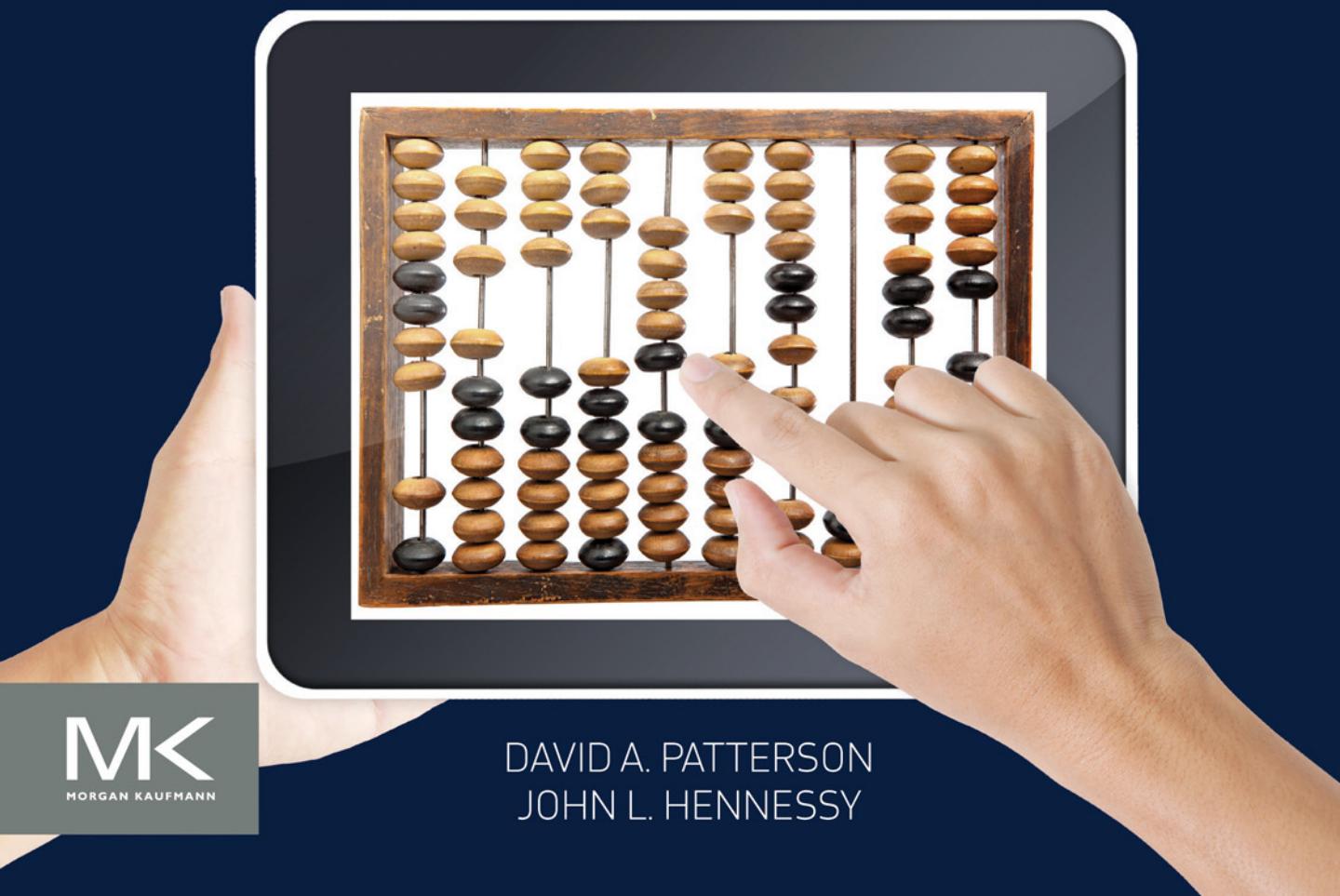


# COMPUTER ORGANIZATION AND DESIGN

THE HARDWARE/SOFTWARE INTERFACE

**RV RISC-V EDITION**



**MK**  
MORGAN KAUFMANN

DAVID A. PATTERSON  
JOHN L. HENNESSY

# 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



MORGAN KAUFMANN PUBLISHERS

AN IMPRINT OF ELSEVIER  
[elsevier.com/books-and-journals](http://elsevier.com/books-and-journals)

COMPUTER SYSTEMS DESIGN  
COMPUTER HARDWARE

ISBN 978-0-12-812275-4



9 780128 122754

## 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

R I S C - V E D I T I O N

# **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

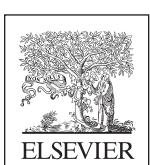
John Nickolls  
NVIDIA

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](http://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](http://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 affiliates).

#### 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>



Working together  
to grow libraries in  
developing countries

[www.elsevier.com](http://www.elsevier.com) • [www.bookaid.org](http://www.bookaid.org)

Publisher: Katelyn 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*

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

Figures 1.7, 1.8 Courtesy of iFixit ([www.ifixit.com](http://www.ifixit.com)).

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

Figure 1.13 Courtesy of Intel.

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

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

Figure 1.10.4 Courtesy of Cray Inc.

Figure 1.10.5 Courtesy of Apple Computer, Inc.

Figure 1.10.6 Courtesy of the Computer History Museum.

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

Figure 5.17.4 Courtesy of MIPS Technologies, Inc.

Figure 6.15.1 Courtesy of NASA Ames Research Center.

# Contents

Preface xv

## CHAPTERS

### 1

### Computer Abstractions and Technology 2

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

### 2

### Instructions: Language of the Computer 60

- 2.1 Introduction 62
- 2.2 Operations of the Computer Hardware 63
- 2.3 Operands of the Computer Hardware 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

**A P P E N D I X****A****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.1 to D.17	 	 
D. RISC Instruction-Set Architectures	3.1 to 3.5	 	 
3. Arithmetic for Computers	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 4.1 (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)	 	 
4. The Processor	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](http://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@mfp.com](mailto:codRISCVbugs@mfp.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 Avizienis** of UC Berkeley, who developed the various versions of matrix multiply and supplied the performance numbers as well. As I worked with his father while I was a graduate student at UCLA, it was a nice symmetry to work with Rimas at UCB.

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

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

I am also very grateful to **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), JaeC.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 (Southern 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

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

**Alfred North Whitehead,**  
*An Introduction to Mathematics*, 1911

## Computer Abstractions and Technology

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

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

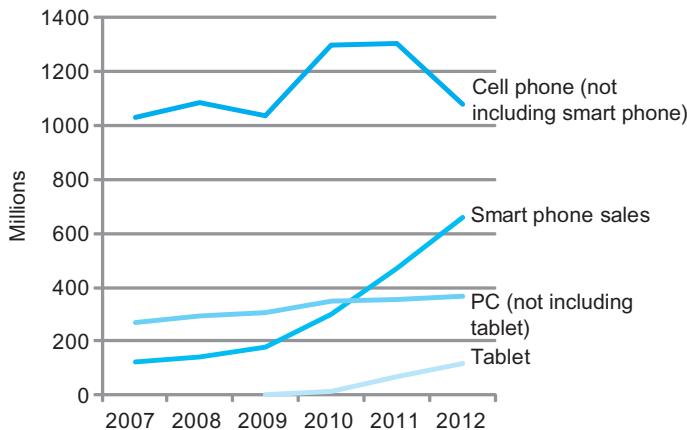
Embedded applications often have unique application requirements that combine a minimum performance with stringent limitations on cost or power. For example, consider a music player: the processor need only to be as fast as necessary to handle its limited function, and beyond that, minimizing cost and 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.

## 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

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

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

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

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

Programmers interested in performance now need to understand the issues that have replaced the simple memory model of the 1960s: the parallel nature of processors and the hierarchical nature of memories. 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.

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

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

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

## Understanding Program Performance

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

To demonstrate the impact of the ideas in this book, 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.

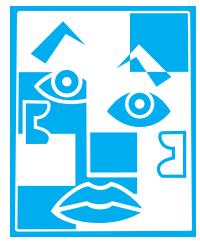
### Design for Moore's Law

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.



### 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.



### Make the Common Case Fast

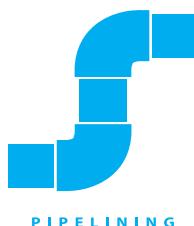
Making the **common case fast** will tend to enhance performance better than optimizing the rare case. Ironically, the common case is often simpler than the rare case and hence is usually easier to enhance. This common sense advice implies that you know what the common case is, which is only possible 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!





## 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.



## Performance via Pipelining

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



## Performance via Prediction

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

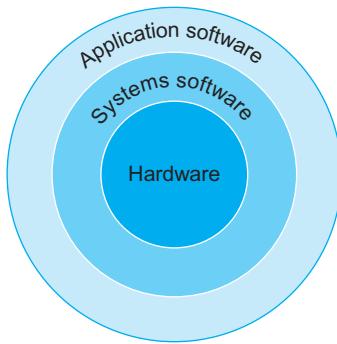
A typical application, such as a word processor or a large database system, may consist of millions of lines of code and rely on sophisticated software libraries that implement complex functions in support of the application. As we will see, the hardware in a computer can only execute extremely simple 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**.

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

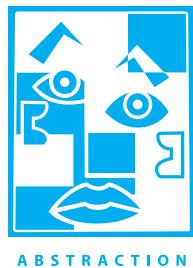
Examples of operating systems in use today are Linux, iOS, and Windows.



**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.

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

Mark Twain, *The Innocents Abroad*, 1869



### systems software

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

### operating system

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

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

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

## 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

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

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**.

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

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

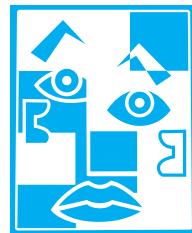
**instruction** A command that computer hardware understands and obeys.

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

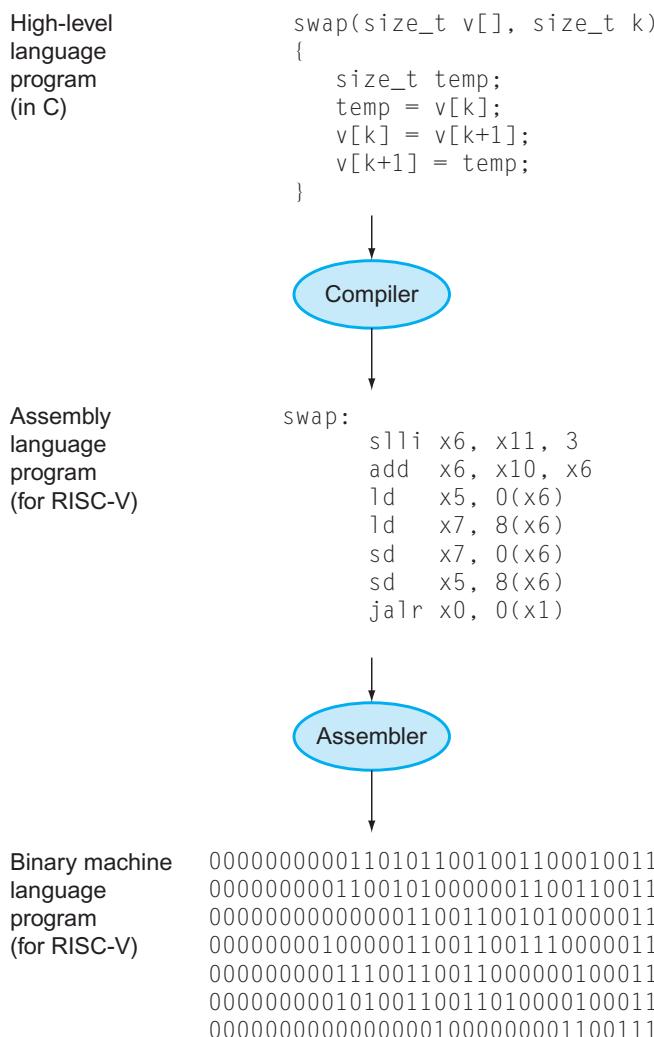
**assembly language** A symbolic representation of machine instructions.

**machine language** A binary representation of machine instructions.

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



ABSTRACTION



### high-level programming language

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

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

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

A + B

The compiler would compile it into this assembly language statement:

add A, B

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

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

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

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

## 1.4

## Under the Covers

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

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

### input device

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

### output device

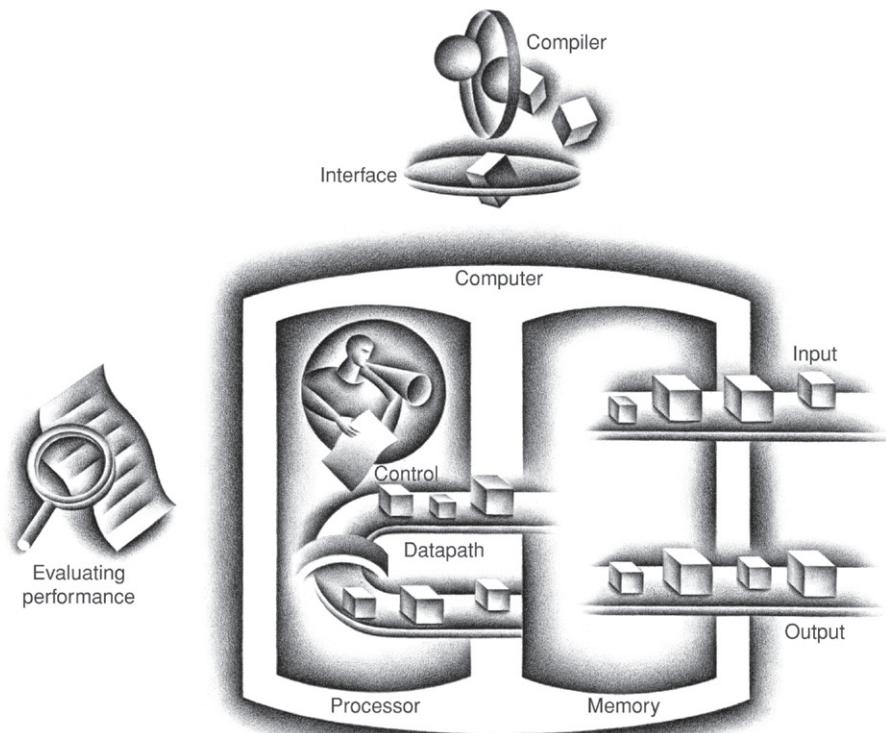
A mechanism that conveys the result of a computation to a user, such as a display, or to another computer.

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

Chapters 5 and 6 describe *input/output* (I/O) devices in more detail, but let's take an introductory tour through the computer hardware, starting with the external I/O devices.

## The 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.*

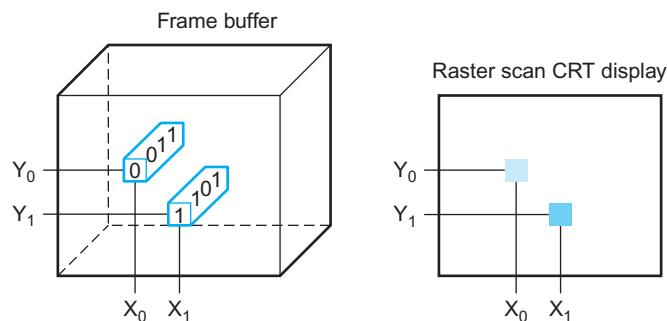
Ivan Sutherland, the “father” of computer graphics, *Scientific American*, 1984

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  $1024 \times 768$  to  $2048 \times 1536$ . 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.

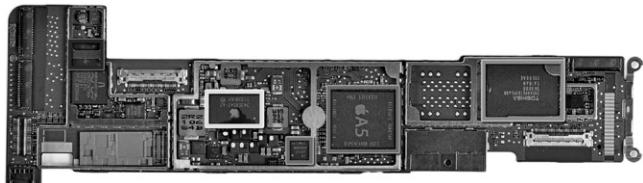
**control** The component of the processor that commands the datapath, memory, and I/O devices according to the instructions of the program.

**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](http://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](http://www.ifixit.com))



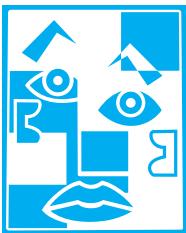
**FIGURE 1.9** The processor integrated circuit inside the A5 package. The size of chip is 12.1 by 10.1 mm, and it was manufactured originally in a 45-nm process (see Section 1.5). It has two identical ARM processors or cores in the middle left of the chip and a PowerVR graphics processing unit (GPU)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

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

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

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





ABSTRACTION

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

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

## The BIG Picture

### implementation

Hardware that obeys the architecture abstraction.

### volatile memory

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

### nonvolatile memory

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

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

An instruction set architecture allows computer designers to talk about functions independently from the hardware that performs them. For example, we can talk about the functions of a digital clock (keeping time, displaying the time, setting the alarm) 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.

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

## A Safe Place for Data

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

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

## Communicating with Other Computers

We've explained how we can input, compute, display, and save data, but there is still one missing item found in today's computers: computer networks. Just as the processor shown in Figure 1.5 is connected to memory and I/O devices, networks interconnect whole computers, allowing computer users to extend the power of computing by including communication. Networks have become so popular that they are the backbone of current computer systems; a new personal mobile device or server without a network interface would be ridiculed. Networked computers have several major advantages:

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

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



HIERARCHY

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

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

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

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

**local area network**

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

**wide area network**

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

hence, it is an example of what is generically called a **local area network**. Local area networks are interconnected with switches that can also provide routing services and security. **Wide area networks** cross continents and are the backbone of the Internet, which supports the web. They are typically based on optical fibers and are leased from telecommunication companies.

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

As networking technology improved, it became 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](#).

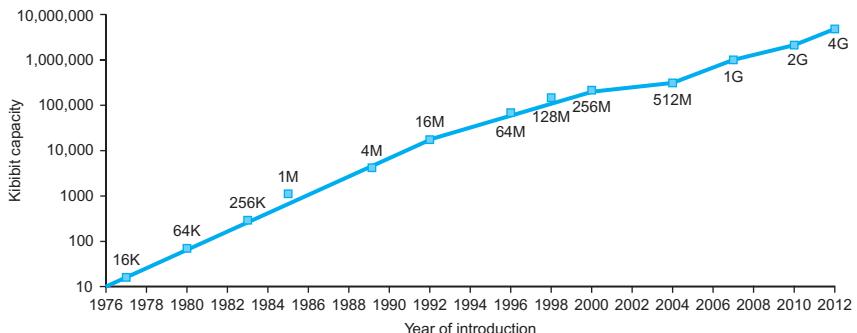
been used over time, with an estimate of the relative performance per unit cost for each technology. Since this technology shapes what computers will be able to do and how quickly they will evolve, we believe all computer professionals should be familiar with the basics of integrated circuits.

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**.

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:

- 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 kibits ( $2^{10}$  bits). The DRAM industry quadrupled capacity almost every three years, a 60% increase per year, for 20 years. In recent years, the rate has slowed down and is somewhat closer to doubling every two to three years.

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

**very large-scale integrated (VLSI)**

**circuit** A device containing hundreds of thousands to millions of transistors.

**silicon** A natural element that is a semiconductor.

**semiconductor**

A substance that does not conduct electricity well.

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

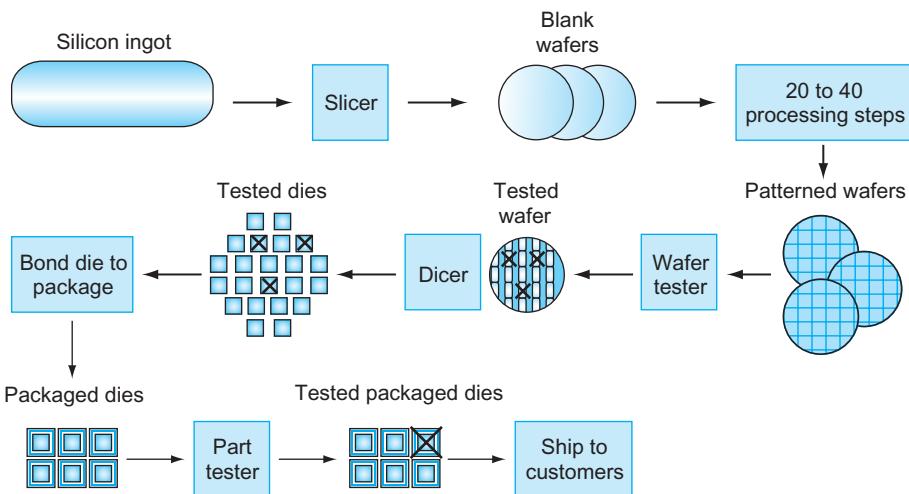
Transistors fall into the last category. A VLSI circuit, then, is just billions of combinations of conductors, insulators, and switches manufactured in a single small package.

### 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.

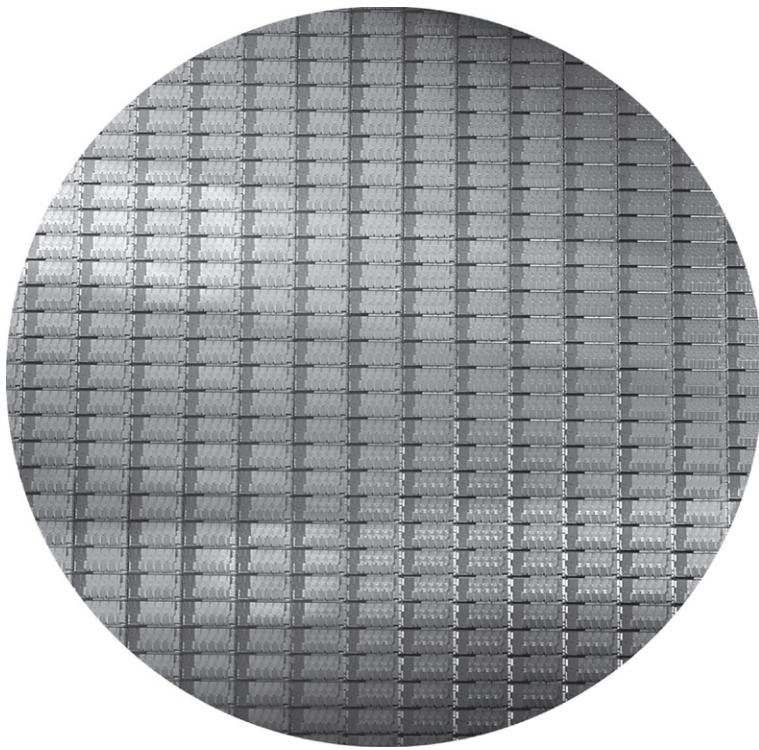
The manufacturing process for integrated circuits is critical to the cost of the chips and hence important to computer designers. Figure 1.12 shows that process. The process starts with a **silicon crystal ingot**, which looks like a giant sausage. Today, ingots are 8–12 inches in diameter and about 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

## ANSWER

### Throughput and Response Time

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

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

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

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

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

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

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

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

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

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

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

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

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

### Relative Performance

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

### EXAMPLE

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

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

### ANSWER

Thus the performance ratio is

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

and A is therefore 1.5 times as fast as B.

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

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

means that

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

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

## Measuring Performance

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

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

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

---

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

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

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

---

## Understanding Program Performance

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

some application environments, the user may care about throughput, response time, or a complex combination of the two (e.g., maximum throughput with a worst-case response time). To improve the performance of a program, one must have a clear definition of what performance metric matters and then proceed to 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.

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

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

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

**clock period** The length of each clock cycle.

## Check Yourself

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

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

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

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

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

---

## EXAMPLE

### Improving Performance

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

## ANSWER

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

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

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

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

CPU time for B can be found using this equation:

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

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

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

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

## Instruction Performance

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

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

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

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

### EXAMPLE

**ANSWER**

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

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

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

Now we can compute the CPU time for each computer:

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

Likewise, for B:

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

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

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

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

### The Classic CPU Performance Equation

**instruction count** The number of instructions executed by the program.

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

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

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

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

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

## Comparing Code Segments

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

### EXAMPLE

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

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

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

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

### ANSWER

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

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

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

This yields

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

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

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

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

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

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

## The BIG Picture

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

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

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

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

### Understanding Program Performance

Hardware or software component	Affects what?	How?
Algorithm	Instruction count, 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*.

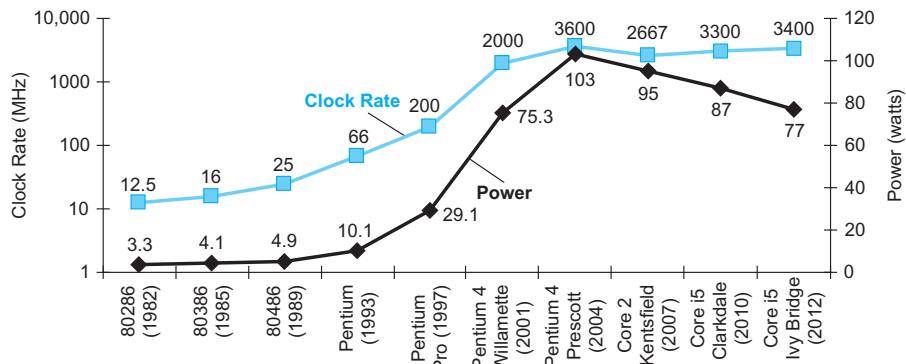
**Check Yourself** A given application written in Java runs 15 seconds on a desktop processor. A new Java compiler is released that requires only 0.6 as many instructions as the old compiler. Unfortunately, it increases the CPI by 1.1. How fast can we expect the application to run using this new compiler?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

- $\frac{15 \times 0.6}{1.1} = 8.2 \text{ sec}$
- $15 \times 0.6 \times 1.1 = 9.9 \text{ sec}$
- $\frac{1.5 \times 1.1}{0.6} = 27.5 \text{ sec}$

## 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:

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

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

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

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

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

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

With regard to [Figure 1.16](#), how could clock rates grow by a factor of 1000 while power 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.

### 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?

### EXAMPLE

**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.

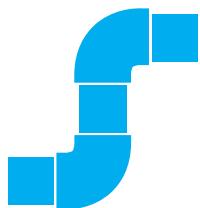


PARALLELISM

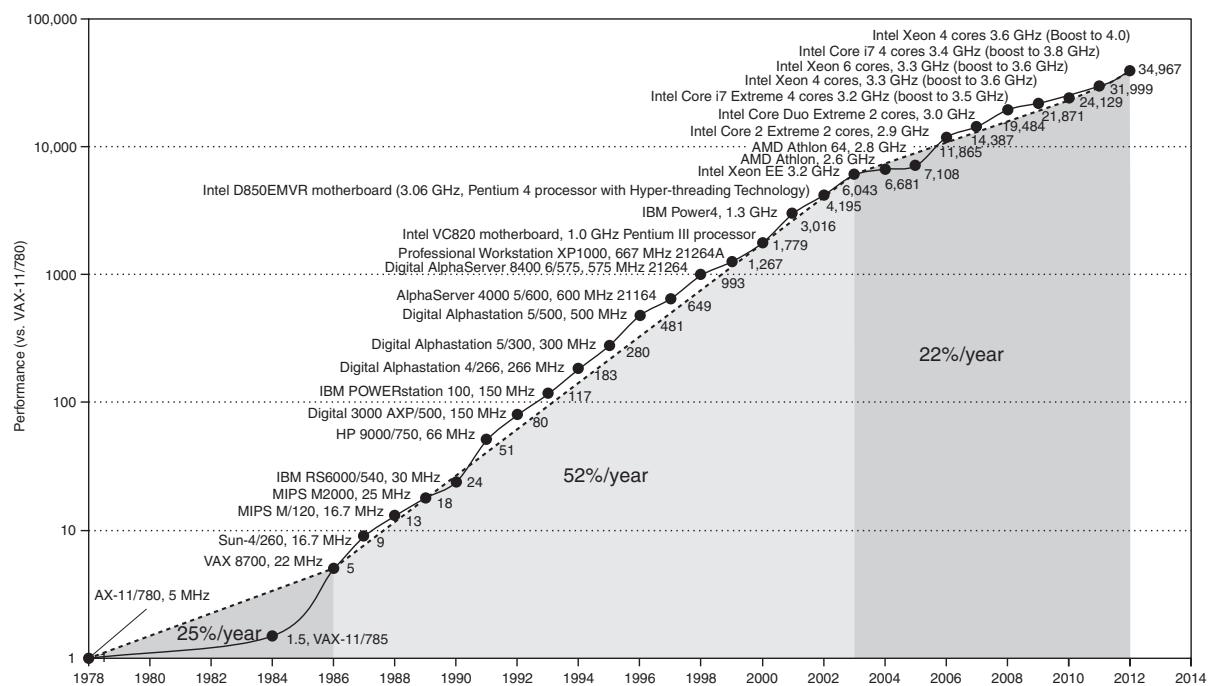
**Parallelism** has always been crucial to performance in computing, but it was often hidden. Chapter 4 will explain **pipelining**, an elegant technique that runs programs faster by overlapping the execution of instructions. This optimization is one example of *instruction-level parallelism*, where 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.

## Hardware/ Software Interface



PIPELINING



**FIGURE 1.17 Growth in processor performance since the mid-1980s.** This chart plots performance relative to the VAX 11/780 as measured by the SPECint benchmarks (see Section 1.10). Prior to the mid-1980s, processor performance growth was largely technology-driven and averaged about 25% per year. The increase in growth to about 52% since then is attributable to more advanced architectural and organizational ideas. The higher annual performance improvement of 52% since the mid-1980s meant performance was about a factor of seven 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.

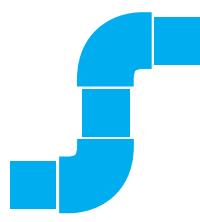
The second reason is that fast for parallel hardware means that the programmer must divide an application so that each processor has roughly the same amount to do at the same time, and that the overhead of scheduling and coordination doesn't fritter away the potential performance benefits of parallelism.

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.





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

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

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



**benchmark** A program selected for use in comparing computer performance.

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

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

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.

## 1.9

### Real Stuff: Benchmarking the Intel Core i7

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

#### SPEC CPU Benchmark

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

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

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

**FIGURE 1.18 SPECINTC2006 benchmarks running on a 2.66GHz Intel Core i7 920.** As the equation on page 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  $\text{Execution time ratio}_i$  is the execution time, normalized to the reference computer, for the  $i$ th program of a total of  $n$  in the workload, and

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

## SPEC Power Benchmark

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

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

**FIGURE 1.19** **SPECpower\_ssj2008** running on a dual socket 2.66 GHz Intel Xeon X5650 with 16 GB of DRAM and one 100 GB SSD disk.

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

boils these numbers down to one number, called “overall ssj\_ops per watt.” The formula for this single summarizing metric is

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

where  $\text{ssj\_ops}_i$  is performance at each 10% increment and  $\text{power}_i$  is power consumed at each performance level.

## 1.10 Fallacies and Pitfalls

The purpose of a section on fallacies and pitfalls, which will be found in every chapter, is to explain some commonly held misconceptions that you might encounter. We call them *fallacies*. When discussing a fallacy, we try to give a counterexample. We also discuss *pitfalls*, or easily made mistakes. Often pitfalls are generalizations of principles that are 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.

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

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

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

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

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

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

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

For this problem:

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



COMMON CASE FAST

### Amdahl's Law

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

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

$$\begin{aligned} 20 \text{ seconds} &= \frac{80 \text{ seconds}}{n} + 20 \text{ seconds} \\ 0 &= \frac{80 \text{ seconds}}{n} \end{aligned}$$

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 Hözle [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<sub>2</sub> 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

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

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:

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

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

Consider the following performance measurements for a program:

### million instructions per second (MIPS)

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

### Check Yourself

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

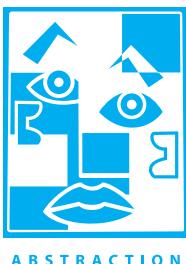
- Which computer has the higher MIPS rating?
- Which computer is faster?

## 1.11

## Concluding Remarks

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

*Popular Mechanics,  
March 1949*



ABSTRACTION

## The BIG Picture

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

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

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

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

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

Execution time is the only valid and unimpeachable measure of performance. Many other metrics have been proposed and found wanting. Sometimes these metrics are flawed from the start by not reflecting execution time; other times a metric that is 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.





## 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



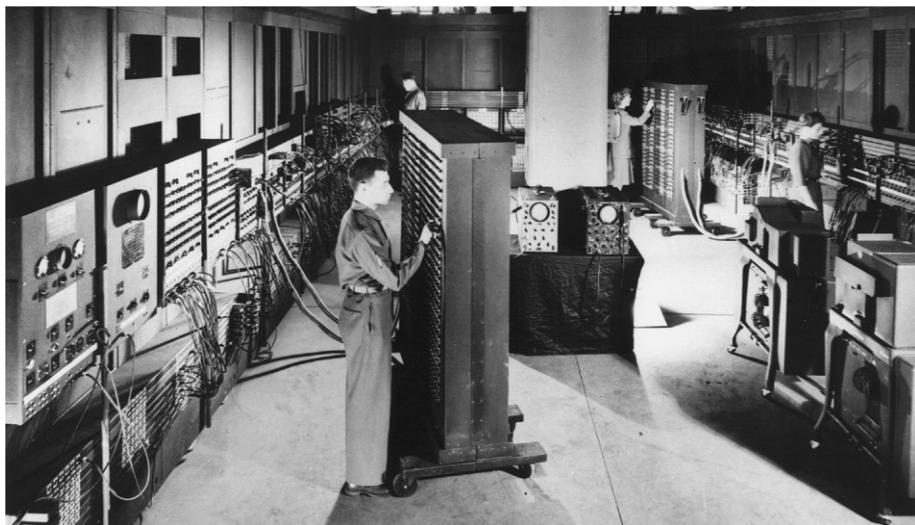
## Historical Perspective and Further Reading

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.

*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

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.*



a.



c.



b.

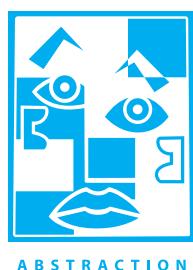


d.

**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.



ABSTRACTION

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](http://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



HIERARCHY



**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!

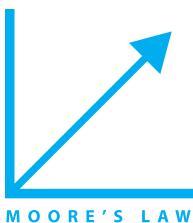
The most recent version of SPEC is SPEC2006. What is perhaps most surprising is that all floating-point programs in SPEC2006 are new, and for integer programs just two are from SPEC2000, one from SPEC95, none from SPEC92, and one from SPEC89. The sole survivor from SPEC89 is the gcc compiler.

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  $1280 \times 1024$ .

- 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<sup>2</sup>. Assume a 20 cm diameter wafer has a cost of 15, contains 100 dies, and has 0.031 defects/cm<sup>2</sup>.

**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<sup>2</sup>.

**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 4GHz 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 4GHz. 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  $50 \times 10^6$  FP instructions,  $110 \times 10^6$  INT instructions,  $80 \times 10^6$  L/S instructions, and  $16 \times 10^6$  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).

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

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

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

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

§1.6, page 40: b.

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

**Answers to  
Check Yourself**

# 2

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

**Charles V, Holy Roman Emperor**  
(1500–1558)

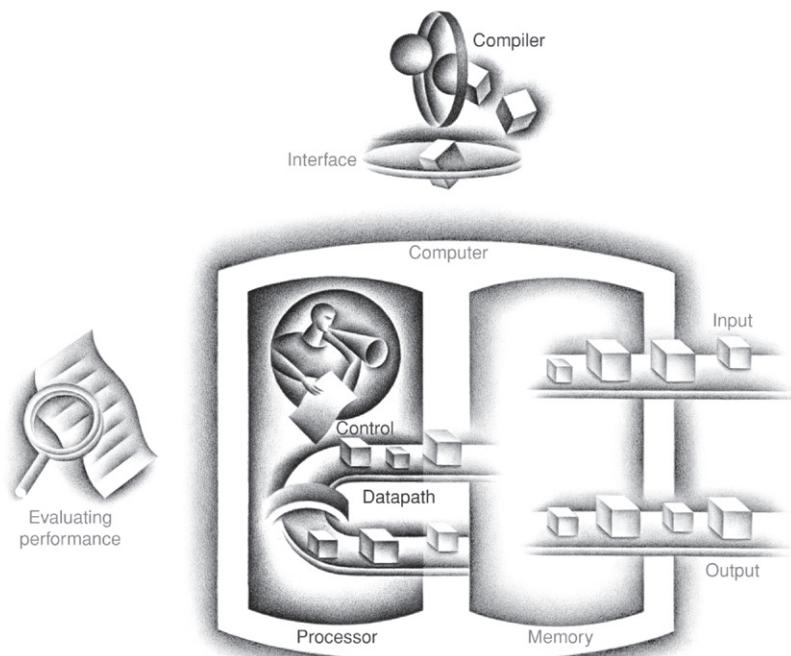
## Instructions: Language of the Computer

- 2.1      Introduction**    62
- 2.2      Operations of the Computer Hardware**    63
- 2.3      Operands of the Computer Hardware**    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

---

## 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.

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

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

## 2.2

## Operations of the Computer Hardware

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

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

Burks, Goldstine, and von Neumann, 1947

### 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)	$x5 = \text{Memory}[x6 + 40]$	Doubleword from memory to register
	Store doubleword	sd x5, 40(x6)	$\text{Memory}[x6 + 40] = x5$	Doubleword from register to memory
	Load word	lw x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Word from memory to register
	Load word, unsigned	lwu x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Unsigned word from memory to register
	Store word	sw x5, 40(x6)	$\text{Memory}[x6 + 40] = x5$	Word from register to memory
	Load halfword	lh x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Halfword from memory to register
	Load halfword, unsigned	lhu x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Unsigned halfword from memory to register
	Store halfword	sh x5, 40(x6)	$\text{Memory}[x6 + 40] = x5$	Halfword from register to memory
	Load byte	lb x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Byte from memory to register
	Load byte, unsigned	lbu x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Byte unsigned from memory to register
	Store byte	sb x5, 40(x6)	$\text{Memory}[x6 + 40] = x5$	Byte from register to memory
	Load reserved	lr.d x5, (x6)	$x5 = \text{Memory}[x6]$	Load; 1st half of atomic swap
	Store conditional	sc.d x7, x5, (x6)	$\text{Memory}[x6] = x5; x7 = 0/1$	Store; 2nd half of atomic swap
	Load upper immediate	lui x5, 0x12345000	$x5 = 0x12345000$	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	$x5 = x6 \ll x7$	Shift left by register
	Shift right logical	srl x5, x6, x7	$x5 = x6 \gg x7$	Shift right by register
	Shift right arithmetic	sra x5, x6, x7	$x5 = x6 \gg x7$	Arithmetic shift right by register
	Shift left logical immediate	slli x5, x6, 3	$x5 = x6 \ll 3$	Shift left by immediate
	Shift right logical immediate	srali x5, x6, 3	$x5 = x6 \gg 3$	Shift right by immediate
	Shift right arithmetic immediate	srai x5, x6, 3	$x5 = x6 \gg 3$	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	<code>beq x5, x6, 100</code>	<code>if (x5 == x6) go to PC+100</code>	PC-relative branch if registers equal
	Branch if not equal	<code>bne x5, x6, 100</code>	<code>if (x5 != x6) go to PC+100</code>	PC-relative branch if registers not equal
	Branch if less than	<code>blt x5, x6, 100</code>	<code>if (x5 &lt; x6) go to PC+100</code>	PC-relative branch if registers less
	Branch if greater or equal	<code>bge x5, x6, 100</code>	<code>if (x5 &gt;= x6) go to PC+100</code>	PC-relative branch if registers greater or equal
	Branch if less, unsigned	<code>bltu x5, x6, 100</code>	<code>if (x5 &lt; x6) go to PC+100</code>	PC-relative branch if registers less, unsigned
	Branch if greater or equal, unsigned	<code>bgeu x5, x6, 100</code>	<code>if (x5 &gt;= x6) go to PC+100</code>	PC-relative branch if registers greater or equal, unsigned
	Jump and link	<code>jal x1, 100</code>	<code>x1 = PC+4; go to PC+100</code>	PC-relative procedure call
Unconditional branch	Jump and link register	<code>jalr x1, 100(x5)</code>	<code>x1 = PC+4; go to x5+100</code>	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

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.

### EXAMPLE

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
```

### ANSWER

**EXAMPLE****ANSWER****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?

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.)

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.

**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.

### Compiling a C Assignment Using Registers

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

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

### EXAMPLE

## ANSWER

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?

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:

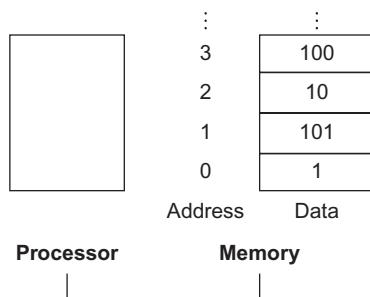
```
add x5, x20, x21 // register x5 contains g + h
add x6, x22, x23 // register x6 contains i + j
sub x19, x5, x6 // f gets x5 - x6, which is (g + h) - (i + j)
```

## 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.

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

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];
```

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*.

### EXAMPLE

### ANSWER

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

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

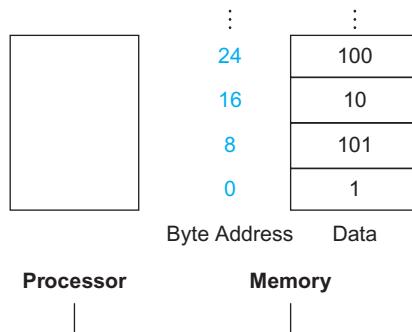
### Hardware/ Software Interface

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  $8 \times 8$ , 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

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];
```

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 ( $8 \times 12$ ) 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](#).

### EXAMPLE

### ANSWER

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.

### Hardware/ Software Interface

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  $x_{22}$ , we could use the code

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

assuming that  $x_3 + \text{AddrConstant4}$  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  $x_{22}$ , 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.

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  $x_0$  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**.



COMMON CASE FAST

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.

### Check Yourself



**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.)

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,

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

represents

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

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  $1011_{\text{two}}$ , (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  $2^{64} - 1$  ( $18,446,774,073,709,551,615_{\text{ten}}$ ):

00000000	00000000	00000000	00000000	00000000	00000000	00000000	$00000000_{\text{two}} = 0_{\text{ten}}$
00000000	00000000	00000000	00000000	00000000	00000000	00000001 <sub>two</sub>	$= 1_{\text{ten}}$
00000000	00000000	00000000	00000000	00000000	00000000	00000010 <sub>two</sub>	$= 2_{\text{ten}}$
...	...						
11111111	11111111	11111111	11111111	11111111	11111111	11111101 <sub>two</sub>	$= 18,446,774,073,709,551,613_{\text{ten}}$
11111111	11111111	11111111	11111111	11111111	11111111	11111110 <sub>two</sub>	$= 18,446,744,073,709,551,614_{\text{ten}}$
11111111	11111111	11111111	11111111	11111111	11111111	11111111 <sub>two</sub>	$= 18,446,744,073,709,551,615_{\text{ten}}$

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  $i$ th bit of  $x$ ):

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

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

---

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.

## Hardware/ Software Interface

---

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 00000000two = 0ten
00000000 00000000 00000000 00000000 00000000 00000001two = 1ten
00000000 00000000 00000000 00000000 00000000 00000010two = 2ten
...
...
01111111 11111111 11111111 11111111 11111111 11111111 11111101two = 9,223,372,036,854,775,805ten
01111111 11111111 11111111 11111111 11111111 11111111 11111110two = 9,223,372,036,854,775,806ten
01111111 11111111 11111111 11111111 11111111 11111111 11111111two = 9,223,372,036,854,775,807ten
10000000 00000000 00000000 00000000 00000000 00000000 00000000two = -9,223,372,036,854,775,808ten
10000000 00000000 00000000 00000000 00000000 00000000 00000000two = -9,223,372,036,854,775,807ten
10000000 00000000 00000000 00000000 00000000 00000000 00000010two = -9,223,372,036,854,775,806ten
...
...
11111111 11111111 11111111 11111111 11111111 11111111 11111101two = -3ten
11111111 11111111 11111111 11111111 11111111 11111111 11111110two = -2ten
11111111 11111111 11111111 11111111 11111111 11111111 11111111two = -1ten
```

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

Two's complement does have one negative number that has no corresponding positive number: -9,223,372,036,854,775,808<sub>ten</sub>. 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:

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

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.

### Binary to Decimal Conversion

### EXAMPLE

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

11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111100<sub>two</sub>

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

### ANSWER

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

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

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

## Hardware/ Software Interface

Signed versus unsigned applies to loads as well as to arithmetic. The *function* of a signed load is to copy the sign repeatedly to fill the rest of the register—called *sign extension*—but its *purpose* is to place a correct representation of the number within that register. Unsigned loads simply fill with 0s to the left of the data, since the number represented by the bit pattern is unsigned.

When loading a 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  $111 \dots 111_{\text{two}}$ , which represents  $-1$ . Since  $x + \bar{x} = -1$ , therefore  $x + \bar{x} + 1 = 0$  or  $\bar{x} + 1 = -x$ . (We use the notation  $\bar{x}$  to mean invert every bit in  $x$  from 0 to 1 and vice versa.)

---

### EXAMPLE

#### Negation Shortcut

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

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

### ANSWER

Negating this number by inverting the bits and adding one,

$$\begin{array}{r}
 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 111111101_{\text{two}} \\
 + \hspace{10em} 1_{\text{two}} \\
 \hline
 = 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_{\text{two}} \\
 = -2_{\text{ten}}
 \end{array}$$

Going the other direction,

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

is first inverted and then incremented:

$$\begin{array}{r}
 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001_{\text{two}} \\
 + \hspace{10em} 1_{\text{two}} \\
 \hline
 = 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000010_{\text{two}} \\
 = 2_{\text{ten}}
 \end{array}$$

Our next shortcut tells us how to convert a binary number represented in  $n$  bits to a number represented with more than  $n$  bits. 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*.

### Sign Extension Shortcut

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

The 16-bit binary version of the number 2 is

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

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:

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

### EXAMPLE

### ANSWER

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

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

becomes

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

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

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

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

## Summary

The main point of this section is that we need to represent both positive and negative integers within a computer, 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?

