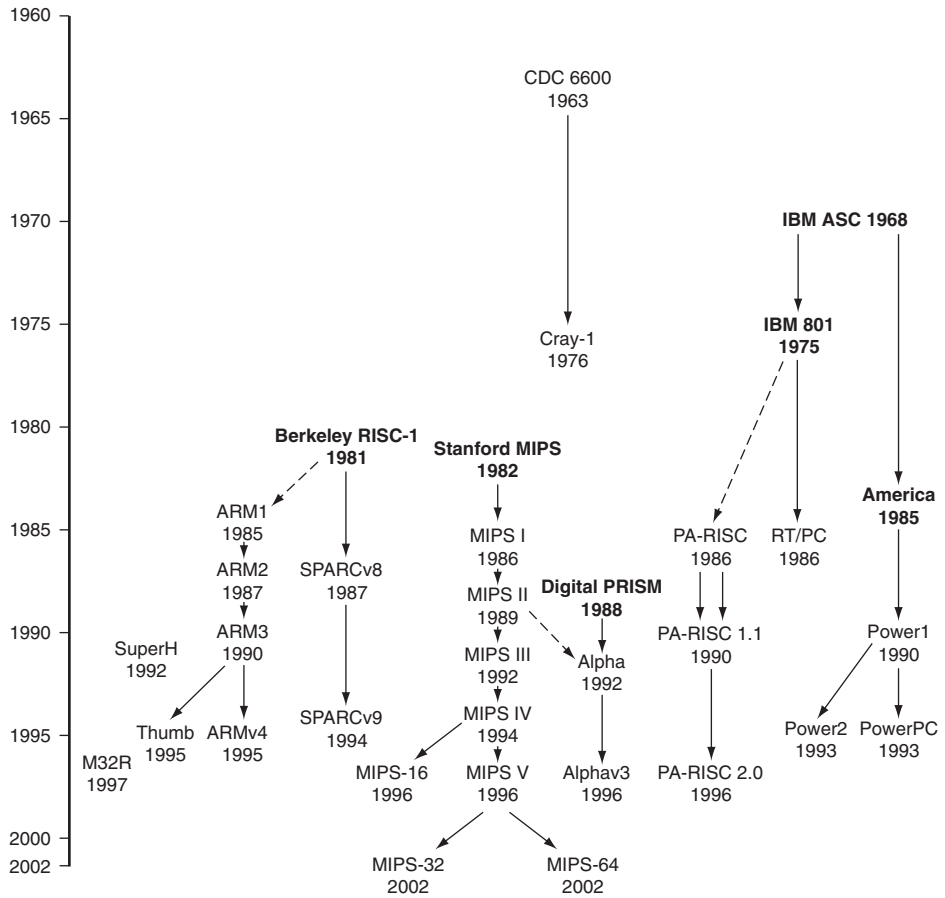


FIGURE D.17.2 The lineage of RISC instruction sets. Commercial machines are shown in plain text and research machines in bold. The CDC 6600 and Cray-1 were load-store machines with register 0 fixed at 0, and with separate integer and floating-point registers. Instructions could not cross word boundaries. An early IBM research machine led to the 801 and America research projects, with the 801 leading to the unsuccessful RT/PC and America leading to the successful Power architecture. Some people who worked on the 801 later joined Hewlett-Packard to work on the PA-RISC. The two university projects were the basis of MIPS and SPARC machines. According to Furber [1996], the Berkeley RISC project was the inspiration of the ARM architecture. While ARM1, ARM2, and ARM3 were names of both architectures and chips, ARM version 4 is the name of the architecture used in ARM7, ARM8, and StrongARM chips. (There are no ARMv4 and ARM5 chips, but ARM6 and early ARM7 chips use the ARM3 architecture.) DEC built a RISC microprocessor in 1988 but did not introduce it. Instead, DEC shipped workstations using MIPS microprocessors for 3 years before they brought out their own RISC instruction set, Alpha 21064, which is very similar to MIPS III and PRISM. The Alpha architecture has had small extensions, but they have not been formalized with version numbers; we used version 3 because that is the version of the reference manual. The Alpha 21164A chip added byte and halfword loads and stores, and the Alpha 21264 includes the MAX multimedia and bit count instructions. Internally, Digital names chips after the fabrication technology: EV4 (21064), EV45 (21064A), EV5 (21164), EV56 (21164A), and EV6 (21264). "EV" stands for "extended VAX."

	PA-RISC			SPARC		MIPS					Power		
Feature	1.0	1.1	2.0	v8	v9	I	II	III	IV	V	1	2	PC
Interlocked loads	X	”	”	X	”		+	”	”		X	”	”
Load-store FP double	X	”	”	X	”		+	”	”		X	”	”
Semaphore	X	”	”	X	”		+	”	”		X	”	”
Square root	X	”	”	X	”		+	”	”			+	”
Single-precision FP ops	X	”	”	X	”	X	”	”	”				+
Memory synchronize	X	”	”	X	”		+	”	”		X	”	”
Coprocessor	X	”	”	X	—	X	”	”	”				
Base + index addressing	X	”	”	X	”				+		X	”	”
Equiv. 32 64-bit FP registers		”	”		+			+	”		X	”	”
Annulling delayed branch	X	”	”	X	”		+	”	”				
Branch register contents	X	”	”		+	X	”	”	”				
Big/little endian		+	”		+	X	”	”	”				+
Branch prediction bit					+		+	”	”		X	”	”
Conditional move					+				+		X	”	—
Prefetch data into cache			+		+				+		X	”	”
64-bit addressing/int. ops			+		+			+	”				+
32-bit multiply, divide		+	”		+	X	”	”	”		X	”	”
Load-store FP quad					+							+	—
Fused FP mul/add			+						+		X	”	”
String instructions	X	”	”								X	”	—
Multimedia support		X	”	X						X			

FIGURE D.17.3 Features added to desktop RISC machines. X means in the original machine, + means added later, ” means continued from prior machine, and — means removed from architecture. Alpha is not included, but it added byte and word loads and stores, and bit count and multimedia extensions, in version 3. MIPS V added the MDMX instructions and paired single floating-point operations.

We would like to thank the following people for comments on drafts of this appendix: Professor Steven B. Furber, University of Manchester; Dr. Dileep Bhandarkar, Intel Corporation; Dr. Earl Killian, Silicon Graphics/MIPS; and Dr. Hiokazu Takata, Mitsubishi Electric Corporation.

Further Reading

Bhandarkar, D. P. [1995]. *Alpha Architecture and Implementations*, Newton, MA: Digital Press.

Darcy, J.D., and D. Gay [1996]. “FLECKmarks: Measuring floating point performance using a full IEEE compliant arithmetic benchmark,” CS 252 class project, U.C. Berkeley (see *www.sonic.net/~jddarcy/Research/fleckmrk.pdf*).

Digital Semiconductor [1996]. *Alpha Architecture Handbook*, Version 3, Maynard, MA: Digital Press, Order number EC-QD2KB-TE (October).

Furber, S. B. [1996]. *ARM System Architecture,* Harlow, England: Addison-Wesley. (See *http://www.pearsonhighered.com/pearsonhigheredus/educator/product/products_detail.page?isbn=9780201675191&forced_logout=forced_logged_out#sthash.QX4WfErc*).

Hewlett-Packard [1994]. *PA-RISC 2.0 Architecture Reference Manual*, 3rd ed.

Hitachi [1997]. *SuperH RISC Engine SH7700 Series Programming Manual*. (See *http://am.renesas.com/products/mpumcu/superh/sh7700/Documentation.jsp*).

IBM [1994]. *The PowerPC Architecture*, San Francisco: Morgan Kaufmann.

Kane, G. [1996]. *PA-RISC 2.0 Architecture*, Upper Saddle River, NJ: Prentice Hall PTR.

Kane, G. and J. Heinrich [1992]. *MIPS RISC Architecture,* Englewood Cliffs, NJ: Prentice Hall.

Kissell, K.D. [1997]. *MIPS16: High-Density for the Embedded Market*.

Magenheimer, D. J., L. Peters, K. W. Pettis, and D. Zuras [1988]. "Integer multiplication and division on the HP precision architecture", *IEEE Trans. on Computers* 37:8, 980–90.

MIPS [1997]. *MIPS16 Application Specific Extension Product Description*.

Mitsubishi [1996]. *Mitsubishi 32-Bit Single Chip Microcomputer M32R Family Soft ware Manual* (September).

Muchnick, S. S. [1988]. "Optimizing compilers for SPARC", *Sun Technology* 1:3(Summer) , 64–77.

Seal, D. *Arm Architecture Reference Manual*, 2nd ed, Morgan Kaufmann, 2000.

Silicon Graphics [1996]. *MIPS V Instruction Set*.

Sites, R. L., and R. Witek (eds.) [1995]. *Alpha Architecture Reference Manual*, 2nd ed. Newton, MA: Digital Press.

Sloss, A. N., D. Symes, and C. Wright, *ARM System Developer's Guide* San Francisco: Elsevier Morgan Kaufmann, 2004.

Sun Microsystems [1989]. *The SPARC Architectural Manual*, Version 8, Part No. 800-1399-09, August 25.

Sweetman, D. *See MIPS Run*, 2nd ed, Morgan Kaufmann, 2006.

Taylor, G., P. Hilfinger, J. Larus, D. Patterson, and B. Zorn [1986]. "Evaluation of the SPUR LISP architecture," *Proc. 13th Symposium on Computer Architecture* (June), Tokyo.

Ungar, D., R. Blau, P. Foley, D. Samples, and D. Patterson [1984]. "Architecture of SOAR: Smalltalk on a RISC," *Proc. 11th Symposium on Computer Architecture* (June), Ann Arbor, MI, 188–97.

Weaver, D. L. and T. Germond [1994]. *The SPARC Architectural Manual,* Version 9 Englewood Cliffs, NJ: Prentice Hall.

Weiss, S. and J. E. Smith [1994]. *Power and PowerPC*, San Francisco: Morgan Kaufmann.

This page intentionally left blank

Answers to Check Yourself

Chapter 1

§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.

Chapter 2

§2.2, page 66: RISC-V, C, Java.

§2.3, page 73: 2) Very slow.

§2.4, page 80: 2) -8_{ten}

§2.5, page 89: 3) `sub x11, x10, x9`

§2.6, page 92: Both. AND with a mask pattern of 1s will leaves 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 doubleword, with 0s in the rest of the doubleword. Note that AND leaves the field where it was originally, and the shift pair moves the field into the rightmost part of the doubleword.

§2.7, page 97: I. All are true. II. 1).

§2.8, page 108: Both are true.

§2.9, page 113: I. 1) and 2) II. 3).

§2.10, page 121: I. 4) ±4 K. II. 4) ± 1 M.

§2.11, page 124: Both are true.

§2.12, page 133: 4) Machine independence.

Chapter 3

§3.2, page 177: 2.

§3.5, page 215: 3.

Chapter 4

§4.1, page 240: 3 of 5: Control, Datapath, Memory. Input and Output are missing.

§4.2, page 243: false. Edge-triggered state elements make simultaneous reading and writing both possible and unambiguous.

§4.3, page 250: I. a. II. c.

§4.4, page 262: Yes, Branch and ALUOp0 are identical. In addition, you can use the flexibility of the don't care bits to combine other signals together. ALUSrc and MemtoReg can be made the same by setting the two don't care bits of MemtoReg to 1 and 0. ALUOp1 and MemtoReg can be made to be inverses of one another by setting the don't care bit of MemtoReg to 1. You don't need an inverter; simply use the other signal and flip the order of the inputs to the MemtoReg multiplexor!

§4.5, page 275: 1. Stall due to a load-use data hazard of the `ld` result. 2. Avoid stalling in the third instruction for the read-after-write data hazard on `x11` by forwarding the `add` result. 3. It need not stall, even without forwarding.

§4.6, page 288: Statements 2 and 4 are correct; the rest are incorrect.

§4.8, page 314: 1. Predict not taken. 2. Predict taken. 3. Dynamic prediction.

§4.9, page 321: The first instruction, since it is logically executed before the others.

§4.10, page 334: 1. Both. 2. Both. 3. Software. 4. Hardware. 5. Hardware. 6. Hardware. 7. Both. 8. Hardware. 9. Both.

§4.12, page 344: First two are false and the last two are true.

Chapter 5

§5.1, page 369: 1 and 4. (3 is false because the cost of the memory hierarchy varies per computer, but in 2016 the highest cost is usually the DRAM.)

§5.3, page 390: 1 and 4: A lower miss penalty can enable smaller blocks, since you don't have that much latency to amortize, yet higher memory bandwidth usually leads to larger blocks, since the miss penalty is only slightly larger.

§5.4, page 409: 1.

§5.8, page 449: 2. (Both large block sizes and prefetching may reduce compulsory misses, so 1 is false.)

Chapter 6

§6.1, page 494: False. Task-level parallelism can help sequential applications and sequential applications can be made to run on parallel hardware, although it is more challenging.

§6.2, page 499: False. *Weak* scaling can compensate for a serial portion of the program that would otherwise limit scalability, but not so for strong scaling.

§6.3, page 504: True, but they are missing useful vector features like gather-scatter and vector length registers that improve the efficiency of vector architectures.

(As an elaboration in this section mentions, the AVX2 SIMD extensions offers indexed loads via a gather operation but *not* scatter for indexed stores. The Haswell generation x86 microprocessor is the first to support AVX2.)

§6.4, page 509: 1. True. 2. True.

§6.5, page 513: False. Since the shared address is a *physical* address, multiple tasks each in their own *virtual* address spaces can run well on a shared memory multiprocessor.

§6.6, page 521: False. Graphics DRAM chips are prized for their higher bandwidth.

§6.7, page 526: 1. False. Sending and receiving a message is an implicit synchronization, as well as a way to share data. 2. True.

§6.8, page 528: True.

§6.10, page 540: True. We likely need innovation at all levels of the hardware and software stack for parallel computing to succeed.

This page intentionally left blank

Glossary

absolute address A variable's or routine's actual address in memory.

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

access bit Also called **use bit** or **reference bit**. A field that is set whenever a page is accessed and that is used to implement LRU or other replacement schemes.

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.

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

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

address translation Also called **address mapping**. The process by which a virtual address is mapped to an address used to access memory.

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

aliasing A situation in which two addresses access the same object; it can occur in virtual memory when there are two virtual addresses for the same physical page.

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

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.

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

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.

antifuse A structure in an integrated circuit that when programmed makes a permanent connection between two wires.

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.

architectural registers The instruction set of visible registers of a processor; for example, in RISC-V, these are the 32 integer and 32 floating-point registers.

arithmetic intensity The ratio of floating-point operations in a program to the number of data bytes accessed by a program from main memory.

arithmetic logic unit (ALU) Hardware that performs addition, subtraction, and usually logical operations such as AND and OR.

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

assembler directive An operation that tells the assembler how to translate a program but does not produce machine instructions; always begins with a period.

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

asserted The signal is logically high or true.

asserted signal A signal that is (logically) true, or 1.

backpatching A method for translating from assembly language to machine instructions in which the assembler builds a (possibly incomplete) binary representation of every instruction in one pass over a program and then returns to fill in previously undefined labels.

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).

behavioral specification Describes how a digital system operates functionally.

benchmark A program selected for use in comparing computer performance.

biased notation A notation that represents the most negative value by $00\ldots000_{two}$ and the most positive value by $11\ldots11_{two}$, with 0 typically having the value $10\ldots00_{two}$, thereby biasing the number such that the number plus the bias has a nonnegative representation.

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

bisection bandwidth The bandwidth between two equal parts of a multiprocessor. This measure is for a worst-case split of the multiprocessor.

block (or line) The minimum unit of information that can be either present or not present in a cache.

blocking assignment In Verilog, an assignment that completes before the execution of the next statement.

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

branch-and-link instruction An instruction that branches to an address and simultaneously saves the address of the following instruction in a register (usually x1 in RISC-V).

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.

branch prediction A method of resolving a branch hazard that assumes a given outcome for the conditional branch and proceeds from that assumption rather than waiting to ascertain the actual outcome.

branch prediction buffer 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.

branch taken A branch where the branch condition is satisfied and the program counter (PC) becomes the branch target. All unconditional branches are taken branches.

branch target address The address specified in a branch, which becomes the new program counter (PC) if the branch is taken. In the RISC-V architecture, the branch target is given by the sum of the immediate field of the instruction and the address of the branch.

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.

bus In logic design, a collection of data lines that are treated together as a single logical signal; also, a shared collection of lines with multiple sources and uses.

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

cache miss A request for data from the cache that cannot be filled because the data are not present in the cache.

callee A procedure that executes a series of stored instructions based on parameters provided by the caller and then returns control to the caller.

callee-saved register A register saved by the routine making a procedure call.

caller The program that instigates a procedure and provides the necessary parameter values.

caller-saved register A register saved by the routine being called.

capacity miss A cache miss that occurs because the cache, even with full associativity, cannot contain all the blocks needed to satisfy the request.

central processing unit (CPU) Also called central processor unit or 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.

clock cycle Also called **tick**, **clock tick**, **clock period**, **clock**, or **cycle**. The time for one clock period, usually of the processor clock.

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

clock period The length of each clock cycle.

clock skew The difference in absolute time between the times when two state elements see a clock edge.

clocking methodology The approach used to determine when data are valid and stable relative to the clock.

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.

cluster A set of computers connected over a local area network that function as a single large multiprocessor.

clusters Collections of computers connected via I/O over standard network switches to form a message-passing multiprocessor.

coarse-grained multithreading A version of hardware multithreading that implies switching between threads only after significant events, such as a last-level cache miss.

combinational element An operational element, such as an AND gate or an ALU.

combinational logic A logic system whose blocks do not contain memory and hence compute the same output given the same input.

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.

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

compulsory miss Also called **cold-start miss**. A cache miss caused by the first access to a block that has never been in the cache.

conditional branch An instruction that tests a value, and that allows for a subsequent transfer of control to a new address in the program based on the outcome of the test.

conflict miss Also called **collision miss**. A cache miss that occurs in a set-associative or direct-mapped cache when multiple blocks compete for the same set and that are eliminated in a fully associative cache of the same size.

context switch A changing of the internal state of the processor to allow a different process to use the processor that includes saving the state needed to return to the currently executing process.

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

control hazard Also called **branch hazard**. Arises 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.

control signal A signal used for multiplexor selection or for directing the operation of a functional unit; contrasts with a **data signal**, which contains information that is operated on by a functional unit.

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.

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

crossbar network A network that allows any node to communicate with any other node in one pass through the network.

D flip-flop A flip-flop with one data input that stores the value of that input signal in the internal memory when the clock edge occurs.

data hazard Also called a **pipeline data hazard**. When a planned instruction cannot execute in the proper clock cycle because data that are needed to execute the instruction are not yet available.

data race Two memory accesses forming a data race if they are from different threads to the same location, at least one is a write, and they occur one after another.

data segment The segment of a UNIX object or executable file that contains a binary representation of the initialized data used by the program.

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

data-level parallelism Parallelism achieved by performing the same operation on independent data.

datapath The component of the processor that performs arithmetic operations.

datapath element A unit used to operate on or hold data within a processor. In the RISC-V implementation, the datapath elements include the instruction and data memories, the register file, the ALU, and adders.

deasserted The signal being logically low or false.

deasserted signal A signal that is (logically) false, or 0.

decoder A logic block that has an n-bit input and $2n$ outputs, where only one output is asserted for each input combination.

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

delayed 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.

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

direct-mapped cache A cache structure in which each memory location is mapped to exactly one location in the cache.

dividend A number being divided.

divisor A number that the dividend is divided by.

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.

double precision A floating-point value represented in a 64-bit doubleword.

doubleword Another natural unit of access in a computer, usually a group of 64 bits; corresponds to the size of a register in the RISC-V architecture.

dynamic branch prediction Prediction of branches at runtime using runtime information.

dynamic multiple issue An approach to implementing a multiple-issue processor where many decisions are made during execution by the processor.

dynamic pipeline scheduling Hardware support for reordering the order of instruction execution so as to avoid stalls.

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.

dynamically linked libraries (DLLs) Library routines that are linked to a program during execution.

edge-triggered clocking A clocking scheme in which all state changes occur on a clock edge.

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

EOR A logical bit-by-bit operation with two operands that calculates the exclusive OR of the two operands. That is, it calculates a 1 only if the values are different in the two operands.

error detection code A code that enables the detection of an error in data, but not the precise location and, hence, correction of the error.

exception Also called an **interrupt**. An unscheduled event that disrupts program execution; used to detect overflow.

exception enable Also called interrupt enable. A signal or action that controls whether the process responds to an exception or not; necessary for preventing the occurrence of exceptions during intervals before the processor has safely saved the state needed to restart.

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.

exponent In the numerical representation system of floating-point arithmetic, the value that is placed in the exponent field.

external label Also called **global label**. A label referring to an object that can be referenced from files other than the one in which it is defined.

false sharing When two unrelated shared variables are located in the same cache block and the full block is exchanged between processors even though the processors are accessing different variables.

field programmable devices (FPD) An integrated circuit containing combinational logic, and possibly memory devices, that are configurable by the end user.

field programmable gate array (FPGA) A configurable integrated circuit containing both combinational logic blocks and flip-flops.

fine-grained multithreading A version of hardware multithreading that implies switching between threads after every instruction.

finite-state machine A sequential logic function consisting of a set of inputs and outputs, a next-state function that maps the current state and the inputs to a new state, and an output function that maps the current state and possibly the inputs to a set of asserted outputs.

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.

flip-flop A memory element for which the output is equal to the value of the stored state inside the element and for which the internal state is changed only on a clock edge.

floating point Computer arithmetic that represents numbers in which the binary point is not fixed.

flush To discard instructions in a pipeline, usually due to an unexpected event.

formal parameter A variable that is the argument to a procedure or macro; replaced by that argument once the macro is expanded.

forward reference A label that is used before it is defined.

forwarding Also called **bypassing**. A method of resolving a data hazard by retrieving the missing data element from internal buffers rather than waiting for it to arrive from programmer-visible registers or memory.

fraction The value, generally between 0 and 1, placed in the fraction field.

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

fully associative cache A cache structure in which a block can be placed in any location in the cache.

fully connected network A network that connects processor-memory nodes by supplying a dedicated communication link between every node.

fused multiply add A floating-point instruction that performs both a multiply and an add, but rounds only once after the add.

gate A device that implements basic logic functions, such as AND or OR.

global miss rate The fraction of references that miss in all levels of a multilevel cache.

global pointer The register that is reserved to point to the static area.

guard The first of two extra bits kept on the right during intermediate calculations of floating-point numbers; used to improve rounding accuracy.

handler Name of a software routine invoked to "handle" an exception or interrupt.

hardware description language A programming language for describing hardware, used for generating simulations of a hardware design and also as input to synthesis tools that can generate actual hardware.

hardware multithreading Increasing utilization of a processor by switching to another thread when one thread is stalled.

hardware synthesis tools Computer-aided design software that can generate a gate-level design based on behavioral descriptions of a digital system.

hexadecimal Numbers in base 16.

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.

hit rate The fraction of memory accesses found in a level of the memory hierarchy.

hit time The time required to access a level of the memory hierarchy, including the time needed to determine whether the access is a hit or a miss.

hold time The minimum time during which the input must be valid after the clock edge.

implementation Hardware that obeys the architecture abstraction.