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

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

**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  $\bar{x}$ . This relation helps explain its name since the complement of  $x$  is  $2^n - x - 1$ . It was also an attempt to be a better solution than sign and magnitude, and several early scientific computers did use the notation. This representation is similar to two's complement except that it also has two 0s: 00 ... 00<sub>two</sub> is positive 0 and 11 ... 11<sub>two</sub> is negative 0. The most negative number, 10 ... 00<sub>two</sub>, 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<sub>two</sub> and the most positive value by 11 ... 11<sub>two</sub>, with 0 typically having the value 10 ... 00<sub>two</sub>. This representation is called a **biased notation**, since it biases 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.

**one's complement** A notation that represents the most negative value by 10 ... 000<sub>two</sub> and the most positive value by 01 ... 11<sub>two</sub>, leaving an equal number of negatives and positives but ending up with two zeros, one positive (00 ... 00<sub>two</sub>) and one negative (11 ... 11<sub>two</sub>). 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<sub>two</sub> and the most positive value by 11 ... 11<sub>two</sub>, with 0 typically having the value 10 ... 00<sub>two</sub>, thereby biasing the number such that the number plus the bias has a non-negative representation.

### 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.

The decimal representation is

0	21	20	0	9	51
---	----	----	---	---	----

### EXAMPLE

### ANSWER

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  $x_{21}$ ), and the third field gives the other source operand for the addition (20 for  $x_{20}$ ). The fifth field contains the number of the register that is to receive the sum (9 for  $x_9$ ). Thus, this instruction adds register  $x_{20}$  to register  $x_{21}$  and places the sum in register  $x_9$ .

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.

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

**hexadecimal** Numbers in base 16.

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.

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

It would appear that you would now be reading and writing long, 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 <sub>hex</sub>	0000 <sub>two</sub>	4 <sub>hex</sub>	0100 <sub>two</sub>	8 <sub>hex</sub>	1000 <sub>two</sub>	c <sub>hex</sub>	1100 <sub>two</sub>
1 <sub>hex</sub>	0001 <sub>two</sub>	5 <sub>hex</sub>	0101 <sub>two</sub>	9 <sub>hex</sub>	1001 <sub>two</sub>	d <sub>hex</sub>	1101 <sub>two</sub>
2 <sub>hex</sub>	0010 <sub>two</sub>	6 <sub>hex</sub>	0110 <sub>two</sub>	a <sub>hex</sub>	1010 <sub>two</sub>	e <sub>hex</sub>	1110 <sub>two</sub>
3 <sub>hex</sub>	0011 <sub>two</sub>	7 <sub>hex</sub>	0111 <sub>two</sub>	b <sub>hex</sub>	1011 <sub>two</sub>	f <sub>hex</sub>	1111 <sub>two</sub>

**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 `0xnnnn` for hexadecimal numbers.

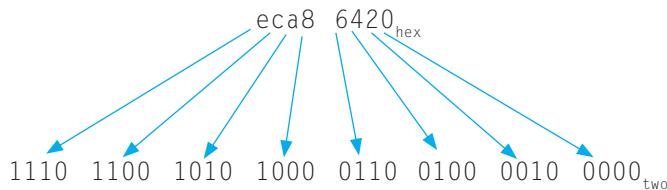
### Binary to Hexadecimal and Back

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

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

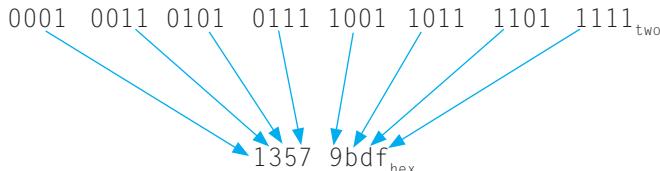
Using Figure 2.4, the answer is just a table lookup one way:

### EXAMPLE



### ANSWER

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 `add i`, 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, `s d`, 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-i-ate	rs2	rs1	funct3	immed-i-ate	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.

### Translating RISC-V Assembly Language into Machine Language

### EXAMPLE

We can now take an example all the way from what the programmer writes to what the computer executes. If  $x10$  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]
```

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

### ANSWER

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

immediate	rs1	funct3	rd	opcode
240	10	3	9	3
funct7	rs2	rs1	funct3	rd
0	9	21	0	9
immediate	rs1	funct3	rd	opcode
1	9	0	9	19
immediate[11:5]	rs2	rs1	funct3	immediate[4:0]
7	9	10	3	16
				35

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

immediate	rs1	funct3	rd	opcode
000011110000	01010	011	01001	0000011
funct7	rs2	rs1	funct3	rd
0000000	01001	10101	000	01001
immediate	rs1	funct3	rd	opcode
000000000001	01001	000	01001	0010011
immediate[11:5]	rs2	rs1	funct3	immediate[4:0]
0000111	01001	01010	011	10000
				0100011

**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.

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.

## Hardware/ Software Interface

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	0110011	addi x1, x2, 1000
ld (load doubleword)	001111101000	00010	011	00001	0000011	0110011	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.

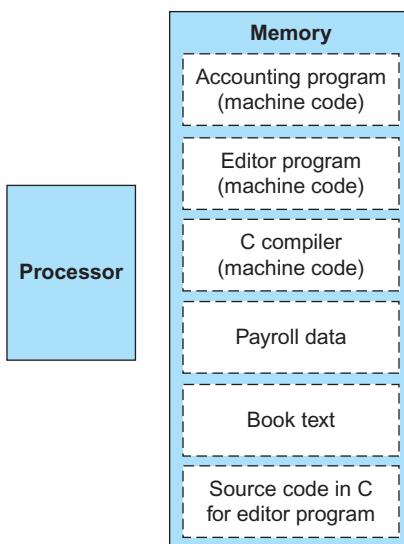
## The BIG Picture

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.

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

### Check Yourself

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

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.

*“Contrariwise,” continued Tweedledee, “if it was so, it might be; and if it were so, it would be; but as it isn’t, it ain’t. That’s logic.”*

Lewis Carroll,  
*Alice’s Adventures in Wonderland*, 1865

Logical operations	C operators	Java operators	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, xorri
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<sub>hex</sub>).

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  $x_{19}$  contained

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

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

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

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* (`srl`). The following instruction performs the operation above, if the original value was in register  $x_{19}$  and the result should go in register  $x_{11}$ :

```
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 `srl`, 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`.

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  $x_{11}$  contains

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

00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000<sub>two</sub>

and register x10 contains

00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000<sub>two</sub>

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<sub>two</sub>

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 00111101 11000000<sub>two</sub>

The final logical operation is a contrarian. **NOT** takes one operand and places a 1 in the result if one operand bit is a 0, and vice versa. Using our prior notation, it calculates  $\bar{x}$ .

In keeping with the three-operand format, the designers of 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 00110001 11000000<sub>two</sub>

**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

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

Which operations can isolate a field in a doubleword?

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

## 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;
```

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:

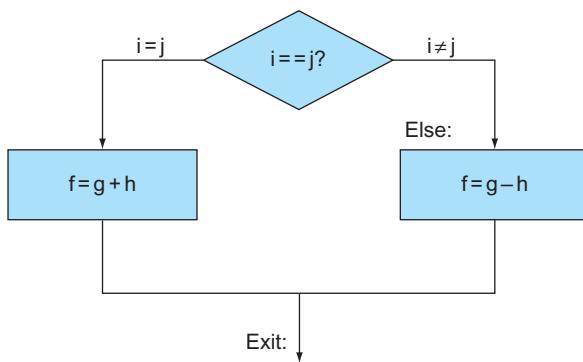
```
beq x0, x0, Exit // if 0 == 0, go to Exit
```

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](#)).

**ANSWER**



**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.)

---

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.

---

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

## Hardware/ Software Interface

**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).

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  $x_{20} \geq x_{11}$  or if  $x_{20}$  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.

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.

## Hardware/ Software Interface

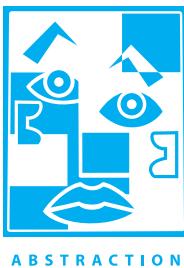
- I. C has many statements for decisions and loops, while RISC-V has few. Which of the following does or does not explain this imbalance? Why?
  1. More decision statements make code easier to read and understand.
  2. Fewer decision statements simplify the task of the underlying layer that is responsible 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.

## Check Yourself

## 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.

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
```

**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).

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  $x_1$ , is called the **return address**. The return address is needed because the same procedure could be called from several parts of the program.

To support 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  $x_1$ —which is just what we want. Thus, the calling program, or **caller**, puts the parameter values in  $x_{10}$ – $x_{17}$  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)`.

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  $x_1$ ) to link to the byte address of the following instruction to set up the procedure return.

**Elaboration:** The jump-and-link instruction can also be used to perform an unconditional branch within a procedure by using  $x_0$  as the destination register. Since  $x_0$  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  $x_2$ , 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**.

**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  $x_1$ .

**caller** The program that instigates a procedure and provides the necessary parameter values.

**callee** A procedure that executes a series of stored instructions based on parameters provided by the caller and then returns control to the caller.

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

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

**stack pointer** A value denoting the most recently allocated address in a stack that shows where registers should be spilled or where old register values can be found. In RISC-V, it is register `sp`, or  $x_2$ .

**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:

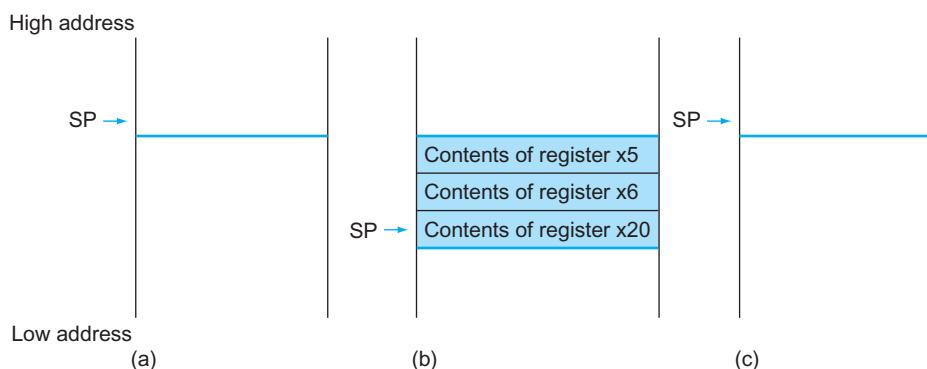
```
ld x20, 0(sp)    // restore register x20 for caller
ld x6, 8(sp)     // restore register x6 for caller
ld x5, 16(sp)    // restore register x5 for caller
addi sp, sp, 24  // adjust stack to delete 3 items
```

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  $x_5$  and  $x_6$  to be preserved across a procedure call, we can drop two stores and two loads from the code. We still must save and restore  $x_{20}$ , 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  $x_{10}$  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  $x_{10}$ . Since A hasn't finished its task yet, there is a conflict over the use of register  $x_{10}$ . Similarly, there is a conflict over the return address in register  $x_1$ , 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 ( $x_{10}-x_{17}$ ) or temporary registers ( $x_5-x_7$  and  $x_{28}-x_{31}$ ) that are needed after the call. The callee pushes the return address register  $x_1$  and any saved registers ( $x_8-x_9$  and  $x_{18}-x_{27}$ ) 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  $x_{10}$ . The compiled program starts with the label of the procedure and then saves two registers on the stack, the return address and  $x_{10}$ :

```
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  $n \geq 1$ .

```
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

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

A C variable is generally a location in storage, and its interpretation depends both on its *type* and *storage class*. 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.

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.

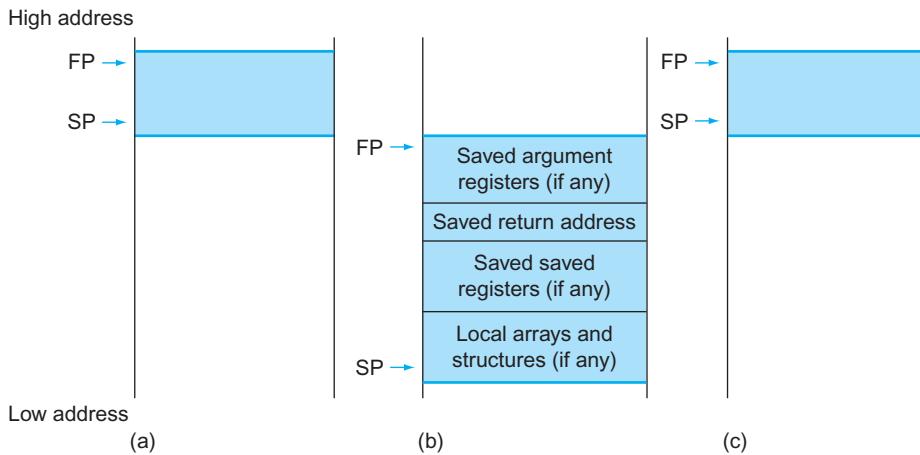
**procedure frame** Also called **activation record**. The segment of the stack containing a procedure's saved registers and local variables.

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

## Allocating Space for New Data on the Stack

The final complexity is that the stack is also used to store variables that are local to the procedure but do not fit in registers, such as local arrays or structures. The segment of the stack containing a procedure's saved registers and local variables is called a **procedure frame** or **activation record**. Figure 2.12 shows the state of the stack before, during, and after the procedure call.

Some 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



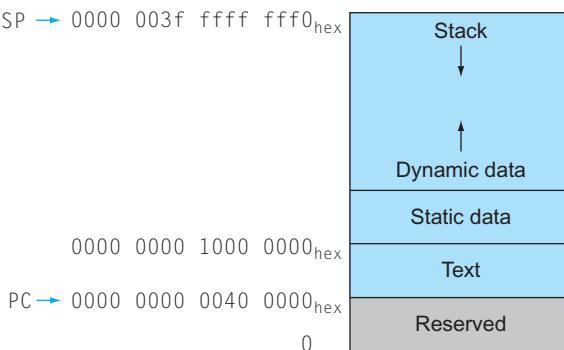
**FIGURE 2.12 Illustration of the stack allocation (a) before, (b) during, and (c) after the procedure call.** The frame pointer (`fp` or `x8`) points to the first doubleword of the frame, often a saved argument register, and the stack pointer (`sp`) points to the top of the stack. The stack is adjusted 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<sub>hex</sub> and grows down toward the data segment. At the other end, the program code (“text”) starts at 0000 0000 0040 0000<sub>hex</sub>. The static data starts immediately after the end of the text segment; in this example, we assume that address is 0000 0000 1000 0000<sub>hex</sub>. 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.



COMMON CASE FAST

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.

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										
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

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

### EXAMPLE

One billion is 1,000,000,000, so it would take 10 ASCII digits, each 8 bits long. Thus the storage expansion would be  $(10 \times 8)/32$  or 2.5. Beyond the expansion in storage, the hardware to add, subtract, multiply, and divide such decimal numbers is difficult and would consume more energy. Such difficulties explain why computing professionals are raised to believe that binary is natural and that the occasional decimal computer is bizarre.

### ANSWER

A series of instructions can extract a byte from a 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<sub>hex</sub>, and Cyrillic at 0400<sub>hex</sub>. 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](http://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* (1h) 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.

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  $1,000,000,000_{\text{ten}}$  takes the most memory space?

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

### Check Yourself

## 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

## ANSWER

### 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
```

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:

```
00000000 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  $1100111_{\text{two}}$  and bne's funct3 code is  $001_{\text{two}}$ .

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

**PC-relative addressing**

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

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**.

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  $\pm 1$  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.

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.

## Hardware/ Software Interface

### EXAMPLE

#### Branching Far Away

Given a branch on register x10 being equal to zero,

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

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:
```

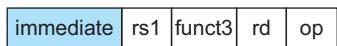
### ANSWER

## 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.

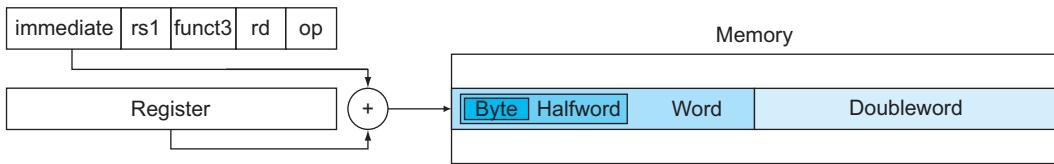
## 1. Immediate addressing



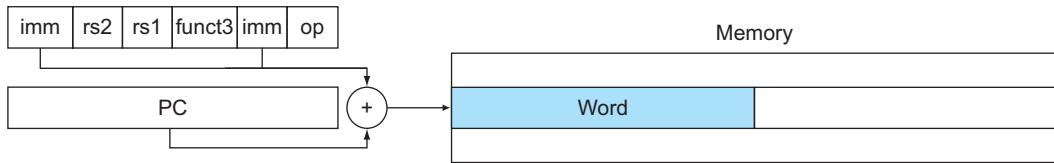
## 2. Register addressing



## 3. Base addressing



## 4. PC-relative addressing



**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
	lrd	0110011	011	0001000
	scd	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.
	srlti	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, srlti, srai) use the funct6 field.

**EXAMPLE****ANSWER****Decoding Machine Code**

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

00578833<sub>hex</sub>

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 size)	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Comments
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immed[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	immed[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immed[31:12]				rd	opcode	Upper immediate format

**FIGURE 2.19** RISC-V instruction formats.

- 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

### Check Yourself

## 2.11

### Parallelism and Instructions: Synchronization

Parallel execution is easier when tasks are independent, but often they need to cooperate. Cooperation usually means some tasks are writing new values that others must read. To know when a task is finished writing so that it is safe for another to read, the tasks need to synchronize. If they don't synchronize, there is a danger of a **data race**, where the results of the program can change depending on how events happen to occur.

For example, recall the analogy of the eight reporters writing a story on 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.



PARALLELISM

**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.

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* (`l r . d`) and a special store called a *store-conditional doubleword* (`s c . 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, `s c . 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**

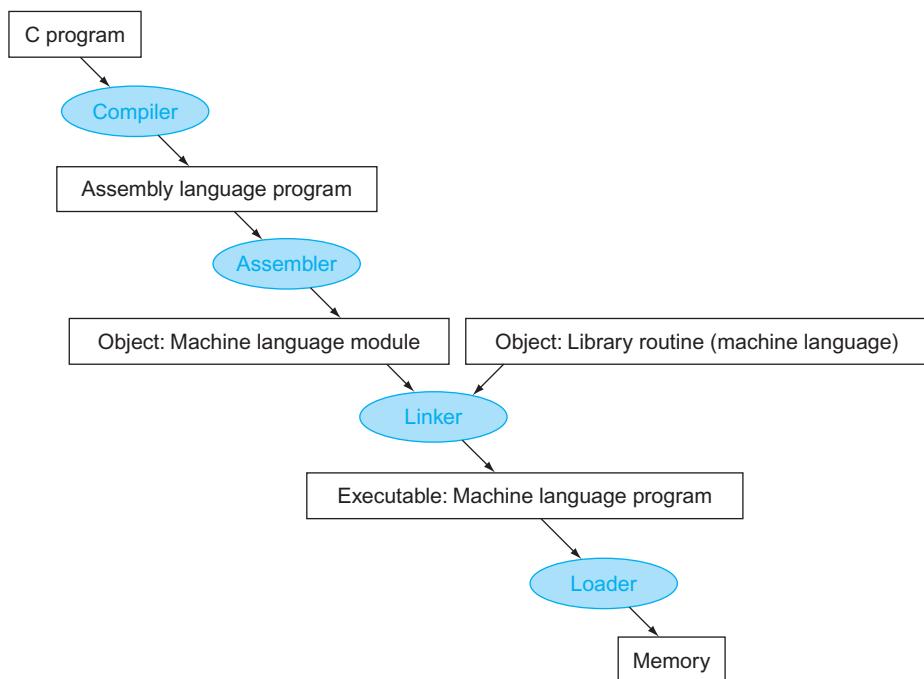
When do you use primitives like load-reserved and store-conditional?

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 `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**.

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

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

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.

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.

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

- 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.

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.

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

**executable file** A functional program in the format of an object file that contains no unresolved references. It can contain symbol tables and debugging information. A “stripped executable” does not contain that information. Relocation information may be included for the loader.

**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.

<b>Object file header</b>			
	Name	Procedure A	
	Text size	100 <sub>hex</sub>	
	Data size	20 <sub>hex</sub>	
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 <sub>hex</sub>	
	Data size	30 <sub>hex</sub>	
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

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<sub>hex</sub> and the data segment at 0000 0000 1000 0000<sub>hex</sub>. 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<sub>hex</sub> bytes and its data is 20<sub>hex</sub> bytes, so the starting address for procedure B text is 40 0100<sub>hex</sub>, and its data starts at 1000 0020<sub>hex</sub>.

## ANSWER

<b>Executable file header</b>		
	Text size	300 <sub>hex</sub>
	Data size	50 <sub>hex</sub>
Text segment	Address	Instruction
	0000 0000 0040 0000 <sub>hex</sub>	<code>ld x10, 0(x3)</code>
	0000 0000 0040 0004 <sub>hex</sub>	<code>jal x1, 252<sub>ten</sub></code>
	...	...
	0000 0000 0040 0100 <sub>hex</sub>	<code>sd x11, 32(x3)</code>
	0000 0000 0040 0104 <sub>hex</sub>	<code>jal x1, -260<sub>ten</sub></code>
	...	...
Data segment	Address	
	0000 0000 1000 0000 <sub>hex</sub>	(X)
	...	...
	0000 0000 1000 0020 <sub>hex</sub>	(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<sub>hex</sub> to go to 40 0100<sub>hex</sub> (the address of procedure B), it must put (40 0100<sub>hex</sub> - 40 0004<sub>hex</sub>) or 252<sub>ten</sub> in its address field. Similarly, since 40 0000<sub>hex</sub> is the address of procedure A, the `jal` at 40 0104<sub>hex</sub> gets the negative number -260<sub>ten</sub> (40 0000<sub>hex</sub> - 40 0104<sub>hex</sub>) 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<sub>hex</sub>. To get the address 0000 0000 1000 0000<sub>hex</sub> (the address of doubleword X), we place 0<sub>ten</sub> in the address field of `ld` at address 40 0000<sub>hex</sub>. Similarly, we place 20<sub>hex</sub> in the address field of `sd` at address 40 0100<sub>hex</sub> to get the address 0000 0000 1000 0020<sub>hex</sub> (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<sub>ten</sub> in the address field of `sd` at address 40 0100<sub>hex</sub> to get the address 0000 0000 1000 0020<sub>hex</sub> (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

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

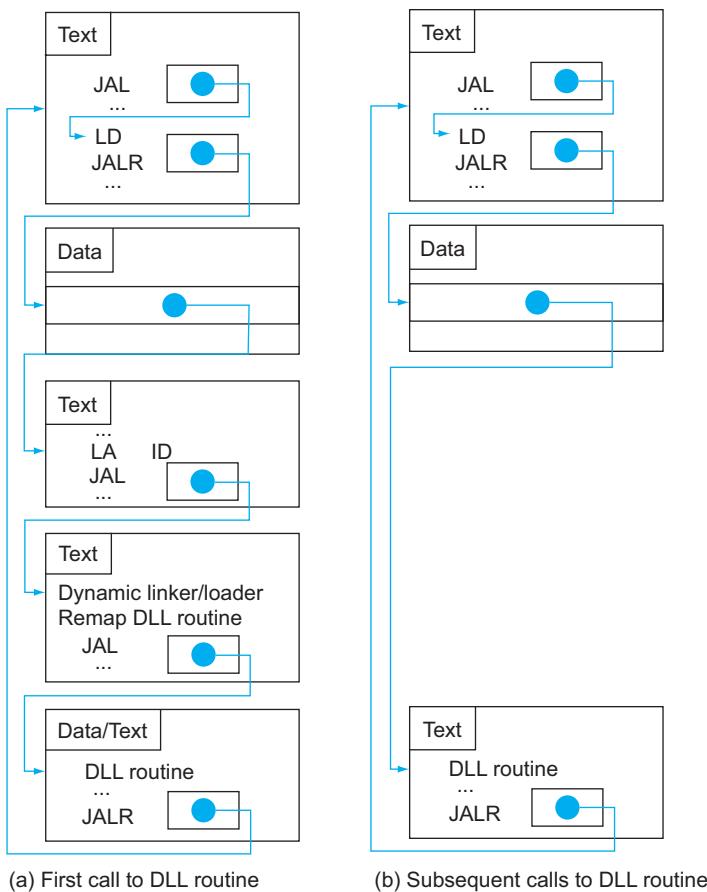
The first part of this section describes the traditional approach to linking libraries before the program is run. Although this static approach is the fastest way to call library routines, it has a few disadvantages:

- 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.

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.

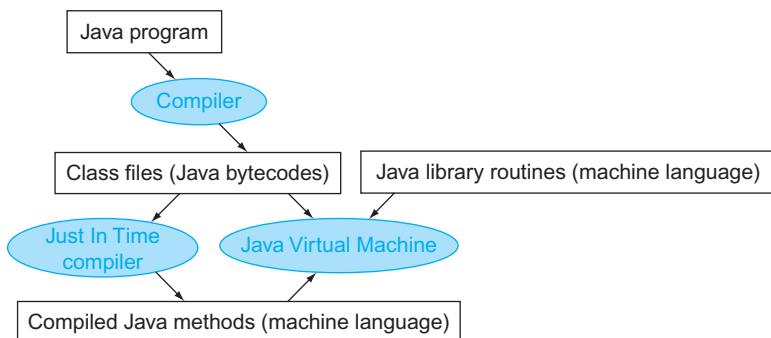
In summary, DLLs require additional space for the information needed for dynamic linking, but do not require that whole libraries be copied or linked. They pay a good deal of overhead the first time a routine is called, but only a single indirect branch thereafter. Note that the return from the library 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.

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.

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



**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.

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.

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

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

**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.

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

### Check Yourself

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  $i \geq n$ . This test takes just one instruction:

```
forltst: bge x19, x11, exit1 // go to exit1 if x19 \geq x1 (i\geq n)
```

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

```
j forltst // branch to test of outer loop
```

```
exit1:
```

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

```

li x19, 0          // i = 0
forltst:
    bge x19, x11, exit1 // go to exit1 if x19 \geq x1 (i\geq 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  $j \geq 0$ .

The second test exits if  $v[j] > v[j + 1]$  is *not* true, or exits if  $v[j] \leq v[j + 1]$ . First we create the address by multiplying *j* by 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  $v[j] \leq v[j + 1]$  to exit the loop:

```
ble      x6, x7, exit2      // go to exit2 if x6 \leq 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 sd x1, 32(sp) sd x22, 24(sp) sd x21, 16(sp) sd x20, 8(sp) sd x19, 0(sp)	# make room on stack for 5 registers # save return address on stack # save x22 on stack # save x21 on stack # save x20 on stack # save x19 on stack
Procedure body		
Move parameters	mv x21, x10 mv x22, x11	# copy parameter x10 into x21 # copy parameter x11 into x22
Outer loop	li x19, 0 for1tst:bge x19, x22, exit1	# i = 0 # go to exit1 if i >= n
Inner loop	addi x20, x19, -1 for2tst:blt x20, x0, exit2 slli x5, x20, 3 add x5, x21, x5 ld x6, 0(x5) ld x7, 8(x5) ble x6, x7, exit2	# j = i - 1 # go to exit2 if j < 0 # x5 = j * 8 # x5 = v + (j * 8) # x6 = v[j] # x7 = v[j + 1] # go to exit2 if x6 < x7
Pass parameters and call	mv x10, x21 mv x11, x20 jal x1, swap	# first swap parameter is v # second swap parameter is j # call swap
Inner loop	addi x20, x20, -1 j for2tst	j for2tst # go to for2tst
Outer loop	exit2: addi x19, x19, 1 j for1tst	# i += 1 # go to for1tst
Restoring registers		
	exit1: ld x19, 0(sp) ld x20, 8(sp) ld x21, 16(sp) ld x22, 24(sp) ld x1, 32(sp) addi sp, sp, 40	# restore x19 from stack # restore x20 from stack # restore x21 from stack # restore x22 from stack # restore return address from stack # 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. (See [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 ( $0.05 \times 2468$ , 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	mv x5, x10	// p = address of array[0]
loop1: slli x6, x5, 3	// x6 = i * 8	slli x6, x11, 3	// x6 = size * 8
add x7, x10, x6	// x7 = address of array[i]	add x7, x10, x6	// x7 = address of array[size]
sd x0, 0(x7)	// array[i] = 0	loop2: sd x0, 0(x5)	// Memory[p] = 0
addi x5, x5, 1	// i = i + 1	addi x5, x5, 8	// p = p + 8
blt x5, x11, loop1	// if (i < size) go to loop1	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



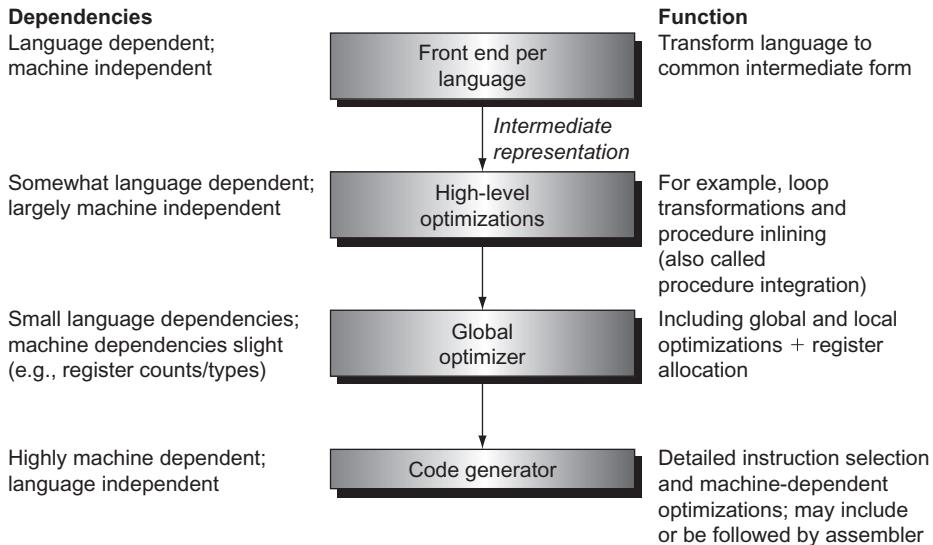
## Advanced Material: Compiling C and Interpreting Java

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 object-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;
```

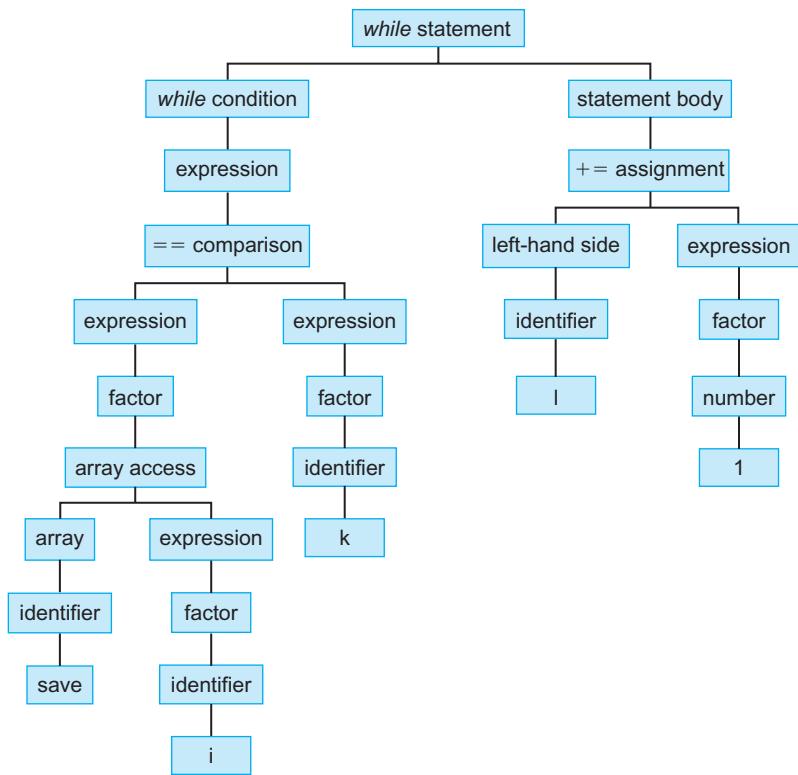
## The Front End

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 descendants 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.

<pre>// 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)</pre>	<pre>// 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)</pre>
---	--

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:

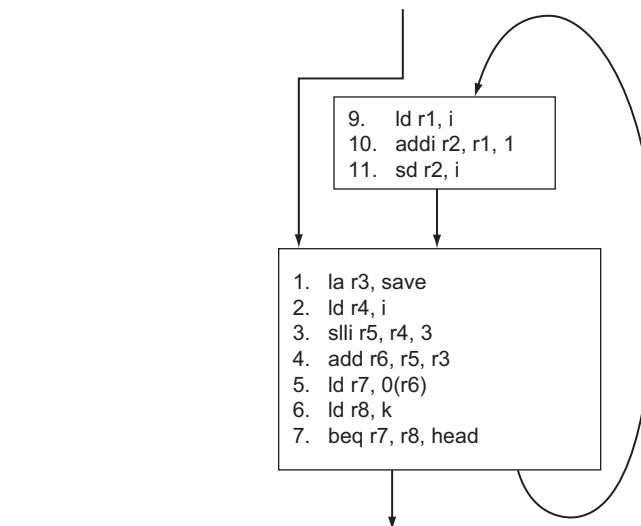
- Consider the case of an opportunity for common subexpression elimination, say, of an IR statement like ADD Rx, R20, R50. To determine whether two such statements compute the same value, we must determine whether the values of R20 and R50 are identical in the two statements. In practice, this means 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!

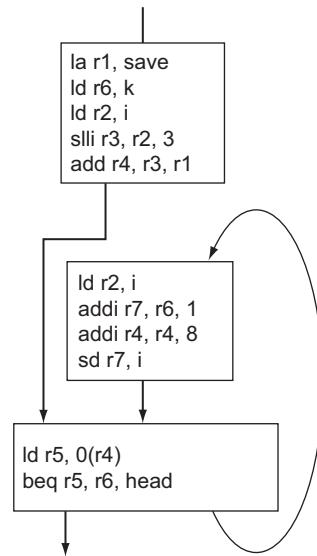


**FIGURE e2.15.4 A control flow graph for the *while* loop example.** Each node represents a basic block, which terminates with a branch or by sequential fall-through into another basic block that is also the target of a branch. The IR statements have been numbered for ease in referring to them. The 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.

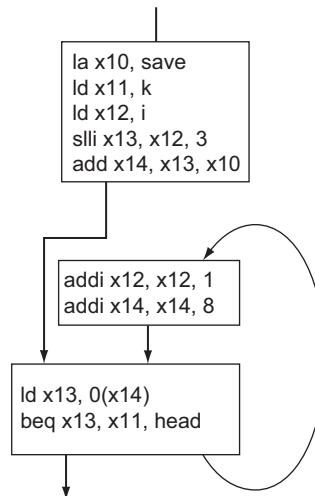
The set of basic blocks found by this technique has a special property: if the designated variable is allocated to a register in all these basic blocks, then there is no need for loading and storing the variable.

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.

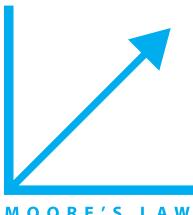
During code generation, the final stages of machine-dependent optimization are also performed. These include some constant folding optimizations, as well as localized instruction scheduling (see [Chapter 4](#)).

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

**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/address s	iload_/aload_{0,1,2,3 }	8	ld	TOS=Frame[{0,1,2,3}]
	store local integer/address s	istore I8/astore I 8	16	sd	Frame[I8]=TOS; pop
	load integer/address from array	iaload/aaloa d	8	ld	NOS=*NOS[TOS]; pop
	store integer/address into array	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 immediate	bipush I8, sipush I1 6	16, 24	addi	push; TOS=I8 or I16
Logical	load immediate	iconst_{-1,0,1,2,3,4,5 }	8	addi	push; TOS={-1,0,1,2,3,4,5}
	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
Conditional branch	shift right	iushr	8	srl	NOS=NOS >> TOS; pop
	branch on equal	if_icompeq I16	24	beq	if TOS == NOS , go to I16; pop2
	branch on not equal	if_icompne I16	24	bne	if TOS != NOS , go to I16; pop2
Unconditional jump	compare	if_icomp{lt,le,gt,ge} I16	24	blt/bge	if TOS {<,<=,>,>=} NOS, go to I16; pop2
	jump	goto I16	24	jal	go to I16
	return	ret, ireturn	8	jalr	
Stack management	jump to subroutine	jsr I16	24	jal	go to I16; push; TOS=PC+3
	remove from stack	pop, pop2	8		pop, pop2
	duplicate on stack	dup	8		push; TOS=NOS
Safety check	swap top 2 positions on stack	swap	8		T=NOS; NOS=TOS; TOS=T
	check for null reference	fnull I16, ifnonnull I16	24		if TOS {==,!=} null, go to I16
	get length of array	arraylength	8		push; TOS = length of array
Invocation	check if object a type	instanceof I16	24		TOS = 1 if TOS matches type of Const[I16]; TOS = 0 otherwise
	invoke method	invokevirtual I16	24		Invoke method in Const[I16] , dispatching on type
	Allocation	new I16	24		Allocate object type Const[I16] on heap
Allocation	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

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?

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_icmpne, 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)
```

**EXAMPLE**

**ANSWER**

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, IndexOutOfBoundsException //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 keyword 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]) > 0; 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:	<pre>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 &lt; 0, goto Error     bge x11, x5, IndexOutOfBounds     # If k &gt;= length, goto Error     addi x6, x11, 1     # Temp reg x6 = k+1     blt x6, x0, IndexOutOfBounds     # If k+1 &lt; 0, goto Error     bge x6, x5, IndexOutOfBounds     # If k+1 &gt;= length, goto Error</pre>
Method body	
	<pre>slli x11, x11, 3     # reg X11 = k * 8     add x11, x11, x10     # reg X11 = v + (k * 8)      ld x12, 0(x11)     ld x13, 8(x11)     # reg x12 = v[k]     # reg x13 = v[k+1]      ld x13, 0(x11)     ld x12, 8(x11)     # v[k] = reg x13     # v[k+1] = reg x12</pre>
Method return	
	<pre>jalr x0,0(x1)           # return to calling routine</pre>

**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	for1tst: li x19, 0 bge x19, x22, exit1	# i = 0 # If i >= length, go to exit1
Inner loop head	for2tst: addi x20, x19, -1 blt x20, x0, exit1	# j = i - 1 # If j < 0, goto exit2
Test if j too big	bge x20, x22, IndexOutOfBoundsException	# If j >= length, goto error
Get v[j]	slli x5, x20, 3 add x5, x21, x5 ld x6, 0(x5)	# x5 = j * 8 # x5 = v + (j * 8) # x6 = v[j]
Test if j+1 < 0 or too big	addi x7, x20, 1 blt x7, x0, IndexOutOfBoundsException bge x7, x22, IndexOutOfBoundsException	# x7 = j + 1 # If j + 1 < 0, goto Error # 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	for2tst: mv x10, x6 mv x11, x7	# 1st parameter is v[j] # 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 mv x11, x20 jal x1, swap	# 1st parameter is v # 2nd parameter is j # Invoke swap routine (Figure 2.34)
Inner loop end	addi x20, x20, -1 j for2tst	# j -= 1 # Go to for2tst
Outer loop end	exit2: addi x19, x19, 1 j for1tst	# i += 1 # 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, IndexOutOfBoundsException // 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  $x10 \leq 0$ :

```
ble x10, x0, exit2  // go to exit2 if v[j] \leq 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, ..., ( $\text{data.length} - 1$ ) and since  $j$  takes on the values  $i - 1, i - 2, \dots, 2, 1, 0$ , there is no need to test if  $j \leq \text{data.length}$  since the largest value  $j$  can be is  $\text{data.length} - 2$ .
3. Following the same reasoning, there is no need to test whether  $j + 1 \leq \text{data.length}$  since the largest value of  $j+1$  is  $\text{data.length} - 1$ .

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.

The rest of  [Section 2.15](#) can be found online.

**object-oriented language** A

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

## 2.16 Real Stuff: 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

**Register-register**

	31	25 24	20 19	15 14	12 11	7 6	0
RISC-V	funct7(7)	rs2(5)	rs1(5)	funct3(3)	rd(5)	opcode(7)	
MIPS	Op(6)	Rs1(5)	Rs2(5)	Rd(5)	Const(5)	Opx(6)	

**Load**

	31	20 19	15 14	12 11	7 6	0
RISC-V	immediate(12)	rs1(5)	funct3(3)	rd(5)	opcode(7)	
MIPS	Op(6)	Rs1(5)	Rs2(5)		Const(16)	

**Store**

	31	25 24	20 19	15 14	12 11	7 6	0
RISC-V	immediate(7)	rs2(5)	rs1(5)	funct3(3)	immediate(5)	opcode(7)	
MIPS	Op(6)	Rs1(5)	Rs2(5)		Const(16)		

**Branch**

	31	25 24	20 19	15 14	12 11	7 6	0
RISC-V	immediate(7)	rs2(5)	rs1(5)	funct3(3)	immediate(5)	opcode(7)	
MIPS	Op(6)	Rs1(5)	Opx/Rs2(5)		Const(16)		

**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.
- **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

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

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.

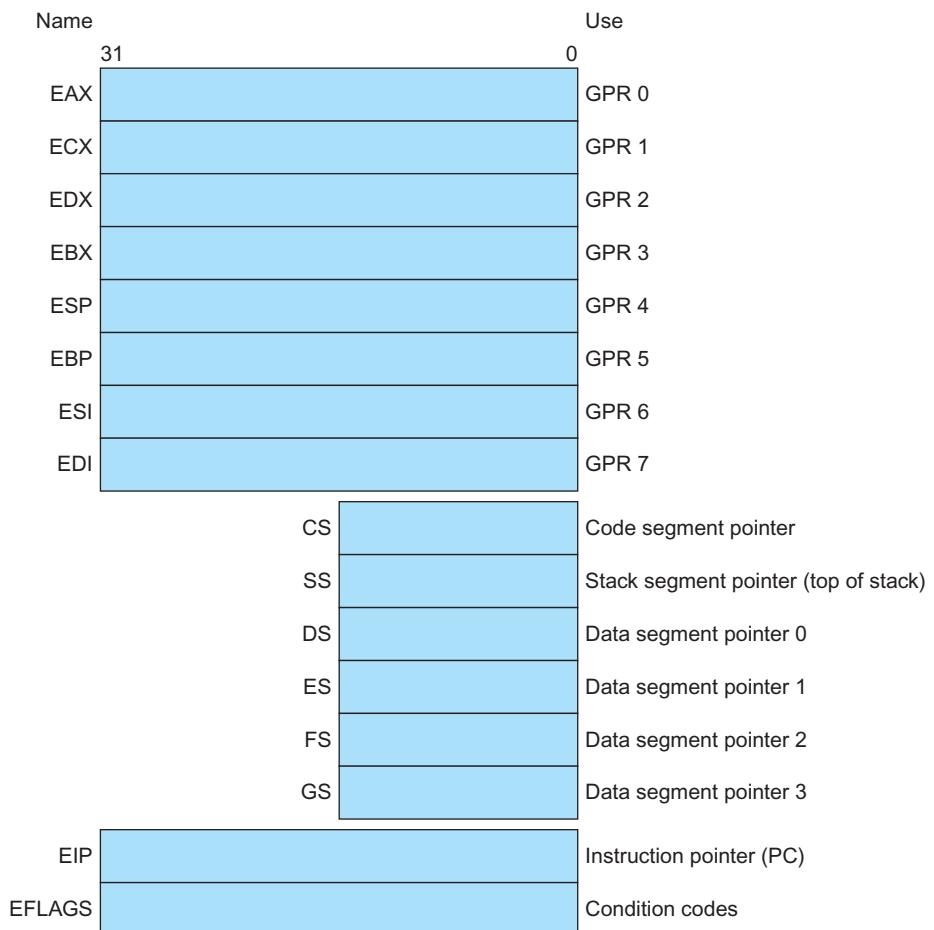
Rather than show the entire 16-bit, 32-bit, and 64-bit instruction set, in this section we concentrate on the 32-bit subset that originated with the 80386. We start our explanation with the registers and addressing modes, move on to the integer operations, and conclude with an examination of instruction 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. Immediate 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	<code>ld x10, 0(x11)</code>
Based mode with 8- or 32-bit displacement	Address is contents of base register plus displacement.	Not ESP	<code>ld x10, 40(x11)</code>
Base plus scaled index	The address is Base + ( $2^{\text{Scale}} \times \text{Index}$ ) where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	<code>slli x12, x12, 3</code> <code>add x11, x11, x12</code> <code>ld x10, 0(x11)</code>
Base plus scaled index with 8- or 32-bit displacement	The address is Base + ( $2^{\text{Scale}} \times \text{Index}$ ) + Displacement where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	<code>slli x12, x12, 3</code> <code>add x11, x11, x12</code> <code>ld x10, 40(x11)</code>

**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
<b>Control</b>	<b>Conditional and unconditional branches</b>
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
<b>Data transfer</b>	<b>Move data between registers or between register and memory</b>
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
<b>Arithmetic, logical</b>	<b>Arithmetic and logical operations using the data registers and memory</b>
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
<b>String</b>	<b>Move between string operands; length given by a repeat prefix</b>
movs	Copies from string source to destination by incrementing ESI and EDI; may be repeated
lod\$	Loads a byte, word, or doubleword of a string into the EAX register

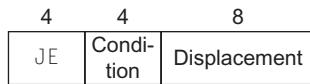
**FIGURE 2.34 Some typical operations on the x86.** Many operations use register-memory format, where either the source or the destination may be memory and the other may be a register or immediate operand.