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.

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.

input device A mechanism through which the computer is fed information, such as a microphone.

instruction A command that computer hardware understands and obeys.

instruction count The number of instructions executed by the program.

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

instruction latency The inherent execution time for an instruction.

instruction-level parallelism The parallelism among instructions.

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

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.

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

interrupt An exception that comes from outside of the processor. (Some architectures use the term *interrupt* for all exceptions.)

interrupt handler A piece of code that is run as a result of an exception or an interrupt.

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.

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.

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

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.

latch A memory element in which the output is equal to the value of the stored state inside the element and the state is changed whenever the appropriate inputs change and the clock is asserted.

latency (pipeline) The number of stages in a pipeline or the number of stages between two instructions during execution.

least recently used (LRU) A replacement scheme in which the block replaced is the one that has been unused for the longest time.

least significant bit The rightmost bit in a RISC-V doubleword.

level-sensitive clocking A timing methodology in which state changes occur at either high or low clock levels but are not instantaneous, as such changes are in edge-triggered designs.

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

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.

load-use data hazard A specific form of data hazard in which the data being loaded by a load instruction have not yet become available when they are needed by another instruction.

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

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

local label A label referring to an object that can be used only within the file in which it is defined.

local miss rate The fraction of references to one level of a cache that miss; used in multilevel hierarchies.

lock A synchronization device that allows access to data to only one processor at a time.

lookup tables (LUTs) In a field programmable device, the name given to the cells because they consist of a small amount of logic and RAM.

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

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

macro A pattern-matching and replacement facility that provides a simple mechanism to name a frequently used sequence of instructions.

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.

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

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

memory hierarchy A structure that uses multiple levels of memories; as the distance from the processor increases, the size of the memories and the access time both increase.

message passing Communicating between multiple processors by explicitly sending and receiving information.

metastability A situation that occurs if a signal is sampled when it is not stable for the required setup and hold times, possibly causing the sampled value to fall into the indeterminate region between a high and low value.

microarchitecture The organization of the processor, including the major functional units, their interconnection, and control.

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.

MIMD or Multiple Instruction streams, Multiple Data streams. A multiprocessor.

minterms Also called **product terms**. A set of logic inputs joined by conjunction (AND operations); the product terms form the first logic stage of the programmable logic array (PLA).

miss penalty The time required to fetch a block into a level of the memory hierarchy from the lower level, including the time to access the block, transmit it from one level to the other, insert it in the level that experienced the miss, and then pass the block to the requestor.

miss rate The fraction of memory accesses not found in a level of the memory hierarchy.

most significant bit The leftmost bit in a RISC-V doubleword.

multicore microprocessor A microprocessor containing multiple processors ("cores") in a single integrated circuit. Virtually all microprocessors today in desktops and servers are multicore.

multilevel cache A memory hierarchy with multiple levels of caches, rather than just a cache and main memory.

multiple issue A scheme whereby multiple instructions are launched in one clock cycle.

multiprocessor A computer system with at least two processors. This computer is in contrast to a uniprocessor, which has one, and is increasingly hard to find today.

multistage network A network that supplies a small switch at each node.

NAND gate An inverted AND gate.

network bandwidth Informally, the peak transfer rate of a network; can refer to the speed of a single link or the collective transfer rate of all links in the network.

next-state function A combinational function that, given the inputs and the current state, determines the next state of a finite-state machine.

nonblocking assignment An assignment that continues after evaluating the right-hand side, assigning the left-hand side the value only after all right-hand sides are evaluated.

nonblocking cache A cache that allows the processor to make references to the cache while the cache is handling an earlier miss.

nonuniform memory access (NUMA) A type of single address space multiprocessor in which some memory accesses are much faster than others depending on which processor asks for which word.

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.

nop An instruction that does no operation to change state.

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.

NOR gate An inverted OR gate.

normalized A number in floating-point notation that has no leading 0s.

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.

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

one's complement A notation that represents the most negative value by $10\ldots000_{two}$ and the most positive value by $01\ldots11_{two}$, leaving an equal number of negatives and positives but ending up with two zeros, one positive ($00\ldots00_{two}$) and one negative ($11\ldots11_{two}$). The term is also used to mean the inversion of every bit in a pattern: 0 to 1 and 1 to 0.

opcode The field that denotes the operation and format of an instruction.

OpenMP An API for shared memory multiprocessing in C, C++, or Fortran that runs on UNIX and Microsoft platforms. It includes compiler directives, a library, and runtime directives.

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

out-of-order execution A situation in pipelined execution when an instruction blocked from executing does not cause the following instructions to wait.

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

page fault An event that occurs when an accessed page is not present in main memory.

page table The table containing the virtual to physical address translations in a virtual memory system. The table, which is stored in memory, is typically indexed by the virtual page number; each entry in the table contains the physical page number for that virtual page if the page is currently in memory.

parallel processing program A single program that runs on multiple processors simultaneously.

PC-relative addressing An addressing regime in which the address is the sum of the program counter (PC) and a constant in the instruction.

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

personal mobile devices (PMDs) 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.

physical address An address in main memory.

physically addressed cache A cache that is addressed by a physical address.

pipeline stall Also called **bubble**. A stall initiated in order to resolve a hazard.

pipelining An implementation technique in which multiple instructions are overlapped in execution, much like an assembly line.

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

pop Remove element from stack.

precise interrupt Also called **precise exception**. An interrupt or exception that is always associated with the correct instruction in pipelined computers.

prefetching A technique in which data blocks needed in the future are brought into the cache early by the use of special instructions that specify the address of the block.

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

procedure call frame A block of memory that is used to hold values passed to a procedure as arguments, to save registers that a procedure may modify but that the procedure's caller does not want changed, and to provide space for variables local to a procedure.

procedure frame Also called **activation record**. The segment of the stack containing a procedure's saved registers and local variables.

process Includes one or more threads, the address space, and the operating system state. Hence, a process switch usually invokes the operating system, but not a thread switch.

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

programmable array logic (PAL) Contains a programmable and-plane followed by a fixed or-plane.

programmable logic array (PLA) A structured-logic element composed of a set of inputs and corresponding input complements and two stages of logic, the first generating product terms of the inputs and input complements, and the second generating sum terms of the product terms. Hence, PLAs implement logic functions as a sum of products.

programmable logic device (PLD) An integrated circuit containing combinational logic whose function is configured by the end user.

programmable ROM (PROM) A form of read-only memory that can be programmed when a designer knows its contents.

propagation time The time required for an input to a flip-flop to propagate to the outputs of the flip-flop.

protection A set of mechanisms for ensuring that multiple processes sharing the processor, memory, or I/O devices cannot interfere, intentionally or unintentionally, with one another by reading or writing each other's data. These mechanisms also isolate the operating system from a user process.

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

Pthreads A UNIX API for creating and manipulating threads. It is structured as a library.

push Add element to stack.

quotient The primary result of a division; a number that when multiplied by the divisor and added to the remainder produces the dividend.

read-only memory (ROM) A memory whose contents are designated at creation time, after which the contents can only be read. ROM is used as structured logic to implement a set of logic functions by using the terms in the logic functions as address inputs and the outputs as bits in each word of the memory.

receive message routine A routine used by a processor in machines with private memories to accept a message from another processor.

recursive procedures Procedures that call themselves either directly or indirectly through a chain of calls.

reduction A function that processes a data structure and returns a single value.

reference bit Also called **use bit** or **access bit**. A field that is set whenever a page is accessed and that is used to implement LRU or other replacement schemes.

reg In Verilog, a register.

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.

register renaming The renaming of registers by the compiler or hardware to remove antidependences.

register use convention Also called **procedure call convention**. A software protocol governing the use of registers by procedures.

relocation information The segment of a UNIX object file that identifies instructions and data words that depend on absolute addresses.

remainder The secondary result of a division; a number that when added to the product of the quotient and the divisor produces the dividend.

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.

reservation station A buffer within a functional unit that holds the operands and the operation.

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.

restartable instruction An instruction that can resume execution after an exception is resolved without the exceptions affecting the result of the instruction.

return address A link to the calling site that allows a procedure to return to the proper address; in RISC-V, it is usually stored in register `x1`.

rotational latency Also called **rotational delay**. The time required for the desired sector of a disk to rotate under the read/write head; usually assumed to be half the rotation time.

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. It is also the name of the second of two extra bits kept on the right during intermediate floating-point calculations, which improves rounding accuracy.

scientific notation A notation that renders numbers with a single digit to the left of the decimal point.

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

sector One of the segments that make up a track on a magnetic disk; a sector is the smallest amount of information that is read or written on a disk.

seek The process of positioning a read/write head over the proper track on a disk.

segmentation A variable-size address mapping scheme in which an address consists of two parts: a segment number, which is mapped to a physical address, and a segment offset.

selector value Also called **control value**. The control signal that is used to select one of the input values of a multiplexor as the output of the multiplexor.

semiconductor A substance that does not conduct electricity well.

send message routine A routine used by a processor in machines with private memories to pass a message to another processor.

sensitivity list The list of signals that specifies when an always block should be re-evaluated.

separate compilation Splitting a program across many files, each of which can be compiled without knowledge of what is in the other files.

sequential logic A group of logic elements that contain memory and hence whose value depends on the inputs as well as the current contents of the memory.

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

set-associative cache A cache that has a fixed number of locations (at least two) where each block can be placed.

setup time The minimum time that the input to a memory device must be valid before the clock edge.

shared memory multiprocessor (SMP) A parallel processor with a single physical address space.

sign-extend Increases 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.

silicon A natural element that is a semiconductor.

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.

SIMD or Single Instruction stream, Multiple Data streams. The same instruction is applied to many data streams, as in a vector processor.

simple programmable logic device (SPLD) Programmable logic device, usually containing either a single PAL or PLA.

simultaneous multithreading (SMT) A version of multithreading that lowers

the cost of multithreading by utilizing the resources needed for multiple issue, dynamically scheduled microarchitecture.

single precision A floating-point value represented in a 32-bit word.

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.

SISD or Single Instruction stream, Single Data stream. A uniprocessor.

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.

source language The high-level language in which a program is originally written.

spatial locality The locality principle stating that if a data location is referenced, data locations with nearby addresses will tend to be referenced soon.

speculation An approach whereby the compiler or processor guesses the outcome of an instruction to remove it as a dependence in executing other instructions.

split cache A scheme in which a level of the memory hierarchy is composed of two independent caches that operate in parallel with each other, with one handling instructions and one handling data.

SPMD Single Program, Multiple Data streams. The conventional MIMD programming model, where a single program runs across all processors.

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 RISC-V, it is register x2, also known as sp.

stack segment The portion of memory used by a program to hold procedure call frames.

state element A memory element, such as a register or a memory.

static data The portion of memory that contains data whose size is known to the compiler and whose lifetime is the program's entire execution.

static multiple issue An approach to implementing a multiple-issue processor where many decisions are made by the compiler before execution.

static random access memory (SRAM) A memory where data are stored statically (as in flip-flops) rather than dynamically (as in DRAM). SRAMs are faster than DRAMs, but less dense and more expensive per bit.

sticky bit A bit used in rounding in addition to guard and round that is set whenever there are nonzero bits to the right of the round bit.

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

strong scaling Speed-up achieved on a multiprocessor without increasing the size of the problem.

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.

structural specification Describes how a digital system is organized in terms of a hierarchical connection of elements.

sum of products A form of logical representation that employs a logical sum (OR) of products (terms joined using the AND operator).

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.

superscalar An advanced pipelining technique that enables the processor to execute more than one instruction per clock cycle by selecting them during execution.

supervisor mode Also called **kernel mode**. A mode indicating that a running process is an operating system process.

swap space The space on the disk reserved for the full virtual memory space of a process.