## 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.”

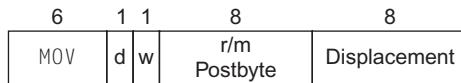
a. JE EIP + displacement



b. CALL



c. MOV EBX, [EDI + 45]



d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



**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 immediate	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 (`l.r.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 (`l.r.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.*

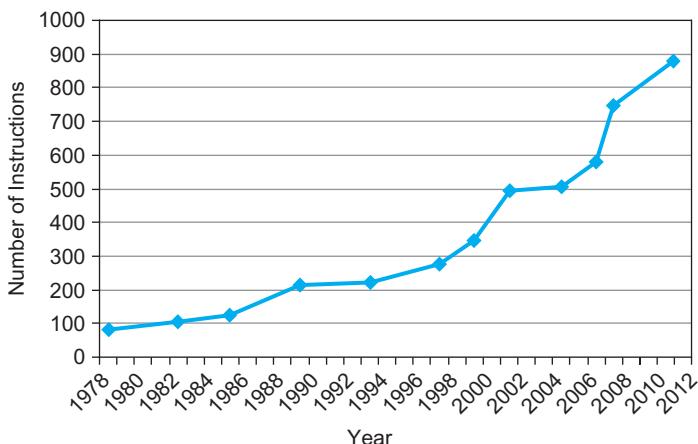
At one time compilers for programming languages produced naïve instruction sequences; the increasing sophistication of compilers means the gap between compiled code and code produced by hand is closing fast. In fact, to compete with current compilers, the assembly language programmer needs to understand 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

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

*Less is more.*

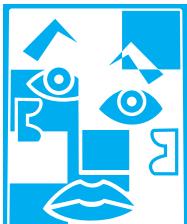
Robert Browning,  
Andrea del Sarto, 1855

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



ABSTRACTION

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**.

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	Floating point
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.

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.

**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

## 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.

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.

**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.

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 instruction 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  $128_{\text{ten}}$ .

**2.11.1** [5] <§2.4> For the instruction  $\text{add } x30, x5, x6$ , what is the range(s) of values for  $x6$  that would result in overflow?

**2.11.2** [5] <§2.4> For the instruction  $\text{sub } x30, x5, x6$ , 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<sub>two</sub>

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 = 0x00000000AAAAAAA, 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  $x_7$  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  $x_7$  for the following sequence of instructions?

```
srl x7, x5, 3  
andi x7, x7, 0xFFE
```

**2.18** [10] <§2.6> Find the shortest sequence of RISC-V instructions that extracts bits 16 down to 11 from register  $x_5$  and uses the value of this field to replace bits 31 down to 26 in register  $x_6$  without changing the other bits of registers  $x_5$  or  $x_6$ . (Be sure to test your code using  $x_5 = 0$  and  $x_6 = 0xfffffffffffff$ . 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  $x_6 = A$ , and  $x_{17}$  is the base address of C.

```
A = C[0] << 4;
```

**2.21** [5] <§2.7> Assume  $x_5$  holds the value 0x00000000001010000. What is the value of  $x_6$  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 0x7fffffff, 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  $x_{10}$ - $x_{14}$ ,  $x_8$ ,  $x_1$ , 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  $x_{10}$  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  $x_{10}$ . If a non-digit character appears anywhere in the string, your program should stop with the value  $-1$  in register  $x_{10}$ . For example, if register  $x_{10}$  points to a sequence of three bytes  $50_{ten}$ ,  $52_{ten}$ ,  $0_{ten}$  (the null-terminated string “24”), then when the program stops, register  $x_{10}$  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  $x_7$  contains the address  $0x10000000$  and the data at address is  $0x1122334455667788$ .

**2.35.1** [5] <§2.3, 2.9> What value is stored in  $0x10000008$  on a big-endian machine?

**2.35.2** [5] <§2.3, 2.9> What value is stored in  $0x10000008$  on a little-endian machine?

**2.36** [5] <§2.10> Write the RISC-V assembly code that creates the 64-bit constant  $0x1122334455667788_{two}$  and stores that value to register  $x_{10}$ .

**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.

§2.2, page 66: RISC-V, C, Java.

§2.3, page 73: 2) Very slow.

§2.4, page 80: 2)  $-8_{\text{ten}}$

§2.5, page 89: 3) `sub x11, x10, x9`

§2.6, page 92: Both. AND with a mask pattern of 1s will leave 0s everywhere but the desired field. Shifting left by the correct amount removes the bits from the left of the field. Shifting right by the appropriate amount puts the field into the right-most 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)  $\pm 4\text{K}$ . II. 4)  $\pm 1\text{M}$ .

§2.11, page 124: Both are true.

§2.12, page 133: 4) Machine independence.

## Answers to Check Yourself

# 3

*Numerical precision  
is the very soul of  
science.*

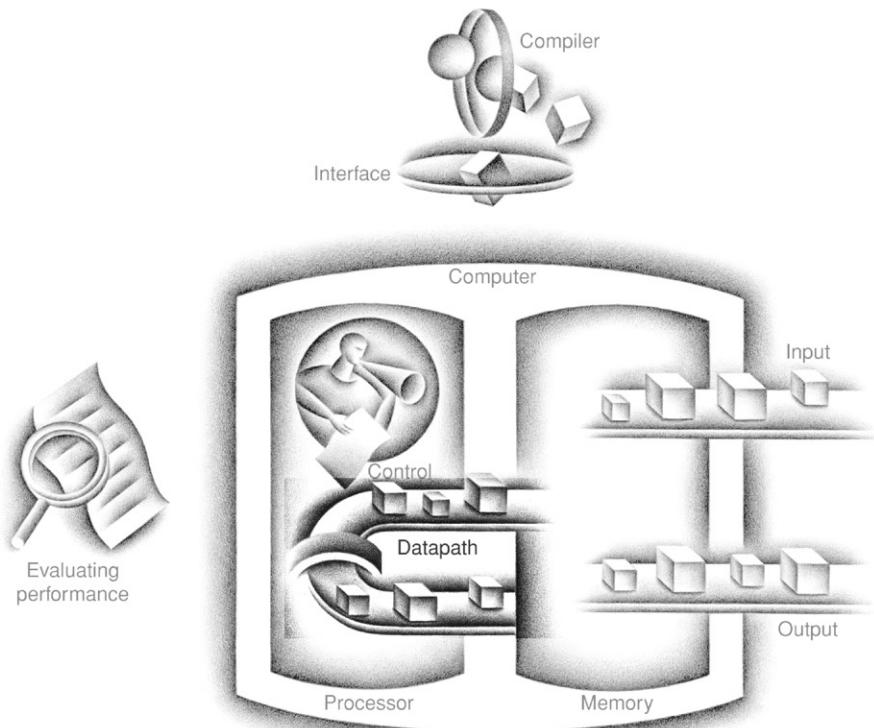
**Sir D'arcy Wentworth Thompson,**  
*On Growth and Form, 1917*

## Arithmetic for Computers

- 3.1      Introduction**    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
- 

## 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

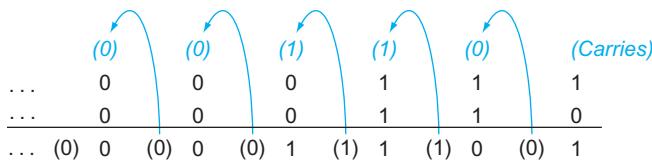
Addition is just what you would expect in computers. Digits are added bit by bit from right to left, with carries passed to the next digit to the left, just as you would do by hand. Subtraction uses addition: the appropriate operand is simply negated before being added.

**EXAMPLE****Binary Addition and Subtraction**

Let's try adding  $6_{\text{ten}}$  to  $7_{\text{ten}}$  in binary and then subtracting  $6_{\text{ten}}$  from  $7_{\text{ten}}$  in binary.

$$\begin{array}{r}
 00000000 00000000 00000000 00000000 00000000 00000000 00000111_{\text{two}} = 7_{\text{ten}} \\
 + 00000000 00000000 00000000 00000000 00000000 00000000 00000110_{\text{two}} = 6_{\text{ten}} \\
 \hline
 = 00000000 00000000 00000000 00000000 00000000 00000000 00001101_{\text{two}} = 13_{\text{ten}}
 \end{array}$$

The 4 bits to the right have all the action; Figure 3.1 shows the sums and carries. 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.

Subtracting  $6_{\text{ten}}$  from  $7_{\text{ten}}$  can be done directly:

$$\begin{array}{r}
 00000000 00000000 00000000 00000000 00000000 00000000 00000111_{\text{two}} = 7_{\text{ten}} \\
 - 00000000 00000000 00000000 00000000 00000000 00000000 00000110_{\text{two}} = 6_{\text{ten}} \\
 \hline
 = 00000000 00000000 00000000 00000000 00000000 00000000 00000001_{\text{two}} = 1_{\text{ten}}
 \end{array}$$

## ANSWER

or via addition using the two's complement representation of  $-6$ :

$$\begin{array}{r}
 00000000 00000000 00000000 00000000 00000000 00000000 00000111_{\text{two}} = 7_{\text{ten}} \\
 + 11111111 11111111 11111111 11111111 11111111 11111111 111111010_{\text{two}} = -6_{\text{ten}} \\
 \hline
 = 00000000 00000000 00000000 00000000 00000000 00000000 00000001_{\text{two}} = 1_{\text{ten}}
 \end{array}$$

Recall that overflow occurs when the result from an operation cannot be represented with the available hardware, in this case a 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
$A + B$	$\geq 0$	$\geq 0$	$< 0$
$A + B$	$< 0$	$< 0$	$\geq 0$
$A - B$	$\geq 0$	$< 0$	$< 0$
$A - B$	$< 0$	$\geq 0$	$\geq 0$

**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.

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**.

**Arithmetic Logic Unit (ALU)** Hardware that performs addition, subtraction, and usually logical operations such as AND and OR.

## 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.

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`.

### Check Yourself

**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

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  $1000_{\text{ten}}$  by  $1001_{\text{ten}}$ :

Multiplicand	$1000$	
Multiplier	$\times$	$1001_{\text{ten}}$
		$1000$
		$0000$
		$0000$
		$1000$
Product	$1001000_{\text{ten}}$	

*Multiplication is vexation, Division is as bad; The rule of three doth puzzle me, And practice drives me mad.*

Anonymous,  
Elizabethan manuscript,  
1570

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 ( $1 \times$  multiplicand) in the proper place if the multiplier digit is a 1, or
2. Place 0 ( $0 \times$  multiplicand) in the proper place if the digit is 0.

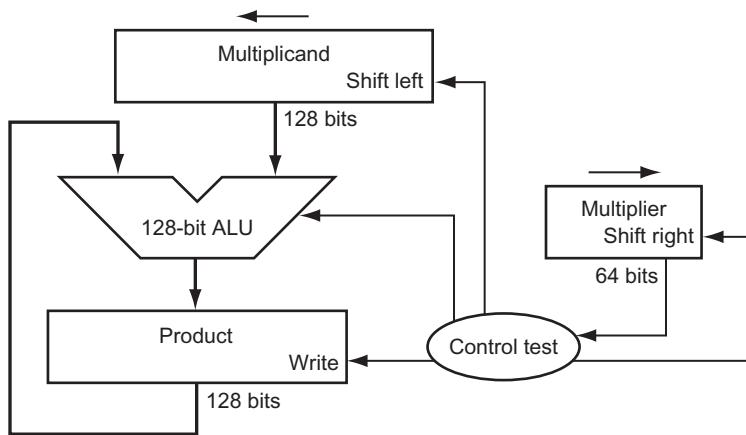
Although the decimal example above happens to use only 0 and 1, multiplication of binary numbers must always use 0 and 1, and thus always offers only these two choices.

Now that we have reviewed the basics of multiplication, the traditional next step is to provide the highly optimized multiply hardware. We break with tradition in the belief that you will gain a better understanding by seeing the evolution of the multiply hardware and algorithm through multiple generations. For now, let's assume that we are multiplying only positive numbers.

## 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.

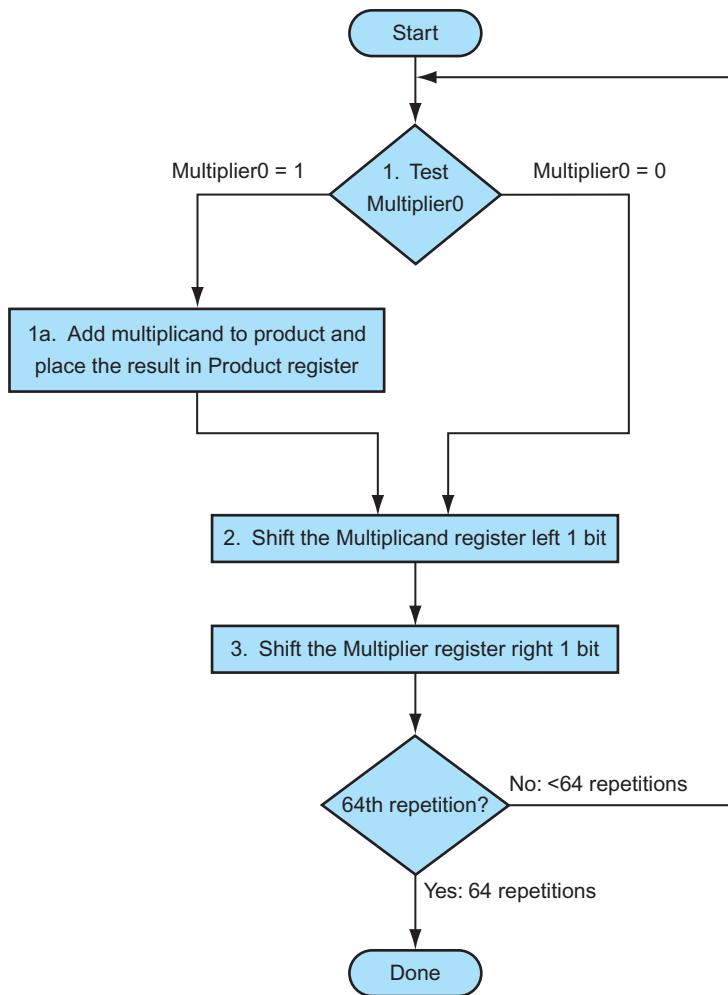
Let's assume that the multiplier is in the 64-bit Multiplier register and that the 128-bit Product register is initialized to 0. From the paper-and-pencil example above, it's clear that we will need to move the multiplicand left one digit each step, as it may be added to the intermediate products. Over 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.

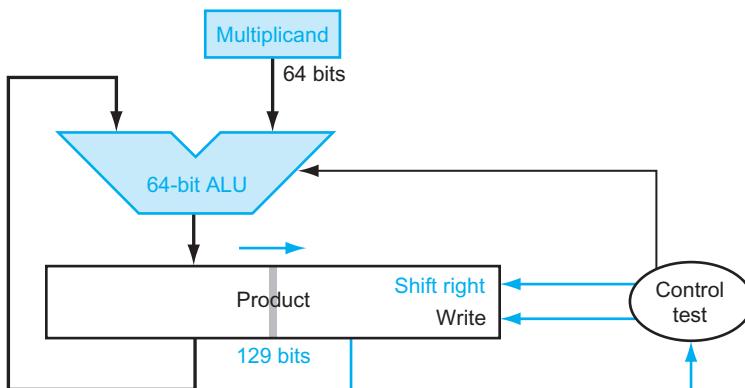
This algorithm and hardware are easily refined to take one clock cycle per step. The speed up comes from performing the operations in parallel: the multiplier and multiplicand are shifted while the multiplicand is added to the product if the multiplier bit is a 1. The hardware just has to ensure that 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	0011	0000 0010	0000 0000
1	1a: 1 $\Rightarrow$ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1 $\Rightarrow$ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1: 0 $\Rightarrow$ No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1: 0 $\Rightarrow$ No operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

**FIGURE 3.6 Multiply example using algorithm in Figure 3.4.** The bit examined to determine the next step is circled in color.

### A Multiply Algorithm

Using 4-bit numbers to save space, multiply  $2_{\text{ten}} \times 3_{\text{ten}}$ , or  $0010_{\text{two}} \times 0011_{\text{two}}$ .

### EXAMPLE

Figure 3.6 shows the value of each register for each of the steps labeled according to Figure 3.4, with the final value of  $0000\ 0110_{\text{two}}$  or  $6_{\text{ten}}$ . 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.

### ANSWER

## 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.

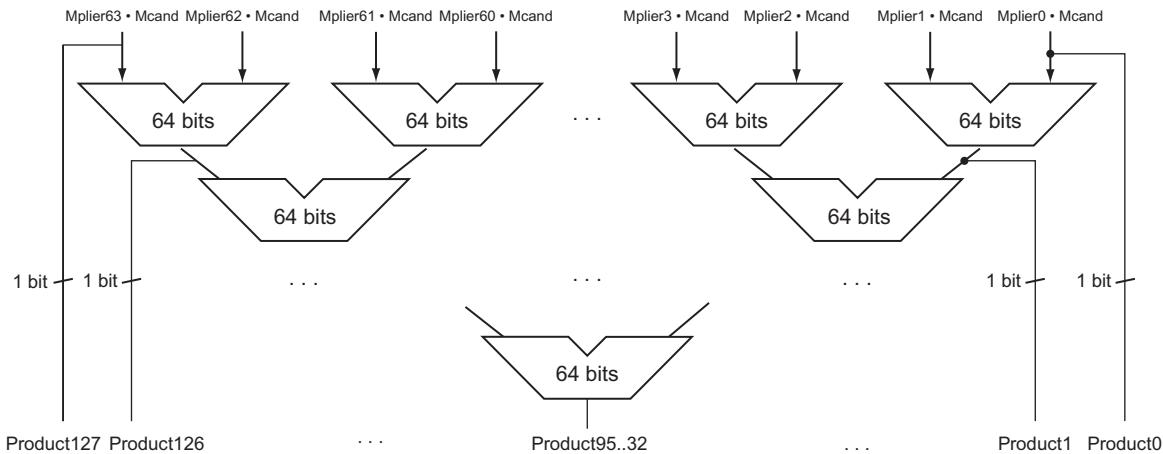
It turns out that the last algorithm will work for signed numbers, if we remember that we are dealing with numbers that have infinite digits, and we are only representing them with 64 bits. Hence, the shifting steps would need to extend the sign of the product for signed numbers. When the algorithm completes, 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.

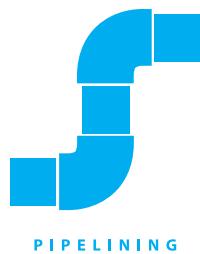
In fact, multiply can go even faster than six add times because of the use of *carry save adders* (see [Section A.6 in Appendix A](#)), and because it is easy to **pipeline** such a design to be able to support many multiplies simultaneously (see [Chapter 4](#)).

## 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.



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.

## Hardware/ Software Interface

## 3.4

## Division

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  $1,001,010_{\text{ten}}$  by  $1000_{\text{ten}}$ :

	$1001_{\text{ten}}$	Quotient
Divisor $1000_{\text{ten}}$	$\overline{1001010}_{\text{ten}}$	Dividend
	$-1000$	
	10	
	101	
	1010	
	$-1000$	
	$10_{\text{ten}}$	Remainder

*Divide et impera.*

Latin for "Divide and rule," ancient political maxim cited by Machiavelli, 1532

**dividend** A number being divided.

**divisor** A number that the dividend is divided by.

**quotient** The primary result of a division; a number that when multiplied by the divisor and added to the remainder produces the dividend.

**remainder** The secondary result of a division; a number that when added to the product of the quotient and the divisor produces the dividend.

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.

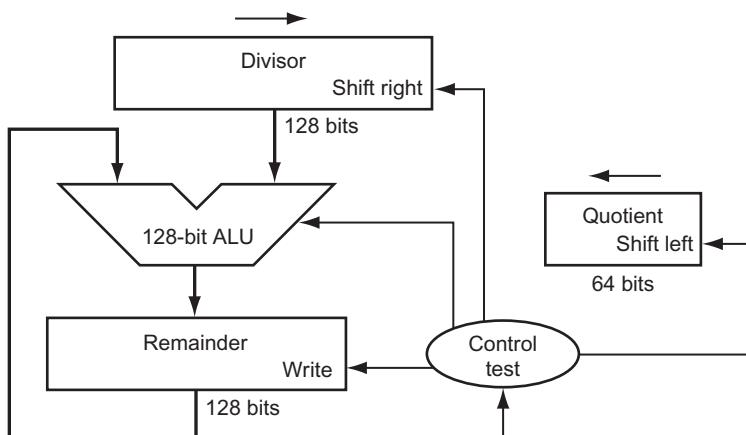
The basic division algorithm from grammar school tries to see how big a number can be subtracted, creating a digit of the quotient on each attempt. Our carefully selected decimal example uses just the numbers 0 and 1, so it's easy to figure out how many times the divisor goes into the portion of the 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.

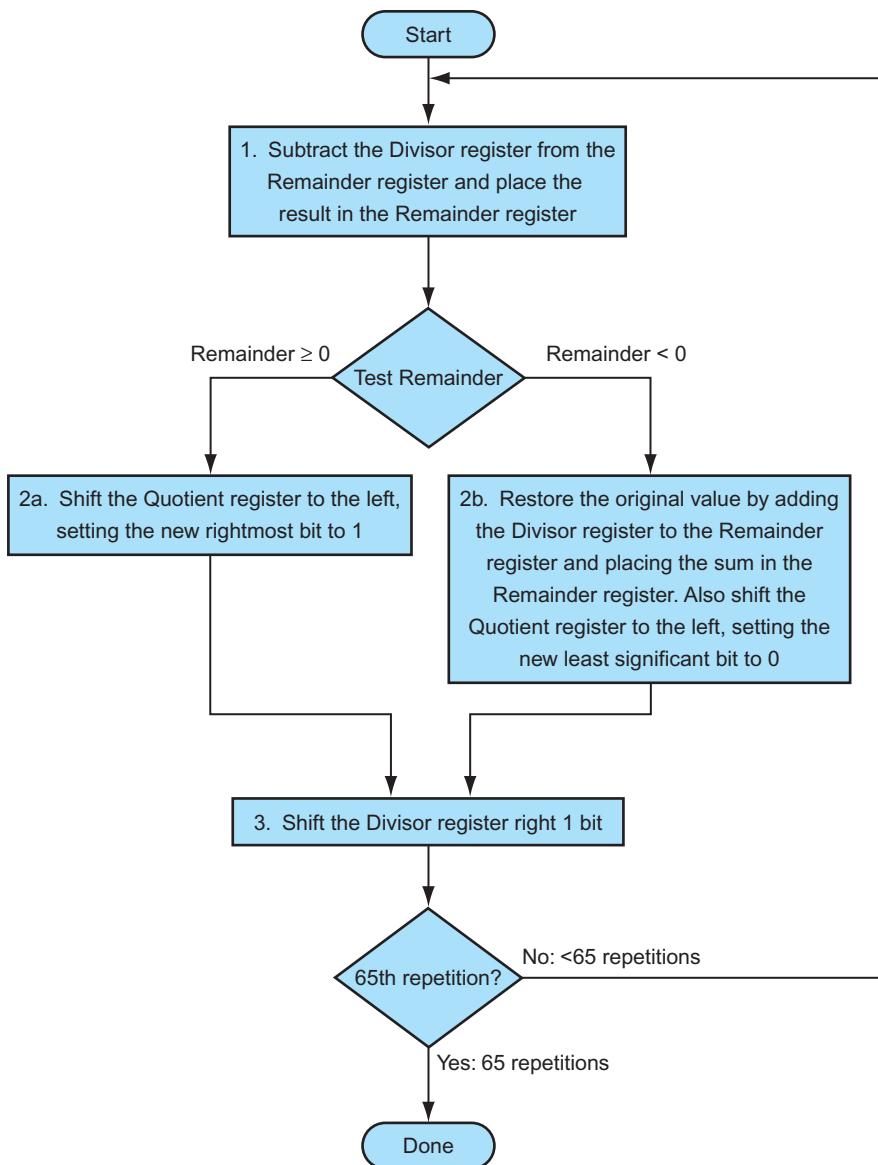
## A Division Algorithm and Hardware

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

## ANSWER

### A Divide Algorithm

Using a 4-bit version of the algorithm to save pages, let's try dividing  $7_{\text{ten}}$  by  $2_{\text{ten}}$ , or  $0000\ 0111_{\text{two}}$  by  $0010_{\text{two}}$ .

Figure 3.10 shows the value of each register for each of the steps, with the quotient being  $3_{\text{ten}}$  and the remainder  $1_{\text{ten}}$ . 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	0110 0111
	2b: Rem < 0 $\Rightarrow$ +Div, SLL Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	0111 0111
	2b: Rem < 0 $\Rightarrow$ +Div, SLL Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	0111 1111
	2b: Rem < 0 $\Rightarrow$ +Div, SLL Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	0000 0011
	2a: Rem $\geq$ 0 $\Rightarrow$ SLL Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	0000 0001
	2a: Rem $\geq$ 0 $\Rightarrow$ SLL Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

**FIGURE 3.10 Division example using the algorithm in Figure 3.9.** The bit examined to determine the next step is circled in color.

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:

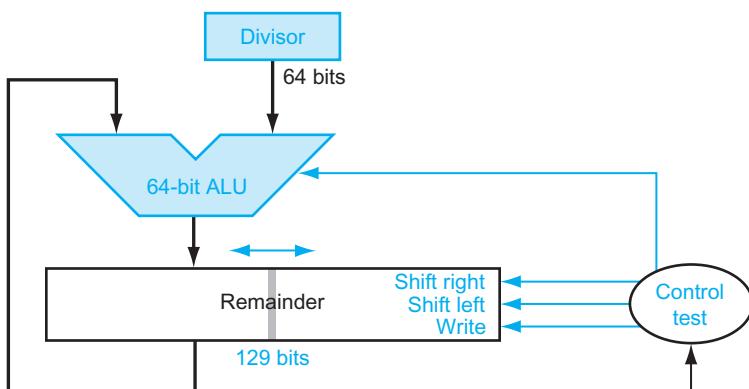
$$\text{Dividend} = \text{Quotient} \times \text{Divisor} + \text{Remainder}$$

To understand how to set the sign of the remainder, let's look at the example of dividing all the combinations of  $\pm 7_{\text{ten}}$  by  $\pm 2_{\text{ten}}$ . The first case is easy:

$$+7 \div +2: \text{Quotient} = +3, \text{Remainder} = +1$$

Checking the results:

$$+7 = 3 \times 2 + (+1) = 6 + 1$$



**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:

$$\begin{aligned}\text{Remainder} &= (\text{Dividend} - \text{Quotient} \times \text{Divisor}) = -7 - (-3x + 2) \\ &= -7 - (-6) = -1\end{aligned}$$

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

$$-(x \div y) \neq (-x) \div y$$

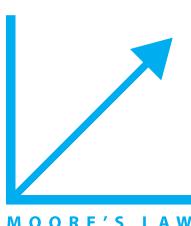
programming would be an even greater challenge. This anomalous behavior is avoided by following the rule that the dividend and remainder must have identical signs, no matter what the signs of the divisor and quotient.

We calculate the other combinations by following the same rule:

$$\begin{aligned}+7 \div -2: \text{Quotient} &= -3, \text{Remainder} = +1 \\ -7 \div -2: \text{Quotient} &= +3, \text{Remainder} = -1\end{aligned}$$

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.



PREDICTION

## 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.

---

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.

## Hardware/ Software Interface

---

**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	$x5 = 1 \text{ if } x5 < x6, \text{ else } 0$	Three register operands
	Set if less than, unsigned	sltu x5, x6, x7	$x5 = 1 \text{ if } x5 < x6, \text{ else } 0$	Three register operands
	Set if less than, immediate	slti x5, x6, x7	$x5 = 1 \text{ if } x5 < x6, \text{ else } 0$	Comparison with immediate
	Set if less than immediate, uns.	sltiu x5, x6, x7	$x5 = 1 \text{ if } x5 < x6, \text{ else } 0$	Comparison with immediate
	Multiply	mul x5, x6, x7	$x5 = x6 \times x7$	Lower 64 bits of 128-bit product
	Multiply high	mulh x5, x6, x7	$x5 = (x6 \times x7) \gg 64$	Upper 64 bits of 128-bit signed product
	Multiply high, unsigned	mulhu x5, x6, x7	$x5 = (x6 \times x7) \gg 64$	Upper 64 bits of 128-bit unsigned product
	Multiply high, signed-unsigned	mulhsu x5, x6, x7	$x5 = (x6 \times x7) \gg 64$	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)	$x5 = \text{Memory}[x6 + 40]$	Doubleword from memory to register
	Store doubleword	sd x5, 40(x6)	$\text{Memory}[x6 + 40] = x5$	Doubleword from register to memory
	Load word	lw x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Word from memory to register
	Load word, unsigned	lwu x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Unsigned word from memory to register
	Store word	sw x5, 40(x6)	$\text{Memory}[x6 + 40] = x5$	Word from register to memory
	Load halfword	lh x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Halfword from memory to register
	Load halfword, unsigned	lhu x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Unsigned halfword from memory to register
	Store halfword	sh x5, 40(x6)	$\text{Memory}[x6 + 40] = x5$	Halfword from register to memory
	Load byte	lb x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Byte from memory to register
	Load byte, unsigned	lbu x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Byte halfword from memory to register
	Store byte	sb x5, 40(x6)	$\text{Memory}[x6 + 40] = x5$	Byte from register to memory
	Load reserved	lr.d x5, (x6)	$x5 = \text{Memory}[x6]$	Load; 1st half of atomic swap
	Store conditional	sc.d x7, x5, (x6)	$\text{Memory}[x6] = x5; x7 = 0/1$	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	$x5 = \text{PC} + 0x12345000$	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	$x5 = x6 \ll x7$	Shift left by register
	Shift right logical	srl x5, x6, x7	$x5 = x6 \gg x7$	Shift right by register
	Shift right arithmetic	sra x5, x6, x7	$x5 = x6 \gg x7$	Arithmetic shift right by register
	Shift left logical immediate	slli x5, x6, 3	$x5 = x6 \ll 3$	Shift left by immediate
	Shift right logical immediate	srali x5, x6, 3	$x5 = x6 \gg 3$	Shift right by immediate
	Shift right arithmetic immediate	srai x5, x6, 3	$x5 = x6 \gg 3$	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 ( $x5 \geq x6$ ) 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 greater/equal, unsigned	bgeu x5, x6, 100	if ( $x5 \geq x6$ ) go to PC+100	PC-relative branch if registers greater or equal
Unconditional branch	Jump and link	jal x1, 100	$x1 = \text{PC}+4; \text{go to PC}+100$	PC-relative procedure call
	Jump and link register	jalr x1, 100(x5)	$x1 = \text{PC}+4; \text{go to } x5+100$	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

Going beyond signed and unsigned integers, programming languages support numbers with fractions, which are called *reals* in mathematics. Here are some examples of reals:

$3.14159265\dots_{\text{ten}}$  (pi)

$2.71828\dots_{\text{ten}}$  (*e*)

$0.000000001_{\text{ten}}$  or  $1.0_{\text{ten}} \times 10^{-9}$  (seconds in a nanosecond)

$3,155,760,000_{\text{ten}}$  or  $3.15576_{\text{ten}} \times 10^9$  (seconds in a typical century)

Notice that in the last case, the number didn't represent a small fraction, but it was bigger than we could represent with a 32-bit signed integer. The alternative notation for the last two numbers is called **scientific notation**, which has a single digit to the left of the decimal point. A number in scientific notation that has no leading 0s is called a **normalized** number, which is the usual way to write it. For example,  $1.0_{\text{ten}} \times 10^{-9}$  is in normalized scientific notation, but  $0.1_{\text{ten}} \times 10^{-8}$  and  $10.0_{\text{ten}} \times 10^{-10}$  are not.

Just as we can show decimal numbers in scientific notation, we can also show binary numbers in scientific notation:

$$1.0_{\text{two}} \times 2^{-1}$$

To keep a binary number in 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

$$1.xxxxxxxx_{\text{two}} \times 2^{yyyy}$$

(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

*Speed gets you nowhere if you're headed the wrong way.*

American proverb

**scientific notation** A notation that renders numbers with a single digit to the left of the decimal point.

**normalized** A number in floating-point notation that has no leading 0s.

**floating point** Computer arithmetic that represents numbers in which the binary point is not fixed.

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.

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

In general, floating-point numbers are of the form

$$(-1)^S \times F \times 2^E$$

*F* involves the value in the fraction field and *E* involves the value in the exponent field; the exact relationship to these fields will be spelled out soon. (We will shortly see that RISC-V does something slightly more sophisticated.)

These chosen sizes of exponent and fraction give RISC-V computer arithmetic an extraordinary range. Fractions almost as small as  $2.0_{\text{ten}} \times 10^{-38}$  and numbers almost as large as  $2.0_{\text{ten}} \times 10^{38}$  can be represented in a computer. Alas, extraordinary differs from infinite, so it is still possible for numbers to be too large. Thus, overflow interrupts can occur in floating-point arithmetic as well as in integer arithmetic. Notice that **overflow** here means that the exponent is too large to be represented in the exponent field.

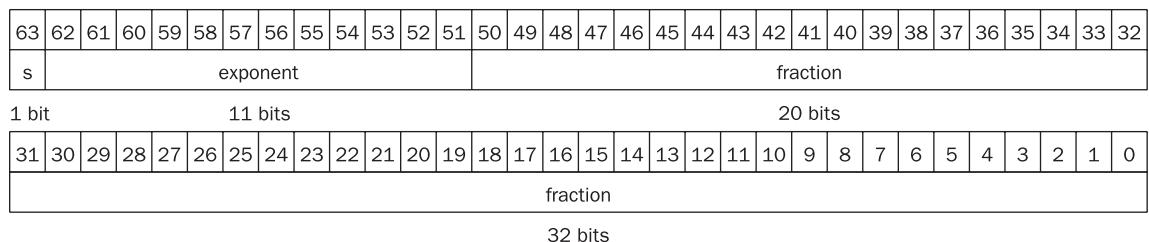
Floating point offers a new kind of exceptional event as well. Just as programmers will want to know when they have calculated a number that is too large to be represented, they will want to know if the nonzero fraction they are calculating has become so small that it cannot be represented; either event could result in a program giving incorrect answers. To distinguish it from overflow, we call this event **underflow**. This situation occurs when the negative exponent is too large to fit in the exponent field.

**overflow (floating-point)** A situation in which a positive exponent becomes too large to fit in the exponent field.

**underflow (floating-point)** A situation in which a negative exponent becomes too large to fit in the exponent field.

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.

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  $2.0_{\text{ten}} \times 10^{-308}$  and almost as large as  $2.0_{\text{ten}} \times 10^{308}$ . Although double precision does increase the exponent range, its primary advantage is its greater precision because of the much larger fraction.

## 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.

### double precision

A floating-point value represented in a 64-bit doubleword.

### single precision

A floating-point value represented in a 32-bit word.

**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<sub>two</sub> 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* s<sub>1</sub>, s<sub>2</sub>, s<sub>3</sub>, ..., then the value is

$$(-1)^S \times (1 + (s_1 \times 2^{-1}) + (s_2 \times 2^{-2}) + (s_3 \times 2^{-3}) + (s_4 \times 2^{-4}) + \dots) \times 2^E$$

[Figure 3.13](#) shows the encodings of IEEE 754 floating-point numbers. Other features of IEEE 754 are special symbols to represent unusual events. For example, instead of interrupting on a divide by 0, software can set the result to a bit pattern representing  $+\infty$  or  $-\infty$ ; the largest exponent is reserved for these special symbols. When the programmer prints the results, the program will 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	$\pm$ denormalized number
1–254	Anything	1–2046	Anything	$\pm$ floating-point number
255	0	2047	0	$\pm$ infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

**FIGURE 3.13 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,  $1.0_{\text{two}} \times 2^{-1}$  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	0

(Remember that the leading 1 is implicit in the significand.) The value  $1.0_{\text{two}} \times 2^{+1}$  would look like the smaller binary number

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

The desirable notation must therefore represent the most negative exponent as  $00\dots00_{\text{two}}$  and the most positive as  $11\dots11_{\text{two}}$ . This convention is called *biased notation*, with the bias being the number subtracted from the normal, unsigned representation to determine the real value.

IEEE 754 uses a bias of 127 for single precision, so an exponent of  $-1$  is represented by the bit pattern of the value  $-1 + 127_{\text{ten}}$ , or  $126_{\text{ten}} = 0111\ 1110_{\text{two}}$ , and  $+1$  is represented by  $1 + 127$ , or  $128_{\text{ten}} = 1000\ 0000_{\text{two}}$ . The exponent bias for double precision is 1023. Biased exponent means that the value represented by a floating-point number is really

$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

The range of single precision numbers is then from as small as

$$\pm 1.000000000000000000000000000000_{\text{two}} \times 2^{-126}$$

to as large as

$$\pm 1.1111111111111111111111111111_{\text{two}} \times 2^{+127}.$$

Let's demonstrate.

**EXAMPLE****ANSWER**

Show the IEEE 754 binary representation of the number  $-0.75_{\text{ten}}$  in single and double precision.

The number  $-0.75_{\text{ten}}$  is also

$$-3/4_{\text{ten}} \text{ or } -3/2^2_{\text{ten}}$$

It is also represented by the binary fraction

$$-11_{\text{two}}/2^2_{\text{ten}} \text{ or } -0.11_{\text{two}}$$

In scientific notation, the value is

$$-0.11_{\text{two}} \times 2^0$$

and in normalized scientific notation, it is

$$-1.1_{\text{two}} \times 2^{-1}$$

The general representation for a single precision number is

$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent}-127)}$$

Subtracting the bias 127 from the exponent of  $-1.1_{\text{two}} \times 2^{-1}$  yields

$$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000_{\text{two}}) \times 2^{(126-127)}$$

The single precision binary representation of  $-0.75_{\text{ten}}$  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	0

1 bit

8 bits

23 bits

The double precision representation is

$$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{\text{two}}) \times 2^{(1022-1023)}$$

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

1 bit

11 bits

20 bits

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	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

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,

### ANSWER

$$\begin{aligned} (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})} &= (-1)^1 \times (1 + 0.25) \times 2^{(129 - 127)} \\ &= -1 \times 1.25 \times 2^2 \\ &= -1.25 \times 4 \\ &= -5.0 \end{aligned}$$

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:  $9.999_{\text{ten}} \times 10^1 + 1.610_{\text{ten}} \times 10^{-1}$ . Assume that we can store only four decimal digits of the significand and two decimal digits of the exponent.

- Step 1. To be able to add these numbers properly, we must align the decimal point of the number that has the smaller exponent. Hence, we need a form of the smaller number,  $1.610_{\text{ten}} \times 10^{-1}$ , that matches the larger exponent. We obtain this by observing that there are multiple representations of an unnormalized floating-point number in scientific notation:

$$1.610_{\text{ten}} \times 10^{-1} = 0.1610_{\text{ten}} \times 10^0 = 0.01610_{\text{ten}} \times 10^1$$

The number on the right is the version we desire, since its exponent matches the exponent of the larger number,  $9.999_{\text{ten}} \times 10^1$ . Thus, the first step shifts the significand of the smaller number to the right until its corrected exponent matches that of the larger number. But we can represent only four decimal digits so, after shifting, the number is really

$$0.016 \times 10^1$$

- Step 2. Next comes the addition of the significands:

$$\begin{array}{r} 9.999_{\text{ten}} \\ + \quad 0.016_{\text{ten}} \\ \hline 10.015_{\text{ten}} \end{array}$$

The sum is  $10.015_{\text{ten}} \times 10^1$ .

- Step 3. This sum is not in normalized scientific notation, so we need to adjust it:

$$10.015_{\text{ten}} \times 10^1 = 1.0015_{\text{ten}} \times 10^2$$

Thus, after the addition we may have to shift the sum to put it into normalized form, adjusting the exponent appropriately. This example shows shifting to the right, but if one number were positive and the other were negative, it would be possible for the sum to have many leading 0s, requiring left shifts. Whenever the exponent is increased or decreased, we must check for overflow or underflow—that is, we must make sure that the exponent still fits in its field.

- Step 4. Since we assumed that the significand 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

$$1.0015_{\text{ten}} \times 10^2$$

is rounded to four digits in the significand to

$$1.002_{\text{ten}} \times 10^2$$

since the fourth digit to the right of the decimal point was between 5 and 9. Notice that if we have bad luck on rounding, such as adding 1 to a string of 9s, the sum may no longer be normalized and we would need to perform step 3 again.

[Figure 3.14](#) shows the algorithm for binary floating-point addition that follows this decimal example. Steps 1 and 2 are similar to the example just discussed: adjust the significand of the number with the smaller exponent and then add the two significands. Step 3 normalizes the results, forcing a check for overflow or underflow. The test for overflow and underflow in step 3 depends on the precision of the operands. Recall that the pattern of all 0 bits in the exponent is reserved and used for the floating-point representation of zero. Moreover, the pattern of all 1 bits in the exponent is reserved for indicating values and situations outside the scope of normal floating-point numbers (see the *Elaboration* on page 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

Try adding the numbers  $0.5_{\text{ten}}$  and  $-0.4375_{\text{ten}}$  in binary using the algorithm in [Figure 3.14](#).

### EXAMPLE

Let's first look at the binary version of the two numbers in normalized scientific notation, assuming that we keep 4 bits of precision:

### ANSWER

$$\begin{array}{lll} 0.5_{\text{ten}} & = 1/2_{\text{ten}} & = 1/2_{\text{ten}}^1 \\ & = 0.1_{\text{two}} & = 0.1_{\text{two}} \times 2^0 & = 1.000_{\text{two}} \times 2^{-1} \\ -0.4375_{\text{ten}} & = -7/16_{\text{ten}} & = -7/2_{\text{ten}}^4 \\ & = -0.0111_{\text{two}} & = -0.0111_{\text{two}} \times 2^0 & = -1.110_{\text{two}} \times 2^{-2} \end{array}$$

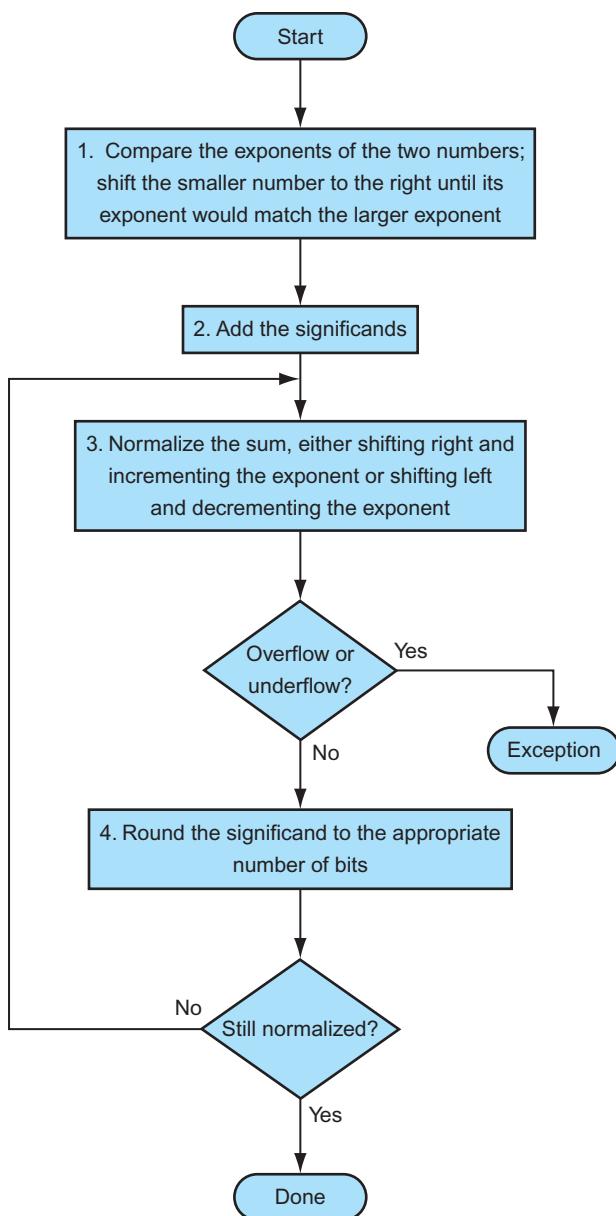
Now we follow the algorithm:

Step 1. The significand of the number with the lesser exponent ( $-1.11_{\text{two}} \times 2^{-2}$ ) is shifted right until its exponent matches the larger number:

$$-1.110_{\text{two}} \times 2^{-2} = -0.111_{\text{two}} \times 2^{-1}$$

Step 2. Add the significands:

$$1.000_{\text{two}} \times 2^{-1} + (-0.111_{\text{two}} \times 2^{-1}) = 0.001_{\text{two}} \times 2^{-1}$$



**FIGURE 3.14 Floating-point addition.** The normal path is to execute steps 3 and 4 once, but if rounding causes the sum to be unnormalized, we must repeat step 3.

Step 3. Normalize the sum, checking for overflow or underflow:

$$\begin{aligned} 0.001_{\text{two}} \times 2^{-1} &= 0.010_{\text{two}} \times 2^{-2} = 0.100_{\text{two}} \times 2^{-3} \\ &= 1.000_{\text{two}} \times 2^{-4} \end{aligned}$$

Since  $127 \geq -4 \geq -126$ , there is no overflow or underflow. (The biased exponent would be  $-4 + 127$ , or 123, which is between 1 and 254, the smallest and largest unreserved biased exponents.)

Step 4. Round the sum:

$$1.000_{\text{two}} \times 2^{-4}$$

The sum already fits exactly in 4 bits, so there is no change to the bits due to rounding.

This sum is then

$$\begin{aligned} 1.000_{\text{two}} \times 2^{-4} &= 0.0001000_{\text{two}} = 0.0001_{\text{two}} \\ &= 1/2^4 \quad = 1/16_{\text{ten}} \quad = 0.0625_{\text{ten}} \end{aligned}$$

This sum is what we would expect from adding  $0.5_{\text{ten}}$  to  $-0.4375_{\text{ten}}$ .

Many computers dedicate hardware to run floating-point operations as fast as possible. [Figure 3.15](#) sketches the basic organization of hardware for floating-point addition.