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

synchronization The process of coordinating the behavior of two or more processes, which may be running on different processors.

synchronizer failure A situation in which a flip-flop enters a metastable state and where some logic blocks reading the output of the flip-flop see a 0 while others see a 1.

synchronous system A memory system that employs clocks and where data signals are read only when the clock indicates that the signal values are stable.

system call A special instruction that transfers control from user mode to a dedicated location in supervisor code space, invoking the exception mechanism in the process.

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

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

tag A field in a table used for a memory hierarchy that contains the address information required to identify whether the associated block in the hierarchy corresponds to a requested word.

task-level parallelism or process-level parallelism Utilizing multiple processors by running independent programs simultaneously.

temporal locality The principle stating that if a data location is referenced, then it will tend to be referenced again soon.

terabyte (TB) Originally 1,099,511,627,776 (240) 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 240 bytes, defining terabyte (TB) to mean 10^{12} bytes. (Figure 1.1 shows the full range of decimal and binary values and names.)

text segment The segment of a UNIX object file that contains the machine language code for routines in the source file.

thread A thread includes the program counter, the register state, and the stack. It is a lightweight process; whereas threads commonly share a single address space, processes don't.

three Cs model A cache model in which all cache misses are classified into one of three categories: compulsory misses, capacity misses, and conflict misses.

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

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.

track One of thousands of concentric circles that makes up the surface of a magnetic disk.

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

translation-lookaside buffer (TLB) A cache that keeps track of recently used address mappings to try to avoid an access to the page 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.

underflow (floating-point) A situation in which a negative exponent becomes too large to fit in the exponent field.

uniform memory access (UMA) A multiprocessor in which latency to any word in main memory is about the same no matter which processor requests the access.

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.

unmapped A portion of the address space that cannot have page faults.

unresolved reference A reference that requires more information from an outside source to be complete.

use bit Also called **reference bit** or **access bit**. A field that is set whenever a page is accessed and that is used to implement LRU or other replacement schemes.

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.

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

valid bit A field in the tables of a memory hierarchy that indicates that the associated block in the hierarchy contains valid data.

vector lane One or more vector functional units and a portion of the vector register file. Inspired by lanes on highways that increase traffic speed, multiple lanes execute vector operations simultaneously.

vectored interrupt An interrupt for which the address to which control is transferred is determined by the cause of the exception.

Verilog One of the two most common hardware description languages.

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

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.

VHDL One of the two most common hardware description languages.

virtual address An address that corresponds to a location in virtual space and is translated by address mapping to a physical address when memory is accessed.

virtual machine A virtual computer that appears to have nondelayed branches and loads and a richer instruction set than the actual hardware.

virtual memory A technique that uses main memory as a "cache" for secondary storage.

virtually addressed cache A cache that is accessed with a virtual address rather than a physical address.

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

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

weak scaling Speed-up achieved on a multiprocessor while expanding the size of the problem proportionally to the increase in the number of processors.

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

wire In Verilog, specifies a combinational signal.

word The natural unit of access in a computer, usually a group of 32 bits.

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.

write buffer A queue that holds data while the data are waiting to be written to memory.

write-back A scheme that handles writes by updating values only to the block in the cache, then writing the modified block to the lower level of the hierarchy when the block is replaced.

write-through A scheme in which writes always update both the cache and the next lower level of the memory hierarchy, ensuring that data are always consistent between the two.

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

Further Reading

Chapter 1

Barroso, L. and U. Hölzle [2007]. "The case for energy-proportional computing", *IEEE Computer*, December.

A plea to change the nature of computer components so that they use much less power when lightly utilized.

Bell, C. G. [1996]. *Computer Pioneers and Pioneer Computers*, ACM and the Computer Museum, videotapes.

Two videotapes on the history of computing, produced by Gordon and Gwen Bell, including the following machines and their inventors: Harvard Mark-I, ENIAC, EDSAC, IAS machine, and many others.

Burks, A.W., H.H. Goldstine, and J. von Neumann [1946]. "Preliminary discussion of the logical design of an electronic computing instrument," *Report to the U.S. Army Ordnance Department*, p. 1; also appears in *Papers of John von Neumann*, W. Aspray and A. Burks (Eds.), MIT Press, Cambridge, MA, and Tomash Publishers, Los Angeles, 1987, 97–146.

A classic paper explaining computer hardware and software before the first stored-program computer was built. We quote extensively from it in Chapter 3. It simultaneously explained computers to the world and was a source of controversy because the first draft did not give credit to Eckert and Mauchly.

Campbell-Kelly, M. and W. Aspray [1996]. *Computer: A History of the Information Machine*, Basic Books, New York.

Two historians chronicle the dramatic story. The New York Times *calls it well written and authoritative.*

Ceruzzi, P. F. [1998]. *A History of Modern Computing*, MIT Press, Cambridge, MA.

Contains a good description of the later history of computing: the integrated circuit and its impact, personal computers, UNIX, and the Internet.

Curnow, H. J. and B. A. Wichmann [1976]. "A synthetic benchmark", *The Computer J.* 19(1):80.

Describes the first major synthetic benchmark, Whetstone, and how it was created.

Flemming, P. J. and J. J. Wallace [1986]. "How not to lie with statistics: The correct way to summarize benchmark results", *Commun. ACM* 29(3 (March)), 218–221.

Describes some of the underlying principles in using different means to summarize performance results.

Goldstine, H. H. [1972]. *The Computer: From Pascal to von Neumann*, Princeton University Press, Princeton, NJ.

A personal view of computing by one of the pioneers who worked with von Neumann.

Hayes, B. [2007]. "Computing in a parallel universe", *American Scientist*, Vol. 95(November–December), 476–480.

An overview of the parallel computing challenge written for the layman.

Hennessy, J. L. and D. A. Patterson [2012]. *Chapter 1 of Computer Architecture: A Quantitative Approach*, fifth edition, Morgan Kaufmann Publishers, Waltham, MA.

Section 1.5 goes into more detail on power, Section 1.6 contains much more detail on the cost of integrated circuits and explains the reasons for the difference between price and cost, and Section 1.8 gives more details on evaluating performance.

Lampson, B.W. [1986]. "Personal distributed computing; The Alto and Ethernet software." In ACM Conference on the History of Personal Workstations (January).

Thacker, C.R. [1986]. "Personal distributed computing: The Alto and Ethernet hardware," In ACM Conference on the History of Personal Workstations (January).

These two papers describe the software and hardware of the landmark Alto.

Metropolis, N., J. Howlett, and G. -C. Rota (Eds.) [1980]. *A History of Computing in the Twentieth Century*, Academic Press, New York.

A collection of essays that describe the people, software, computers, and laboratories involved in the first experimental and commercial computers. Most of the authors were personally involved in the projects. An excellent bibliography of early reports concludes this interesting book.

Public Broadcasting System [1992]. *The Machine That Changed the World*, videotapes.

These five 1-hour programs include rare footage and interviews with pioneers of the computer industry.

Slater, R. [1987]. *Portraits in Silicon*, MIT Press, Cambridge, MA.

Short biographies of 31 computer pioneers.

Stern, N. [1980]. "Who invented the first electronic digital computer?" *Annals of the History of Computing* 2:4 (October), 375–376.

A historian s perspective on Atanasoff versus Eckert and Mauchly.

Wilkes, M. V. [1985]. *Memoirs of a Computer Pioneer*, MIT Press, Cambridge, MA.

A personal view of computing by one of the pioneers.

Chapter 2

Bayko, J. [1996]. "Great microprocessors of the past and present," search for it on the *http://www.cpushack.com/CPU/cpu.html*.

A personal view of the history of both representative and unusual microprocessors, from the Intel 4004 to the Patriot Scientific ShBoom!

Kane, G. and J. Heinrich [1992]. *MIPS RISC Architecture*, Prentice Hall, Englewood Cliffs, NJ.

This book describes the MIPS architecture in greater detail than Appendix A.

Levy, H. and R. Eckhouse [1989]. *Computer Programming and Architecture: The VAX*, Digital Press, Boston.

This book concentrates on the VAX, but also includes descriptions of the Intel 8086, IBM 360, and CDC 6600.

Morse, S., B. Ravenal, S. Mazor, and W. Pohlman [1980]. "Intel microprocessors—8080 to 8086", *Computer* 13:10 (October).

The architecture history of the Intel from the 4004 to the 8086, according to the people who participated in the designs.

Wakerly, J. [1989]. *Microcomputer Architecture and Programming*, Wiley, New York.

The Motorola 6800 is the main focus of the book, but it covers the Intel 8086, Motorola 6809, TI 9900, and Zilog Z8000.

Waterman, A. Y. Lee, D. Patterson, and K. Asanović [2016]. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1.

The canonical reference manual for the RISC-V instruction set architecture, this technical report discusses the rationale behind the myriad tradeoffs in the ISA's design. Download from http://riscv.org/specifications/.

Chapter 3

If you are interested in learning more about floating point, two publications by David Goldberg [1991, 2002] are good starting points; they abound with pointers to further reading. Several of the stories told in this section come from Kahan [1972, 1983]. The latest word on the state of the art in computer arithmetic is often found in the *Proceedings* of the latest IEEE-sponsored Symposium on Computer Arithmetic, held every 2 years; the 16th was held in 2003.

Burks, A.W., H.H. Goldstine, and J. von Neumann [1946]. "Preliminary discussion of the logical design of an electronic computing instrument," *Report to the U.S. Army Ordnance Dept*., p. 1; also in *Papers of John von Neumann*, W. Aspray and A. Burks (Eds.), MIT Press, Cambridge, MA, and Tomash Publishers, Los Angeles, 1987, 97–146.

This classic paper includes arguments against floating-point hardware.

Goldberg, D. [2002]. "Computer arithmetic". Appendix J of *Computer Architecture: A Quantitative Approach*, fifth edition, J. L. Hennessy and D. A. Patterson, Morgan Kaufmann Publishers, Waltham, MA.

A more advanced introduction to integer and floating-point arithmetic, with emphasis on hardware. It covers Sections 3.4–3.6 of this book in just 10 pages, leaving another 45 pages for advanced topics.

Goldberg, D. [1991]. "What every computer scientist should know about floating-point arithmetic", *ACM Computing Surveys* 23(1) 5–48.

Another good introduction to floating-point arithmetic by the same author, this time with emphasis on software.

Kahan, W. [1972]. "A survey of error-analysis". *Info. Processing 71 (Proc. IFIP Congress 71 in Ljubljana)*, Vol. 2, North-Holland Publishing, Amsterdam, 1214–1239

This survey is a source of stories on the importance of accurate arithmetic.

Kahan, W. [1983]. "Mathematics written in sand", *Proc. Amer. Stat. Assoc. Joint Summer Meetings of 1983, Statistical Computing Section*, 12–26.

The title refers to silicon and is another source of stories illustrating the importance of accurate arithmetic.

Kahan, W. [1990]. *"On the advantage of the 8087's stack," unpublished course notes, Computer Science Division*, University of California, Berkeley.

What the 8087 floating-point architecture could have been.

Kahan, W. [1997]. Available at http://www.cims.nyu.edu/~dbindel/class/cs279/87stack.pdf.

A collection of memos related to floating point, including "Beastly numbers" (another less famous Pentium bug), "Notes on the IEEE floating point arithmetic" (including comments on how some features are atrophying), and "The baleful effects of computing benchmarks" (on the unhealthy preoccupation on speed versus correctness, accuracy, ease of use, flexibility, …).

Koren, I. [2002]. *Computer Arithmetic Algorithms*, second edition, A. K. Peters, Natick, MA.

A textbook aimed at seniors and first-year graduate students that explains fundamental principles of basic arithmetic, as well as complex operations such as logarithmic and trigonometric functions.