## Floating-Point Multiplication

Now that we have explained floating-point addition, let's try floating-point multiplication. We start by multiplying decimal numbers in scientific notation by hand:  $1.110_{\text{ten}} \times 10^{10} \times 9.200_{\text{ten}} \times 10^{-5}$ . Assume that we can store only four digits of the significand and two digits of the exponent.

Step 1. Unlike addition, we calculate the exponent of the product by simply adding the exponents of the operands together:

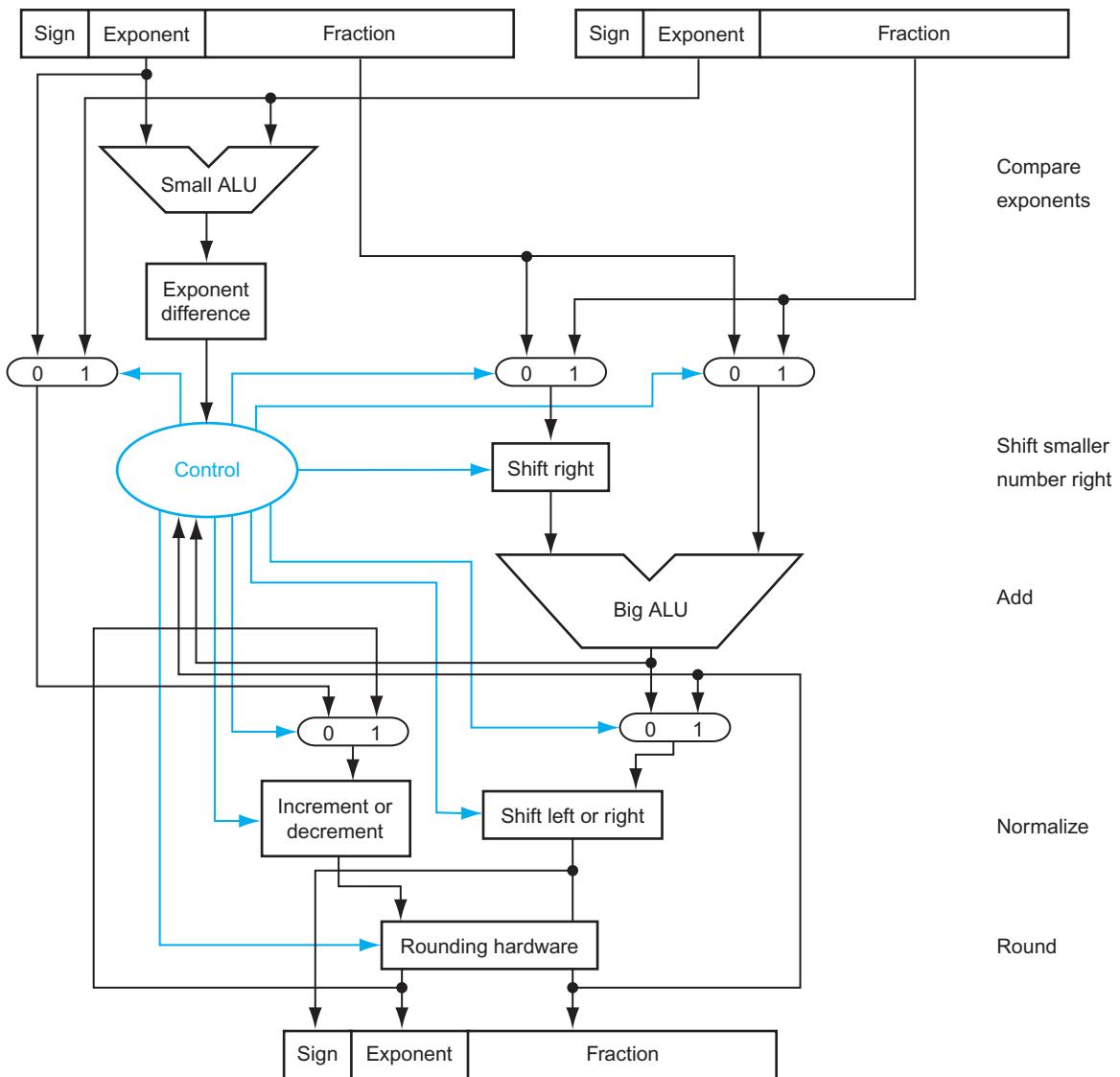
$$\text{New exponent} = 10 + (-5) = 5$$

Let's do this with the biased exponents as well to make sure we obtain the same result:  $10 + 127 = 137$ , and  $-5 + 127 = 122$ , so

$$\text{New exponent} = 137 + 122 = 259$$

This result is too large for the 8-bit exponent field, so something is amiss! The problem is with the bias because we are adding the biases as well as the exponents:

$$\text{New exponent} = (10 + 127) + (-5 + 127) = (5 + 2 \times 127) = 259$$



**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:

$$\begin{array}{r} 1.110_{\text{ten}} \\ \times 9.200_{\text{ten}} \\ \hline 0000 \\ 0000 \\ 2220 \\ 9990 \\ \hline 1110000_{\text{ten}} \end{array}$$

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  $10.212 \times 10^5$ .

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

$$1.0212_{\text{ten}} \times 10^6$$

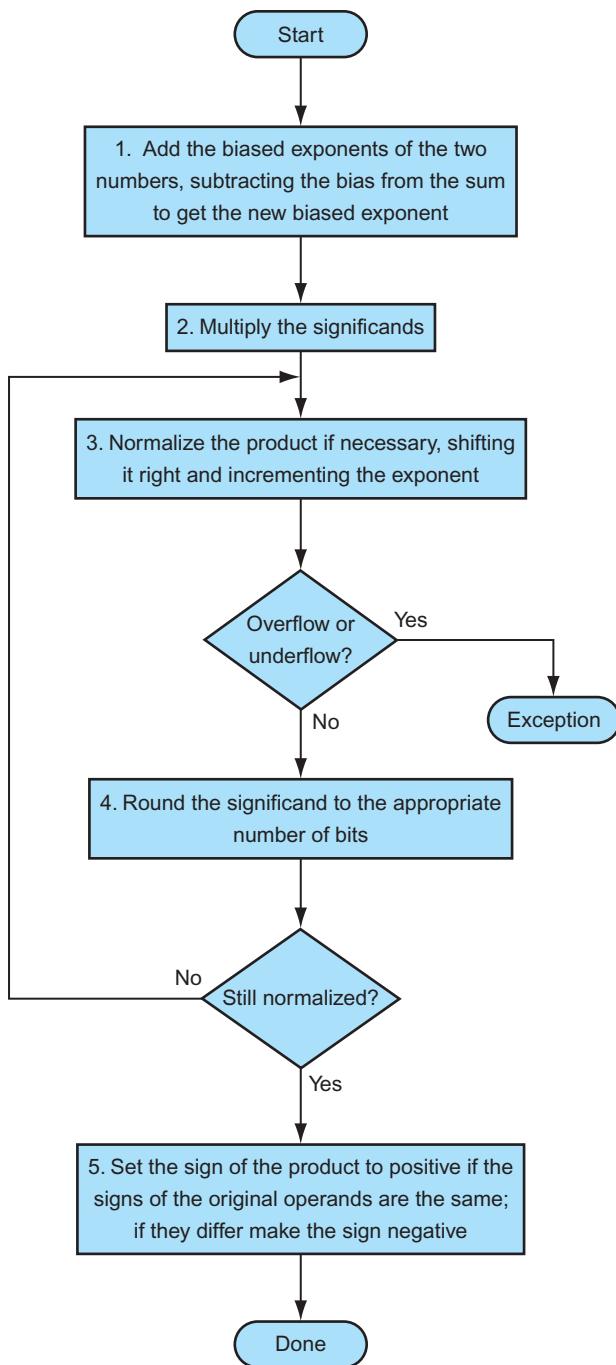
is rounded to four digits in the significand to

$$1.021_{\text{ten}} \times 10^6$$

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  $0.5_{\text{ten}}$  and  $-0.4375_{\text{ten}}$ , using the steps in Figure 3.16.

In binary, the task is multiplying  $1.000_{\text{two}} \times 2^{-1}$  by  $-1.110_{\text{two}} \times 2^{-2}$ .

### ANSWER

Step 1. Adding the exponents without bias:

$$-1 + (-2) = -3$$

or, using the biased representation:

$$\begin{aligned} (-1 + 127) + (-2 + 127) - 127 &= (-1 - 2) + (127 + 127 - 127) \\ &= -3 + 127 = 124 \end{aligned}$$

Step 2. Multiplying the significands:

$$\begin{array}{r} 1.000_{\text{two}} \\ \times \underline{1.110}_{\text{two}} \\ \hline 0000 \\ 1000 \\ 1000 \\ 1000 \\ \hline 1110000_{\text{two}} \end{array}$$

The product is  $1.110000_{\text{two}} \times 2^{-3}$ , but we need to keep it to 4 bits, so it is  $1.110_{\text{two}} \times 2^{-3}$ .

Step 3. Now we check the product to make sure it is normalized, and then check the exponent for overflow or underflow. The product is already normalized and, since  $127 \geq -3 \geq -126$ , there is no overflow or underflow. (Using the biased representation,  $254 \geq 124 \geq 1$ , so the exponent fits.)

Step 4. Rounding the product makes no change:

$$1.110_{\text{two}} \times 2^{-3}$$

Step 5. Since the signs of the original operands differ, make the sign of the product negative. Hence, the product is

$$-1.110_{\text{two}} \times 2^{-3}$$

Converting to decimal to check our results:

$$\begin{aligned} -1.110_{\text{two}} \times 2^{-3} &= -0.001110_{\text{two}} = -0.00111_{\text{two}} \\ &= -7/2^5_{\text{ten}} = -7/32_{\text{ten}} = -0.21875_{\text{ten}} \end{aligned}$$

The product of  $0.5_{\text{ten}}$  and  $-0.4375_{\text{ten}}$  is indeed  $-0.21875_{\text{ten}}$ .

## 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* (fddiv.s) and *division, double* (fddiv.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	$f0 = \sqrt{f1}$	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	$f0 = \sqrt{f1}$	FP square root (double precision)
Comparison	FP equality single	feq.s x5, f0, f1	$x5 = 1 \text{ if } f0 == f1, \text{ else } 0$	FP comparison (single precision)
	FP less than single	flt.s x5, f0, f1	$x5 = 1 \text{ if } f0 < f1, \text{ else } 0$	FP comparison (single precision)
	FP less than or equals single	fle.s x5, f0, f1	$x5 = 1 \text{ if } f0 \leq f1, \text{ else } 0$	FP comparison (single precision)
	FP equality double	feq.d x5, f0, f1	$x5 = 1 \text{ if } f0 == f1, \text{ else } 0$	FP comparison (double precision)
	FP less than double	flt.d x5, f0, f1	$x5 = 1 \text{ if } f0 < f1, \text{ else } 0$	FP comparison (double precision)
	FP less than or equals double	fle.d x5, f0, f1	$x5 = 1 \text{ if } f0 \leq f1, \text{ else } 0$	FP comparison (double precision)
Data transfer	FP load word	flw f0, 4(x5)	$f0 = \text{Memory}[x5 + 4]$	Load single-precision from memory
	FP load doubleword	fld f0, 8(x5)	$f0 = \text{Memory}[x5 + 8]$	Load double-precision from memory
	FP store word	fsw f0, 4(x5)	$\text{Memory}[x5 + 4] = f0$	Store single-precision to memory
	FP store doubleword	fsd f0, 8(x5)	$\text{Memory}[x5 + 8] = f0$	Store double-precision to 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):

```
f1w    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?

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

### ANSWER

$c[i][j]$  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  $32 \times 32$ , 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 * 25(size of row of c)
```

Now we add the second index to select the  $j$ th 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 * 25(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 * 25(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.

## 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.

**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.

### Rounding with Guard Digits

Add  $2.56_{\text{ten}} \times 10^0$  to  $2.34_{\text{ten}} \times 10^2$ , assuming that we have three significant decimal digits. Round to the nearest decimal number with three significant decimal digits, first with guard and round digits, and then without them.

First we must shift the smaller number to the right to align the exponents, so  $2.56_{\text{ten}} \times 10^0$  becomes  $0.0256_{\text{ten}} \times 10^2$ . Since we have guard and round digits, we are able to represent the two least significant digits when we align exponents. The guard digit holds 5 and the round digit holds 6. The sum is

$$\begin{array}{r} 2.3400_{\text{ten}} \\ + 0.0256_{\text{ten}} \\ \hline 2.3656_{\text{ten}} \end{array}$$

Thus the sum is  $2.3656_{\text{ten}} \times 10^2$ . Since we have two digits to round, we want values 0 to 49 to round down and 51 to 99 to round up, with 50 being the tiebreaker. Rounding the sum up with three significant digits yields  $2.37_{\text{ten}} \times 10^2$ .

Doing this *without* guard and round digits drops two digits from the calculation. The new sum is then

$$\begin{array}{r} 2.34_{\text{ten}} \\ + 0.02_{\text{ten}} \\ \hline 2.36_{\text{ten}} \end{array}$$

The answer is  $2.36_{\text{ten}} \times 10^2$ , off by 1 in the last digit from the sum above.

Since the worst case for rounding would be when the actual number is halfway between two floating-point representations, accuracy in floating point is normally measured in terms of the number of bits in error in the least significant bits of the significand; the measure is called the number of **units in the last place**, or **ulp**. If a number were off by 2 in the least significant bits, it would be called off by 2 ulps. Provided there are no overflow, underflow, or invalid operation exceptions, IEEE 754 guarantees that the computer uses the number that is within one-half ulp.

### EXAMPLE

### ANSWER

**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  $0.50 \dots 00_{\text{ten}}$  and  $0.50 \dots 01_{\text{ten}}$  when rounding.

The sticky bit may be set, for example, during addition, when the smaller number is shifted to the right. Suppose we added  $5.01_{\text{ten}} \times 10^{-1}$  to  $2.34_{\text{ten}} \times 10^2$  in the example above. Even with guard and round, we would be adding 0.0050 to 2.34, with a sum of 2.3450. The sticky bit would be set, since there are nonzero bits to the right. Without the sticky bit to remember whether any 1s were shifted off, we would assume the number is equal to 2.345000 ... 00 and round to the nearest even of 2.34. With the sticky bit to remember that the number is larger than 2.345000 ... 00, we round instead to 2.35.

**Elaboration:** 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.

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	f1w, fsw	fadd.s, fsub.s, fmul.s, fdiv.s, feq.s, flt.s, fle.s
double	double	f1d, fsd	fadd.d, fsub.d, fmul.d, fdiv.d, feq.d, flt.d, fle.d

In the last chapter, we presented the storage classes of the programming language C (see the *Hardware/Software Interface* section in [Section 2.7](#)). The table above shows some of the C and Java data types, the data transfer instructions, and instructions that operate on those types that appear in [Chapter 2](#) and this chapter. Note that Java omits unsigned integers.

## Hardware/ Software Interface

The revised IEEE 754-2008 standard added a 16-bit floating-point format with five exponent bits. What do you think is the likely range of numbers it could represent?

## Check Yourself

1.  $1.0000 \ 00 \times 2^0$  to  $1.1111 \ 1111 \ 11 \times 2^{31}, 0$
2.  $\pm 1.0000 \ 0000 \ 0 \times 2^{-14}$  to  $\pm 1.1111 \ 1111 \ 1 \times 2^{15}, \pm 0, \pm \infty, \text{NaN}$
3.  $\pm 1.0000 \ 0000 \ 00 \times 2^{-14}$  to  $\pm 1.1111 \ 1111 \ 11 \times 2^{15}, \pm 0, \pm \infty, \text{NaN}$
4.  $\pm 1.0000 \ 0000 \ 00 \times 2^{-15}$  to  $\pm 1.1111 \ 1111 \ 11 \times 2^{14}, \pm 0, \pm \infty, \text{NaN}$

**Elaboration:** To accommodate comparisons that may include NaNs, the standard includes *ordered* and *unordered* as options for compares. 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

$$1.0000\ 0000\ 0000\ 0000\ 0000_{\text{two}} \times 2^{-126}$$

but the smallest single precision denormalized number is

$$0.0000\ 0000\ 0000\ 0000\ 0001_{\text{two}} \times 2^{-126}, \text{ or } 1.0_{\text{two}} \times 2^{-149}$$

For double precision, the denorm gap goes from  $1.0 \times 2^{-1022}$  to  $1.0 \times 2^{-1074}$ .

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, MOVSUSS, 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**.



P ARALLELISM

**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 2.6GHz to 3.3GHz 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  $x_i$  means the  $i$ th bit, represents the number

$$\dots + (x^3 \times 2^3) + (x^2 \times 2^2) + (x^1 \times 2^1) + (x^0 \times 2^0)$$

Shifting the bits of  $c$  right by  $n$  bits would seem to be the same as dividing by  $2^n$ . And this is true for unsigned integers. The problem is with signed integers. For example, suppose we want to divide  $-5_{\text{ten}}$  by  $4_{\text{ten}}$ ; the quotient should be  $-1_{\text{ten}}$ . The two's complement representation of  $-5_{\text{ten}}$  is

11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111011<sub>two</sub>

According to this fallacy, shifting right by two should divide by  $4_{\text{ten}}$  ( $2^2$ ):

00111111 11111111 11111111 11111111 11111111 11111111 11111111 11111110<sub>two</sub>

With a 0 in the sign bit, this result is clearly wrong. The value created by the shift right is actually  $4,611,686,018,427,387,902_{\text{ten}}$  instead of  $-1_{\text{ten}}$ .

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  $-5_{\text{ten}}$  produces

11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111110<sub>two</sub>

The result is  $-2_{\text{ten}}$  instead of  $-1_{\text{ten}}$ ; close, but no cigar.

*Pitfall: Floating-point addition is not associative.*

Associativity holds for a sequence of two's complement integer additions, even if the computation overflows. Alas, because floating-point numbers are approximations of real numbers and because computer arithmetic has limited precision, it does not hold for floating-point numbers. Given the great range of numbers that can be represented in floating point, problems occur when adding two large numbers of opposite signs plus a small number. For example, let's see if  $c + (a + b) = (c + a) + b$ . Assume  $c = -1.5_{\text{ten}} \times 10^{38}$ ,  $a = 1.5_{\text{ten}} \times 10^{38}$ , and  $b = 1.0$ , and that these are all single precision numbers.

$$\begin{aligned} c + (a + b) &= -1.5_{\text{ten}} \times 10^{38} + (1.5_{\text{ten}} \times 10^{38} + 1.0) \\ &= -1.5_{\text{ten}} \times 10^{38} + (1.5_{\text{ten}} \times 10^{38}) \\ &= 0.0 \end{aligned}$$

$$\begin{aligned} c + (a + b) &= (-1.5_{\text{ten}} \times 10^{38} + 1.5_{\text{ten}} \times 10^{38}) + 1.0 \\ &= (0.0_{\text{ten}}) + 1.0 \\ &= 1.0 \end{aligned}$$

Since floating-point numbers have limited precision and result in approximations of real results,  $1.5_{\text{ten}} \times 10^{38}$  is so much larger than  $1.0_{\text{ten}}$  that  $1.5_{\text{ten}} \times 10^{38} + 1.0$  is still  $1.5_{\text{ten}} \times 10^{38}$ . That is why the sum of  $c$ ,  $a$ , and  $b$  is 0.0 or 1.0, depending on the order of the floating-point additions, so  $c + (a + b) \neq (c + a) + b$ . Therefore, floating-point addition is *not* associative.

*Fallacy: Parallel execution strategies that work for integer data types also work for floating-point data types.*

Programs have typically been written first to run sequentially before being rewritten to run concurrently, so a natural question is, “Do the two versions get the same answer?” If the answer is no, you presume there is a bug in the parallel version that you need to track down.

This approach assumes that computer arithmetic does not affect the results when going from sequential to parallel. That is, if you were to add a million numbers together, you would get the same results whether you used 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 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

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.

*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

## 3.12 Exercises

**3.1** [5] <§3.2> What is  $5ED4 - 07A4$  when these values represent unsigned 16-bit hexadecimal numbers? The result should be written in hexadecimal. Show your work.

**3.2** [5] <§3.2> What is  $5ED4 - 07A4$  when these values represent signed 16-bit hexadecimal numbers stored in sign-magnitude format? The result should be written in hexadecimal. Show your work.

**3.3** [10] <§3.2> Convert  $5ED4$  into a binary number. What makes base 16 (hexadecimal) an attractive numbering system for representing values in computers?

**3.4** [5] <§3.2> What is  $4365 - 3412$  when these values represent unsigned 12-bit octal numbers? The result should be written in octal. Show your work.

**3.5** [5] <§3.2> What is  $4365 - 3412$  when these values represent signed 12-bit octal numbers stored in sign-magnitude format? The result should be written in octal. Show your work.

**3.6** [5] <§3.2> Assume 185 and 122 are unsigned 8-bit decimal integers. Calculate  $185 - 122$ . Is there overflow, underflow, or neither?

*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



## Historical Perspective and Further Reading

This section surveys the history of the floating point going back to von Neumann, including the surprisingly controversial IEEE standards effort, 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.*

*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*

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 descendants 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  $9 \times 6$ , for example, can be written  $(2 \times 2 \times 2 + 1) \times 6$ , we can calculate  $9 \times 6$  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  $-1.5625 \times 10^{-1}$  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  $-1.5625 \times 10^{-1}$  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  $-1.5625 \times 10^{-1}$  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  $2.6125 \times 10^1$  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  $-8.0546875 \times 10^0$  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  $8.625 \times 10^1$  divided by  $-4.875 \times 10^0$ . 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

*In a major matter, no details are small.*

**French Proverb**

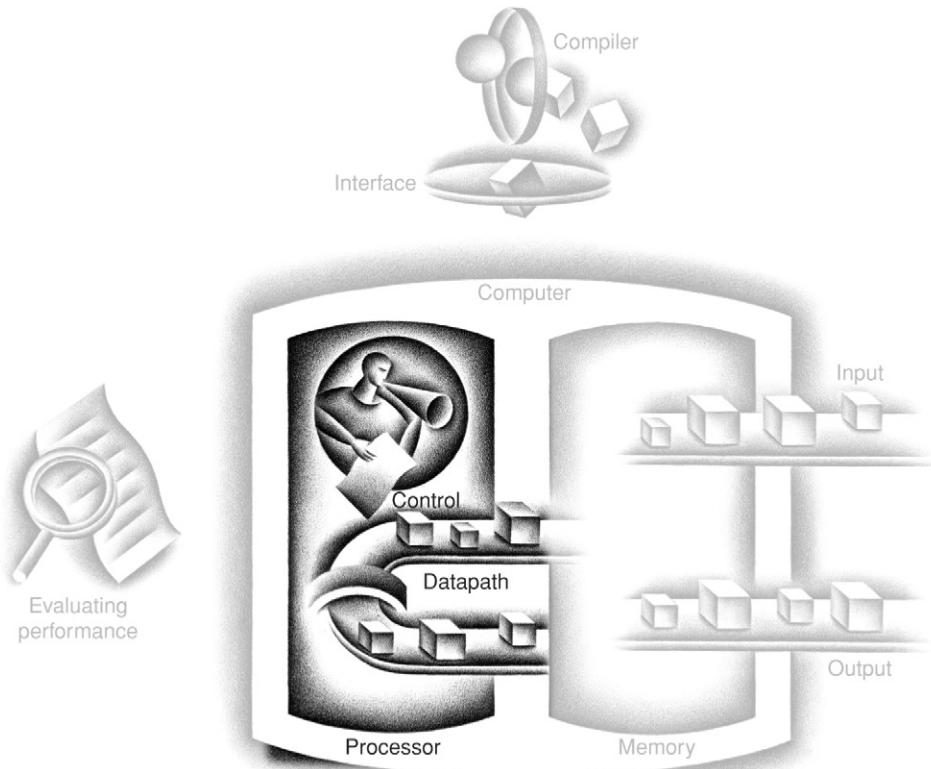
## The Processor

- 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

- 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

---

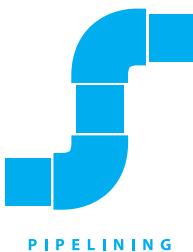
## The Five Classic Components of a Computer



## 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*, *or*, and *xor*
- 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.

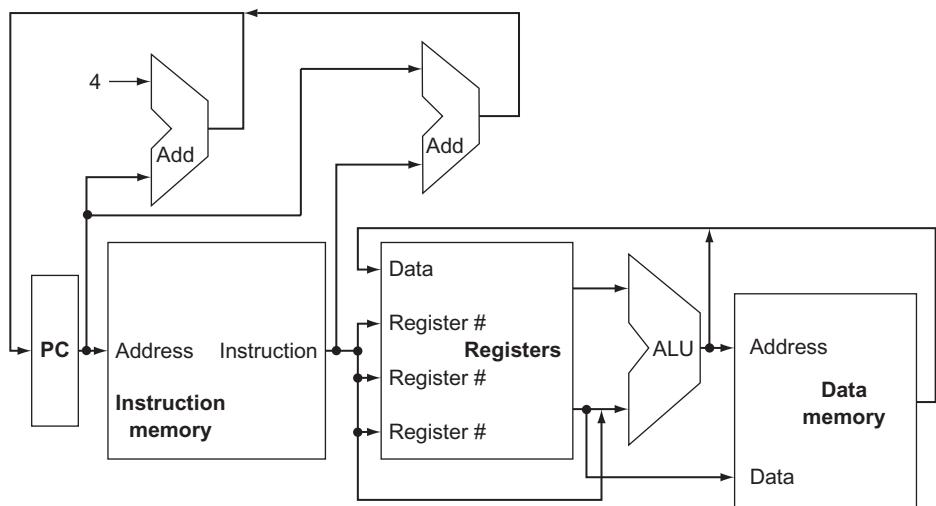
For example, all instruction classes use the arithmetic-logical unit (ALU) after reading the registers. The memory-reference instructions use the ALU for an address calculation, the arithmetic-logical instructions for the operation execution, and conditional branches for the equality test. After using 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.

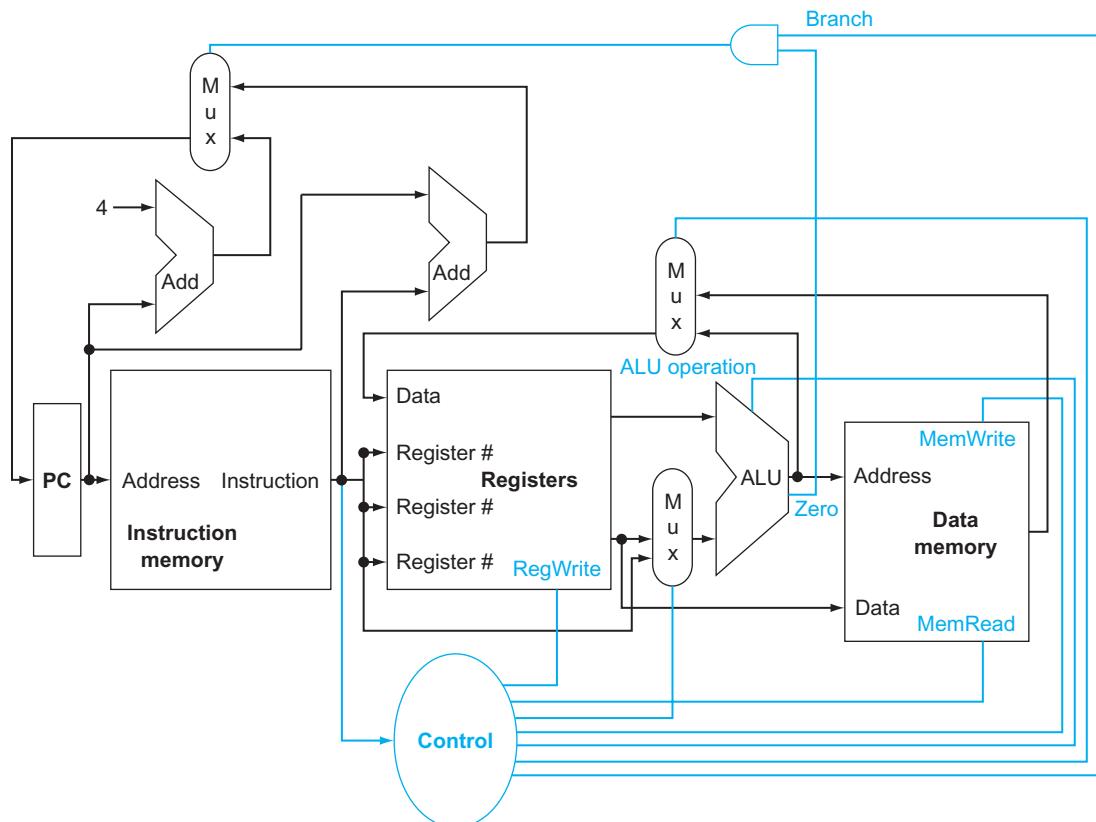
The second omission in Figure 4.1 is that several of the units must be controlled depending on the type of instruction. For example, the data memory must read on a load and write on a store. The register file must be written only on a load or



**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  $\text{PC} + 4$  or the branch destination address is written into the PC, is set based on the Zero output of the ALU, which is used to perform the comparison of a `beq` instruction. The regularity and simplicity of the 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.

Other elements in the design are not combinational, but instead contain *state*. An element contains state if it has some internal storage. We call these elements **state elements** because, if we pulled the power plug on the computer, we could restart it accurately by loading the state elements with the values they contained before we pulled the plug. Furthermore, if we saved and restored the state elements, it would be as if the computer had never lost power. Thus, these state elements completely characterize the computer. In [Figure 4.1](#), the instruction and data memories, as well as the registers, are all examples of state elements.

### combinational element

An operational element, such as an AND gate or an ALU

### state element

A memory element, such as a register or a memory.

A state element has at least two inputs and one output. The required inputs are the data value to be written into the element and the clock, which determines when the data value is written. The output from a state element provides the value that was written in an earlier clock cycle. For example, one of the logically simplest state elements is a D-type flip-flop (see [Appendix 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.

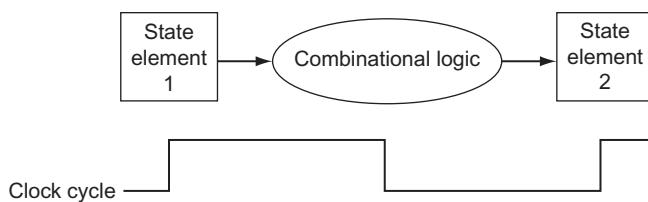
### Clocking Methodology

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.

For simplicity, we will assume an **edge-triggered clocking** methodology. An edge-triggered clocking methodology means that any values stored in a sequential logic element are updated only on a clock edge, which is a quick transition from low to high or vice versa (see [Figure 4.3](#)). Because only state elements can store a data value, any collection of combinational logic must have its inputs come from a set of state elements and its outputs written into a set of state elements. The inputs are values that were written in a previous clock cycle, while the outputs are values that can be used in a following clock cycle.

**clocking methodology** The approach used to determine when data are valid and stable relative to the clock.

**edge-triggered clocking** A clocking scheme in which all state changes occur on a clock edge.



**FIGURE 4.3 Combinational logic, state elements, and the clock are closely related.** In a synchronous digital system, the clock determines when elements with state will write values into internal storage. Any inputs to a state element must reach a stable value (that is, have reached a value from which they will not change until after the clock edge) before the active clock edge causes the state to be updated. All state elements in this chapter, including memory, are assumed positive edge-triggered; that is, they change on the rising clock edge.

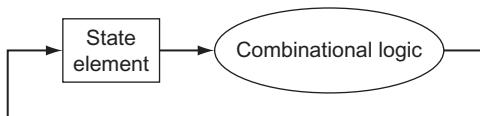
**Figure 4.3** shows the two state elements surrounding a block of combinational logic, which operates in a single clock cycle: all signals must propagate from state element 1, through the combinational logic, and to state element 2 in the time of one clock cycle. The time necessary for the signals to reach state element 2 defines the length of the clock cycle.

For simplicity, we do not show a write **control signal** when a state element is written on every active clock edge. In contrast, if a state element is not updated on every clock, then an explicit write control signal is required. Both the clock signal and the write control signal are inputs, and the state element is changed only when the write control signal is asserted and a clock edge occurs.

We will use the word **asserted** to indicate a signal that is logically high and *assert* to specify that a signal should be driven logically high, and **deasserted** or **deasserted** to represent logically low. We use the terms assert and deassert because when we implement hardware, at times 1 represents logically high and at times it can represent logically low.

An edge-triggered methodology allows us to read the contents of a register, send the value through some combinational logic, and write that register in the same clock cycle. **Figure 4.4** gives a generic example. It doesn't matter whether we assume that all writes take place on the rising clock edge (from low to high) or on the falling clock edge (from high to low), since the inputs to the combinational logic block cannot change except on the chosen clock edge. In this book, we use the rising clock edge. With an edge-triggered timing methodology, there is *no* feedback within a single clock cycle, and the logic in **Figure 4.4** works correctly. In [Appendix 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.

**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.

**asserted** The signal is logically high or true.

**deasserted** The signal is logically low or false.

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.

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.

### Check Yourself

**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.

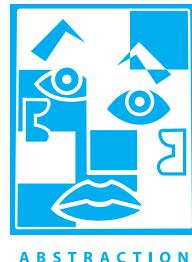
## 4.3 Building a Datapath

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.

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.

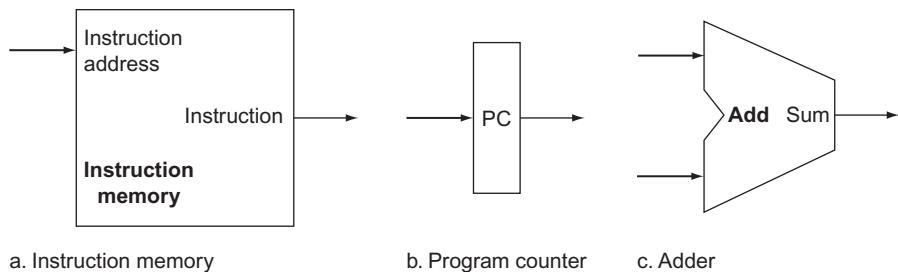
Now let's consider the R-format instructions (see Figure 2.19 on page 120). They all read two registers, perform an ALU operation on the contents of the registers, and write the result to a register. We call these instructions either *R-type instructions* or *arithmetic-logical instructions* (since they perform arithmetic or logical operations). This instruction class includes add, sub, and, and or, which



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 4.5 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.

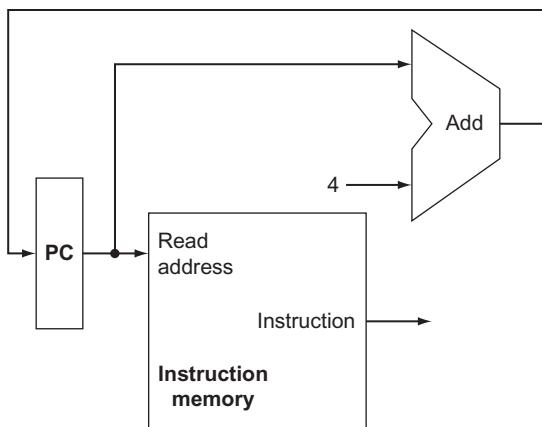
**register file** A state element that consists of a set of registers that can be read and written by supplying a register number to be accessed.

were introduced in [Chapter 2](#). Recall that a typical instance of such an instruction is `add x1, x2, x3`, which reads  $x_2$  and  $x_3$  and writes the sum into  $x_1$ .

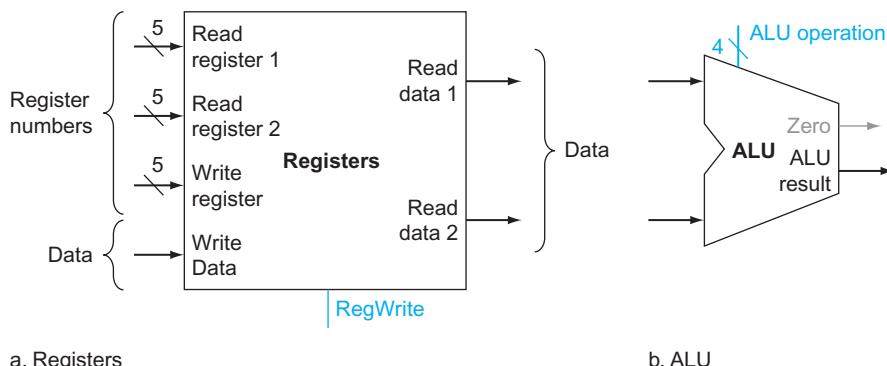
The processor's 32 general-purpose registers are stored in a structure called a **register file**. A register file is a collection of registers in which any register can be read or written by specifying the number of the register in the file. The register file contains the register state of the computer. 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  $x_2$ , 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  $x_1$ . If the instruction is a load, the value read from memory must be written into the register file in the specified register, which is  $x_1$ . 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.

#### 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.

#### 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 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.

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.

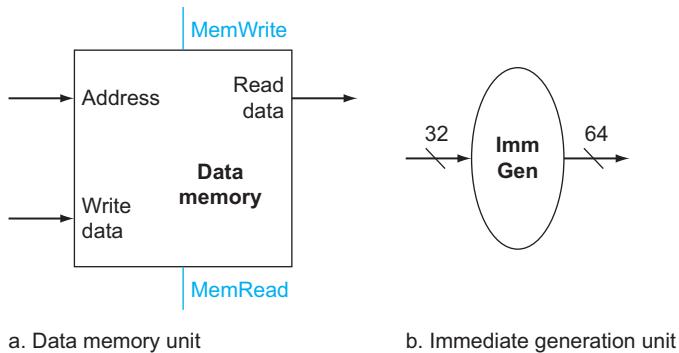
The `beq` instruction has three operands, two registers that are compared for equality, and a 12-bit offset used to compute the **branch target address** relative to the branch instruction address. Its form is `beq x1, x2, offset`. To implement this instruction, we must compute the branch target address 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.

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**.

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



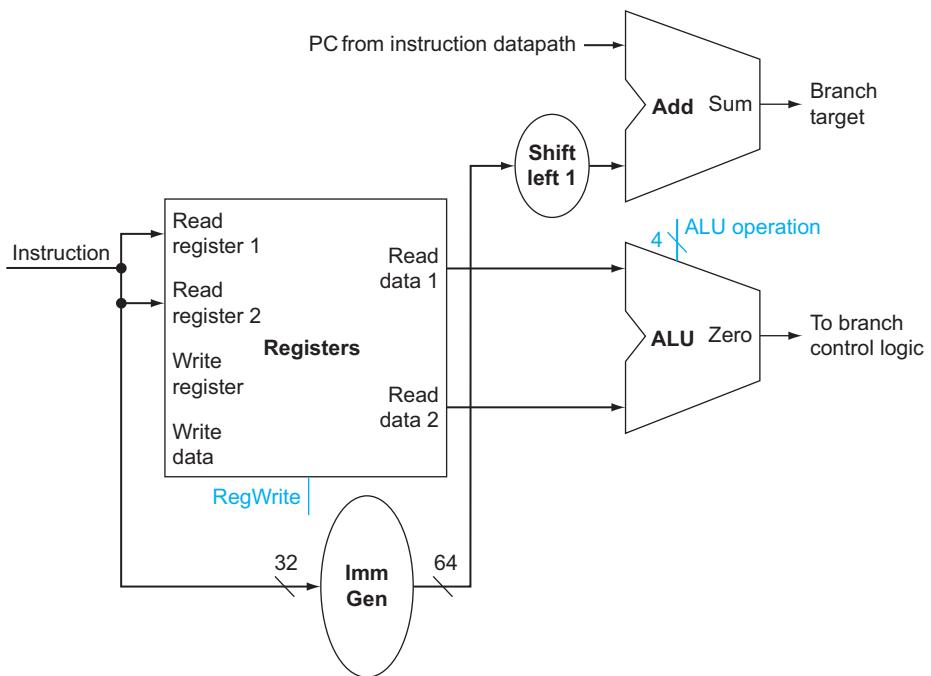
**FIGURE 4.8 The two units needed to implement loads and stores, in addition to the register file and ALU of Figure 4.7, are the data memory unit and the immediate generation unit.** The memory unit is a state element with inputs for the address and the write data, and a single output for the read result. 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  $0_{\text{two}}$  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.

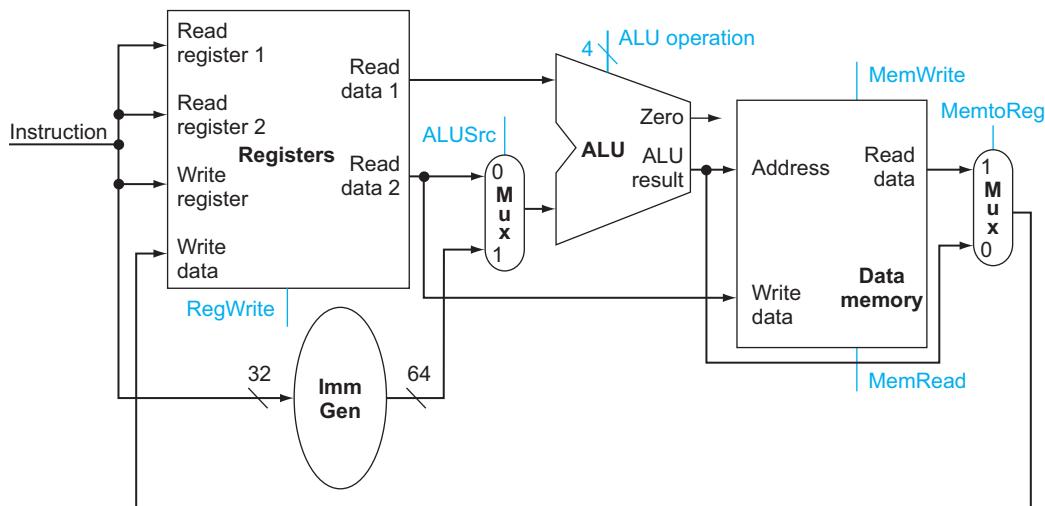
- The value stored into a destination register comes from the ALU (for an R-type instruction) or the memory (for a load).

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.

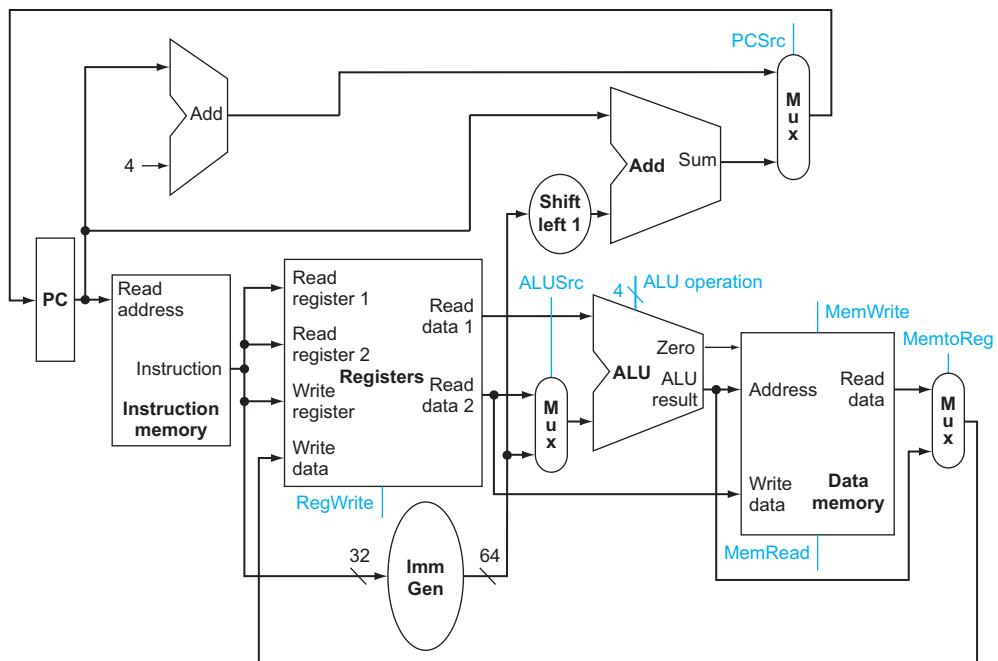
To create a datapath with only a single register file and a single ALU, we must support two different sources for the second ALU input, as well as two different sources for the data stored into the register file. Thus, one multiplexor is placed at the ALU input and another at the data input to the register file. [Figure 4.10](#) shows the operational portion of the combined datapath.

## ANSWER

Now we can combine all the pieces to make a simple datapath for the core 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](#).)

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.

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

**truth table** From logic, a representation of a logical operation by listing all the values of the inputs and then in each case showing what the resulting outputs should be.

**don't-care term** An element of a logical function in which the output does not depend on the values of all the inputs. Don't-care terms may be specified in different ways.

ALUOp		Funct7 field										Funct3 field			Operation
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)	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 ( $\text{opcode} = 51_{\text{ten}}$ ), 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 ( $\text{opcode} = 3_{\text{ten}}$ ). 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 ( $\text{opcode} = 35_{\text{ten}}$ ). 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 ( $\text{opcode} = 99_{\text{ten}}$ ). 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.

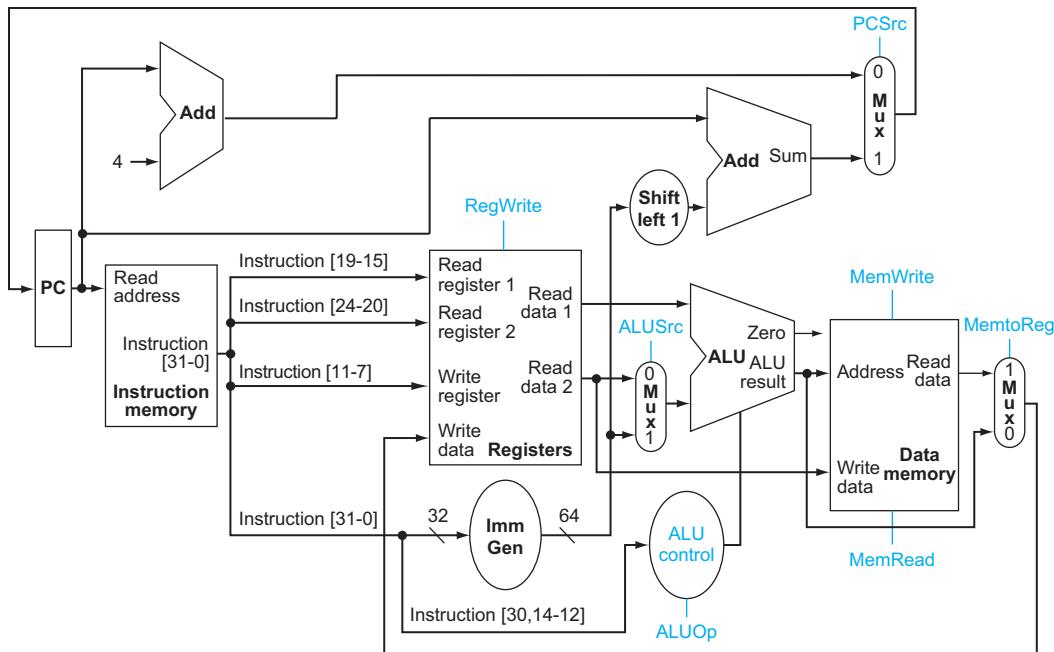
the formats of the four instruction classes: arithmetic, load, store, and conditional branch instructions. Figure 4.14 shows these formats.

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.)

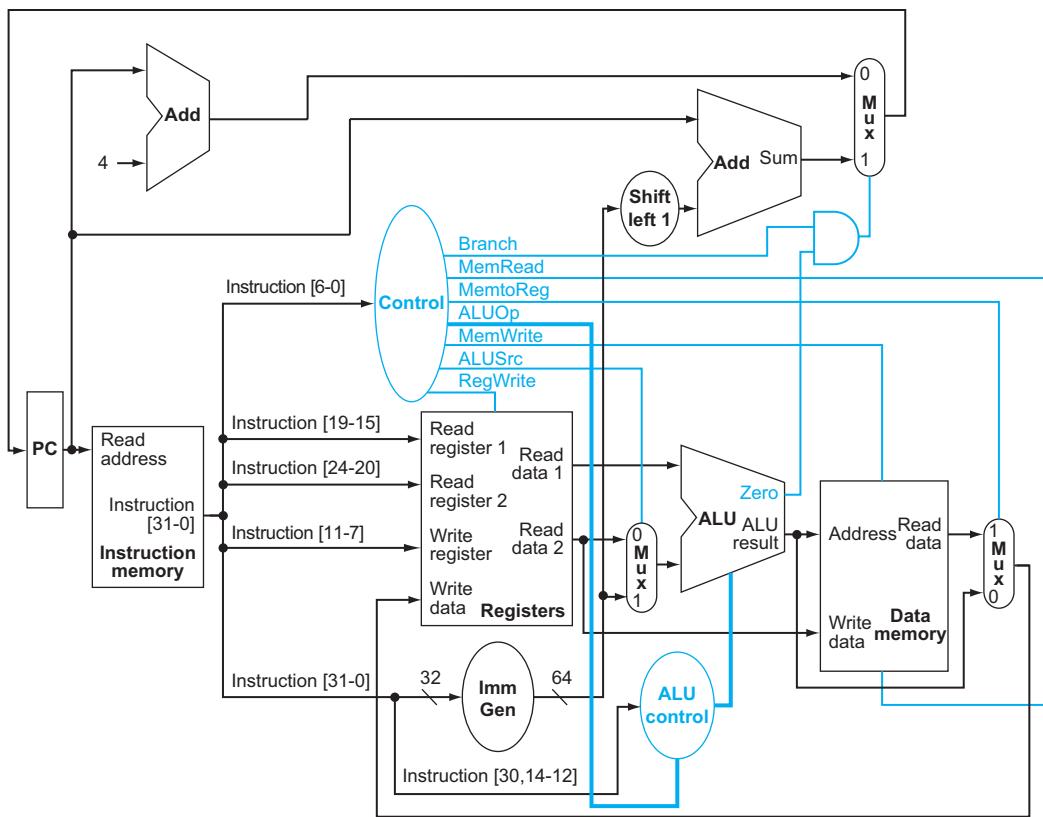
These eight control signals (six from [Figure 4.16](#) and two for ALUOp) can now be set based on the input signals to the control unit, which are the opcode bits 6:0. [Figure 4.17](#) shows the datapath with the control unit and the control signals.

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  $\text{add } x_1, x_2, x_3$ . Although everything occurs in one clock cycle, we can think



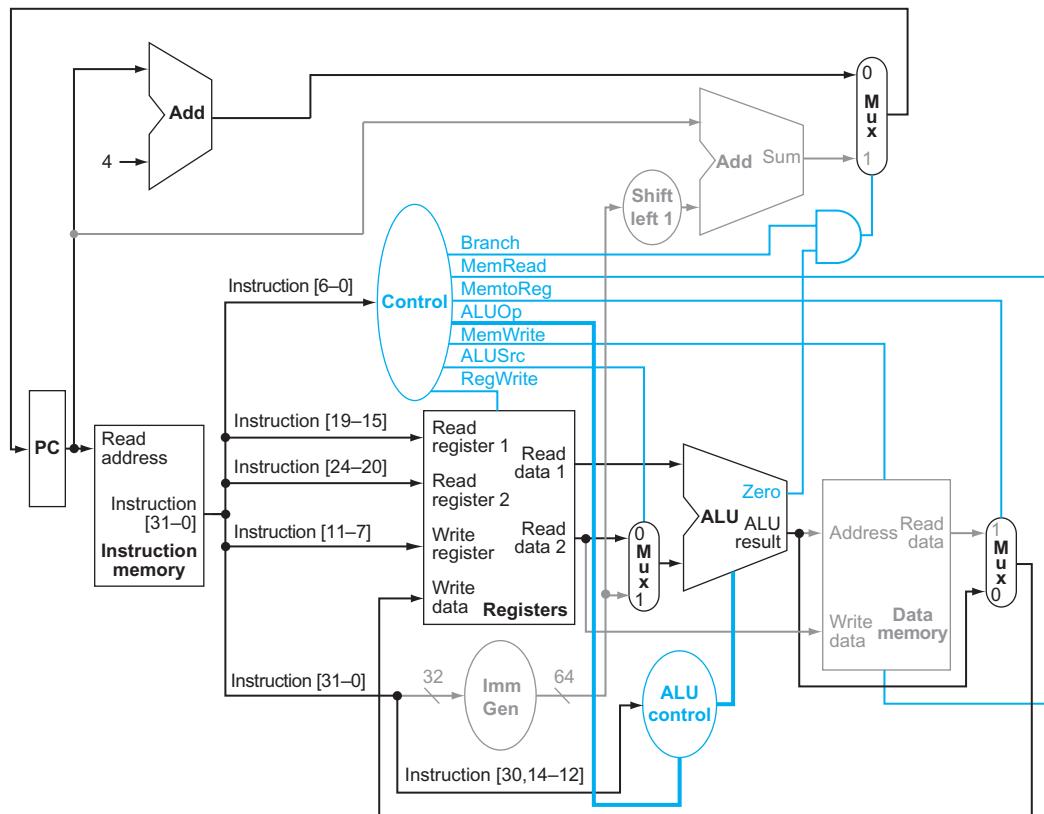
**FIGURE 4.17 The simple datapath with the control unit.** The input to the control unit is the 7-bit opcode field from the instruction. The outputs of the control unit consist of two 1-bit signals that are used to control multiplexors (ALUSrc and MemtoReg), three signals for controlling reads and writes 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,  $x_2$  and  $x_3$ , 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 ( $x_1$ ) 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.



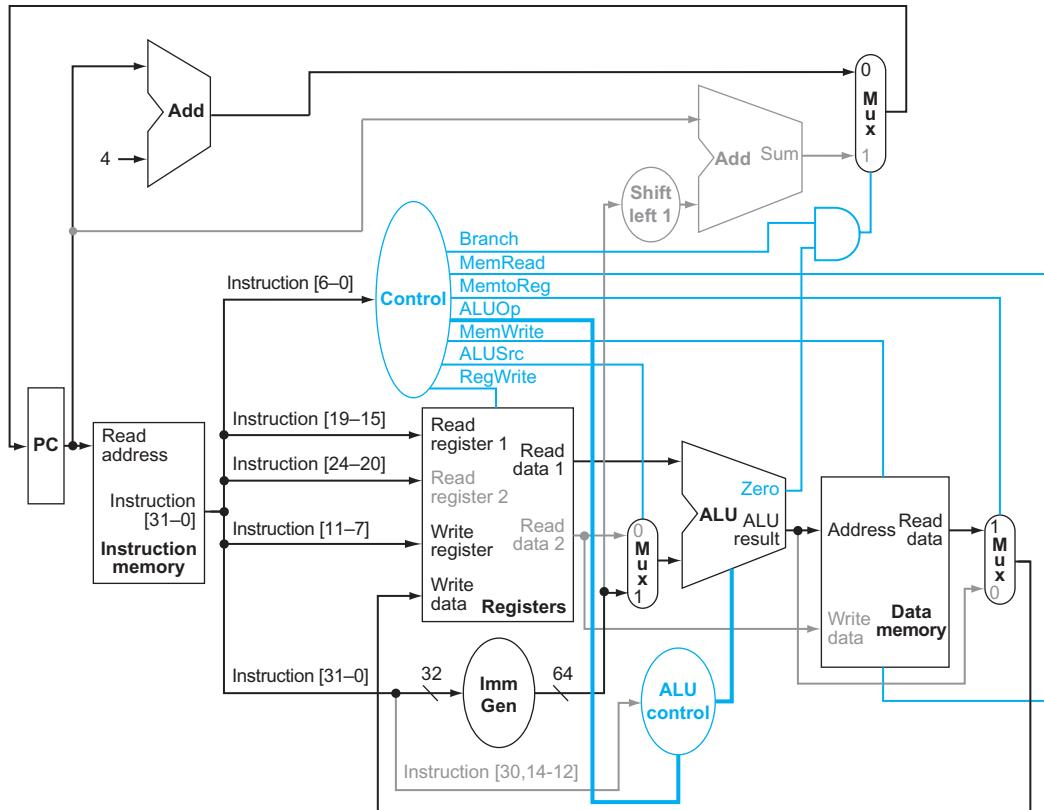
**FIGURE 4.19 The datapath in operation for an R-type instruction, such as `add x1, x2, x3`.** The control lines, datapath units, and connections that are active are highlighted.

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):

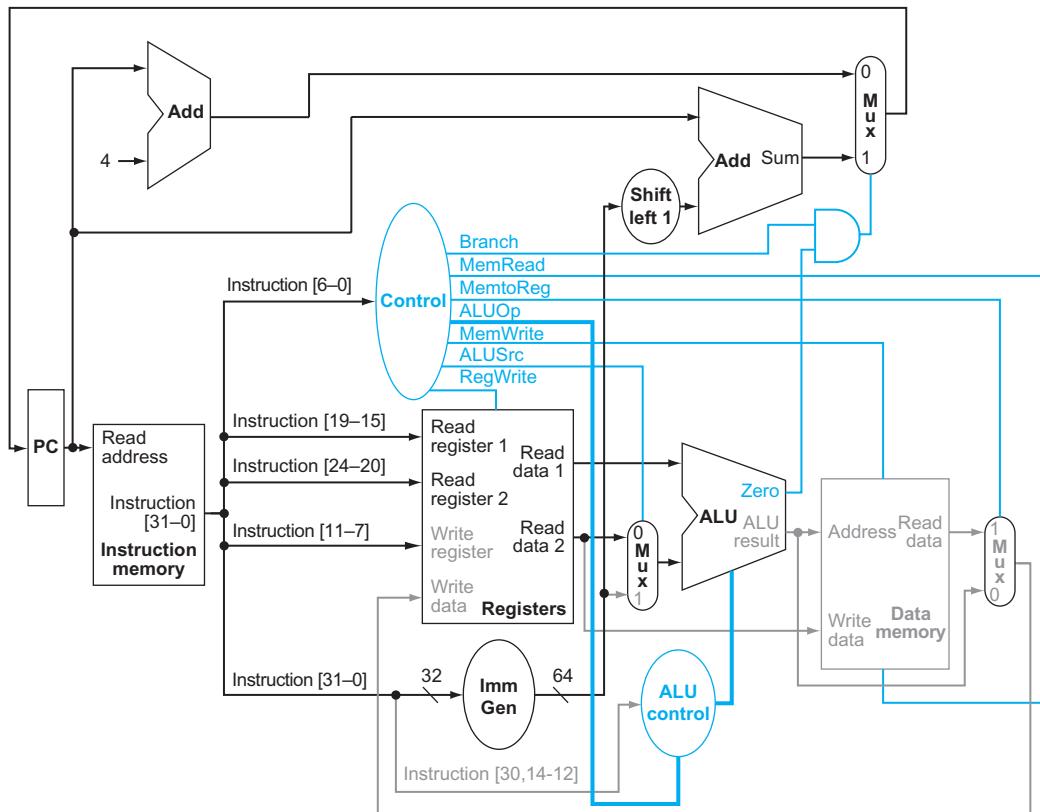
1. An instruction is fetched from the instruction memory, and the PC is incremented.
2. A register ( $x_2$ ) value is read from the register file.
3. The ALU computes the sum of the value read from the register file and the sign-extended 12 bits of the instruction (offset).
4. The sum from the ALU is used as the address for the data memory.
5. The data from the memory unit is written into the register file ( $x_1$ ).



**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  $\text{PC} + 4$  or the branch target address. Figure 4.21 shows the four steps in execution:

1. An instruction is fetched from the instruction memory, and the PC is incremented.
2. Two registers,  $x_1$  and  $x_2$ , 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  $Op4 \cdot Op5$ , 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.



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

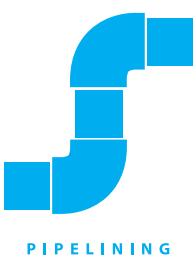
Look at the control signals in [Figure 4.22](#). Can you combine any together? Can any control signal output in the figure be replaced by the inverse of another? (Hint: take into account the don't cares.) If so, can you use one signal for the other without adding an inverter?

## 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](#).

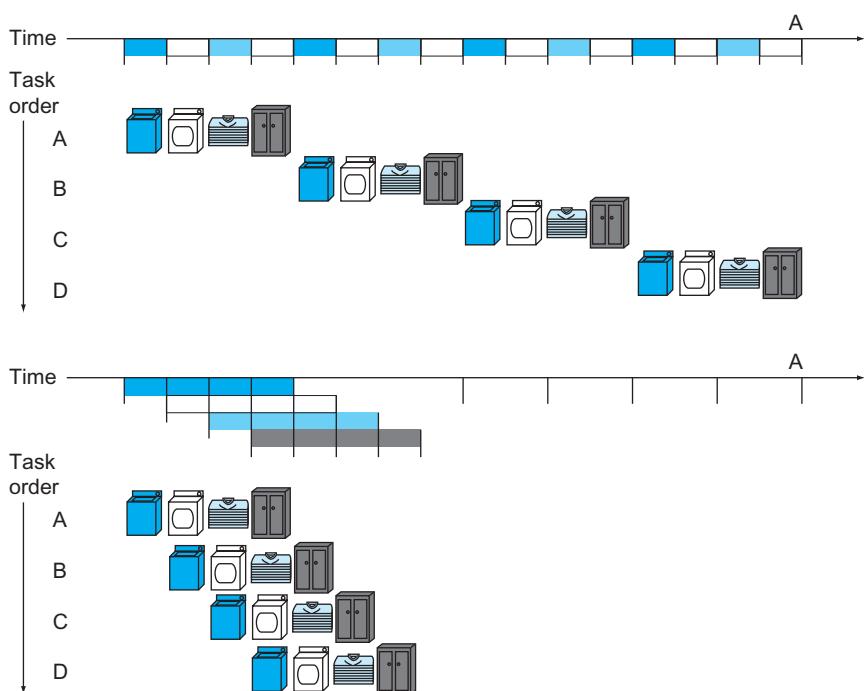
Anyone who has done a lot of laundry has intuitively used pipelining. The *non-pipelined* approach to laundry would be as follows:

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.

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  $3 \times 800$  ps or 2400 ps.

## ANSWER

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  $3 \times 200$  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

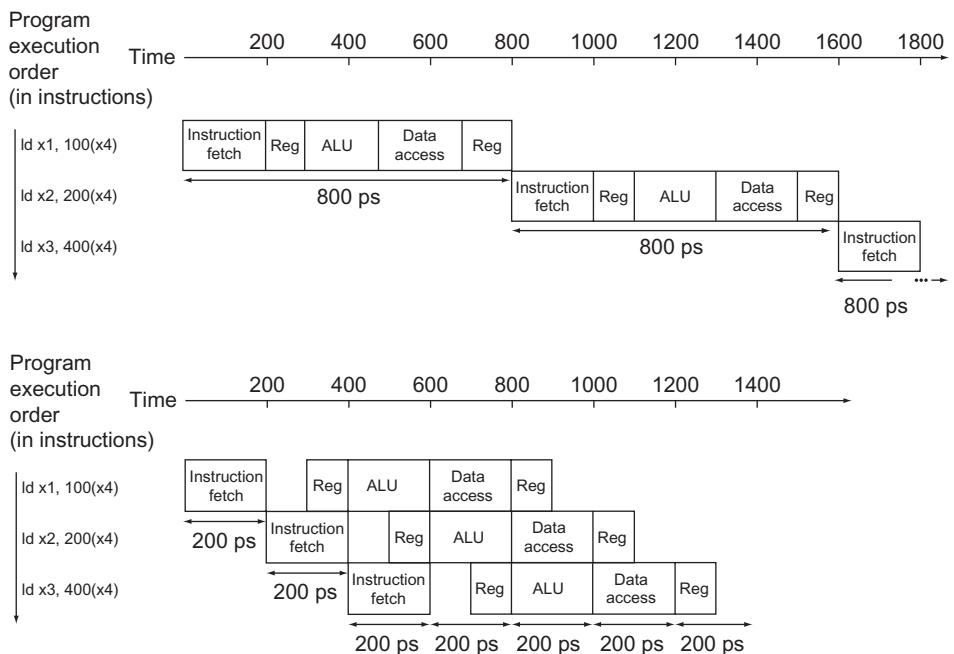
$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of pipe stages}}$$

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 ( <code>ld</code> )	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store doubleword ( <code>sd</code> )	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 ( <code>beq</code> )	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  $1,000,000 \times 200 \text{ ps} + 1400 \text{ ps}$ , or 200,001,400 ps. In the nonpipelined example, we would add 1,000,000 instructions, each taking 800 ps, so total execution time would be  $1,000,000 \times 800 \text{ ps} + 2400 \text{ ps}$ , or 800,002,400 ps. Under these

conditions, the ratio of total execution times for real programs on nonpipelined to pipelined processors is close to the ratio of times between instructions:

$$\frac{800,002,400 \text{ ps}}{200,001,400 \text{ ps}} \approx \frac{800 \text{ ps}}{200 \text{ ps}} = 4.00$$

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

The first hazard is called a **structural hazard**. It means that the hardware cannot support the combination of instructions that we want to execute in the same clock cycle. A structural hazard in the laundry room would occur if we used a washer-dryer combination instead of a separate washer and dryer, or if our roommate was busy doing something else and wouldn't put clothes away. Our carefully scheduled pipeline plans would then be foiled.

**structural hazard** When a planned instruction cannot execute in the proper clock cycle because the hardware does not support the combination of instructions that are set to execute.

As we said above, the 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.

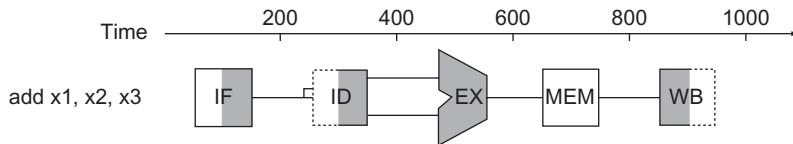
Although we could try to rely on compilers to remove all such hazards, the results would not be satisfactory. These dependences happen just too often and the delay is far too long to expect the compiler to rescue us from this dilemma.

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](#).



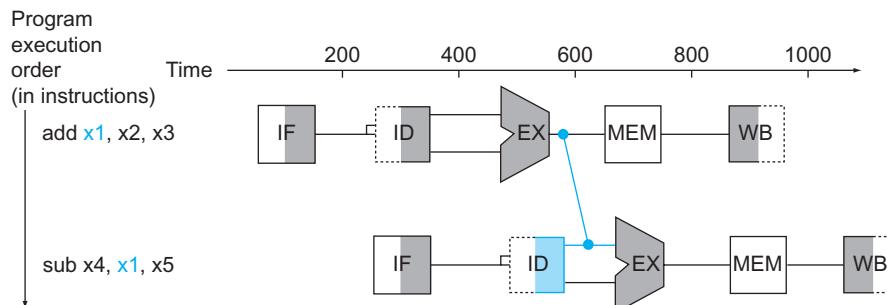
**FIGURE 4.26 Graphical representation of the instruction pipeline, similar in spirit to the laundry pipeline in Figure 4.23.** Here we use symbols representing the physical resources with the abbreviations for pipeline stages used throughout the chapter. The symbols for the five stages: *IF* for the 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.

Figure 4.27 shows the connection to forward the value in  $x_1$  after the execution stage of the *add* instruction as input to the execution stage of the *sub* instruction.

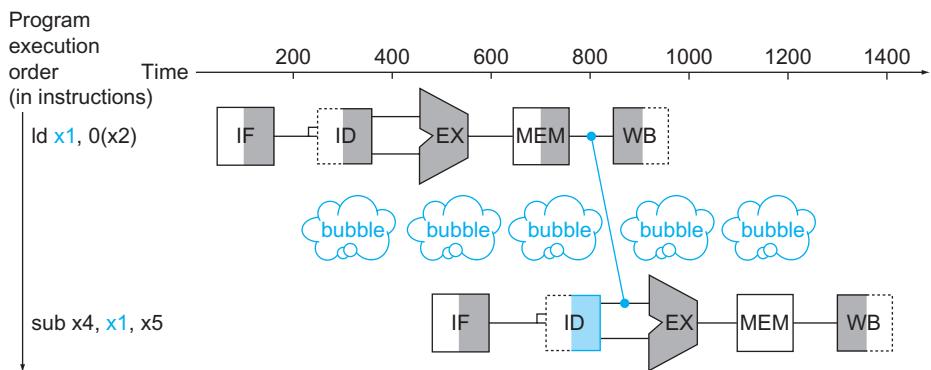
**ANSWER**

In this graphical representation of events, forwarding paths are valid only if the destination stage is later in time than the source stage. For example, there cannot be a valid forwarding path from the output of the memory access stage in the first instruction to the input of the execution stage of the following, since that would mean going backward in time.

Forwarding works very well and is described in detail in Section 4.7. It cannot prevent all pipeline stalls, however. For example, suppose the first instruction was a load of  $x_1$  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  $x_1$  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.

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.

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.

**ANSWER**

**control hazard** Also called **branch hazard**. When the proper instruction cannot execute in the proper pipeline clock cycle because the instruction that was fetched is not the one that is needed; that is, the flow of instruction addresses is not what the pipeline expected.

The equivalent decision task in a computer is the 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.

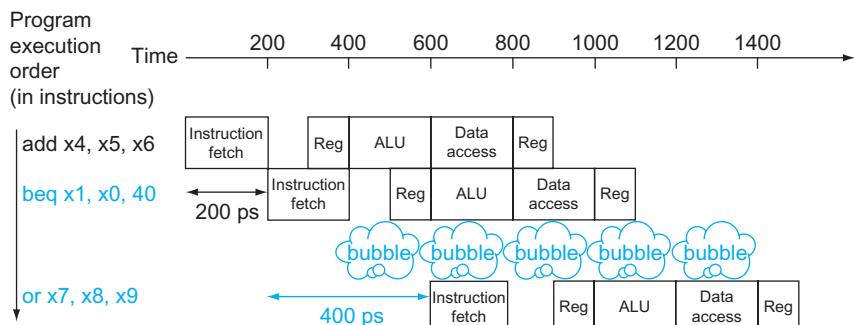
## EXAMPLE

## ANSWER

### 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.

[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.



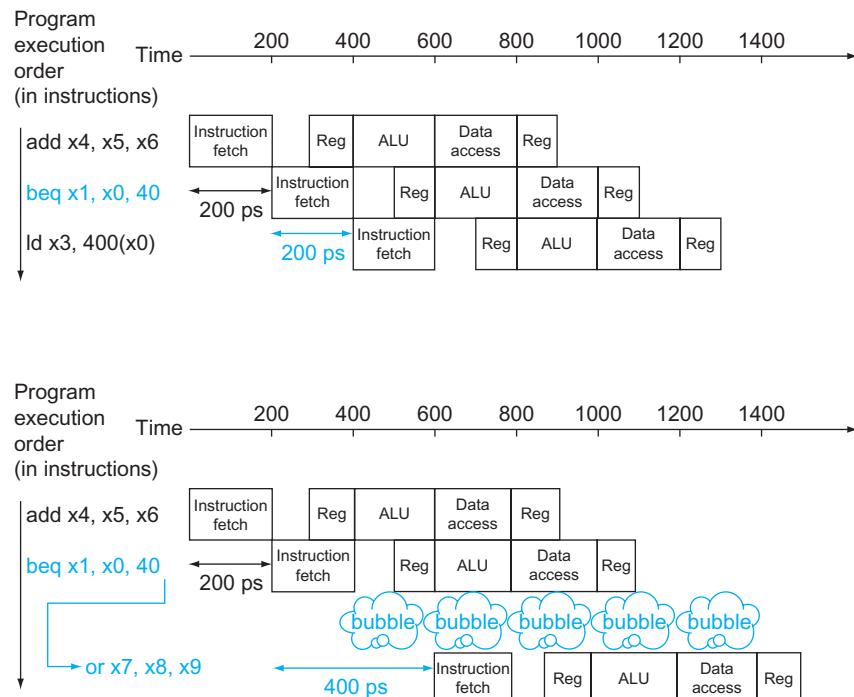
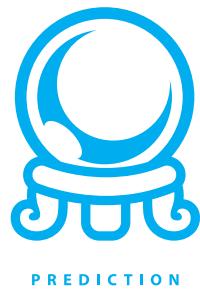
**FIGURE 4.29 Pipeline showing stalling on every conditional branch as solution to control hazards.** This example assumes the conditional branch is taken, and the instruction at the destination of the branch is the `or` instruction. There is a one-stage pipeline stall, or bubble, after the branch. In 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.

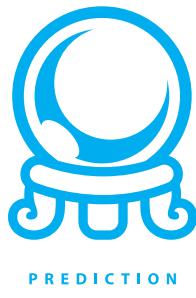
Computers do indeed use **prediction** to handle conditional branches. One simple approach is to predict always that conditional branches will be untaken. When you're right, the pipeline proceeds at full speed. Only when conditional branches are taken does the pipeline stall. [Figure 4.30](#) shows such 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.



PREDICTION

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.

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.

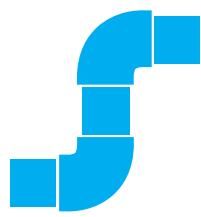
Alternatively, if you are interested in understanding how pipelining is implemented and the challenges of dealing with hazards, you can proceed to examine the design of a pipelined datapath and the basic control, explained in [Section 4.6](#). You can then use this understanding to explore the implementation 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](#).

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



PARALLELISM



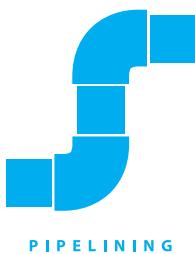
PIPELINING

## Check Yourself

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

## Understanding Program Performance



## The BIG Picture

**latency (pipeline)** The number of stages in a pipeline or the number of stages between two instructions during execution.



*There is less in this than meets the eye.*

Tallulah Bankhead, remark to Alexander Woollcott, 1922

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.

**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.

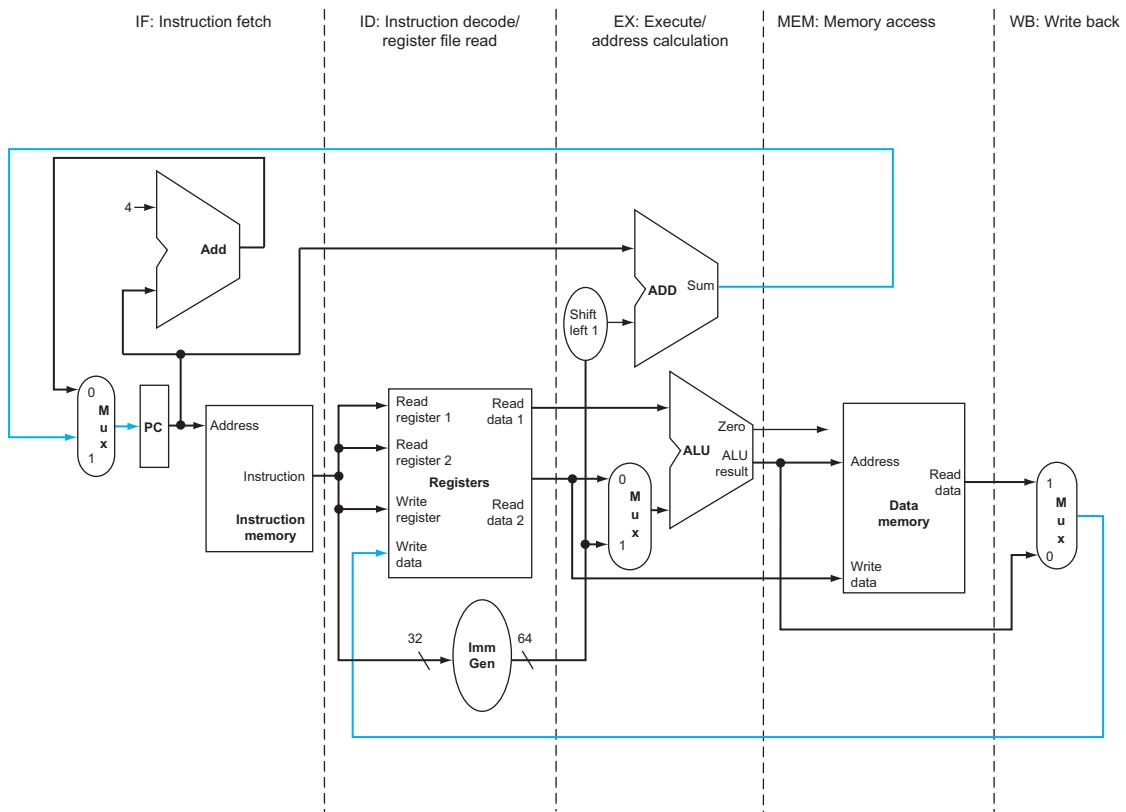
### 4.6

## 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

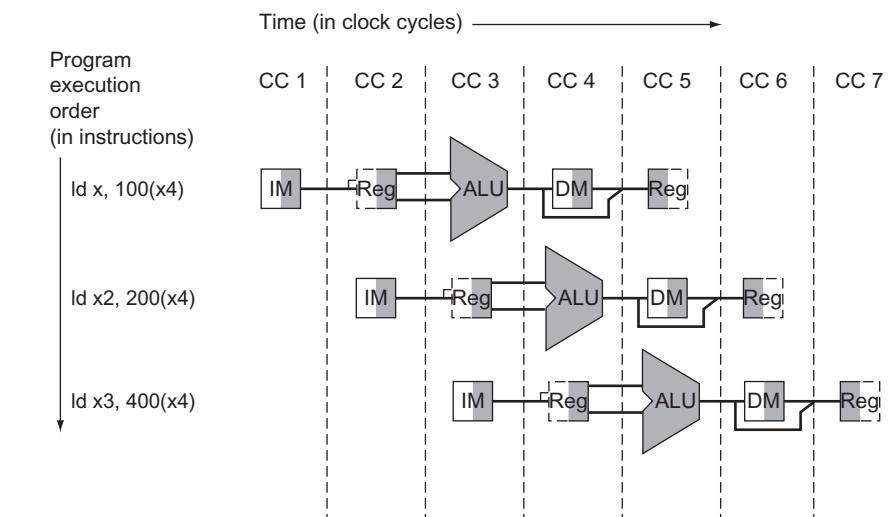
Data flowing from right to left do not affect the current instruction; these reverse data movements influence only later instructions in the pipeline. Note that the first

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.



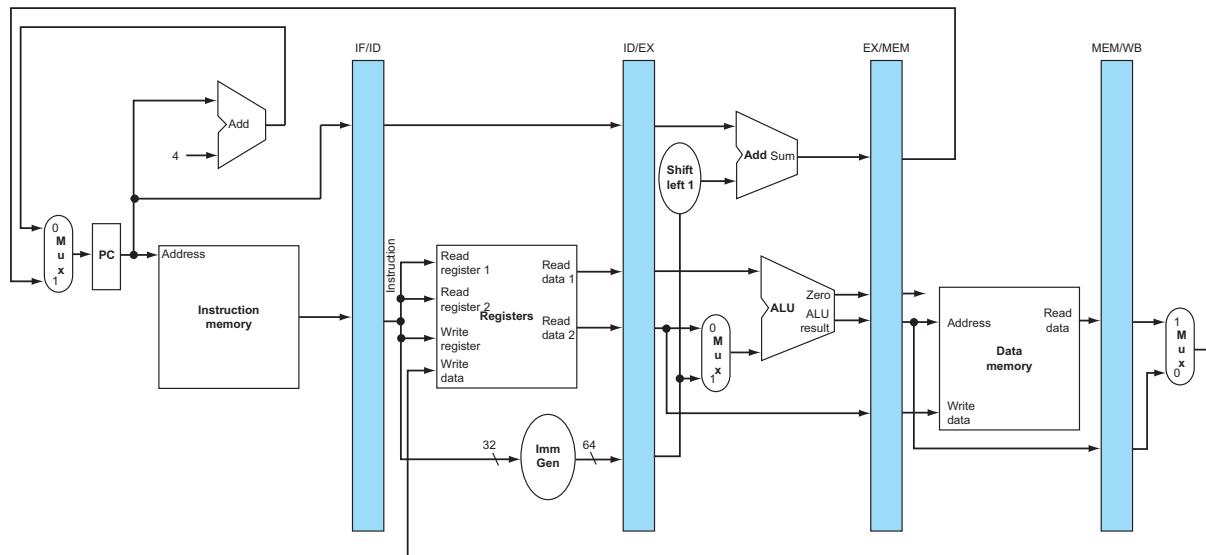
**FIGURE 4.32 Instructions being executed using the single-cycle datapath in Figure 4.31, assuming pipelined execution.** Similar to Figures 4.26 through 4.28, this figure pretends that each instruction has its own datapath, and shades each portion according to use. Unlike those figures, each stage is 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.

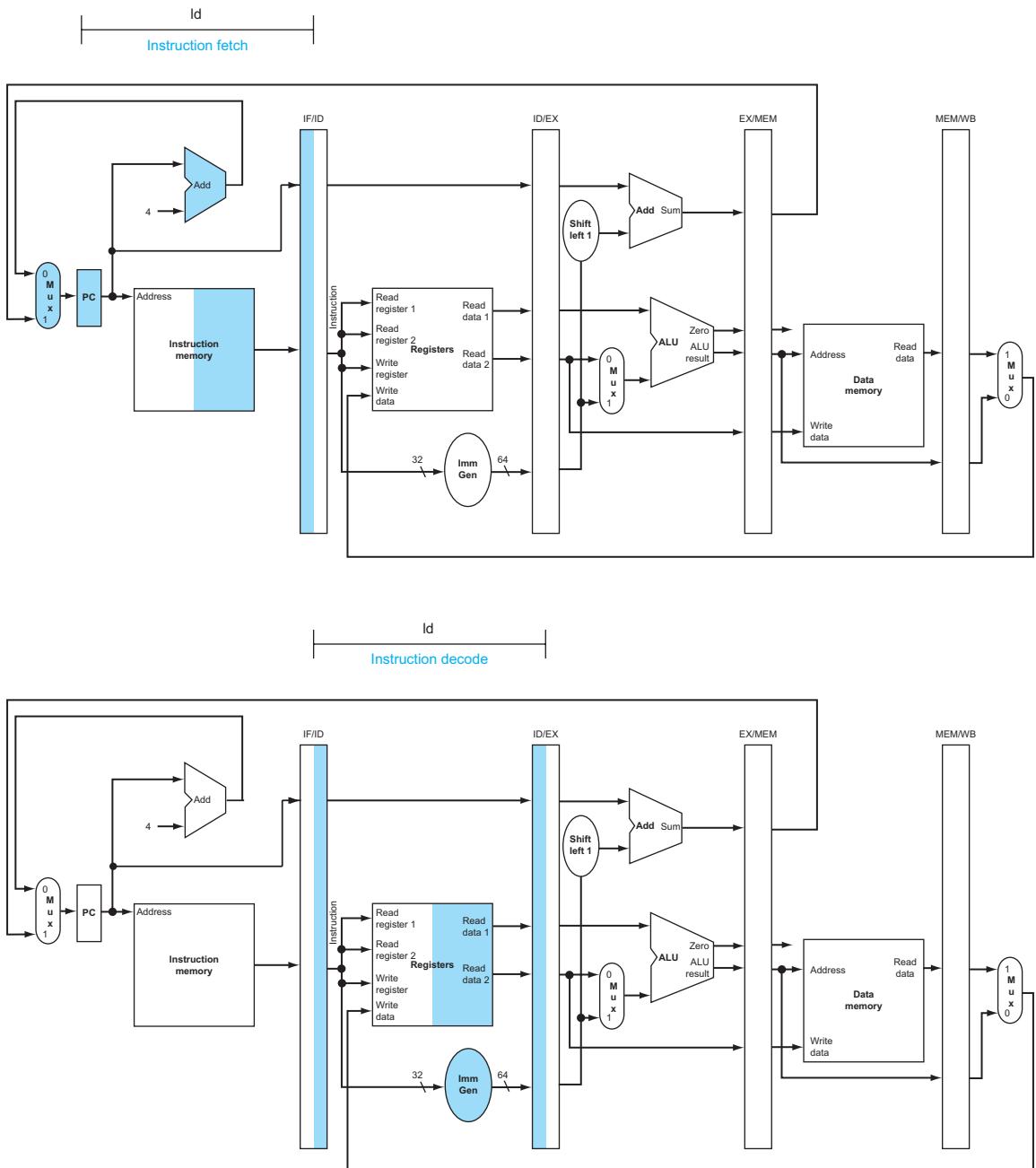
less time than it might appear, because you can compare them to see what changes occur in each clock cycle. [Section 4.7](#) describes what happens when there are data hazards between pipelined instructions; ignore them for now.

[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*.

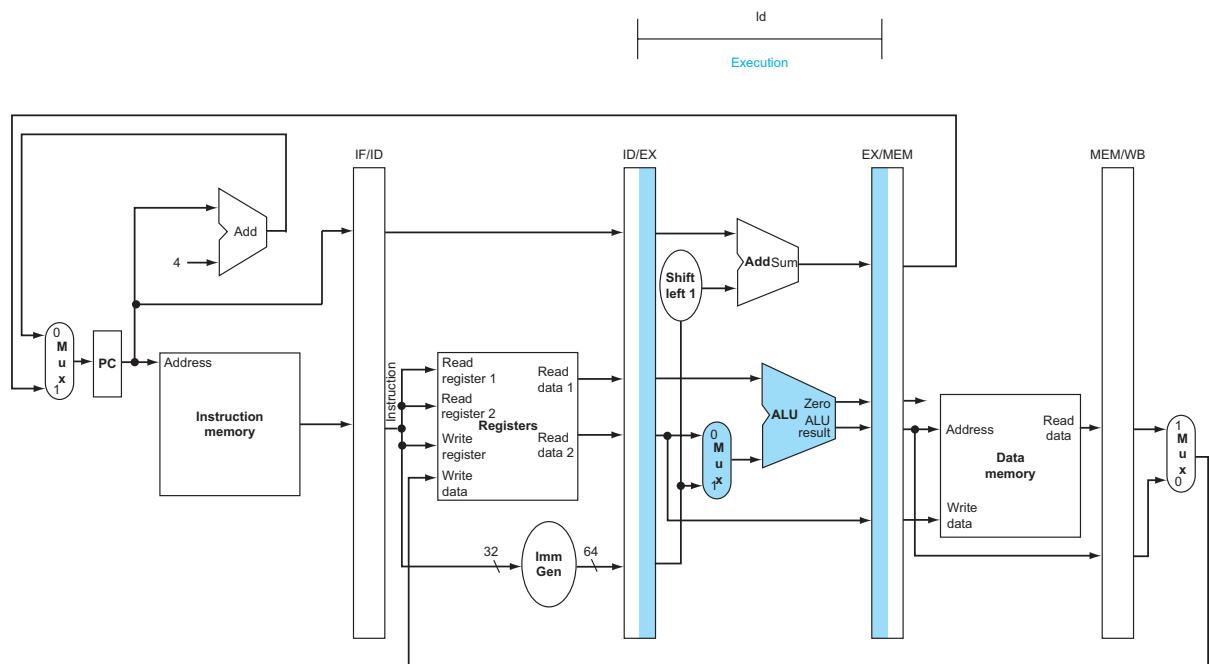
We show the instruction `ld` with the name of the pipe stage that is active in each figure. The five stages are the following:

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 `bne`. 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:

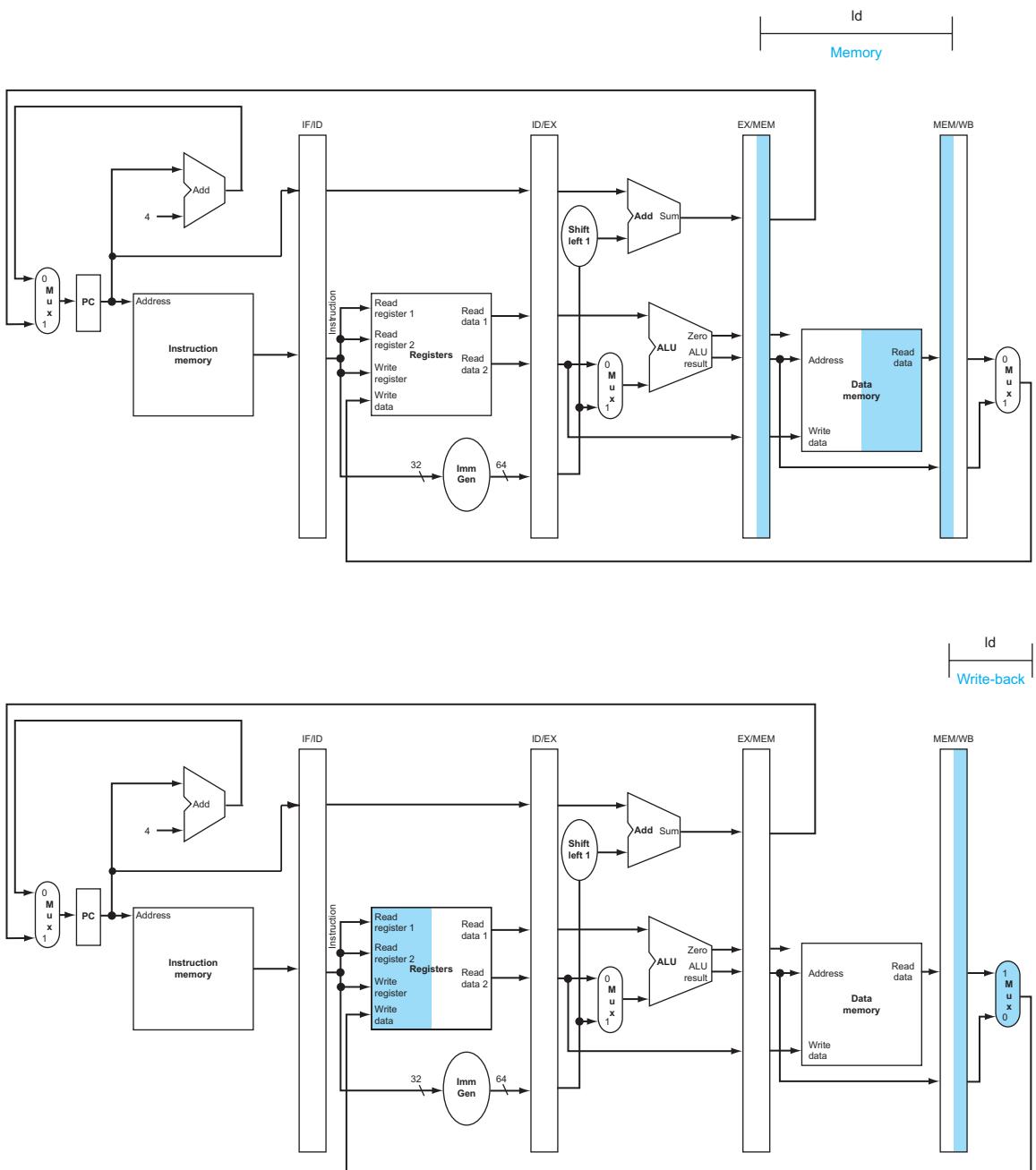


**FIGURE 4.34 IF and ID: First and second pipe stages of an instruction, with the active portions of the datapath in Figure 4.33 highlighted.** The highlighting convention is the same as that used in Figure 4.26. As in Section 4.2, there is no confusion when reading and writing registers, because the 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.