Wilkes, M. V. [1985]. *Memoirs of a Computer Pioneer*, MIT Press, Cambridge, MA.

This computer pioneer's recollections include the derivation of the standard hardware for multiply and divide developed by von Neumann.

Chapter 4

Bhandarkar, D. and D. W. Clark [1991]. "Performance from architecture: Comparing a RISC and a CISC with similar hardware organizations," *Proc. Fourth Conf. on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (April), Palo Alto, CA, 310–319.

A quantitative comparison of RISC and CISC written by scholars who argued for CISCs as well as built them; they conclude that MIPS is between 2 and 4 times faster than a VAX built with similar technology, with a mean of 2.7.

Fisher, J. A. and B. R. Rau [1993]. *Journal of Supercomputing (January)*, Kluwer.

This entire issue is devoted to the topic of exploiting ILP. It contains papers on both the architecture and software and is a wonderful source for further references.

Hennessy, J. L. and D. A. Patterson [2012]. *Computer Architecture: A Quantitative Approach*, fifth edition, Morgan Kaufmann, Waltham, MA.

Chapter 3 and Appendix C go into considerably more detail about pipelined processors (almost 200 pages), including superscalar processors and VLIW processors. Appendix G describes Itanium.

Jouppi, N. P. and D. W. Wall [1989]. "Available instruction-level parallelism for superscalar and superpipelined processors," *Proc. Third Conf. on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (April), Boston, 272–82.

A comparison of deeply pipelined (also called superpipelined) and superscalar systems.

Kogge, P. M. [1981]. *The Architecture of Pipelined Computers*, McGraw-Hill, New York.

A formal text on pipelined control, with emphasis on underlying principles.

Russell, R. M. [1978]. "The CRAY-1 computer system", *Commun. ACM* 21:1 (January), 63–72.

A short summary of a classic computer that uses vectors of operations to remove pipeline stalls.

Smith, A. and J. Lee [1984]. "Branch prediction strategies and branch target buffer design", *Computer* 17:1 (January), 6–22.

An early survey on branch prediction.

Smith, J. E. and A. R. Plezkun [1988]. "Implementing precise interrupts in pipelined processors", *IEEE Trans. on Computers* 37:5 (May), 562–573.

Covers the difficulties in interrupting pipelined computers.

Thornton, J. E. [1970]. *Design of a Computer: The Control Data 6600*, Scott, Foresman, Glenview, IL.

A classic book describing a classic computer, considered the first supercomputer.

Chapter 5

Cantin, J. F. and M. D. Hill [2001]. "Cache performance for selected SPEC CPU2000 benchmarks", *SIGARCH Computer Architecture News* 29:4 (September), 13–18.

A reference paper of cache miss rates for many cache sizes for the SPEC2000 benchmarks.

Conti, C, D. H. Gibson, and S. H. Pitowsky [1968]. "Structural aspects of the System/360 Model 85, part I: General organization", *IBM Systems J.* 7:1, 2–14.

A classic paper that describes the first commercial computer to use a cache and its resulting performance.

Hennessy, J. and D. Patterson [2012]. *Chapter 2 and Appendix B in Computer Architecture: A Quantitative Approach*, fifth edition, Morgan Kaufmann Publishers, Waltham, MA.

For more in-depth coverage of a variety of topics including protection, cache performance of out-of-order processors, virtually addressed caches, multilevel caches, compiler optimizations, additional latency tolerance mechanisms, and cache coherency.

Kilburn, T., D. B. G Edwards, M. J. Lanigan, and F. H. Sumner [1962]. "One-level storage system," *IRE Transactions on Electronic Computers* EC-11 (April), 223–35. Also appears in D. P. Siewiorek, C G Bell, and A. Newell [1982], *Computer Structures: Principles and Examples*, McGraw-Hill, New York, 135–48.

This classic paper is the first proposal for virtual memory.

LaMarca, A. and R. E. Ladner [1996]. "The influence of caches on the performance of heaps," *ACM J. of Experimental Algorithmics* Vol. 1.

This paper shows the difference between complexity analysis of an algorithm, instruction count performance, and memory hierarchy for four sorting algorithms.

McCalpin, J.D. [1995]. "STREAM: Sustainable Memory Bandwidth in High Performance Computers," https://www.cs.virginia.edu/stream/.

A widely used microbenchmark that measures the performance of the memory system behind the caches.

Przybylski, S. A. [1990]. *Cache and Memory Hierarchy Design: A Performance-Directed Approach*, Morgan Kaufmann Publishers, San Francisco.

A thorough exploration of multilevel memory hierarchies and their performance.

Ritchie, D. [1984]. "The evolution of the UNIX time-sharing system", *AT& T Bell Laboratories Technical Journal* 1984:1577–1593.

The history of UNIX from one of its inventors.

Ritchie, D. M. and K. Thompson [1978]. "The UNIX time-sharing system", *Bell System Technical Journal* (August), 1991–2019.

A paper describing the most elegant operating system ever invented.

Silberschatz, A., P. Galvin, and G. Grange [2003]. *Operating System Concepts*, sixth edition, Addison-Wesley, Reading, MA.

An operating systems textbook with a thorough discussion of virtual memory processes and process management, and protection issues.

Smith, A. J. [1982]. "Cache memories", *Computing Surveys* 14:3 (September), 473–530.

The classic survey paper on caches. This paper defined the terminology for the field and has served as a reference for many computer designers.

Smith, D. K. and R. C. Alexander [1988]. *Fumbling the Future: How Xerox Invented, Then Ignored, the First Personal Computer*, Morrow, New York.

A popular book that explains the role of Xerox PARC in laying the foundation for today's computing, but which Xerox did not substantially benefit from.

Tanenbaum, A. [2001]. *Modern Operating Systems*, second edition, Upper Saddle River: Prentice Hall, NJ.

An operating system textbook with a good discussion of virtual memory.

Waterman, A. Y. Lee, D. Patterson, and K. Asanović [2016]. The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.9.1.

The RISC-V Privileged Architecture manual discusses in more detail the layered privilege mode design and the memory address-translation and protection schemes described in Chapter 5.

Wilkes, M. [1965]. "Slave memories and dynamic storage allocation", *IEEE Trans. Electronic Computers EC* 14(2 (April)), 270–271.

The first classic paper on caches.

Chapter 6

Almasi, G. S. and A. Gottlieb [1989]. *Highly Parallel Computing*, Benjamin/Cummings, Redwood City, CA.

A textbook covering parallel computers.

Amdahl, G.M. [1967]. "Validity of the single processor approach to achieving large scale computing capabilities," *Proc. AFIPS Spring Joint Computer Conf.*, Atlantic City, NJ (April), 483–85.

Written in response to the claims of the Illiac IV, this three-page article describes Amdahl's law and gives the classic reply to arguments for abandoning the current form of computing.

Andrews, G. R. [1991]. Concurrent Programming: Principles and Practice, Benjamin/Cummings, Redwood City, CA.

A text that gives the principles of parallel programming.

Archibald, J. and J.-L. Baer [1986]. "Cache coherence protocols: Evaluation using a multiprocessor simulation model", *ACM Trans. on Computer Systems* 4:4 (November), 273–98.

Classic survey paper of shared-bus cache coherence protocols.

Arpaci-Dusseau, A., R. Arpaci-Dusseau, D. Culler, J. Hellerstein, and D. Patterson [1997]. "High-performance sorting on networks of workstations," *Proc. ACM SIG MOD/PODS Conference on Management of Data*, Tucson, AZ (May), 12–15.

How a world record sort was performed on a cluster, including architecture critique of the workstation and network interface. By April 1, 1997, they pushed the record to 8.6 GB in 1 minute and 2.2 seconds to sort 100 MB.

Asanovic, K., R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick [2006]. "The landscape of parallel computing research: A view from Berkeley". *Tech. Rep. UCB/EECS-2006-183*, EECS Department, University of California, Berkeley. (December 18).

Nicknamed the "Berkeley View," this report lays out the landscape of the multicore challenge.

Bailey, D.H., E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrishnan, and S.K. Weeratunga. [1991]. "The NAS parallel benchmarks—summary and preliminary results," *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing* (August).

Describes the NAS parallel benchmarks.

Bell, C. G. [1985]. "Multis: A new class of multiprocessor computers", *Science* 228(April 26), 462–467.

Distinguishes shared address and nonshared address multiprocessors based on microprocessors.

Bienia, C., S. Kumar, J.P. Singh, and K. Li [2008]. "The PARSEC benchmark suite: characterization and architectural implications," Princeton University Technical Report TR-81 1-008 (January).

Describes the PARSEC parallel benchmarks. Also see http://parsec.cs.princeton.edu/.

Cooper, B.F., A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears [2010]. Benchmarking cloud serving systems with YCSB, In: *Proceedings of the 1st ACM Symposium on Cloud Computing*, Indianapolis, Indiana, USA. http://dx.doi.org/10.1145/1807128.1807152 (June).

Presents the "Yahoo! Cloud Serving Benchmark" (YCSB) framework, with the goal of facilitating performance comparisons of the new generation of cloud data serving systems.

Culler, D. E., J. P. Singh, and A. Gupta [1998]. *Parallel Computer Architecture*, Morgan Kaufmann, San Francisco.

A textbook on parallel computers.

Dongarra, J. J., J. R. Bunch, G. B. Moler, and G. W. Stewart [1979]. *LINPACK Users' Guide*, Society for Industrial Mathematics.

The original document describing Linpack, which became a widely used parallel benchmark.

Falk, H. [1976]. "Reaching for the gigaflop", *IEEE Spectrum* 13:10 (October), 65–70.

Chronicles the sad story of the Illiac IV: four times the cost and less than one-tenth the performance of original goals.

Flynn, M. J. [1966]. "Very high-speed computing systems", *Proc. IEEE* 54:12 (December), 1901–09.

Classic article showing SISD/SIMD/MISD/MIMD classifications.

Hennessy, J. and D. Patterson [2012]. "Chapters 5 and Appendices F and I". In *Computer Architecture: A Quantitative Approach*, fifth edition, Morgan Kaufmann Publishers, Waltham, MA.

A more in-depth coverage of a variety of multiprocessor and cluster topics, including programs and measurements.

Henning, J. L. [2007]. "SPEC CPU suite growth: an historical perspective", *Computer Architecture News* Vol. 35, no. 1 (March).

Gives the history of SPEC, including the use of SPECrate to measure performance on independent jobs, which is being used as a parallel benchmark.

Hord, R. M. [1982]. *The Illiac-IV, the First Supercomputer*, Computer Science Press, Rockville, MD.

A historical accounting of the Illiac IV project.

Hwang, K. [1993]. *Advanced Computer Architecture with Parallel Programming*, McGraw-Hill, New York.

Another textbook covering parallel computers.

Kozyrakis, C. and D. Patterson [2003]. "Scalable vector processors for embedded systems", *IEEE Micro* 23:6 (November–December), 36–45.

Examination of a vector architecture for the MIPS instruction set in media and signal processing.

Laprie, J.-C. [1985]. "Dependable computing and fault tolerance: Concepts and terminology," *15th Annual Int'l Symposium on Fault-Tolerant Computing FTCS 15*, Digest of Papers, Ann Arbor, MI (June 19–21) 2–11.

The paper that introduced standard definitions of dependability, reliability, and availability.

Menabrea, L. F. [1842]. "Sketch of the analytical engine invented by Charles Babbage", *Bibliothèque Universelle de Genève* (October).

Certainly the earliest reference on multiprocessors, this mathematician made this comment while translating papers on Babbage's mechanical computer.

Patterson, D., G. Gibson, and R. Katz [1988]. "A case for redundant arrays of inexpensive disks (RAID)," *SIGMOD Conference*, 109–16.

Pfister, G. F. [1998]. *In Search of Clusters: The Coming Battle in Lowly Parallel Computing*, second edition, Prentice Hall, Upper Saddle River, NJ.