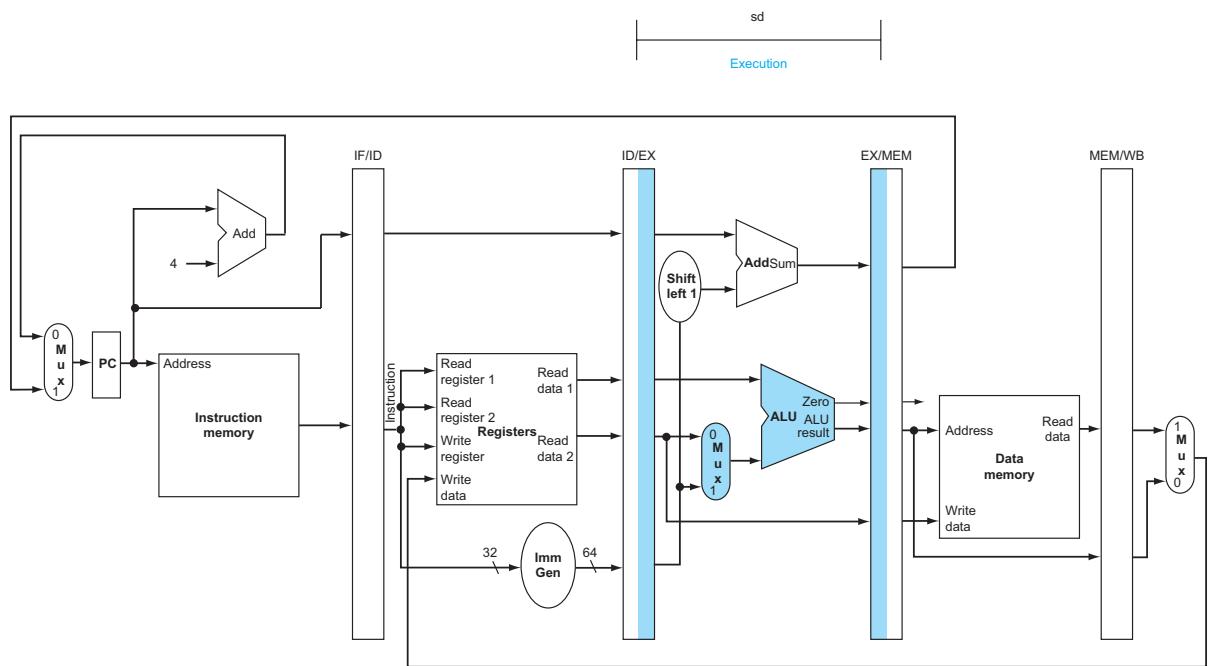


**FIGURE 4.35 EX: The third pipe stage of a load instruction, highlighting the portions of the datapath in Figure 4.33 used in this pipe stage.** The register is added to the sign-extended immediate, and the sum is placed in the EX/MEM pipeline register.

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.



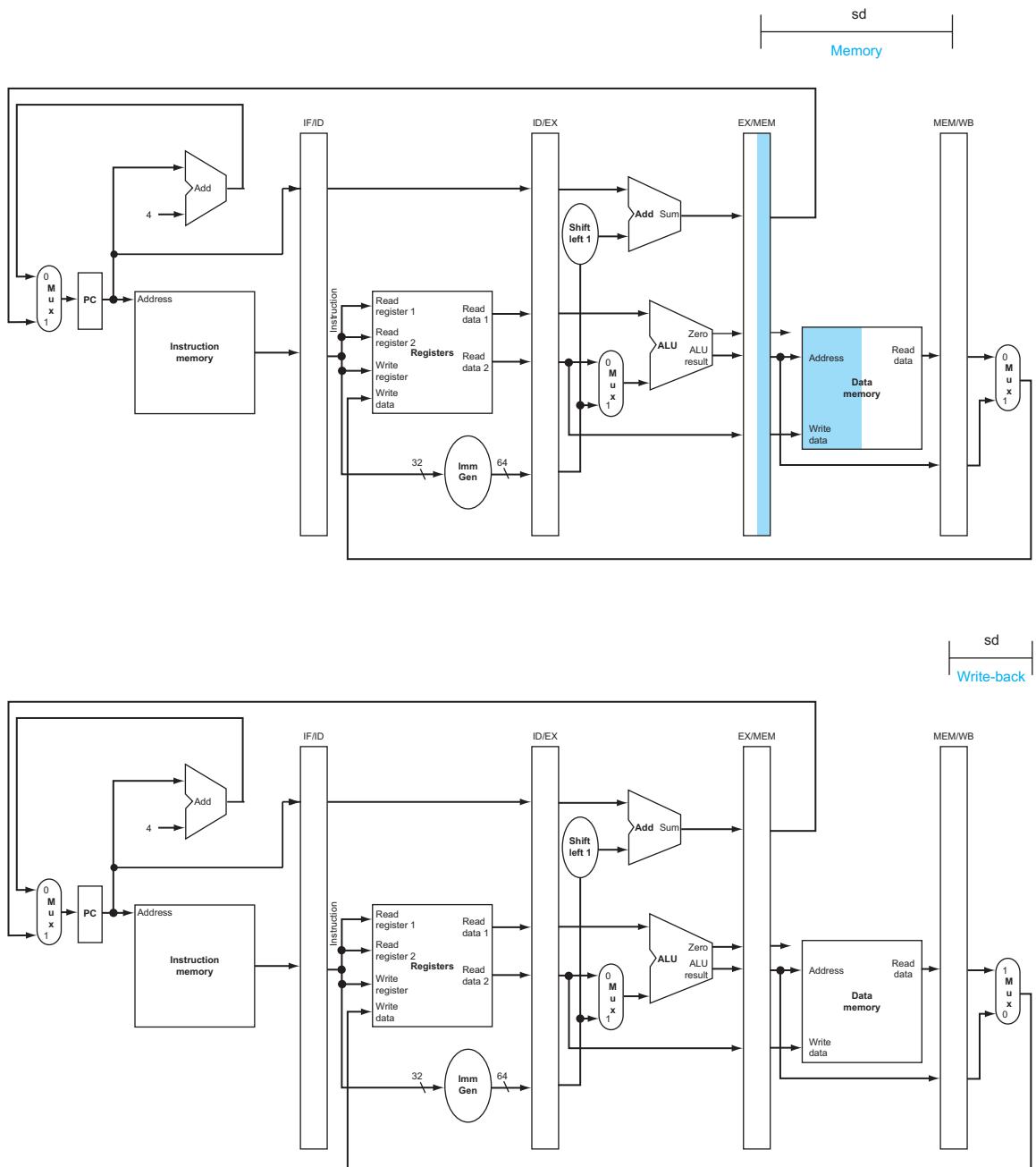
**FIGURE 4.37 EX: The third pipe stage of a store instruction.** Unlike the third stage of the load instruction in Figure 4.35, the second register value is loaded into the EX/MEM pipeline register to be used in the next stage. Although it wouldn't hurt to always write this second register into the EX/MEM pipeline register, we write the second register only on a store instruction to make the pipeline easier to understand.

- 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.

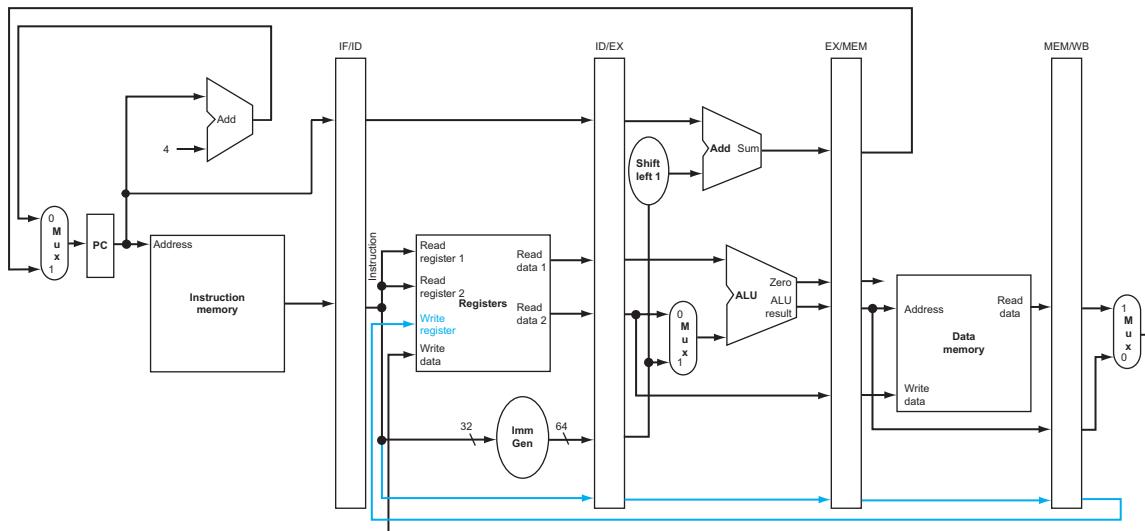
which instruction supplies the write register number? The instruction in the IF/ID pipeline register supplies the write register number, yet this instruction occurs considerably *after* the load instruction!

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



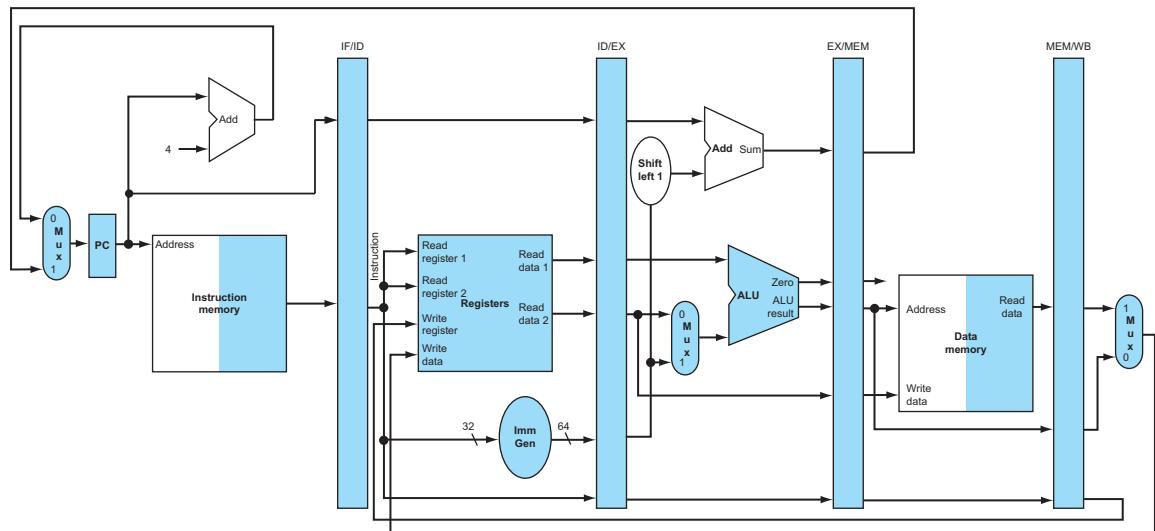
**FIGURE 4.39 The corrected pipelined datapath to handle the load instruction properly.** The write register number now comes from the MEM/WB pipeline register along with the data. The register number is passed from the ID pipe stage until it reaches the MEM/WB pipeline register, adding five more bits 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:

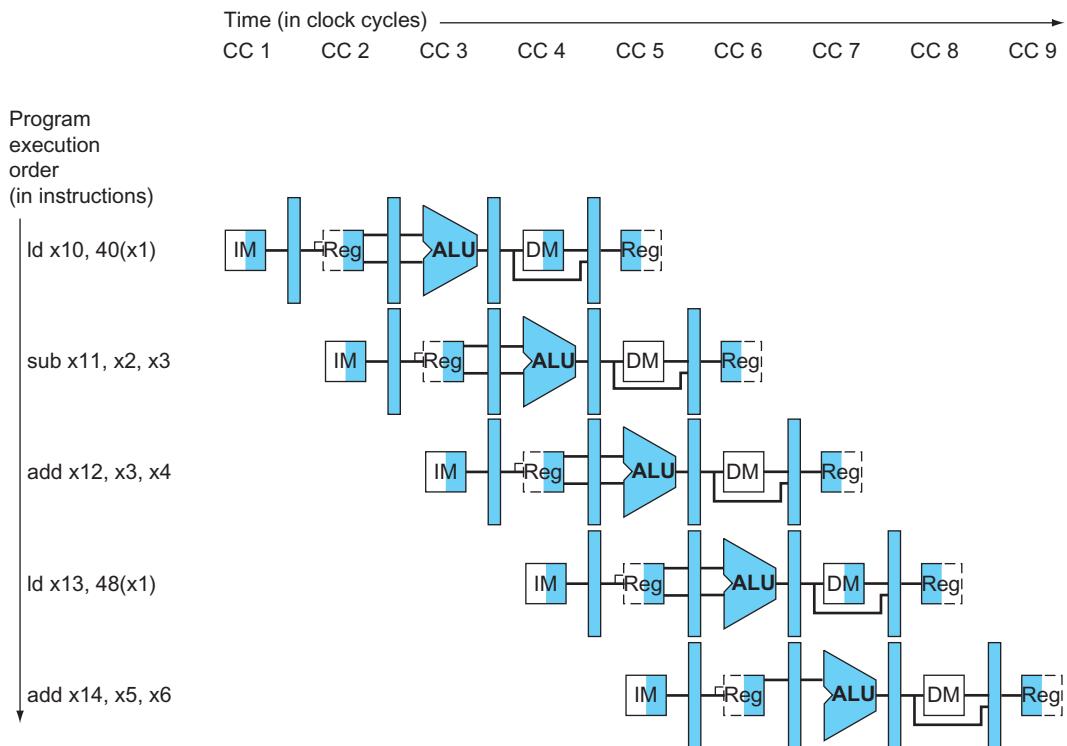
ld	x10, 40(x1)
sub	x11, x2, x3
add	x12, x3, x4
ld	x13, 48(x1)
add	x14, x5, x6

[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.

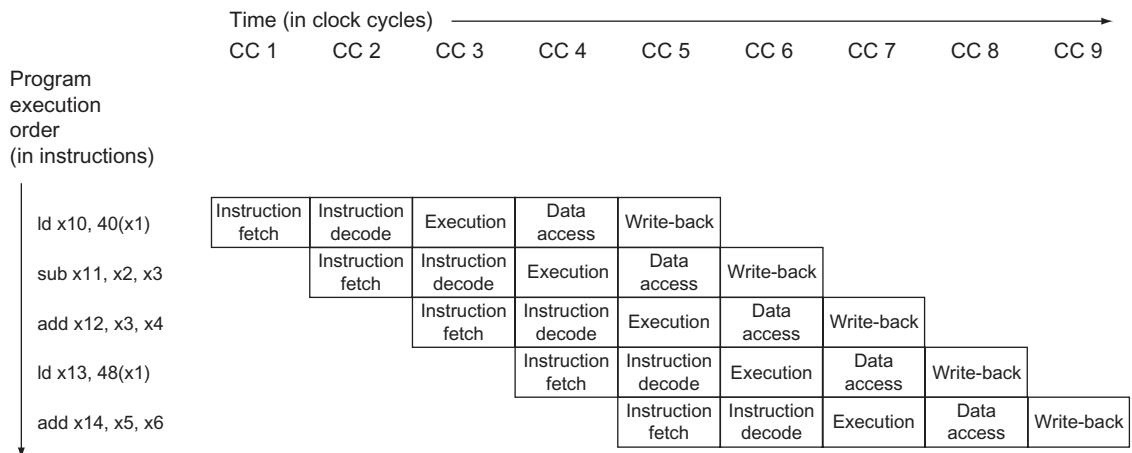


**FIGURE 4.41 Multiple-clock-cycle pipeline diagram of five instructions.** This style of pipeline representation shows the complete execution of instructions in a single figure. Instructions are listed in instruction execution order from top to bottom, and clock cycles move from left to right. Unlike Figure 4.26, here we show the pipeline registers between each stage. Figure 4.42 shows the traditional way to draw this diagram.

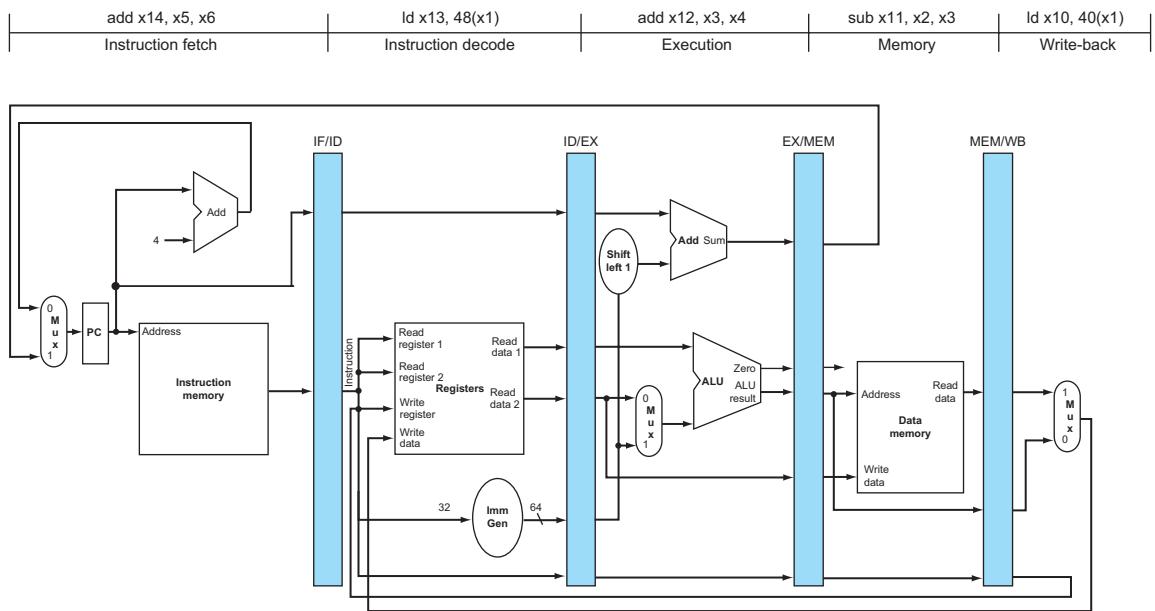
cycle; typically, the drawings appear in groups to show pipeline operation over a sequence of clock cycles. We use multiple-clock-cycle diagrams to give overviews of pipelining situations. (🌐 [Section 4.13](#) gives more illustrations of single-clock diagrams if you would like to see more details about 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.

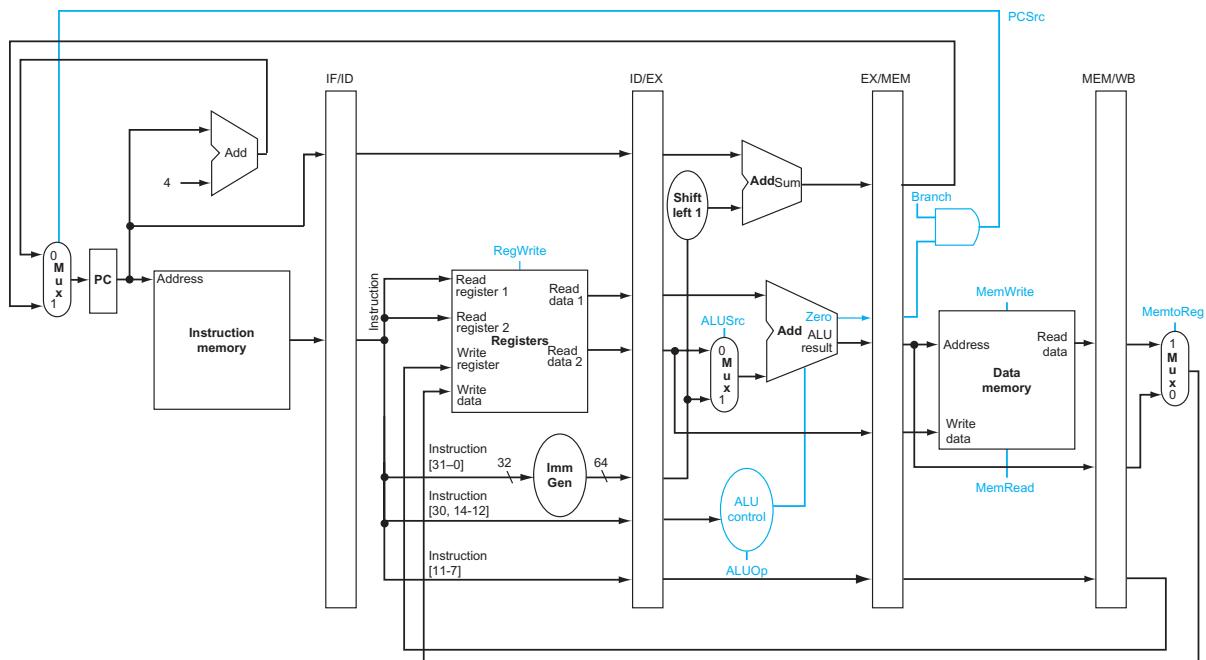
*In the 6600 Computer, perhaps even more than in any previous computer, the control system is the difference.*

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

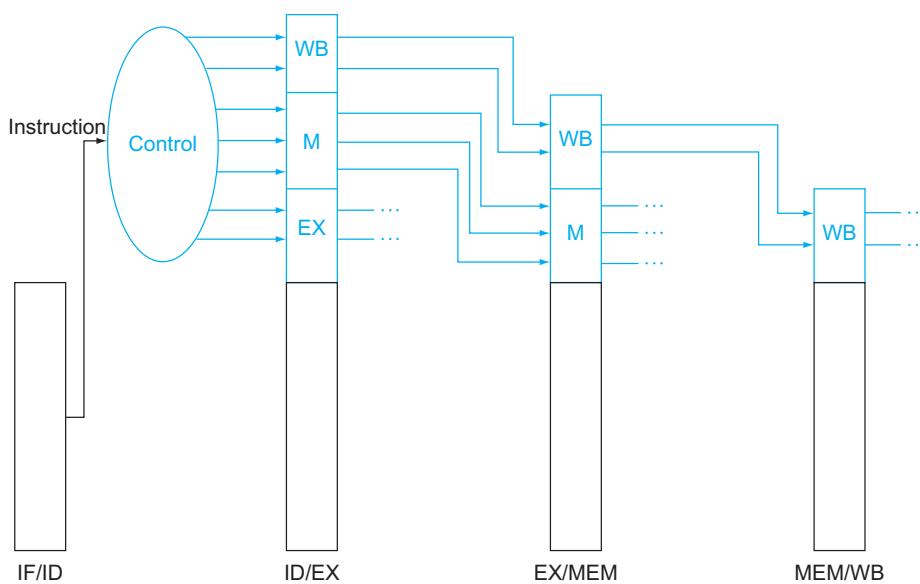
**FIGURE 4.45 A copy of Figure 4.12.** This figure shows how the ALU control bits are set depending on the ALUOp control bits and the different opcodes for the R-type instruction.

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

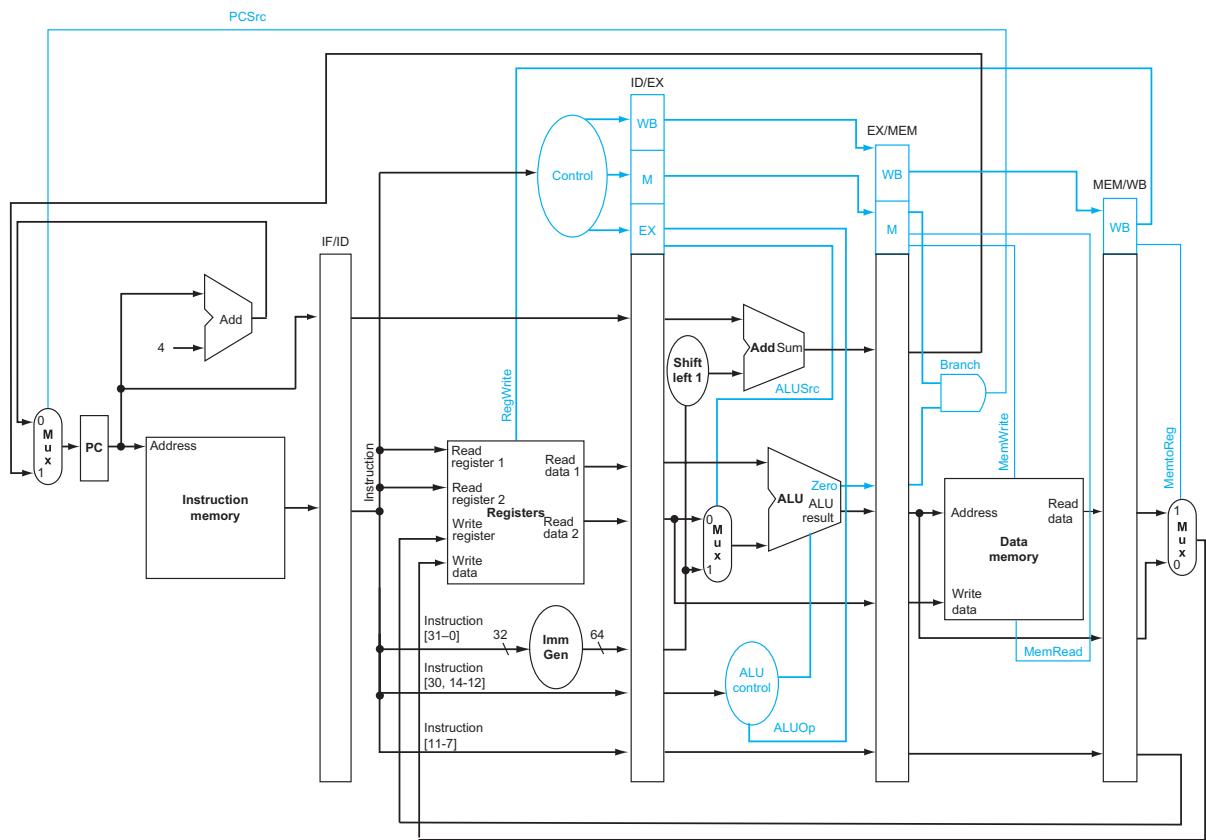
**FIGURE 4.47 The values of the control lines are the same as in Figure 4.18, but they have been shuffled into three groups corresponding to the last three pipeline stages.**



**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  $x_2$  of the first instruction. If register  $x_2$  had the value 10 before the subtract instruction and  $-20$  afterwards, the programmer intends that  $-20$  will be used in the following instructions that refer to register  $x_2$ .

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  $x_2$ , which changes during the middle of clock cycle 5, when the sub instruction writes its result.

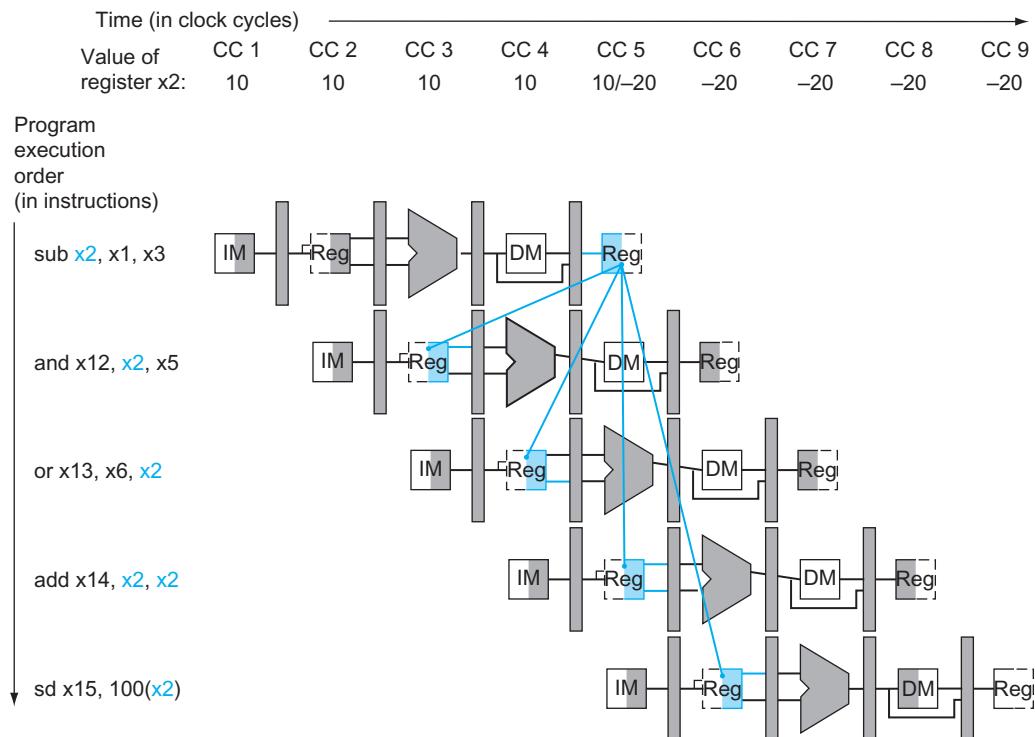
The last potential hazard can be resolved by the design of the register file hardware: What happens when a register is read and written in the same clock cycle? We assume that the write is in the first half of the clock cycle and the read is in the second half, so the read delivers what is written. As is the case for many implementations of register files, we have no data hazard in this case.

[Figure 4.50](#) shows that the values read for register  $x_2$  would *not* be the result of the sub instruction unless the read occurred during clock cycle 5 or later. Thus, the instructions that would get the correct value of  $-20$  are add and 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  $\text{sub } x2, x1, x3$  and the first read operand of  $\text{and } x12, x2, x5$ . 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:

$$\text{EX/MEM.RegisterRd} = \text{ID/EX.RegisterRs1} = x2$$

### Dependence Detection

### EXAMPLE

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
```

As mentioned above, the sub-and is a type 1a hazard. The remaining hazards are as follows:

### ANSWER

- The sub-or is a type 2b hazard:

$$\text{MEM/WB.RegisterRd} = \text{ID/EX.RegisterRs2} = x2$$

- 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  $\text{EX/MEM.RegisterRd} \neq 0$  to the first hazard condition and  $\text{MEM/WB.RegisterRd} \neq 0$  to the second.

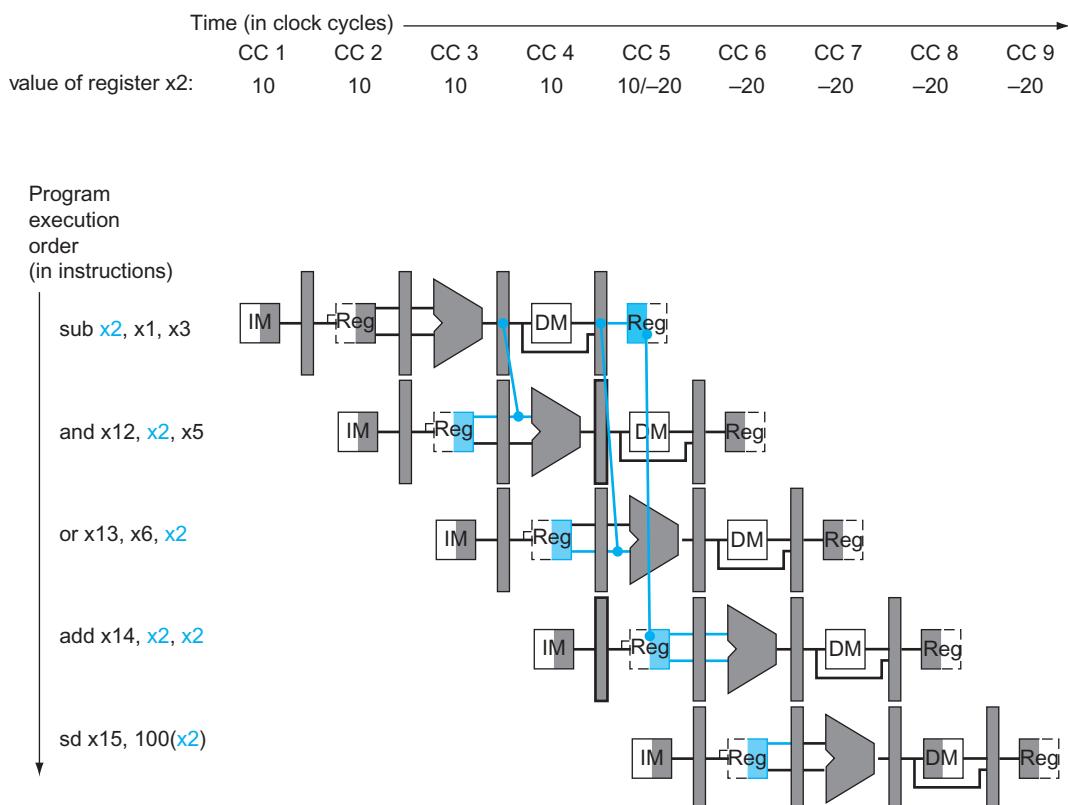
Now that we can detect hazards, half of the problem is resolved—but we must still forward the proper data.

[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

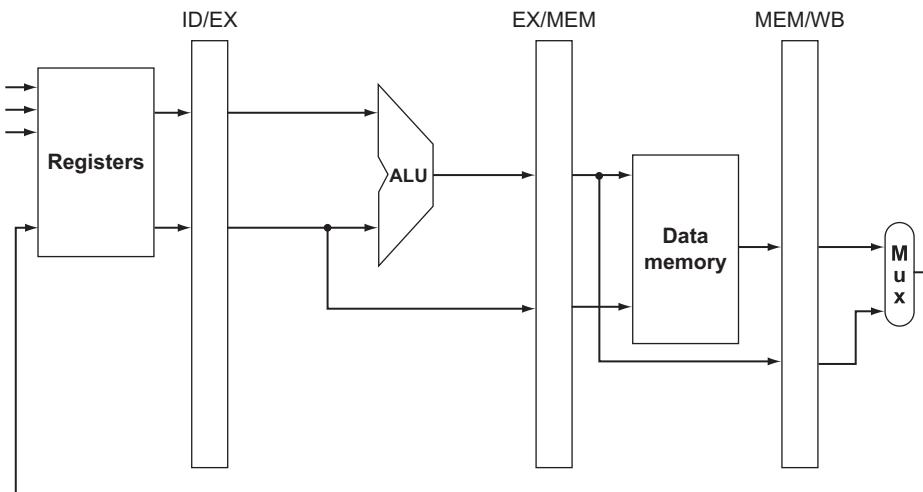
dependence begins from a *pipeline register*, rather than waiting for the WB stage to write the register file. Thus, the required data exist in time for later instructions, with the pipeline registers holding the data to be forwarded.

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.

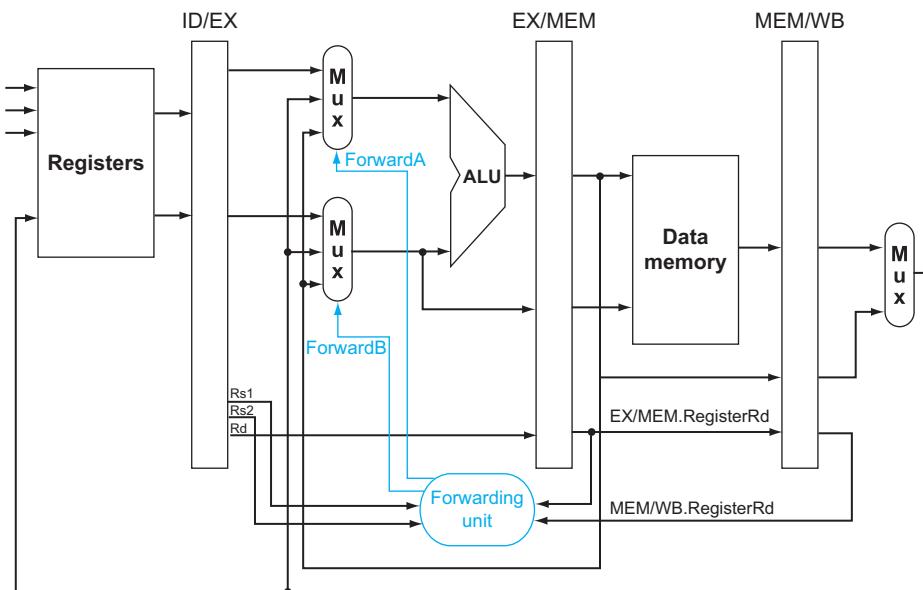
For now, we will assume the only instructions we need to forward are the four R-format instructions: add, sub, and, and or. [Figure 4.52](#) shows a close-up of the ALU and pipeline register before and after adding forwarding. [Figure 4.53](#)



**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):

```

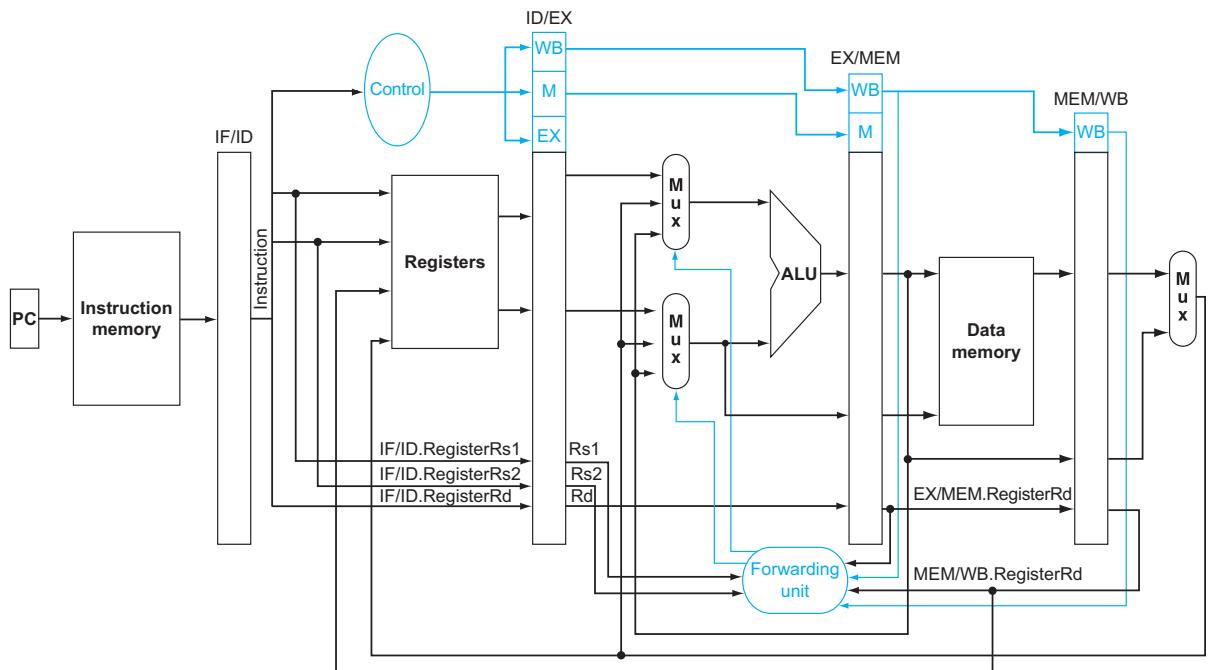
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
       and (EX/MEM.RegisterRd = ID/EX.RegisterRs1))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 01

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.

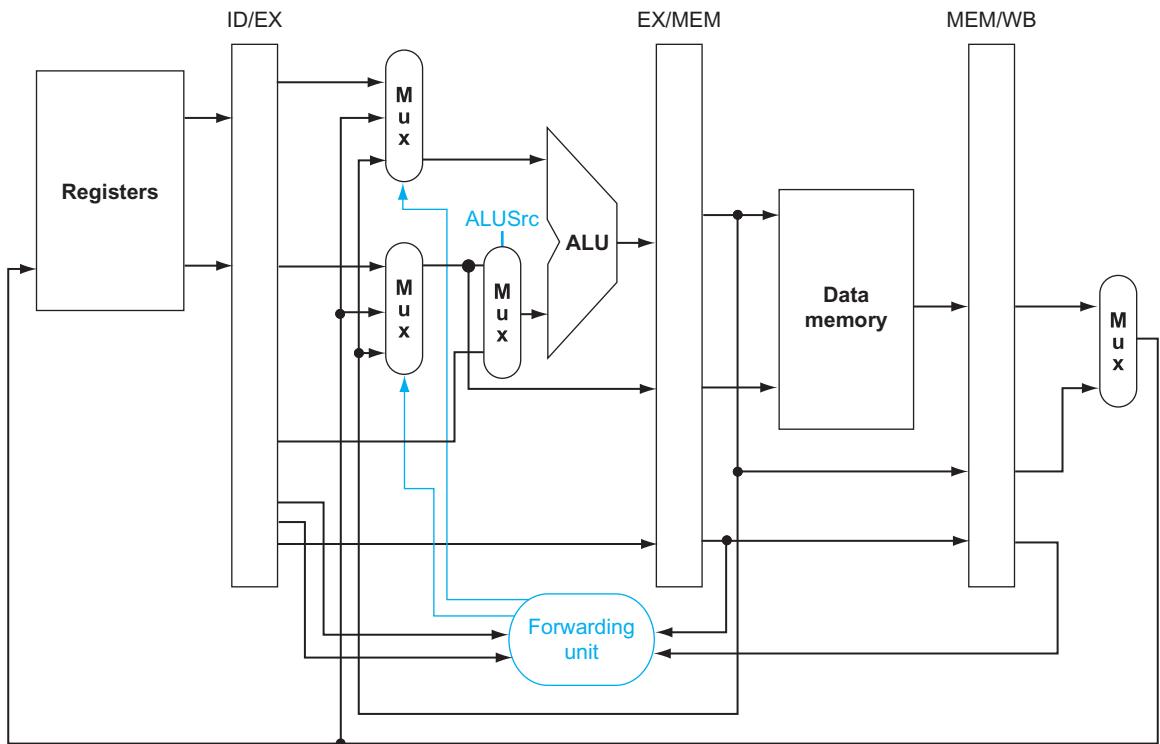
If you would like to see more illustrated examples using single-cycle pipeline drawings, [Section 4.13](#) has figures that show two pieces of RISC-V code with hazards that cause forwarding.