An entertaining book that advocates clusters and is critical of NUMA multiprocessors.

Regnier, G., S. Makineni, R. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong [2004]. TCP onloading for data center servers. IEEE Computer, 37(11):48–58.

Describes the work of researchers at Intel Labs, who have experimented with alternative solutions that improve the server's ability to process TCP/IP packets efficiently and at very high rates.

Seitz, C. [1985]. "The Cosmic Cube", *Comm. ACM* 28:1 (January), 22–31.

A tutorial article on a parallel processor connected via a hypertree. The Cosmic Cube is the ancestor of the Intel supercomputers.

Slotnick, D. L. [1982]. "The conception and development of parallel processors—a personal memoir", *Annals of the History of Computing* 4:1 (January), 20–30.

Recollections of the beginnings of parallel processing by the architect of the Illiac IV.

Williams, S., A. Waterman, and D. Patterson [2009]. "Roofline: An insightful visual performance model for multicore architectures", *Communications of the ACM*, 52:4 (April), 65–76.

Williams, S., J. Carter, L. Oliker, J. Shalf, and K. Yelick [2008]. "Lattice Boltzmann simulation optimization on leading multicore platforms," *International Parallel & Distributed Processing Symposium (IPDPS)*.

Paper containing the results of the four multicores for LBMHD.

Williams, S., L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel [2007]. "Optimization of sparse matrix-vector multiplication on emerging multicore platforms", *Supercomputing (SC)*

Paper containing the results of the four multicores for SPmV.

Williams, S. [2008]. *Autotuning Performance of Multicore Computers*, Ph.D. Dissertation, U.C. Berkeley.

Dissertation containing the roofline model.

Woo, S.C., M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. "The SPLASH-2 programs: characterization and methodological considerations," *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA '95)*, May, 24–36.

Paper describing the second version of the Stanford parallel benchmarks.

Appendix A

There are a number of good texts on logic design. Here are some you might like to look into.

Ashenden, P. [2007]. *Digital Design: An Embedded Systems Approach Using VHDL/Verilog*, Waltham, MA: Morgan Kaufmann.

Ciletti, M. D. [2002]. *Advanced Digital Design with the Verilog HDL*, Englewood Cliff s, NJ: Prentice Hall.

A thorough book on logic design using Verilog.

Harris, D. and S. Harris [2012]. *Digital Design and Computer Architecture*, Waltham, MA: Morgan Kaufmann.

A unique and modern approach to digital design using VHDL and SystemVerilog.

Katz, R. H. [2004]. *Modern Logic Design*, 2nd ed., Reading, MA: Addison-Wesley.

A general text on logic design.

Wakerly, J. F. [2000]. *Digital Design: Principles and Practices*, 3rd ed., Englewood Cliff s, NJ: Prentice Hall.

A general text on logic design.

Appendix B

Akeley, K. and T. Jermoluk [1988]. "High-Performance Polygon Rendering," *Proc. SIGGRAPH 1988* (August), 239–46.

Akeley, K. [1993]. "RealityEngine Graphics." *Proc. SIGGRAPH 1993* (August), 109–16.

Blelloch, G. B. [1990]. "Prefix Sums and Their Applications". In John H. Reif (Ed.), *Synthesis of Parallel Algorithms*, Morgan Kaufmann Publishers, San Francisco.

Blythe, D. [2006]. "The Direct3D 10 System", *ACM Trans. Graphics* Vol. 25 no. 3, (July), 724–734.

Buck, I., T. Foley, D. Horn, J. Sugerman, K. Fatahlian, M. Houston, and P. Hanrahan [2004]. "Brook for GPUs: Stream Computing on Graphics Hardware." *Proc. SIGGRAPH 2004*, 777–86, August. http://doi.acm.org/10.1145/1186562.1015800.

Elder, G. [2002] "Radeon 9700." Eurographics/SIGGRAPH Workshop on Graphics Hardware, Hot3D Session. www.graphicshardware.org/previous/www_2002/presentations/Hot3D-RADEON9700.ppt.

Fernando, R. and M. J. Kilgard [2003]. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*, Addison-Wesley, Reading, MA.

Fernando, R. (Ed.), [2004]. *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*, Addison-Wesley, Reading, MA. https://developer.nvidia.com/gpugems/GPUGems/gpugems_pref01.html.

Foley, J., A. van Dam, S. Feiner, and J. Hughes [1995]. *Computer Graphics: Principles and Practice, second edition in C*, Addison- Wesley, Reading, MA.

Hillis, W. D. and G. L. Steele [1986]. "Data parallel algorithms", *Commun. ACM* 29:12 (Dec.), 1170–83. http://doi.acm.org/10.1145/7902.7903.

IEEE 754R Working Group [2006]. *DRAFT Standard for Floating-Point Arithmetic P754*. www.validlab.com/754R/drafts/archive/2006-10-04.pdf.

Industrial Light and Magic [2003]. *OpenEXR*, www.openexr.com.

Intel Corporation [2007]. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. November. Order Number: 248966-016. Also: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

Kessenich, J. [2006]. *The OpenGL Shading Language, Language Version 1.20, Sept. 2006*. www.opengl.org/documentation/specs/.

Kirk, D. and D. Voorhies [1990]. "The Rendering Architecture of the DN10000VS." *Proc. SIGGRAPH* 1990 (August), 299–307.

Lindholm E., M.J. Kilgard, and H. Moreton [2001]. "A User-Programmable Vertex Engine." *Proc. SIGGRAPH* 2001 (August), 149–58.

Lindholm, E., J. Nickolls, S. Oberman, and J. Montrym [2008]. "NVIDIA Tesla: A Unified Graphics and Computing Architecture", *IEEE Micro* Vol. 28, no. 2 (March–April), 39–55.

Microsoft Corporation. *Microsoft DirectX Specification*, https://msdn.microsoft.com/en-us/library/windows/apps/hh452744.aspx.

Microsoft Corporation. [2003]. *Microsoft DirectX 9 Programmable Graphics Pipeline*, Microsoft Press, Redmond, WA.

Montrym, J., D. Baum, D. Dignam, and C. Migdal [1997]. "InfiniteReality: A Real-Time Graphics System." *Proc. SIGGRAPH 1997* (August), 293–301.

Montrym, J. and H. Moreton [2005]. "The GeForce 6800", *IEEE Micro* Vol. 25, no. 2 (March–April), 41–51.

Moore, G. E. [1965]. "Cramming more components onto integrated circuits", *Electronics*, Vol. 38, no. 8 (April 19).

Nguyen, H. ed. [2008]. *GPU Gems 3*, Addison-Wesley, Reading, MA.

Nickolls, J., I. Buck, M. Garland, and K. Skadron [2008]. "Scalable Parallel Programming with CUDA", *ACM Queue*, Vol. 6, no. 2 (March–April), 40–53.

NVIDIA [2007]. CUDA Zone. http://www.nvidia.com/object/cuda_home_new.html.

NVIDIA [2007]. *CUDA Programming Guide 1.1*. https://developer.nvidia.com/nvidia-gpu-programming-guide.

NVIDIA [2007]. *PTX: Parallel Thread Execution ISA version 1.1*. www.nvidia.com/object/io_1195170102263.html.

Nyland, L., M. Harris, and J. Prins [2007]. "Fast N-Body Simulation with CUDA". In *GPU Gems 3*, H. Nguyen (Ed.), Addison-Wesley, Reading, MA.

Oberman, S.F. and M.Y. Siu [2005]. "A High-Performance Area-Efficient Multifunction Interpolator," *Proc. Seventeenth IEEE Symp. Computer Arithmetic*, 272–79.

Pharr, M. (Ed.), [2005]. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Addison-Wesley, Reading, MA.

Satish, N., M. Harris, and M. Garland [2008]. "Designing Efficient Sorting Algorithms for Manycore GPUs," NVIDIA Technical Report NVR-2008-001.

Segal, M. and K. Akeley [2006]. *The OpenGL Graphics System: A Specification, Version 2.1, Dec. 1, 2006*. www.opengl.org/documentation/specs/.

Sengupta, S., M. Harris, Y. Zhang, and J.D. Owens [2007]. "Scan Primitives for GPU Computing." In *Proc. of Graphics Hardware 2007* (August), 97–106.

Volkov, V. and J. Demmel [2008]. "LU, QR and Cholesky Factorizations using Vector Capabilities of GPUs," Technical Report No. UCB/EECS-2008-49, 1–11. http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-49.pdf.

Williams, S., L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel [2007]. "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," In *Proc. Supercomputing 2007*, November.

Appendix D

Bhandarkar, D. P. [1995]. *Alpha Architecture and Implementations*, Newton, MA: Digital Press.

Darcy, J.D., and D. Gay [1996]. "FLECKmarks: Measuring floating point performance using a compliant arithmetic benchmark," CS 252 class project, U.C Berkeley (see *www.sonic.net/~jddarcy/Research/fleckmrk.pdf*)).

Digital Semiconductor. [1996]. *Alpha Architecture Handbook, Version 3*, Digital Press, Maynard, MA. Order number EC-QD2KB-TE (October).

Furber, S. B. [1996]. *ARM System Architecture*, Addison-Wesley, Harlow, England. (See *http://www.pearsonhighered.com/pearsonhigheredus/educator/product/products_detail.page?isbn=9780201675191&forced_logout=forced_logged_out#sthash.QX4WfErc*).

Hewlett-Packard [1994]. *PA-RISC 2.0 Architecture Reference Manual*, 3rd ed.

Hitachi [1997]. *SuperH RISC Engine SH7700 Series Programming Manual*. (See *http://am.renesas.com/products/mpumcu/superh/sh7700/Documentation.jsp*).

IBM. [1994]. *The PowerPC Architecture*, San Francisco: Morgan Kaufmann.

Kane, G. [1996]. *PA-RISC 2.0 Architecture*, Upper Saddle River, NJ: Prentice Hall PTR.

Kane, G. and J. Heinrich [1992]. *MIPS RISC Architecture*, Englewood Cliffs, NJ: Prentice Hall.

Kissell, K.D. [1997]. *MIPS16: High-Density for the Embedded Market*.

Magenheimer, D. J., L. Peters, K. W. Pettis, and D. Zuras [1988]. "Integer multiplication and division on the HP precision architecture", *IEEE Trans. on Computers* 37(8), 980–990.

MIPS [1997]. *MIPS16 Application Specific Extension Product Description*.

Mitsubishi [1996]. *Mitsubishi 32-Bit Single Chip Microcomputer M32R Family Software Manual* (September).

Muchnick, S. S. [1988]. "Optimizing compilers for SPARC", *Sun Technology* 1:3 (Summer), 64–77.

Seal, D. *Arm Architecture Reference Manual*, 2nd ed, Morgan Kaufmann, 2000.

Silicon Graphics [1996]. *MIPS V Instruction Set*.

Sites, R. L., and R. Witek (Eds.) [1995]. *Alpha Architecture Reference Manual*, 2nd ed., Newton, MA: Digital Press.

Sloss, A. N., D. Symes, and C. Wright, *ARM System Developer's Guide*, San Francisco: Elsevier Morgan Kaufmann, 2004.

Sun Microsystems [1989]. *The SPARC Architectural Manual*, Version 8, Part No. 800-1399-09, August 25.

Sweetman, D. *See MIPS Run*, 2nd ed, Morgan Kaufmann, 2006.

Taylor, G., P. Hilfinger, J. Larus, D. Patterson, and B. Zorn [1986]. "Evaluation of the SPUR LISP architecture," *Proc. 13th Symposium on Computer Architecture* (June), Tokyo.

Ungar, D., R. Blau, P. Foley, D. Samples, and D. Patterson [1984]. "Architecture of SOAR: Smalltalk on a RISC," *Proc. 11th Symposium on Computer Architecture* (June), Ann Arbor, MI, 188–97.

Weaver, D. L. and T. Germond [1994]. *The SPARC Architectural Manual, Version 9*, Prentice Hall, Englewood Cliffs, NJ.

Weiss, S. and J. E. Smith [1994]. *Power and PowerPC*, San Francisco: Morgan Kaufmann.

RISC-V Reference Data Card ("Green Card") 1. Pull along perforation to separate card 2. Fold bottom side (columns 3 and 4) together

RISC-V

Reference Data

①

RV64I BASE INTEGER INSTRUCTIONS, in alphabetical order

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
add,addw	R	ADD (Word)	R[rd] = R[rs1] + R[rs2]	1)
addi,addiw	I	ADD Immediate (Word)	R[rd] = R[rs1] + imm	1)
and	R	AND	R[rd] = R[rs1] & R[rs2]	
andi	I	AND Immediate	R[rd] = R[rs1] & imm	
auipc	U	Add Upper Immediate to PC	R[rd] = PC + {imm, 12'b0}	
beq	SB	Branch EQual	if(R[rs1]==R[rs2) PC=PC+{imm,1b'0}	
bge	SB	Branch Greater than or Equal	if(R[rs1]>=R[rs2) PC=PC+{imm,1b'0}	
bgeu	SB	Branch ≥ Unsigned	if(R[rs1]>=R[rs2) PC=PC+{imm,1b'0}	2)
blt	SB	Branch Less Than	if(R[rs1]<R[rs2) PC=PC+{imm,1b'0}	
bltu	SB	Branch Less Than Unsigned	if(R[rs1]<R[rs2) PC=PC+{imm,1b'0}	2)
bne	SB	Branch Not Equal	if(R[rs1]!=R[rs2) PC=PC+{imm,1b'0}	
csrrc	I	Cont./Stat.RegRead&Clear	R[rd] = CSR;CSR = CSR & ~R[rs1]	
csrrci	I	Cont./Stat.RegRead&Clear Imm	R[rd] = CSR;CSR = CSR & ~imm	
csrrs	I	Cont./Stat.RegRead&Set	R[rd] = CSR; CSR = CSR | R[rs1]	
csrrsi	I	Cont./Stat.RegRead&Set Imm	R[rd] = CSR; CSR = CSR | imm	
csrrw	I	Cont./Stat.RegRead&Write	R[rd] = CSR; CSR = R[rs1]	
csrrwi	I	Cont./Stat.Reg Read&Write Imm	R[rd] = CSR; CSR = imm	
ebreak	I	Environment BREAK	Transfer control to debugger	
ecall	I	Environment CALL	Transfer control to operating system	
fence	I	Synch thread	Synchronizes threads	
fence.i	I	Synch Instr & Data	Synchronizes writes to instruction stream	
jal	UJ	Jump & Link	R[rd] = PC+4; PC = PC + {imm,1b'0}	
jalr	I	Jump & Link Register	R[rd] = PC+4; PC = R[rs1]+imm	3)
lb	I	Load Byte	R[rd] = {56'bM[](7),M[R[rs1]+imm](7:0)}	4)
lbu	I	Load Byte Unsigned	R[rd] = {56'b0,M[R[rs1]+imm](7:0)}	
ld	I	Load Doubleword	R[rd] = M[R[rs1]+imm](63:0)	
lh	I	Load Halfword	R[rd] = {48'bM[](15),M[R[rs1]+imm](15:0)}	4)
lhu	I	Load Halfword Unsigned	R[rd] = {48'b0,M[R[rs1]+imm](15:0)}	
lui	U	Load Upper Immediate	R[rd] = {32b'imm<31>, imm, 12'b0}	
lw	I	Load Word	R[rd] = {32'bM[](31),M[R[rs1]+imm](31:0)}	4)
lwu	I	Load Word Unsigned	R[rd] = {32'b0,M[R[rs1]+imm](31:0)}	
or	R	OR	R[rd] = R[rs1] | R[rs2]	
ori	I	OR Immediate	R[rd] = R[rs1] | imm	
sb	S	Store Byte	M[R[rs1]+imm](7:0) = R[rs2](7:0)	
sd	S	Store Doubleword	M[R[rs1]+imm](63:0) = R[rs2](63:0)	
sh	S	Store Halfword	M[R[rs1]+imm](15:0) = R[rs2](15:0)	
sll,sllw	R	Shift Left (Word)	R[rd] = R[rs1] << R[rs2]	1)
slli,slliw	I	Shift Left Immediate (Word)	R[rd] = R[rs1] << imm	1)
slt	R	Set Less Than	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	
slti	I	Set Less Than Immediate	R[rd] = (R[rs1] < imm) ? 1 : 0	
sltiu	I	Set < Immediate Unsigned	R[rd] = (R[rs1] < imm) ? 1 : 0	2)
sltu	R	Set Less Than Unsigned	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	2)
sra,sraw	R	Shift Right Arithmetic (Word)	R[rd] = R[rs1] >> R[rs2]	1,5)
srai,sraiw	I	Shift Right Arith Imm (Word)	R[rd] = R[rs1] >> imm	1,5)
srl,srlw	R	Shift Right (Word)	R[rd] = R[rs1] >> R[rs2]	1)
srli,srliw	I	Shift Right Immediate (Word)	R[rd] = R[rs1] >> imm	1)
sub,subw	R	SUBtract (Word)	R[rd] = R[rs1] – R[rs2]	1)
sw	S	Store Word	M[R[rs1]+imm](31:0) = R[rs2](31:0)	
xor	R	XOR	R[rd] = R[rs1] ^ R[rs2]	
xori	I	XOR Immediate	R[rd] = R[rs1] ^ imm	

Notes: 1) The Word version only operates on the rightmost 32 bits of a 64-bit registers
2) Operation assumes unsigned integers (instead of 2's complement)
3) The least significant bit of the branch address in jalr is set to 0
4) (signed) Load instructions extend the sign bit of data to fill the 64-bit register
5) Replicates the sign bit to fill in the leftmost bits of the result during right shift
6) Multiply with one operand signed and one unsigned
7) The Single version does a single-precision operation using the rightmost 32 bits of a 64-bit F register
8) Classify writes a 10-bit mask to show which properties are true (e.g., –inf, -0, +0, +inf, denorm, ...)
9) Atomic memory operation; nothing else can interpose itself between the read and the write of the memory location
The immediate field is sign-extended in RISC-V

ARITHMETIC CORE INSTRUCTION SET

②

RV64M Multiply Extension

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
mul,mulw	R	MULtiply (Word)	R[rd] = (R[rs1] * R[rs2])(63:0)	1)
mulh	R	MULtiply High	R[rd] = (R[rs1] * R[rs2])(127:64)	
mulhu	R	MULtiply High Unsigned	R[rd] = (R[rs1] * R[rs2])(127:64)	2)
mulhsu	R	MULtiply upper Half Sign/Uns	R[rd] = (R[rs1] * R[rs2])(127:64)	6)
div,divw	R	DIVide (Word)	R[rd] = (R[rs1] / R[rs2])	1)
divu	R	DIVide Unsigned	R[rd] = (R[rs1] / R[rs2])	2)
rem,remw	R	REMainder (Word)	R[rd] = (R[rs1] % R[rs2])	1)
remu,remuw	R	REMainder Unsigned (Word)	R[rd] = (R[rs1] % R[rs2])	1,2)

RV64F and RV64D Floating-Point Extensions

fld,flw	I	Load (Word)	F[rd] = M[R[rs1]+imm]	1)
fsd,fsw	S	Store (Word)	M[R[rs1]+imm] = F[rd]	1)
fadd.s,fadd.d	R	ADD	F[rd] = F[rs1] + F[rs2]	7)
fsub.s,fsub.d	R	SUBtract	F[rd] = F[rs1] – F[rs2]	7)
fmul.s,fmul.d	R	MULtiply	F[rd] = F[rs1] * F[rs2]	7)
fdiv.s,fdiv.d	R	DIVide	F[rd] = F[rs1] / F[rs2]	7)
fsqrt.s,fsqrt.d	R	SQuare RooT	F[rd] = sqrt(F[rs1])	7)
fmadd.s,fmadd.d	R	Multiply-ADD	F[rd] = F[rs1] * F[rs2] + F[rs3]	7)
fmsub.s,fmsub.d	R	Multiply-SUBtract	F[rd] = F[rs1] * F[rs2] - F[rs3]	7)
fnmadd.s,fnmadd.d	R	Negative Multiply-ADD	F[rd] = –(F[rs1] * F[rs2] + F[rs3])	7)
fnmsub.s,fnmsub.d	R	Negative Multiply-SUBtract	F[rd] = –(F[rs1] * F[rs2] – F[rs3])	7)
fsgnj.s,fsgnj.d	R	SiGN source	F[rd] = { F[rs2]<63>,F[rs1]<62:0>}	7)
fsgnjn.s,fsgnjn.d	R	Negative SiGN source	F[rd] = { (~F[rs2]<63>), F[rs1]<62:0>}	7)
fsgnjx.s,fsgnjx.d	R	Xor SiGN source	F[rd] = {F[rs2]<63>^F[rs1]<63>, F[rs1]<62:0>}	7)
fmin.s,fmin.d	R	MINimum	F[rd] = (F[rs1] < F[rs2]) ? F[rs1] : F[rs2]	7)
fmax.s,fmax.d	R	MAXimum	F[rd] = (F[rs1] > F[rs2]) ? F[rs1] : F[rs2]	7)
feq.s,feq.d	R	Compare Float EQual	R[rd] = (F[rs1]== F[rs2]) ? 1 : 0	7)
flt.s,flt.d	R	Compare Float Less Than	R[rd] = (F[rs1]< F[rs2]) ? 1 : 0	7)
fle.s,fle.d	R	Compare Float Less than or =	R[rd] = (F[rs1]<= F[rs2]) ? 1 : 0	7)
fclass.s,fclass.d	R	Classify Type	R[rd] = class(F[rs1])	7,8)
fmv.s.x,fmv.d.x	R	Move from Integer	F[rd] = R[rs1]	7)
fmv.x.s,fmv.x.d	R	Move to Integer	R[rd] = F[rs1]	7)
fcvt.s.d	R	Convert to SP from DP	F[rd] = single(F[rs1])	
fcvt.d.s	R	Convert to DP from SP	F[rd] = double(F[rs1])	
fcvt.s.w,fcvt.d.w	R	Convert from 32b Integer	F[rd] = float(R[rs1](31:0))	7)
fcvt.s.l,fcvt.d.l	R	Convert from 64b Integer	F[rd] = float(R[rs1](63:0))	7)
fcvt.s.wu,fcvt.d.wu	R	Convert from 32b Int Unsigned	F[rd] = float(R[rs1](31:0))	2,7)
fcvt.s.lu,fcvt.d.lu	R	Convert from 64b Int Unsigned	F[rd] = float(R[rs1](63:0))	2,7)
fcvt.w.s,fcvt.w.d	R	Convert to 32b Integer	R[rd](31:0) = integer(F[rs1])	7)
fcvt.l.s,fcvt.l.d	R	Convert to 64b Integer	R[rd](63:0) = integer(F[rs1])	7)
fcvt.wu.s,fcvt.wu.d	R	Convert to 32b Int Unsigned	R[rd](31:0) = integer(F[rs1])	2,7)
fcvt.lu.s,fcvt.lu.d	R	Convert to 64b Int Unsigned	R[rd](63:0) = integer(F[rs1])	2,7)

RV64A Atomtic Extension

amoadd.w,amoadd.d	R	ADD	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] + R[rs2]	9)
amoand.w,amoand.d	R	AND	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] & R[rs2]	9)
amomax.w,amomax.d	R	MAXimum	R[rd] = M[R[rs1]], if (R[rs2] > M[R[rs1]]) M[R[rs1]] = R[rs2]	9)
amomaxu.w,amomaxu.d	R	MAXimum Unsigned	R[rd] = M[R[rs1]], if (R[rs2] > M[R[rs1]]) M[R[rs1]] = R[rs2]	2,9)
amomin.w,amomin.d	R	MINimum	R[rd] = M[R[rs1]], if (R[rs2] < M[R[rs1]]) M[R[rs1]] = R[rs2]	9)
amominu.w,amominu.d	R	MINimum Unsigned	R[rd] = M[R[rs1]], if (R[rs2] < M[R[rs1]]) M[R[rs1]] = R[rs2]	2,9)
amoor.w,amoor.d	R	OR	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] | R[rs2]	9)
amoswap.w,amoswap.d	R	SWAP	R[rd] = M[R[rs1]], M[R[rs1]] = R[rs2]	9)
amoxor.w,amoxor.d	R	XOR	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] ^ R[rs2]	9)
lr.w,lr.d	R	Load Reserved	R[rd] = M[R[rs1]], reservation on M[R[rs1]]	
sc.w,sc.d	R	Store Conditional	if reserved, M[R[rs1]] = R[rs2], R[rd] = 0; else R[rd] = 1	

CORE INSTRUCTION FORMATS

	31 27	26 25	24 20	19 15	14 12	11 7	6 0
R	funct7		rs2	rs1	funct3	rd	Opcode
I	imm[11:0]			rs1	funct3	rd	Opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
UJ	imm[20|10:1|11|19:12]					rd	opcode

© 2018 by Elsevier, Inc. All rights reserved. From Patterson and Hennessy, *Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition*

PSEUDO INSTRUCTIONS ③

MNEMONIC	NAME	DESCRIPTION	USES
beqz	Branch = zero	if(R[rs1]==0) PC=PC+{imm,1b'0}	beq
bnez	Branch ≠ zero	if(R[rs1]!=0) PC=PC+{imm,1b'0}	bne
fabs.s,fabs.d	Absolute Value	F[rd] = (F[rs1]< 0) ? –F[rs1] : F[rs1]	fsgnx
fmv.s,fmv.d	FP Move	F[rd] = F[rs1]	fsgnj
fneg.s,fneg.d	FP negate	F[rd] = –F[rs1]	fsgnjn
j	Jump	PC = {imm,1b'0}	jal
jr	Jump register	PC = R[rs1]	jalr
la	Load address	R[rd] = address	auipc
li	Load imm	R[rd] = imm	addi
mv	Move	R[rd] = R[rs1]	addi
neg	Negate	R[rd] = –R[rs1]	sub
nop	No operation	R[0] = R[0]	addi
not	Not	R[rd] = ~R[rs1]	xori
ret	Return	PC = R[1]	jalr
seqz	Set = zero	R[rd] = (R[rs1]== 0) ? 1 : 0	sltiu
snez	Set ≠ zero	R[rd] = (R[rs1]!= 0) ? 1 : 0	sltu

OPCODES IN NUMERICAL ORDER BY OPCODE

MNEMONIC	FMT	OPCODE	FUNCT3	FUNCT7 OR IMM	HEXADECIMAL
lb	I	0000011	000		03/0
lh	I	0000011	001		03/1
lw	I	0000011	010		03/2
ld	I	0000011	011		03/3
lbu	I	0000011	100		03/4
lhu	I	0000011	101		03/5
lwu	I	0000011	110		03/6
fence	I	0001111	000		0F/0
fence.i	I	0001111	001		0F/1
addi	I	0010011	000		13/0
slli	I	0010011	001	0000000	13/1/00
slti	I	0010011	010		13/2
sltiu	I	0010011	011		13/3
xori	I	0010011	100		13/4
srli	I	0010011	101	0000000	13/5/00
srai	I	0010011	101	0100000	13/5/20
ori	I	0010011	110		13/6
andi	I	0010011	111		13/7
auipc	U	0010111			17
addiw	I	0011011	000		1B/0
slliw	I	0011011	001	0000000	1B/1/00
srliw	I	0011011	101	0000000	1B/5/00
sraiw	I	0011011	101	0100000	1B/5/20
sb	S	0100011	000		23/0
sh	S	0100011	001		23/1
sw	S	0100011	010		23/2
sd	S	0100011	011		23/3
add	R	0110011	000	0000000	33/0/00
sub	R	0110011	000	0100000	33/0/20
sll	R	0110011	001	0000000	33/1/00
slt	R	0110011	010	0000000	33/2/00
sltu	R	0110011	011	0000000	33/3/00
xor	R	0110011	100	0000000	33/4/00
srl	R	0110011	101	0000000	33/5/00
sra	R	0110011	101	0100000	33/5/20
or	R	0110011	110	0000000	33/6/00
and	R	0110011	111	0000000	33/7/00
lui	U	0110111			37
addw	R	0111011	000	0000000	3B/0/00
subw	R	0111011	000	0100000	3B/0/20
sllw	R	0111011	001	0000000	3B/1/00
srlw	R	0111011	101	0000000	3B/5/00
sraw	R	0111011	101	0100000	3B/5/20
beq	SB	1100011	000		63/0
bne	SB	1100011	001		63/1
blt	SB	1100011	100		63/4
bge	SB	1100011	101		63/5
bltu	SB	1100011	110		63/6
bgeu	SB	1100011	111		63/7
jalr	I	1100111	000		67/0
jal	UJ	1101111			6F
ecall	I	1110011	000	000000000000	73/0/000
ebreak	I	1110011	000	000000000001	73/0/001
CSRRW	I	1110011	001		73/1
CSRRS	I	1110011	010		73/2
CSRRC	I	1110011	011		73/3
CSRRWI	I	1110011	101		73/5
CSRRSI	I	1110011	110		73/6
CSRRCI	I	1110011	111		73/7

REGISTER NAME, USE, CALLING CONVENTION ④

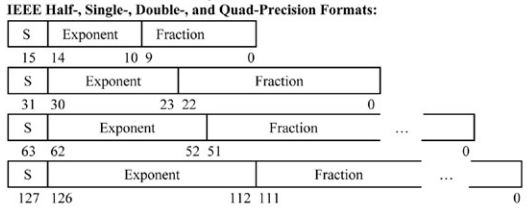
REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5–x7	t0–t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10–x11	a0–a1	Function arguments/Return values	Caller
x12–x17	a2–a7	Function arguments	Caller
x18–x27	s2–s11	Saved registers	Callee
x28–x31	t3–t6	Temporaries	Caller
f0–f7	ft0–ft7	FP Temporaries	Caller
f8–f9	fs0–fs1	FP Saved registers	Callee
f10–f11	fa0–fa1	FP Function arguments/Return values	Caller
f12–f17	fa2–fa7	FP Function arguments	Caller
f18–f27	fs2–fs11	FP Saved registers	Callee
f28–f31	ft8–ft11	R[rd] = R[rs1] + R[rs2]	Caller

IEEE 754 FLOATING-POINT STANDARD

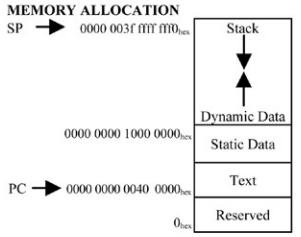
$(-1)^{S} \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$

where Half-Precision Bias = 15, Single-Precision Bias = 127, Double-Precision Bias = 1023, Quad-Precision Bias = 16383

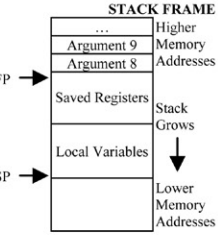
IEEE Half-, Single-, Double-, and Quad-Precision Formats:



MEMORY ALLOCATION



STACK FRAME



SIZE PREFIXES AND SYMBOLS

SIZE	PREFIX	SYMBOL	SIZE	PREFIX	SYMBOL
10^{3}	Kilo-	K	2^{10}	Kibi-	Ki
10^{6}	Mega-	M	2^{20}	Mebi-	Mi
10^{9}	Giga-	G	2^{30}	Gibi-	Gi
10^{12}	Tera-	T	2^{40}	Tebi-	Ti
10^{15}	Peta-	P	2^{50}	Pebi-	Pi
10^{18}	Exa-	E	2^{60}	Exbi-	Ei
10^{21}	Zetta-	Z	2^{70}	Zebi-	Zi
10^{24}	Yotta-	Y	2^{80}	Yobi-	Yi
10^{-3}	milli-	m	10^{-15}	femto-	f
10^{-6}	micro-	μ	10^{-18}	atto-	a
10^{-9}	nano-	n	10^{-21}	zepto-	z
10^{-12}	pico-	p	10^{-24}	yocto-	y

© 2018 by Elsevier, Inc. All rights reserved. From Patterson and Hennessy, *Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition*

RISC-V Reference Data Card ("Green Card") 1. Pull along perforation to separate card 2. Fold bottom side (columns 3 and 4) together

COMPUTER ORGANIZATION AND DESIGN

THE HARDWARE/SOFTWARE INTERFACE

RISC-V EDITION

DAVID A. PATTERSON | JOHN L. HENNESSY

"It is rare to encounter a practicing engineer who was not introduced to computer architecture through earlier editions of this classic textbook from these superlative teachers, and so it's thrilling to now see *Computer Organization and Design* adopt the free and open RISC-V architecture. RISC-V was created to support research and education but is now poised to become an industry-standard instruction set. Future generations of students will learn about this most modern of RISC architectures from the original RISC pioneers, and will be able to dive straight into making their own RISC-V cores while benefiting from, and hopefully contributing to, the large open-source hardware and software ecosystem developing around this new standard."

- Professor Krste Asanović, University of California, Berkeley; Chairman of the Board, RISC-V Foundation

The new RISC-V Edition of *Computer Organization and Design* has been updated to feature the free and open RISC-V architecture, which is used to present the fundamentals of hardware technologies, assembly language, computer arithmetic, pipelining, memory hierarchies, and I/O.

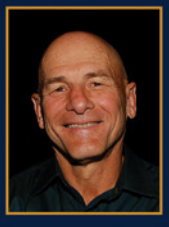
With the post-PC era now upon us, ***Computer Organization and Design*** moves forward to explore this generational change with examples, exercises, and material highlighting the emergence of mobile computing and the Cloud. Updated content featuring tablet computers, Cloud infrastructure, and the x86 (cloud computing) and ARM (mobile computing devices) architectures is included.

An online companion website provides links to RISC-V software tools, as well as additional advanced content for further study, appendices, glossary, references, and recommended reading.

RISC-V EDITION FEATURES

- Covers parallelism in depth with examples and content highlighting parallel hardware and software topics
- Features the Intel Core i7, ARM Cortex-A53, and NVIDIA Fermi GPU as real-world examples throughout the book
- Adds a new concrete example, "Going Faster," to demonstrate how understanding hardware can inspire software optimizations that improve performance by 200 times
- Discusses and highlights the "Eight Great Ideas" of computer architecture: Performance via Parallelism; Performance via Pipelining; Performance via Prediction; Design for Moore's Law; Hierarchy of Memories; Abstraction to Simplify Design; Make the Common Case Fast; and Dependability via Redundancy
- Includes a full set of updated and improved exercises

ABOUT THE AUTHORS



David A. Patterson
Pardee Chair of Computer Science, Emeritus
University of California at Berkeley



John L. Hennessy
Professor of Electrical Engineering and Computer Science
Stanford University

COMPUTER SYSTEMS DESIGN
COMPUTER HARDWARE

MORGAN KAUFMANN PUBLISHERS
AN IMPRINT OF ELSEVIER
elsevier.com/books-and-journals

ISBN 978-0-12-812275-4
9 780128 122754