**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.

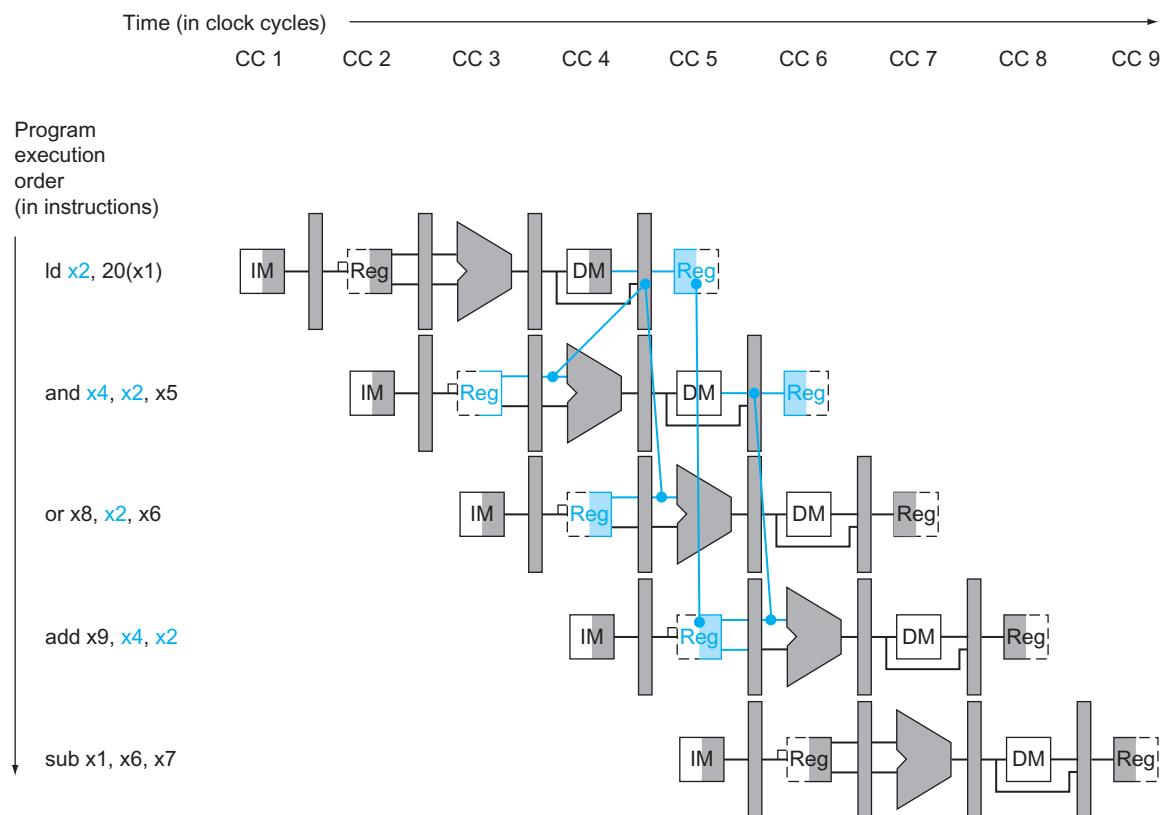
## Data Hazards and Stalls

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
```

If at first you don't succeed, redefine success.  
Anonymous



**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.)

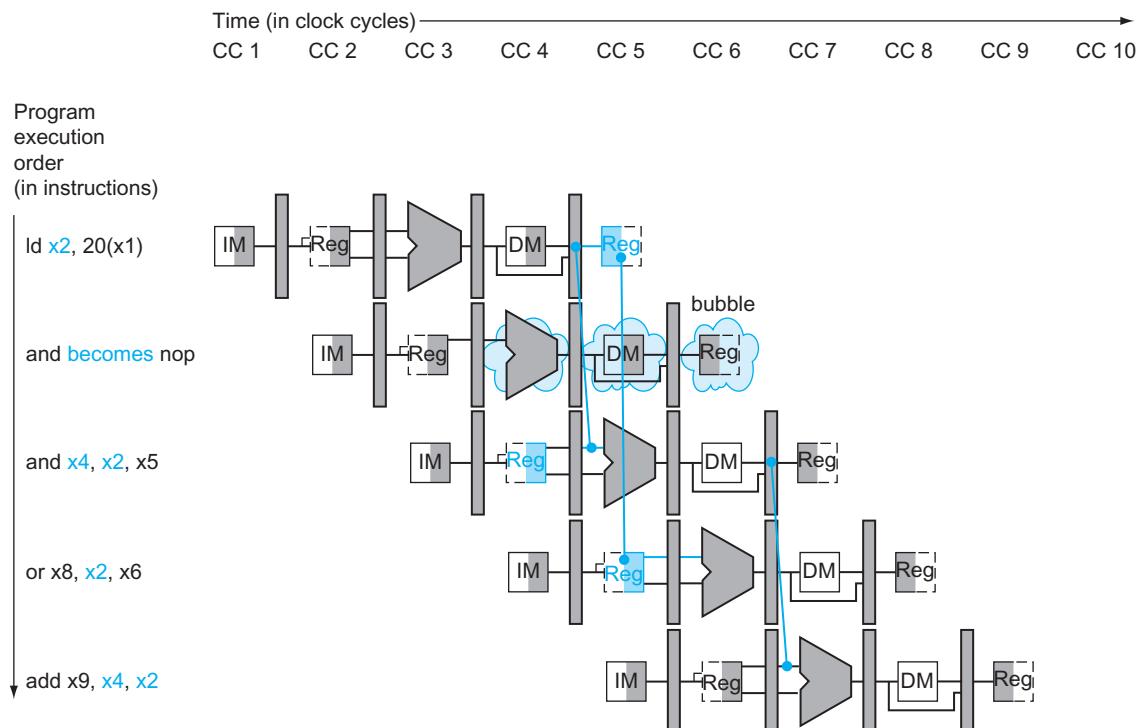
If the instruction in the ID stage is stalled, then the instruction in the IF stage must also be stalled; otherwise, we would lose the fetched instruction. Preventing these two instructions from making progress is accomplished simply by preventing the PC register and the IF/ID pipeline register from 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**.

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

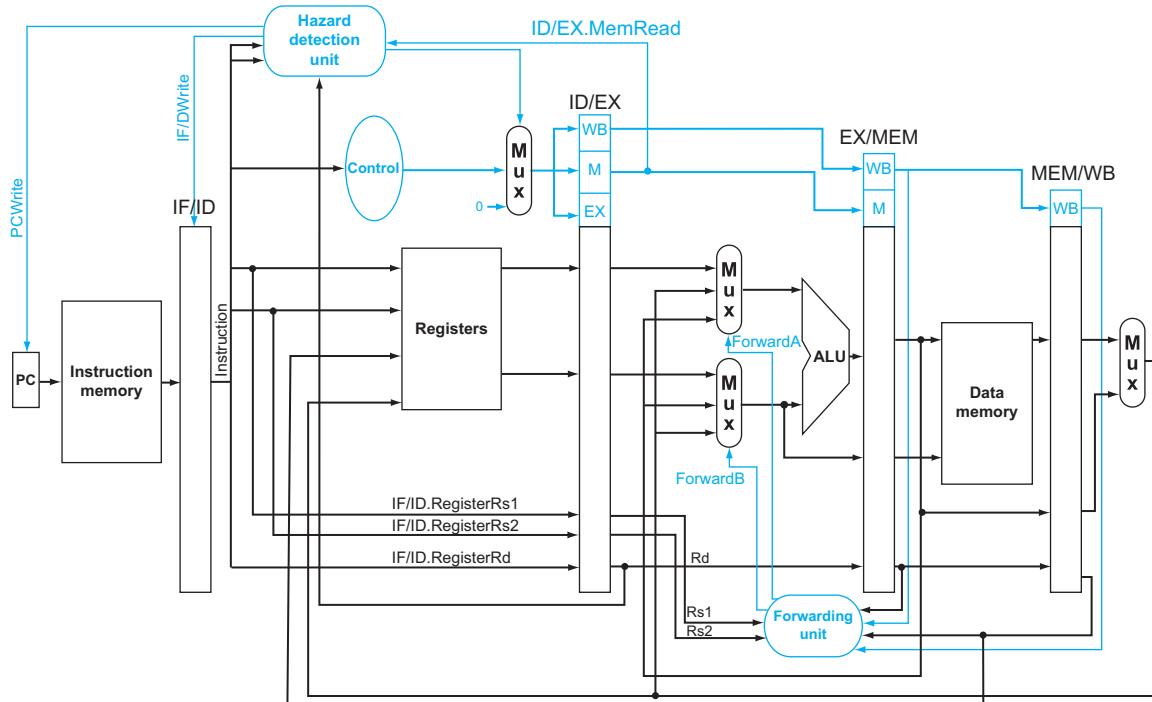
**nops** An instruction that does no operation to change state.



**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.

## The BIG Picture

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.

**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

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.

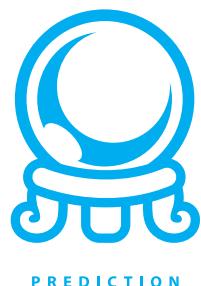
This section on control hazards is shorter than the previous sections on data hazards. The reasons are that control hazards are relatively simple to understand, they occur less frequently than data hazards, and there is nothing as effective against control hazards as forwarding is against data hazards. Hence, we use simpler schemes. We look at two schemes for resolving control hazards and one optimization to improve these schemes.

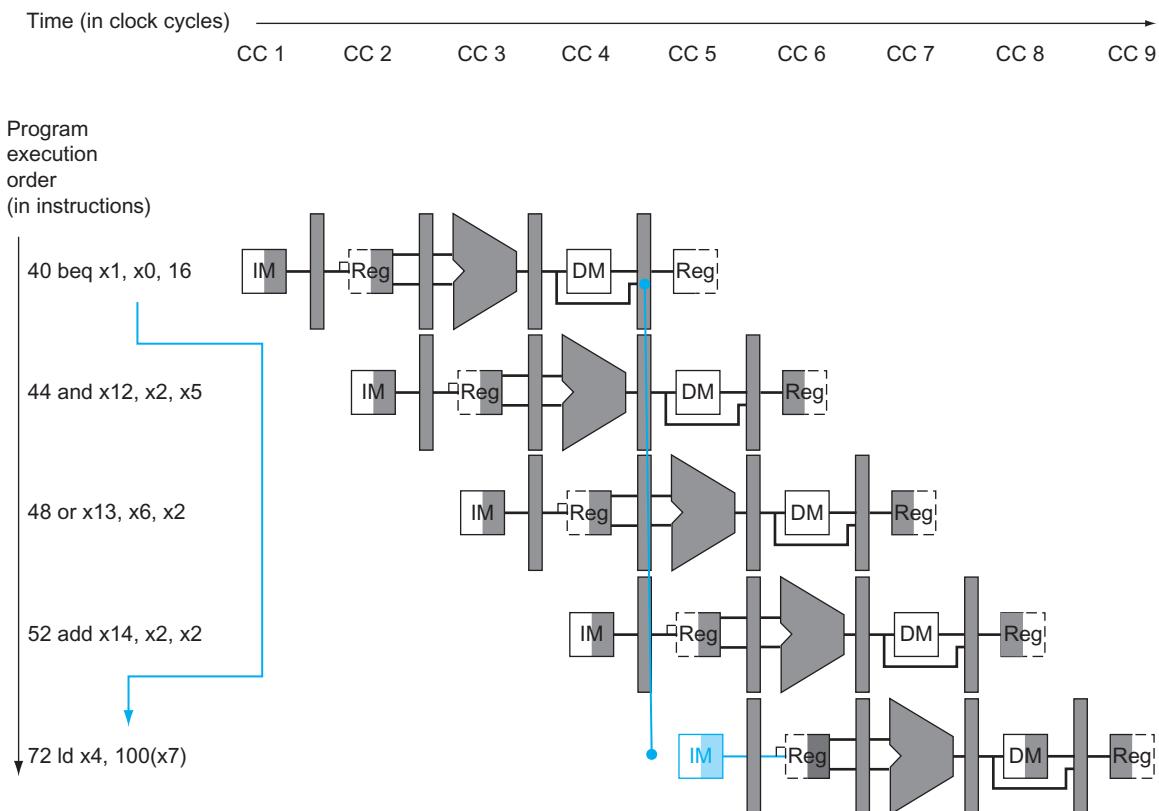
### Assume Branch Not Taken

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.

*There are a thousand  
hacking at the  
branches of evil to one  
who is striking at the  
root.*

Henry David Thoreau,  
*Walden*, 1854





**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 1 d 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****ANSWER**

PREDICTION

**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.

**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)

```

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.

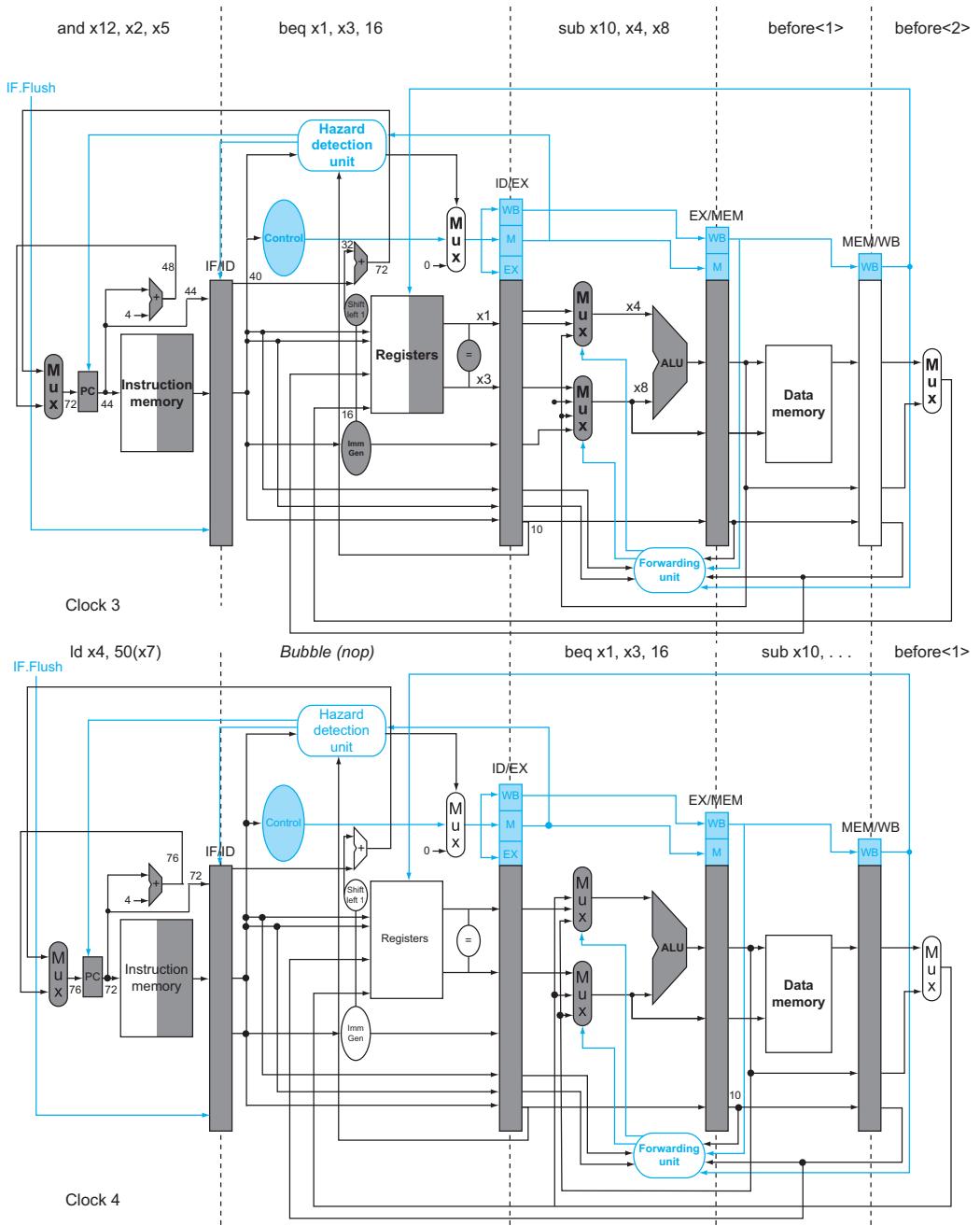
**Dynamic Branch Prediction**

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.

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

## ANSWER

### 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?

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](#).

**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**.

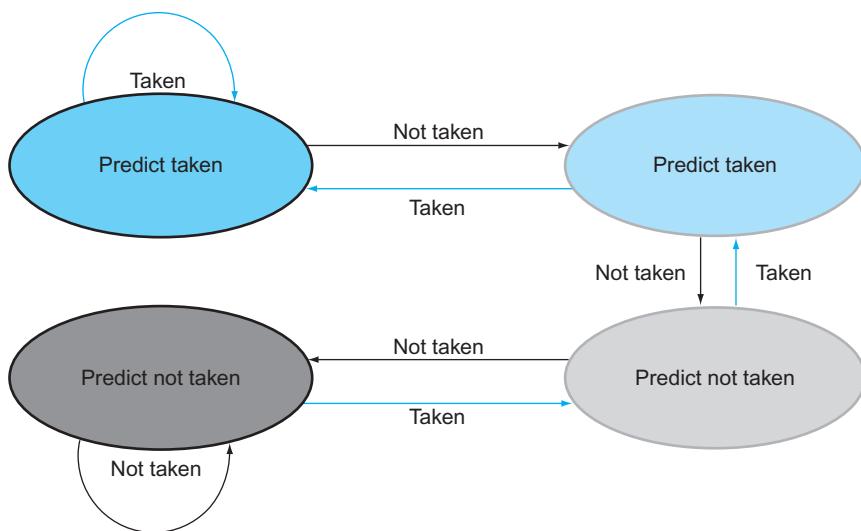
The 2-bit dynamic prediction scheme uses only information about a particular branch. Researchers noticed that using information about both a local branch and the global behavior of recently executed branches together yields greater prediction accuracy for the same number of prediction bits. Such predictors are called **correlating predictors**. A typical correlating predictor might have two 2-bit

### branch target buffer

A structure that caches the destination PC or destination instruction for a branch. It is usually organized as a cache with tags, making it more costly than a simple prediction buffer.

### correlating predictor

A branch predictor that combines local behavior of a particular branch and global information about the behavior of some recent number of executed branches.



**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.

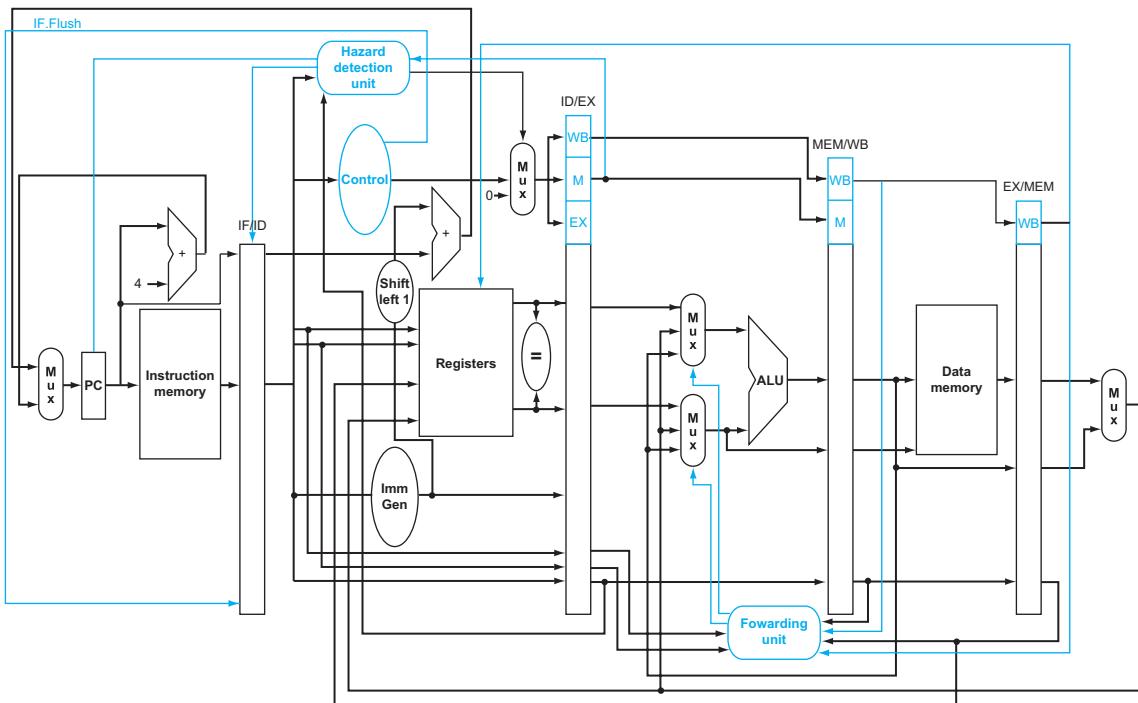
**Elaboration:** One way to reduce the number of conditional branches is to add *conditional move* instructions. Instead of changing the PC with a conditional branch, the instruction conditionally changes the destination register of the move. For example, the ARMv8 instruction set architecture has a 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

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:

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.

*To make a computer with automatic program-interruption facilities behave [sequentially] was not an easy matter, because the number of instructions in various stages of processing when an interrupt signal occurs may be large.*

Fred Brooks, Jr.,  
*Planning a Computer System: Project Stretch*, 1962

**exception** Also called **interrupt**. An unscheduled event that disrupts program execution; used to detect undefined instructions.

**interrupt** An exception that comes from outside of the processor. (Some architectures use the term *interrupt* for all exceptions.)

## 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.

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 <sub>two</sub>
System Error (hardware malfunction)	01 1000 0000 <sub>two</sub>

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<sub>hex</sub>. (Implementing

**vectored interrupt** An interrupt for which the address to which control is transferred is determined by the cause of the exception.

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.

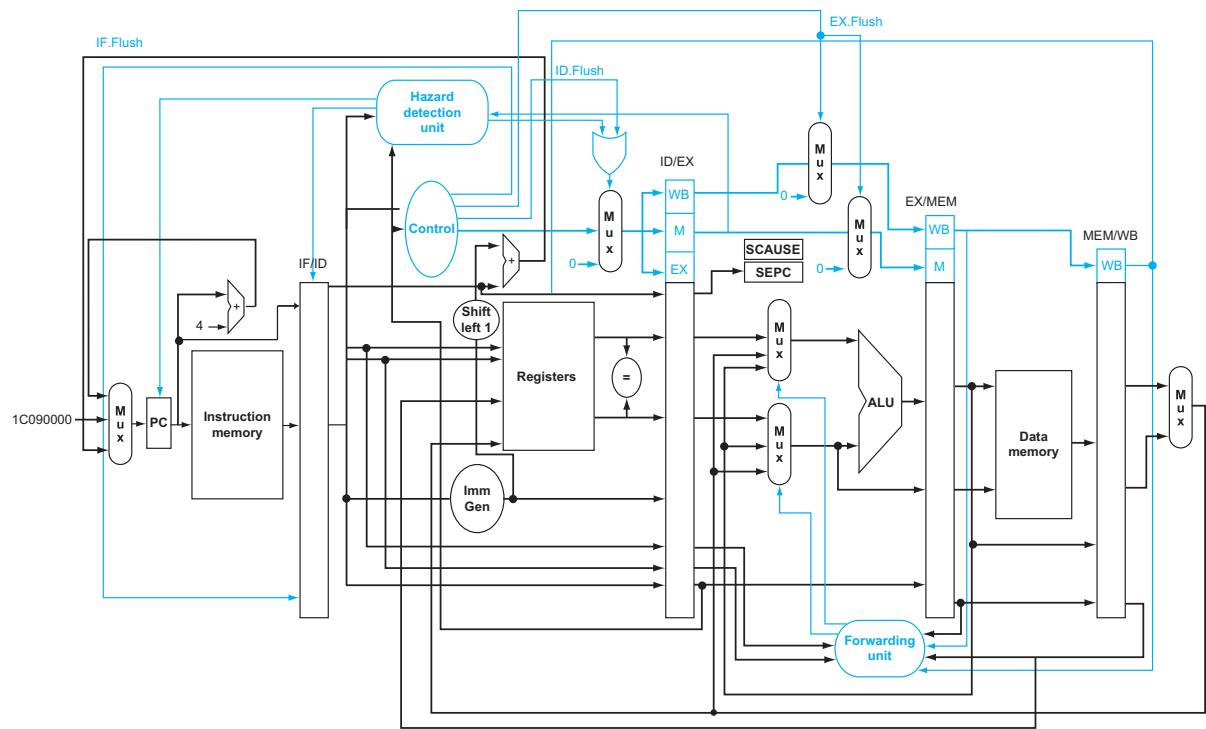
## Exceptions in a Pipelined Implementation

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<sub>hex</sub> 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<sub>hex</sub> 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.

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<sub>hex</sub> 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<sub>hex</sub>.

## ANSWER

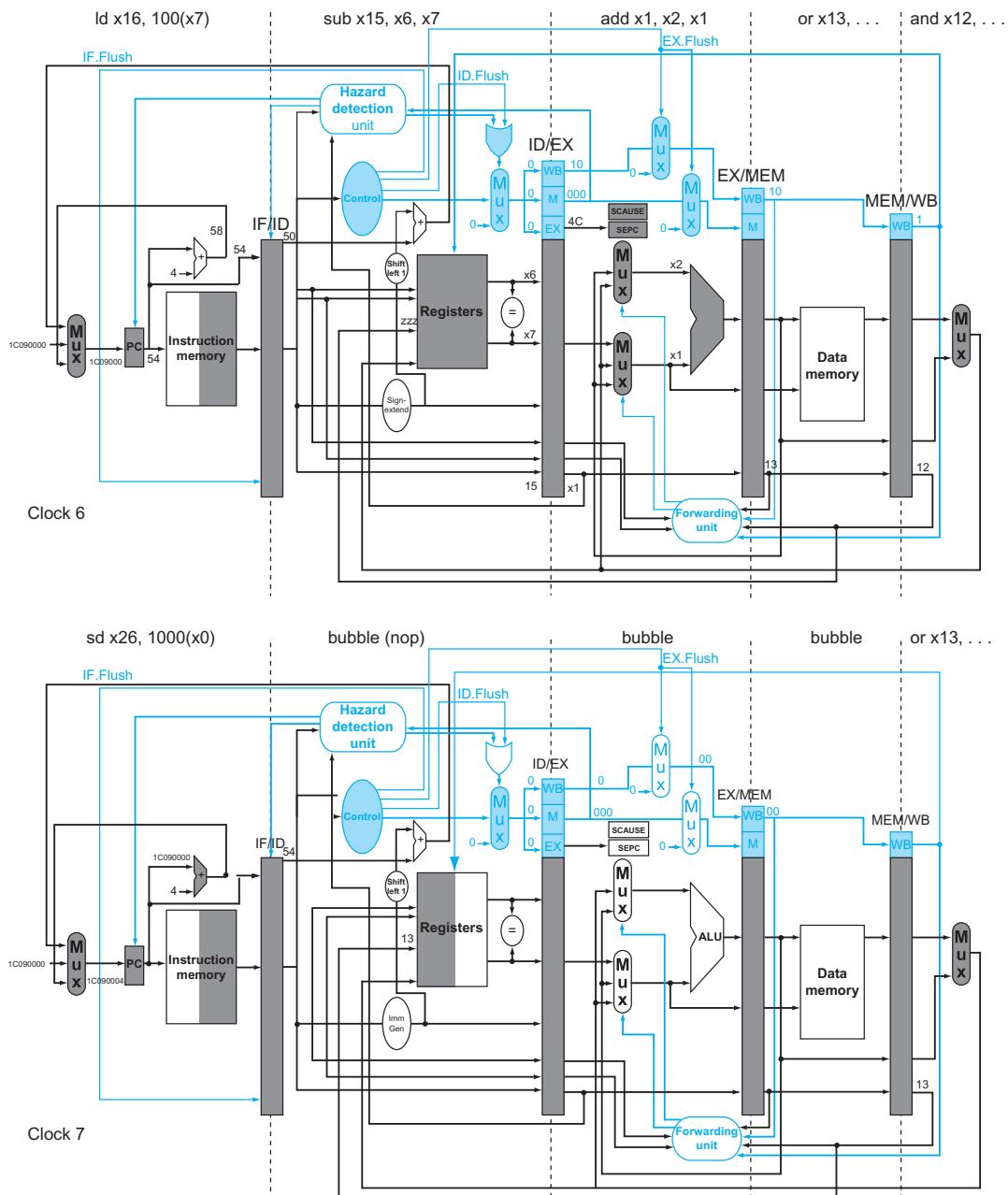
We mentioned several examples of exceptions on page 315, and we will see others in Chapter 5. With five instructions active in any clock cycle, the challenge is to associate an exception with the appropriate instruction. Moreover, multiple exceptions can occur simultaneously in a single clock cycle. 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.

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.

## Hardware/ Software Interface



**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<sub>hex</sub>). 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<sub>hex</sub>. 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](#).)

**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.

**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.

Which exception should be recognized first in this sequence?

1. `xxx x11, x12, x11` // undefined instruction
2. `sub x11, x12, x11` // hardware error

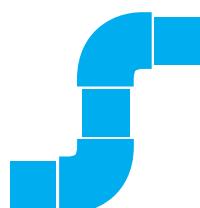
### Check Yourself

## 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



PIPELINING



PARALLELISM

**instruction-level parallelism** The parallelism among instructions.

**multiple issue** A scheme whereby multiple instructions are launched in one clock cycle.

**static multiple issue** An approach to implementing a multiple-issue processor where many decisions are made by the compiler before execution.

**dynamic multiple issue** An approach to implementing a multiple-issue processor where many decisions are made during execution by the processor.

**issue slots** The positions from which instructions could issue in a given clock cycle; by analogy, these correspond to positions at the starting blocks for a sprint.

move from a four-stage to a six-stage pipeline. To get the full speed-up, we need to rebalance the remaining steps so they are the same length, in processors or in laundry. The amount of parallelism being exploited is higher, since there are more operations being overlapped. Performance is potentially greater since the clock cycle can be shorter.

Another approach is to replicate the internal components of the computer so that it can launch multiple instructions in every pipeline stage. The general name for this technique is **multiple issue**. A multiple-issue laundry would replace our household washer and dryer with, say, three washers and three dryers. You would also have to recruit more assistants to fold and put away three times as much laundry in the same amount of time. The downside is the extra work to keep all the machines busy and transferring the loads to the next pipeline stage.

Launching multiple instructions per stage allows the instruction execution rate to exceed the clock rate or, stated alternatively, the CPI to be less than 1. As mentioned in [Chapter 1](#), it is sometimes useful to flip the metric and use *IPC*, or *instructions per clock cycle*. Hence, a 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.

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:

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.

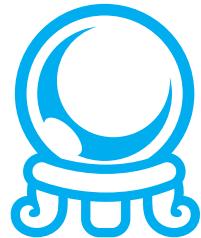
## The Concept of Speculation

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 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.



PREDICTION

**speculation** An approach whereby the compiler or processor guesses the outcome of an instruction to remove it as a dependence in executing other instructions.

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

**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.

Static multiple-issue processors all use the compiler to assist with packaging instructions and handling hazards. In a static issue processor, you can think of the set of instructions issued in a given clock cycle, which is called an **issue packet**, as one large instruction with multiple operations. This view is more than an analogy. Since a static multiple-issue processor usually restricts what mix of instructions can be initiated in a given clock cycle, it is useful to think of the issue packet as a single instruction allowing several operations in certain predefined fields. This view led to the original name for this approach: **Very Long Instruction Word (VLIW)**.

Most static issue processors also rely on the compiler to take on some responsibility for handling data and control hazards. The compiler's responsibilities may include static branch prediction and code scheduling to reduce or prevent all hazards. Let's look at a simple static issue version of 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
Load or store instruction				IF	ID	EX	WB

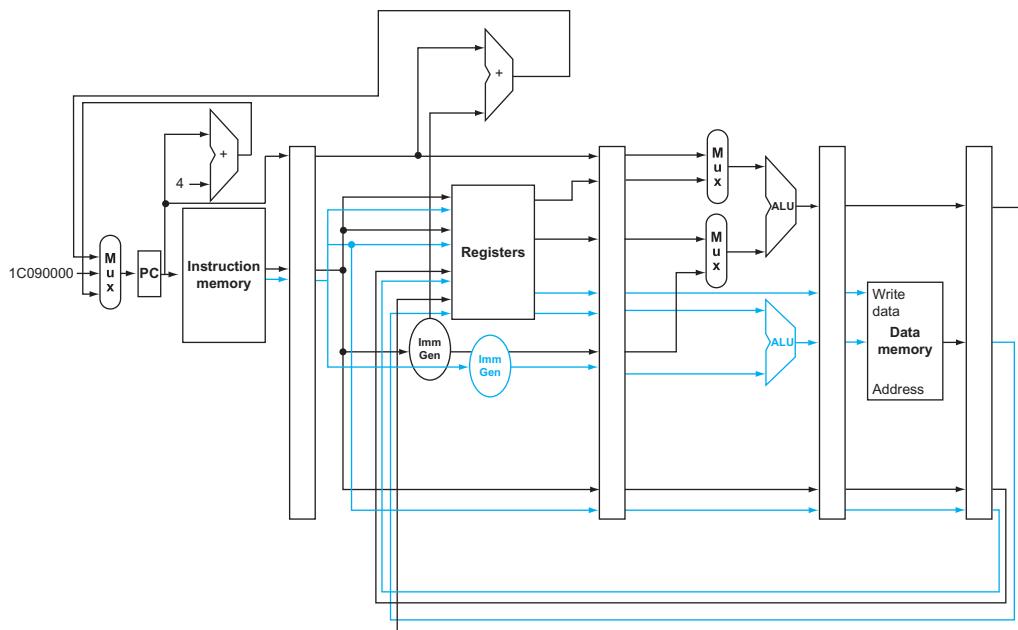
**FIGURE 4.65 Static two-issue pipeline in operation.** The ALU and data transfer instructions are issued at the same time. Here we have assumed the same five-stage structure as used for the single-issue pipeline. Although this is not strictly necessary, it does have some advantages. In particular, keeping 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.

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** 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 Unrolling for Multiple-Issue Pipelines

See how well loop unrolling and scheduling work in the example above. For simplicity, assume that the loop index is a multiple of four.

### EXAMPLE

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.

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.

### ANSWER

**register renaming** The renaming of registers by the compiler or hardware to remove antidependences.

**antidependence** **Also called name dependence** An ordering forced by the reuse of a name, typically a register, rather than by a true dependence that carries a value between two instructions.

	<b>ALU or branch instruction</b>	<b>Data transfer instruction</b>	<b>Clock cycle</b>
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

Dynamic multiple-issue processors are also known as **superscalar** processors, or simply superscalars. In the simplest superscalar processors, instructions issue in order, and the processor decides whether zero, one, or more instructions can issue in a given clock cycle. Obviously, achieving good performance on such a processor still requires the compiler to try to schedule instructions to move dependences apart and thereby improve the instruction issue rate. Even with such compiler scheduling, there is an important difference between this simple superscalar and a VLIW processor: the code, whether scheduled or not, is guaranteed by the hardware to execute correctly. Furthermore, compiled code will always run correctly independent of the issue rate or pipeline structure of the processor. In some VLIW designs, this has not been the case, and recompilation was required when moving across different processor models; in other static issue processors, code would run correctly across different implementations, but often so poorly as to make compilation effectively required.

Many superscalars extend the basic framework of dynamic issue decisions to include **dynamic pipeline scheduling**. Dynamic pipeline scheduling chooses which instructions to execute in a given clock cycle while trying to avoid hazards

**superscalar** An advanced pipelining technique that enables the processor to execute more than one instruction per clock cycle by selecting them during execution.

**dynamic pipeline scheduling** Hardware support for reordering the order of instruction execution to avoid stalls.

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
```

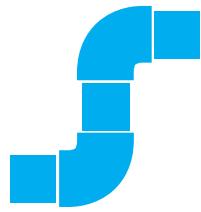
Even though the `sub` instruction is ready to execute, it must wait for the `ld` and `add` to complete first, which might take many clock cycles if memory is slow. (Chapter 5 explains cache misses, the reason that memory accesses are sometimes very slow.) Dynamic **pipeline** scheduling allows such hazards to be avoided either fully or partially.

### 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.

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:

- 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.

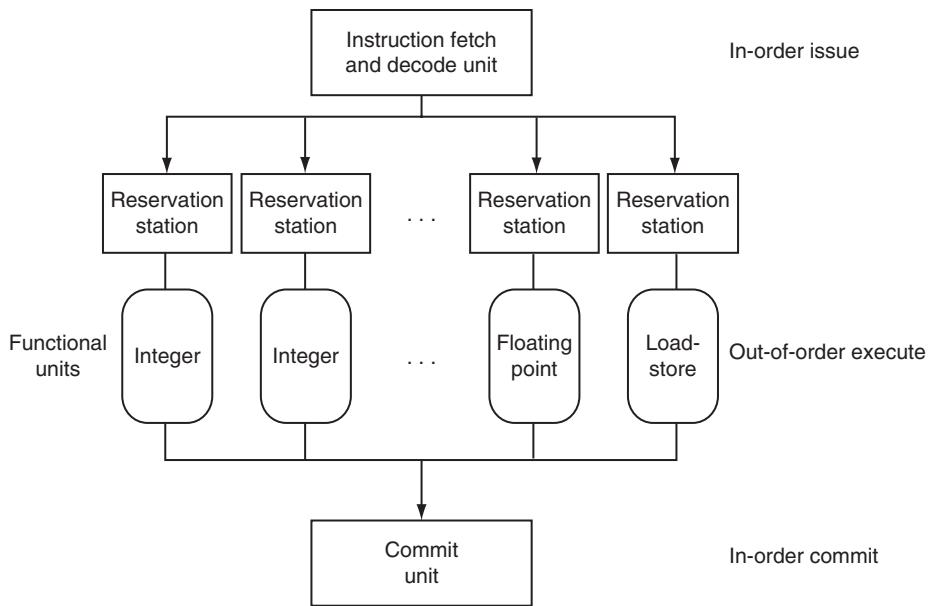


PIPELINING

**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.



**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.

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

#### 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.

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.

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.

## Understanding Program Performance

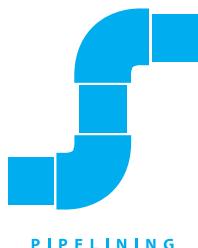


HIERARCHY



PREDICTION

## 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

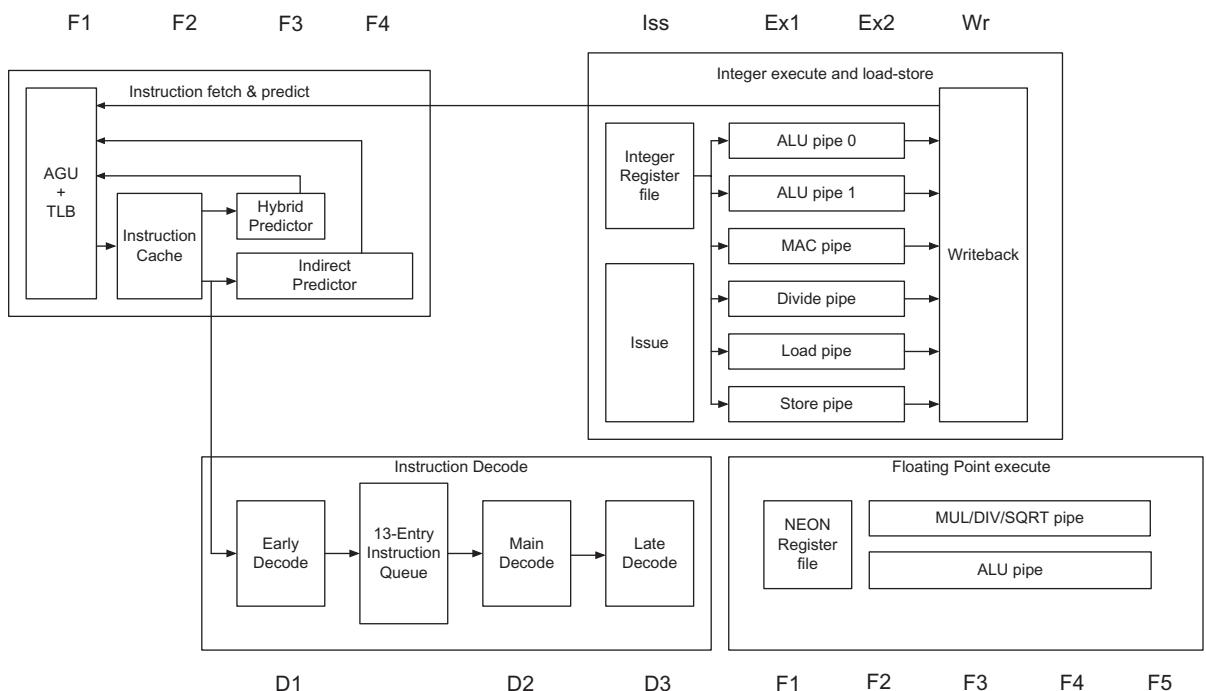
### Real Stuff: The ARM Cortex-A53 and Intel Core i7 Pipelines

[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 Cortex-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.

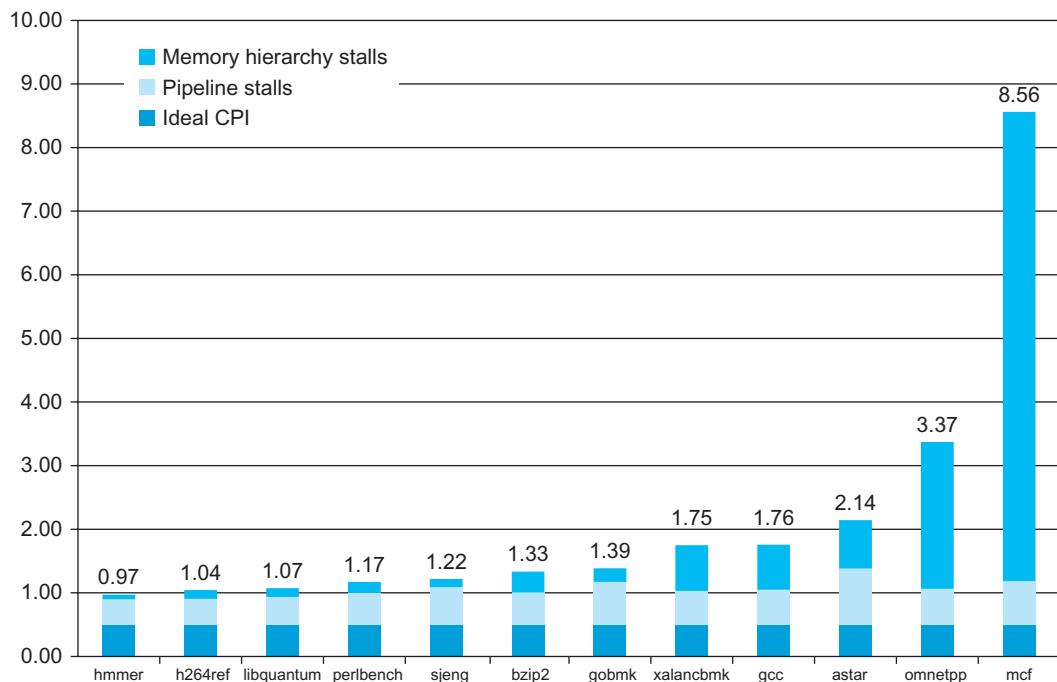
The first three stages fetch two instructions at a time and try to keep a 13-entry instruction queue full. It uses a 6k-bit hybrid conditional branch predictor, a 256-entry indirect branch predictor, and an 8-entry return address stack to predict future function returns. The prediction of indirect branches 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.

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

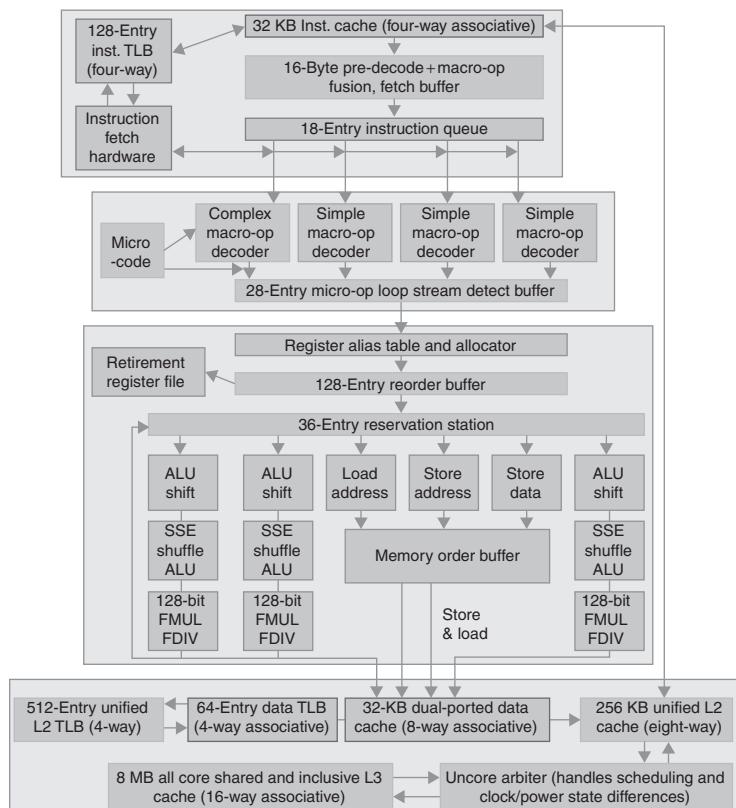
**microarchitecture** The organization of the processor, including the major functional units, their interconnection, and control.

**architectural registers** The instruction set of visible registers of a processor; for example, in 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.

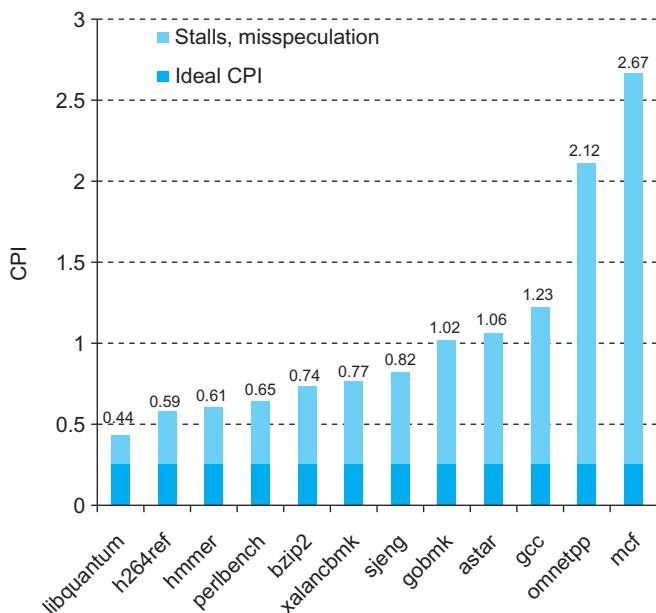
**Elaboration:** Hardware in the second and fourth steps can combine or *fuse* operations together to reduce the number of operations that must be performed. *Macro-op fusion* in the second step takes x86 instruction combinations, such as compare followed by a branch, and fuses them into a single operation. *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

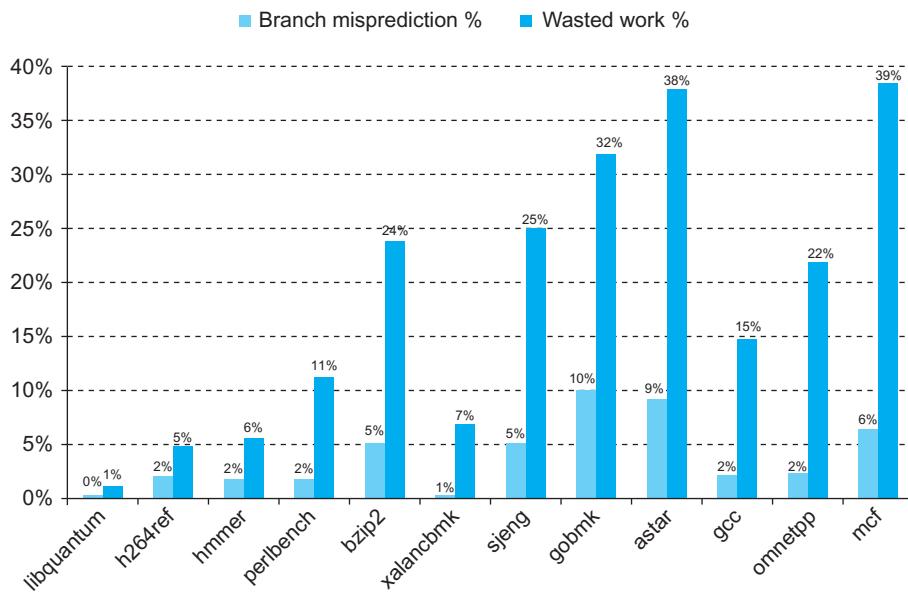
Figure 4.75 shows the CPI of the Intel Core i7 for each of the SPEC2006 benchmarks. While the ideal CPI is 0.25, the best case achieved is 0.44, the median case is 0.79, and the worst case is 2.67.

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.**

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

## Understanding Program Performance



## 4.12

### Going Faster: Instruction-Level Parallelism and Matrix Multiply

Returning to the DGEMM example from [Chapter 3](#), we can see the impact of instruction-level parallelism by unrolling the loop so that the multiple-issue, out-of-order execution processor has more instructions to work with. [Figure 4.77](#) 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.

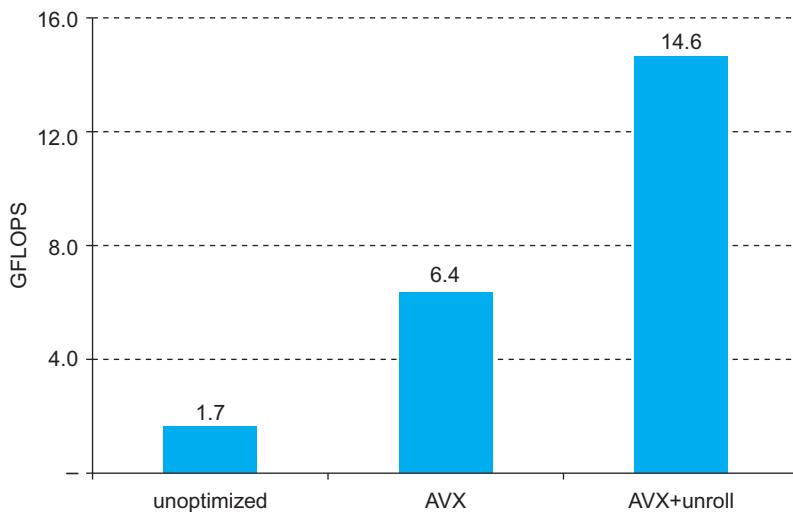
[Figure 4.78](#) shows the assembly language output of the unrolled code. As expected, in [Figure 4.78](#) there are four versions of each of the AVX instructions in [Figure 3.23](#), with one exception. We only need one copy of the vbroadcastsd instruction, since we can use the four copies of the B 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  $32 \times 32$  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  $32 \times 32$  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 + {{5{IDEXIR[31]}}, IDEXIR[30:20]};
    else if (IDEXOp == SD)
        EXMEMALUOut <= IDEXA + {{5{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.

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.

### Check Yourself

## 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'b0000_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, IDEXR, 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 = IDEXR[6:0];
    assign IDEXrs1 = IDEXR[19:15];
    assign IDEXrs2 = IDEXR[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 IDEXR 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 IDEXR 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; IDEXR = 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
  IDEXR <= 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]}}, IDEXR[30:20]};
  else if (IDEXOp == SD)
    EXMEMALUOut <= IDEXA + {{53{IDEXIR[31]}}, IDEXR[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 <= IDEXR; 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; IDEXR = 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
            IDEXR <= 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]}}, IDEXR[30:20]};
            else if (IDEXop == SD)
                EXMEMALUOut <= IDEXA + {{53{IDEXIR[31]}}, IDEXR[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 <= IDEXR; 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 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:

```

**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*)

```

INDEXB;
    // The signal for detecting a stall based on the use of a result from
LW
    assign stall = (MEMWBop == LD) && ( // source instruction is a load
        (((INDEXop == LD) || (INDEXop == SD)) && (INDEXrs1 ==
MEMWBrd)) || // stall for address calc
        (((INDEXop == ALUop) && ((INDEXrs1 == MEMWBrd) ||
(INDEXrs2 == 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; INDEXIR = 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
        end
    // second instruction in pipeline is fetching registers
    INDEXA <= Regs[IFIDrs1]; INDEXB <= Regs[IFIDrs2]; // get two
registers
    INDEXIR <= IFIDIR; // pass along IR--can happen anywhere, since this
affects next stage only!

    // third instruction is doing addresses calculation or ALU operation
    if (INDEXop == LD)

        EXMEMALUOut <= INDEXA + {{53{INDEXIR[31]}}, INDEXIR[30:20]};
    else if (INDEXop == SD)
        EXMEMALUOut <= INDEXA + {{53{INDEXIR[31]}}, INDEXIR[30:25],
INDEXIR[11:7]};
    else if (INDEXop == ALUop)
        case (INDEXIR[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 <= IDEXR; 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
    MEMBIR <= 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 e4.13.4 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
                    end
            end
    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 ALUBininput (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)
RISCVVALU 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;

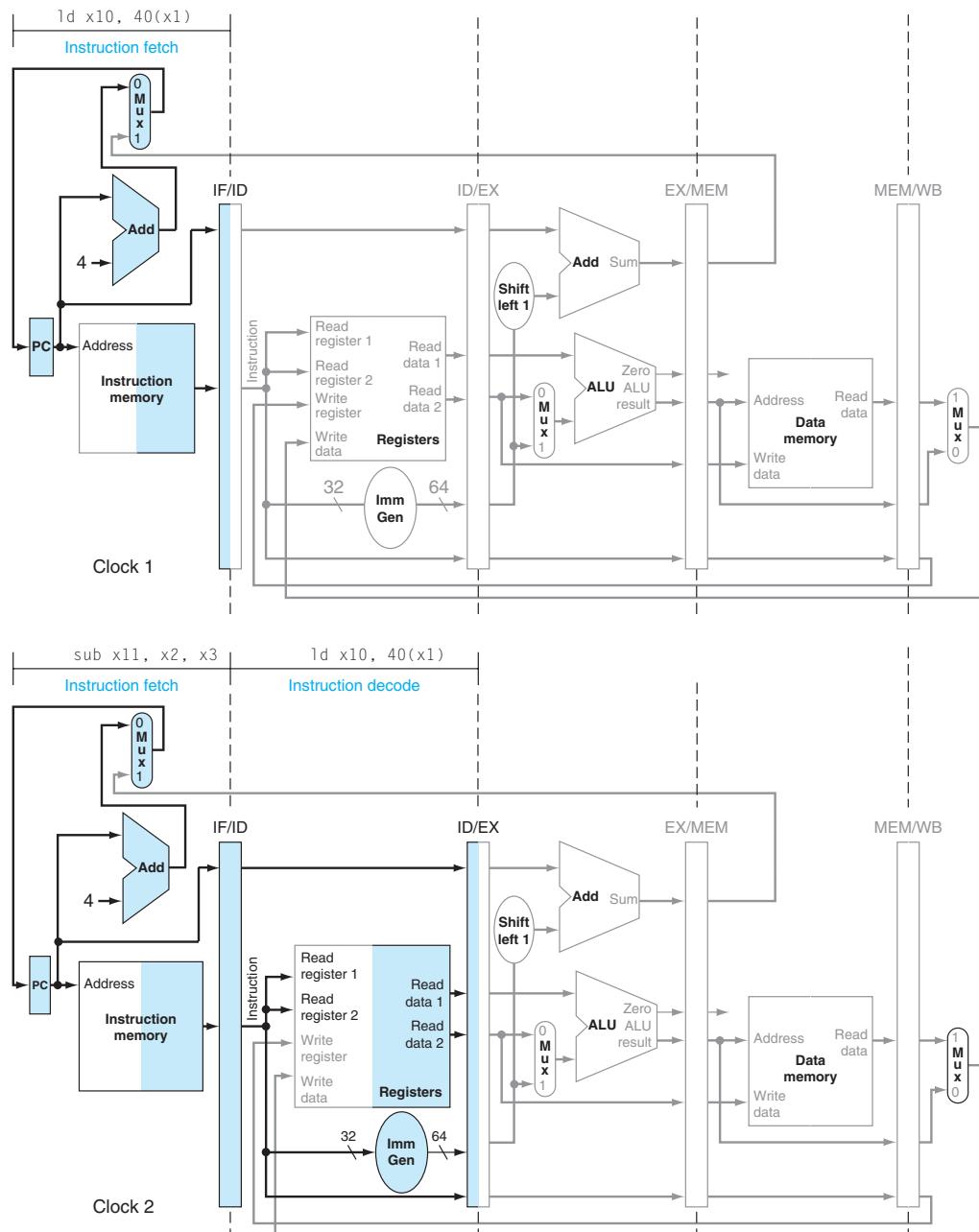
    // Create an instance of the RISC-V datapath, the inputs are the
    control signals; opcode is only output
    Datapath RISCVDP (ALUOp, MemtoReg, MemRead, MemWrite, IorD, RegWrite,
    IRWrite,
                    PCWrite, PCWriteCond, ALUSrcA, ALUSrcB, PCSource,
    opcode, clock);

    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;

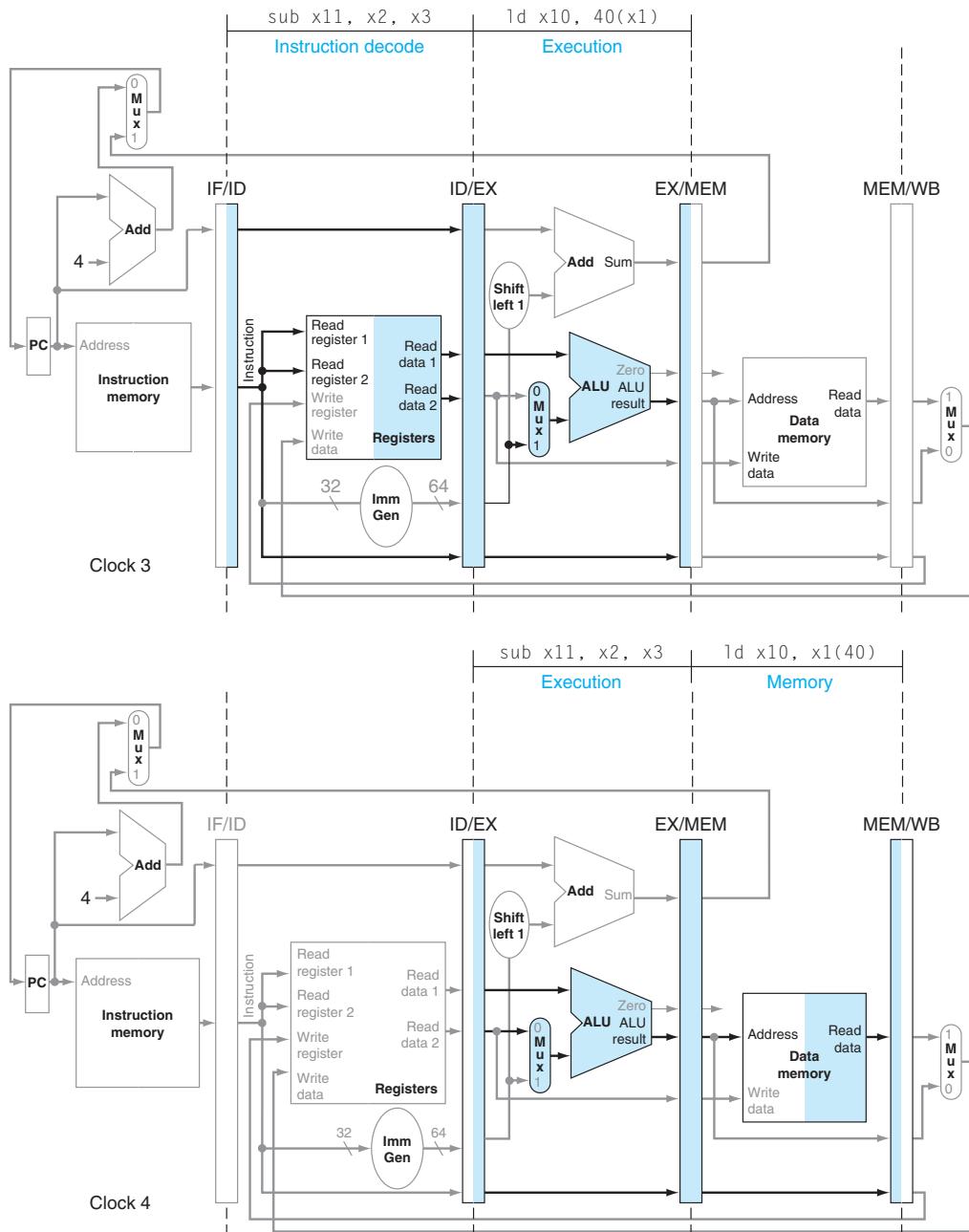
    // Here is the state machine, which only has to sequence states
    always @ (posedge clock)
    begin // all state updates on a positive clock edge
        case (state)
            1: state <= 2; // unconditional next state
            2: state <= 3; // unconditional next state
            3: state <= (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

```

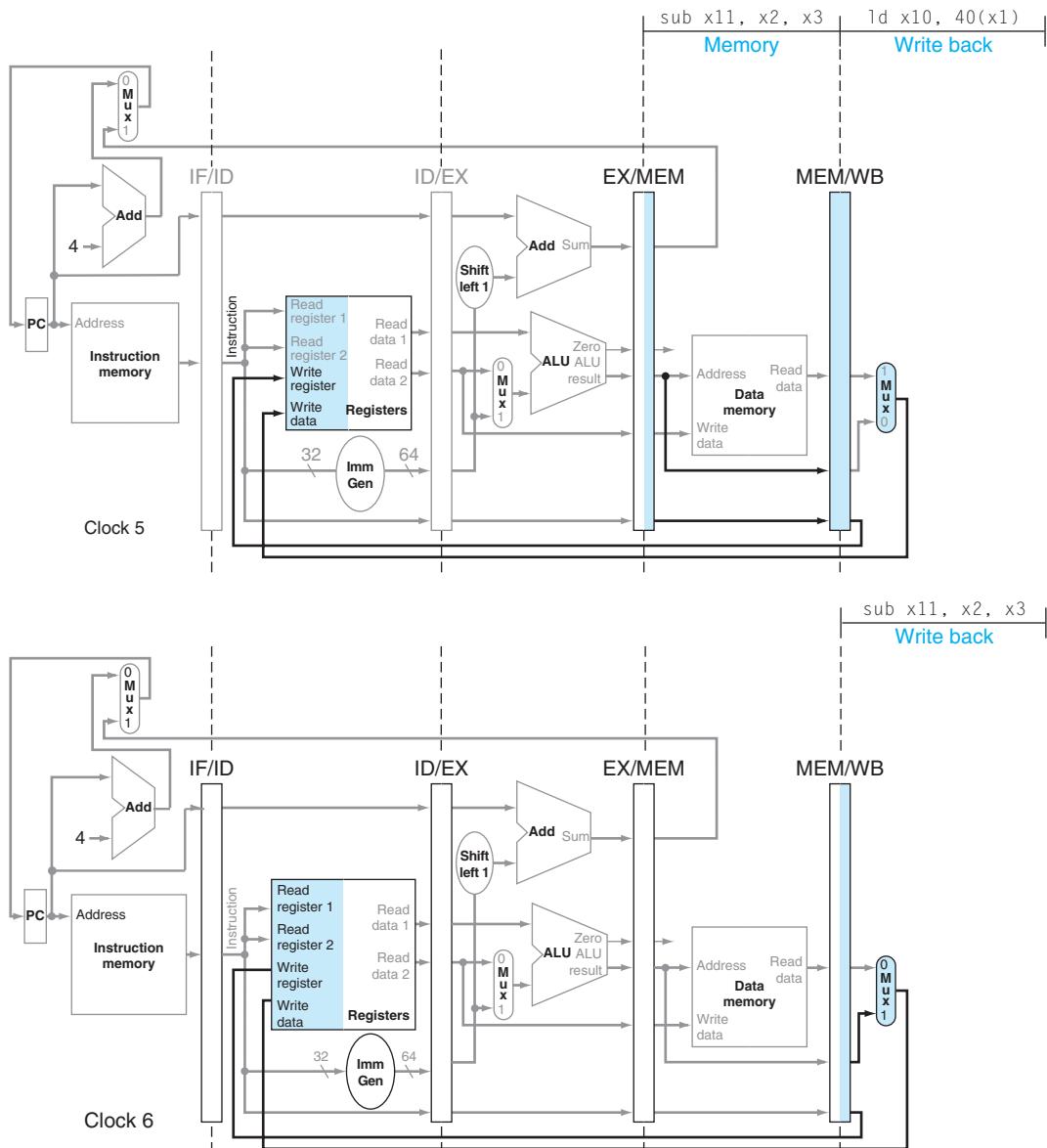
**FIGURE e4.13.7 The RISC-V CPU using the datapath from Figure e4.13.6.**



**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, 1d 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.

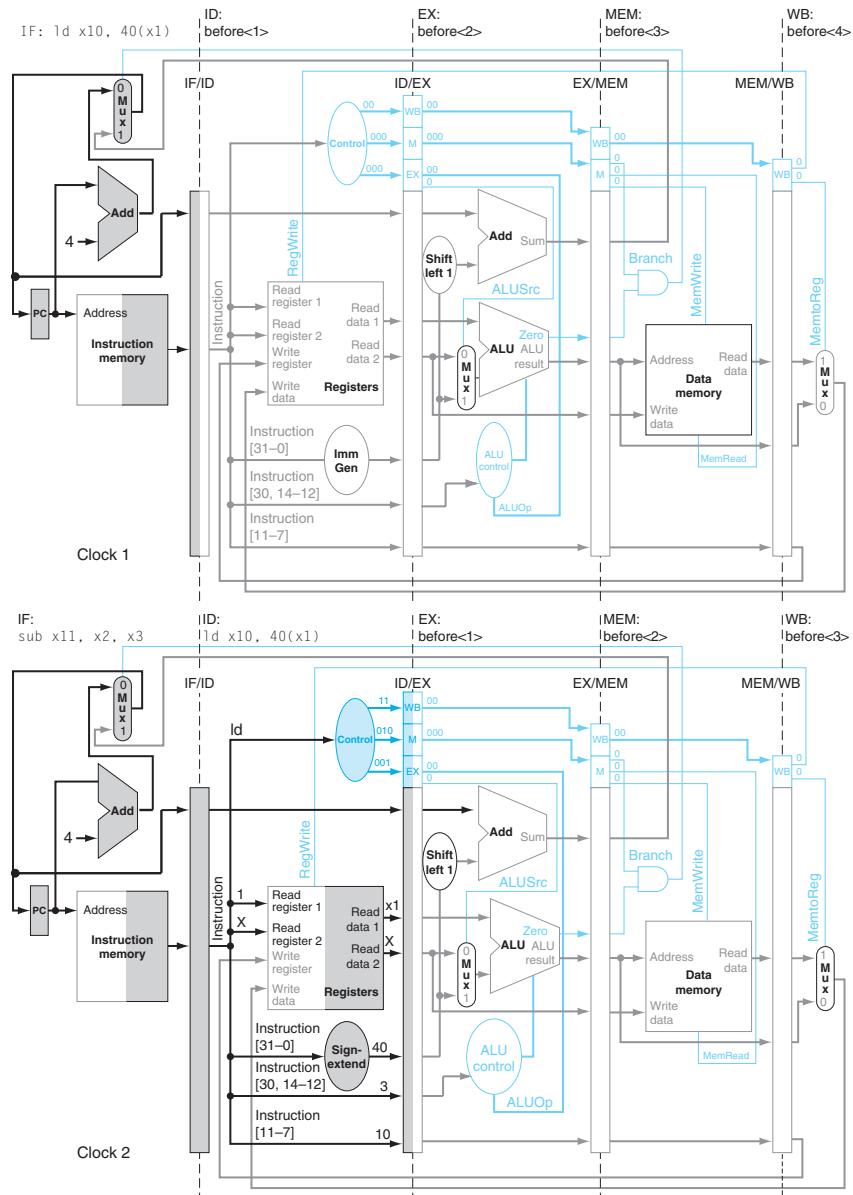
Thus, in clock cycle 5, the forwarding unit selects the EX/MEM pipeline register for the upper input to the ALU and the MEM/WB pipeline register for the lower input to the ALU. The following add instruction reads both register  $x_4$ , the target of the and instruction, and register  $x_2$ , which the sub instruction has already written. Notice that the prior two instructions both write register  $x_4$ , 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.

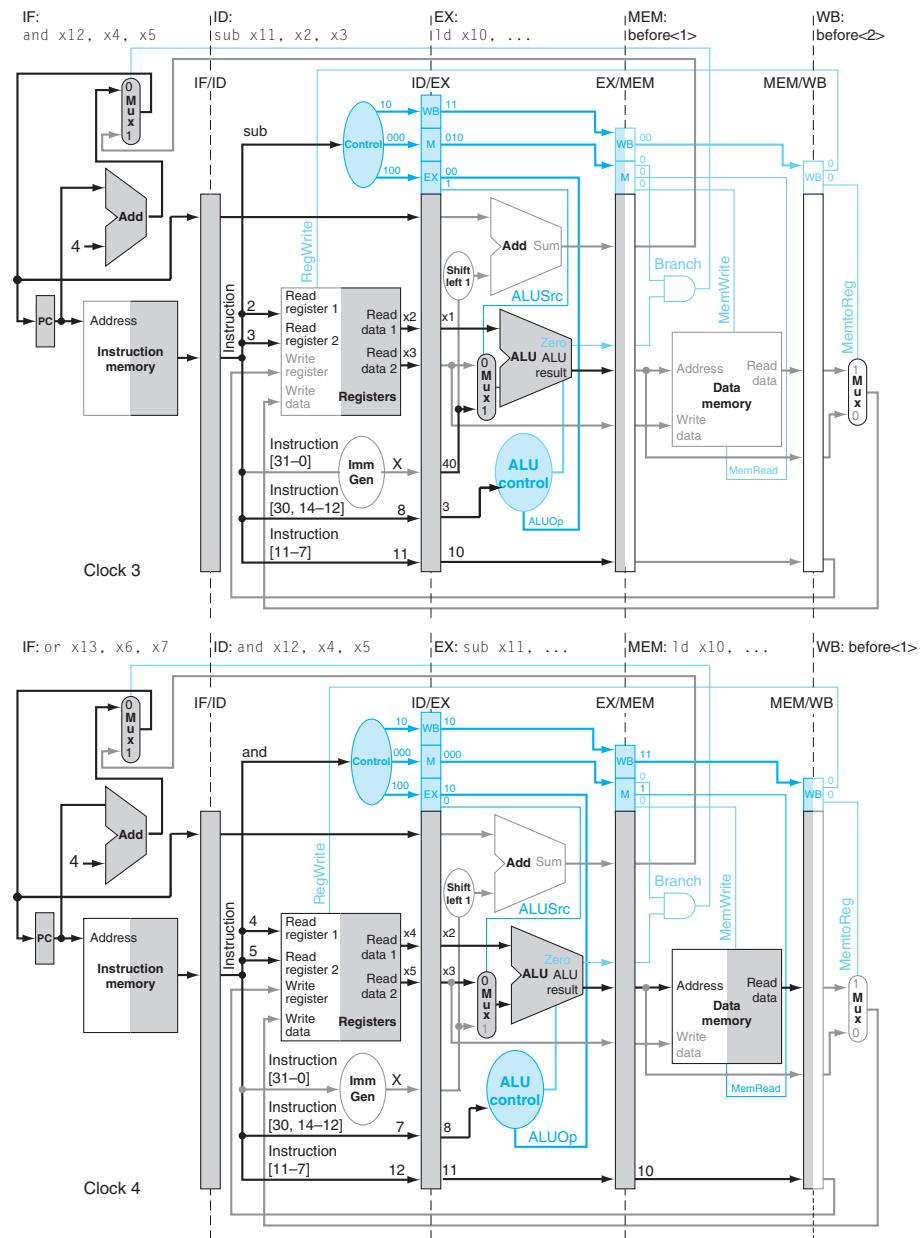
### Illustrating Pipelines with Stalls and Forwarding

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):

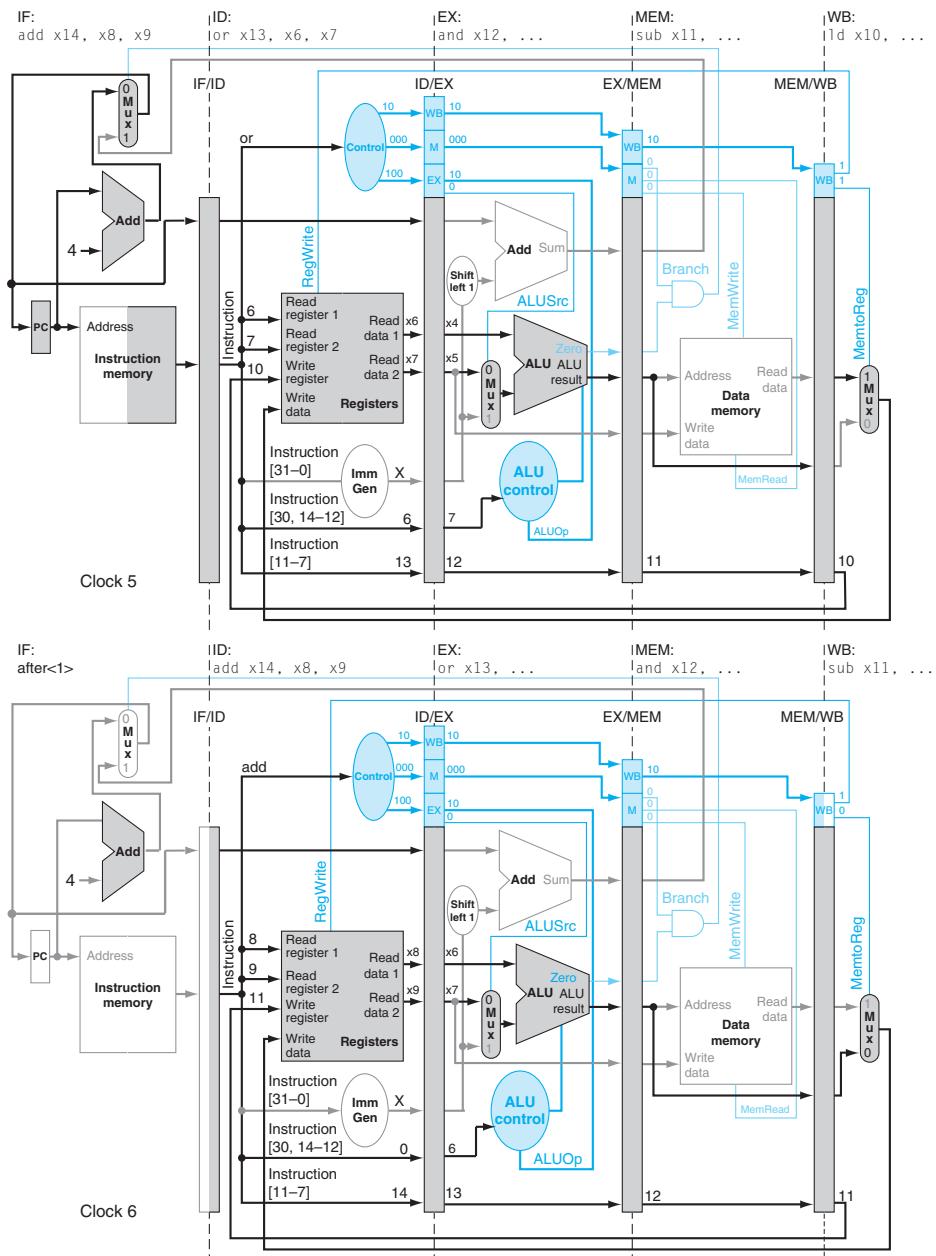
```
ld   x2, 40(x1)
and x4, x2, x5
or  x4, x4, x2
add x9, x4, x2
```



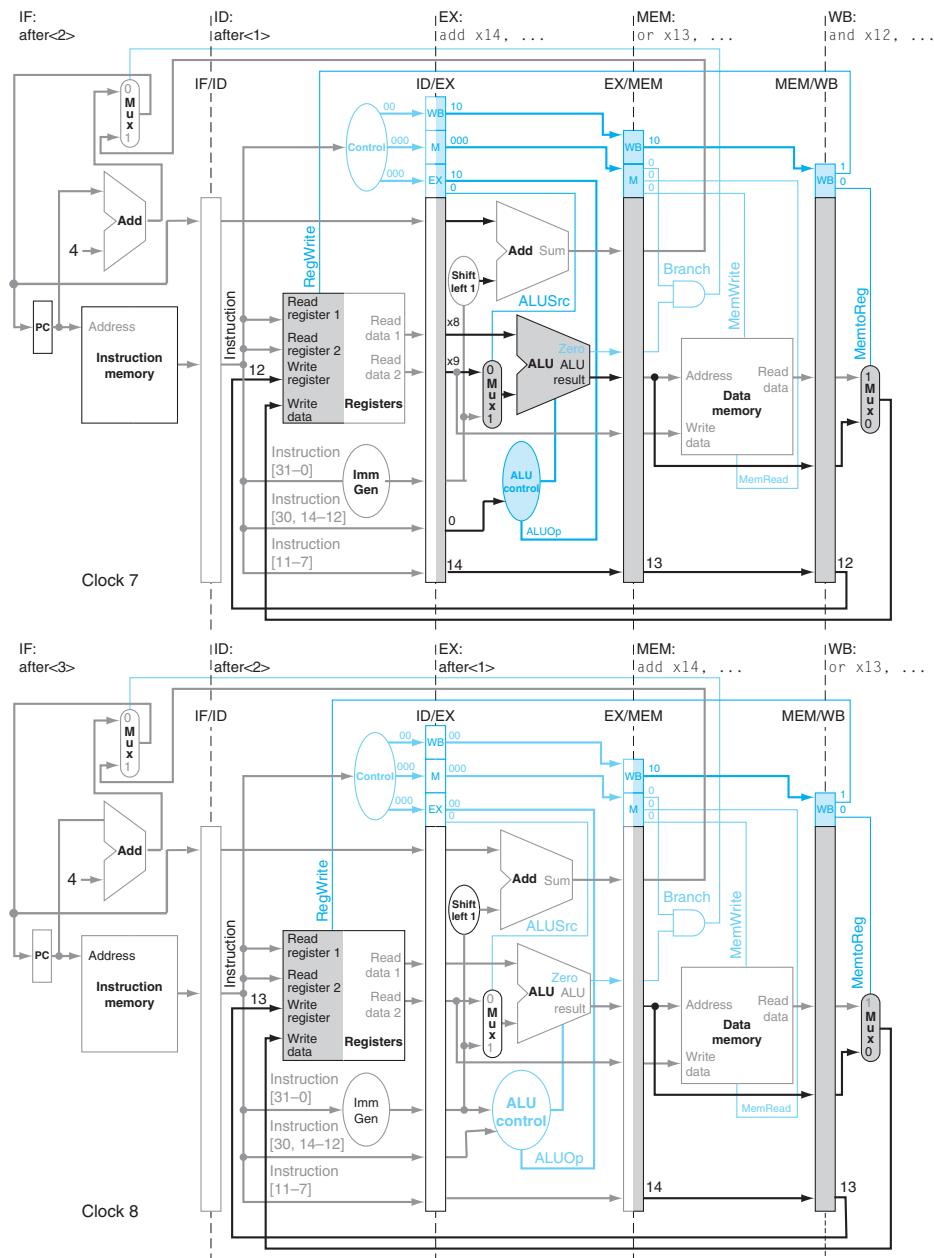
**FIGURE e4.13.11 Clock cycles 1 and 2.** The phrase “before  $<i>$ ” means the  $i$ th instruction before  $ld$ . The  $ld$  instruction in the top datapath is in the IF stage. At the end of the clock cycle, the  $ld$  instruction is in the IF/ID pipeline registers. In the second clock cycle, seen in 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.



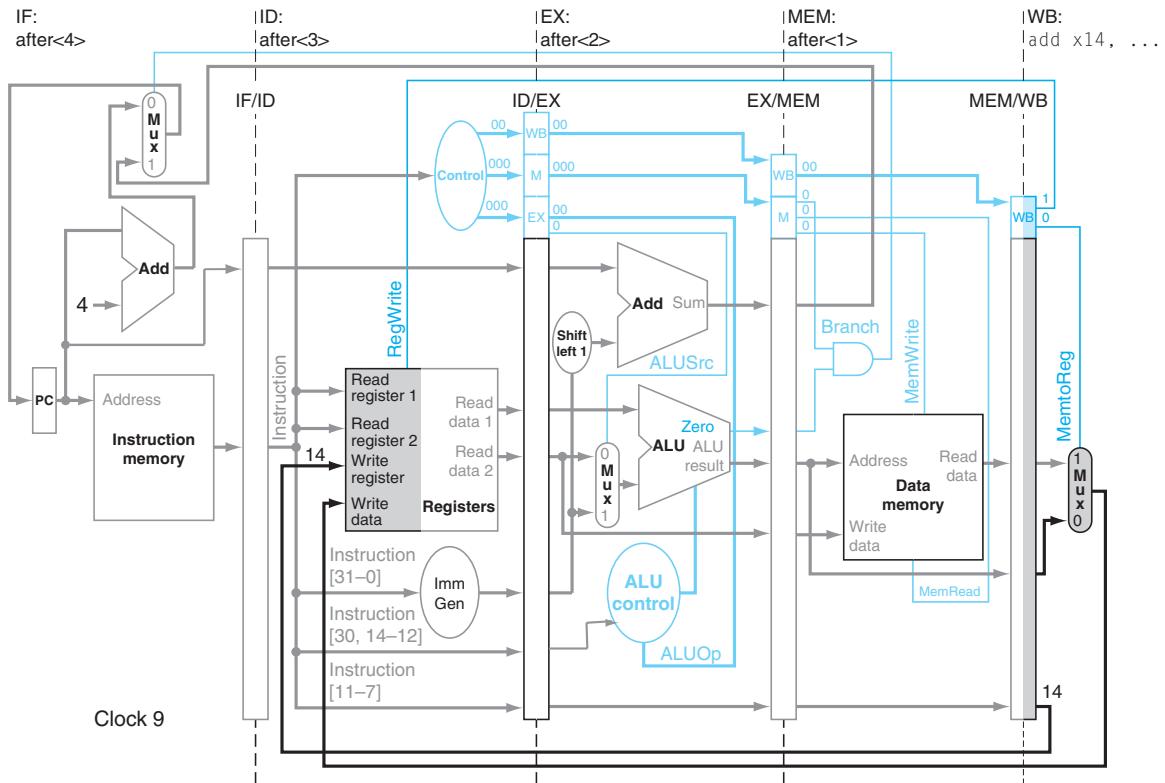
**FIGURE e4.13.12 Clock cycles 3 and 4.** In the top diagram,  $l_d$  enters the EX stage in the third clock cycle, adding  $x_1$  and 40 to form the address in the EX/MEM pipeline register. (The  $l_d$  instruction is written  $l_d \ x_{10}, \dots$  upon reaching EX, because the identity of instruction 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  $x_2$  and  $x_3$ , and the  $and$  instruction starts IF. In the fourth clock cycle (bottom datapath),  $l_d$  moves into MEM stage, reading memory using the value in EX/MEM as the address. In the same clock cycle, the ALU subtracts  $x_3$  from  $x_2$  and places the difference into EX/MEM, reads registers  $x_4$  and  $x_5$  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.



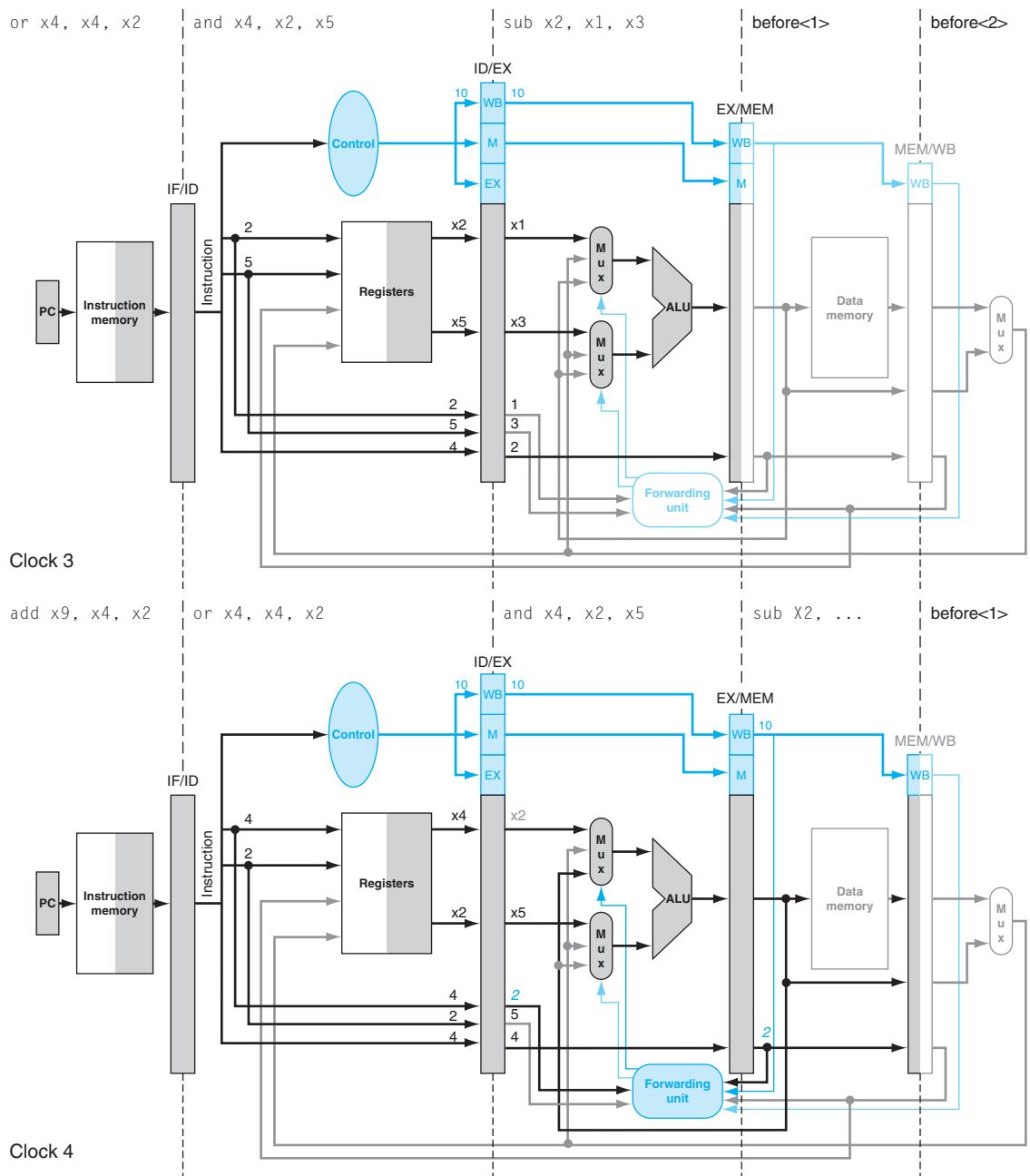
**FIGURE e4.13.13 Clock cycles 5 and 6.** With add, the final instruction in this example, entering IF in the top datapath, all instructions are engaged. By writing the data in MEM/WB into register 10, ldi completes; both the data and the register number are in MEM/WB. In the same clock cycle, sub sends 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.



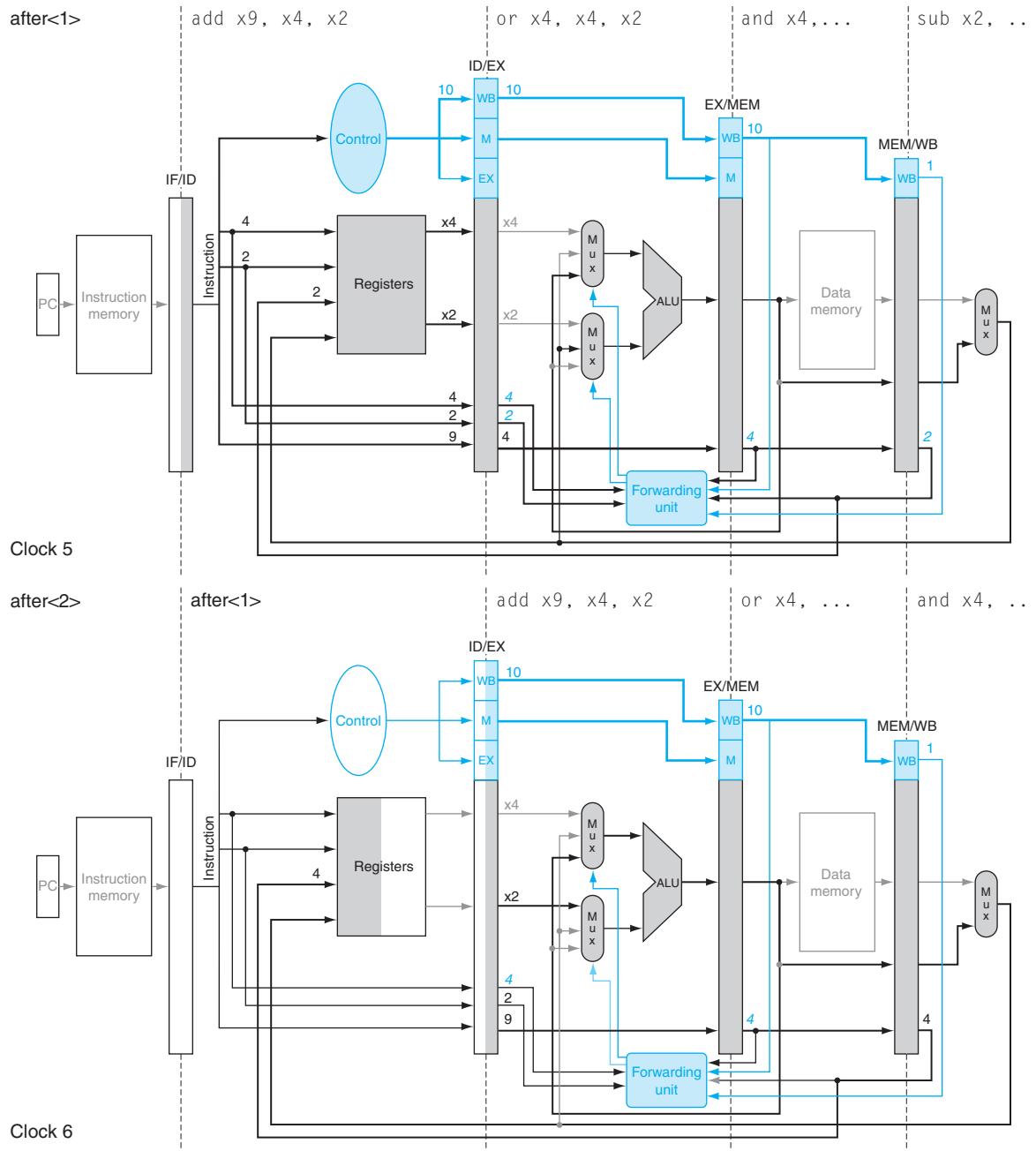
**FIGURE e4.13.14 Clock cycles 7 and 8.** In the top datapath, the add instruction brings up the rear, adding the values corresponding to registers  $x_8$  and  $x_9$  during the EX stage. The result of the  $\text{OR}$  instruction is passed from EX/MEM to MEM/WB in the MEM stage, and the WB stage writes the result of the and instruction in MEM/WB to register  $x_{12}$ . 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  $x_{13}$ , thereby completing  $\text{OR}$ , 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.



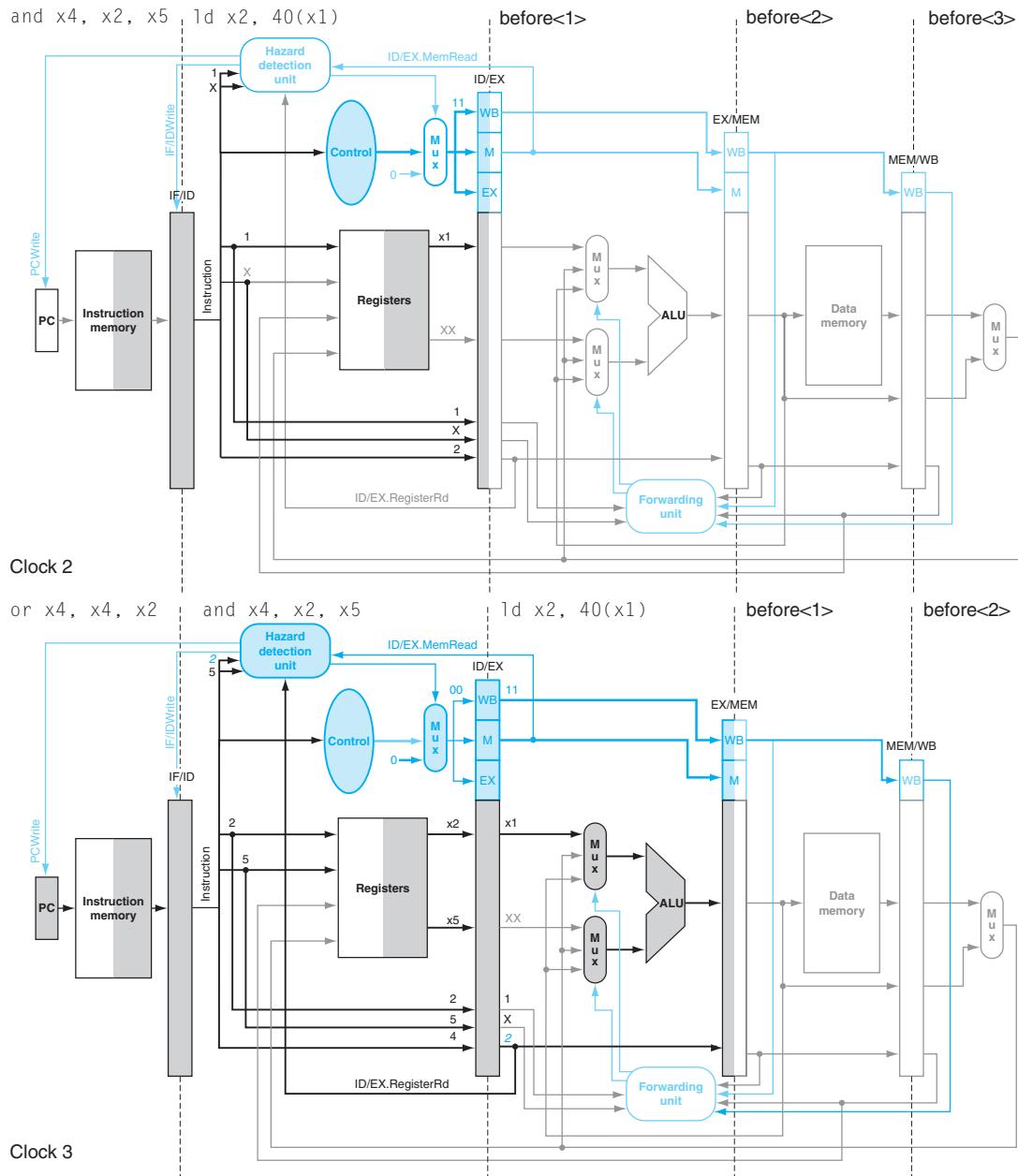
**FIGURE e4.13.15 Clock cycle 9.** The WB stage writes the ALU result in MEM/WB into register  $\times 14$ , completing add and the five-instruction sequence. The instructions after add are shown as inactive for pedagogical reasons.



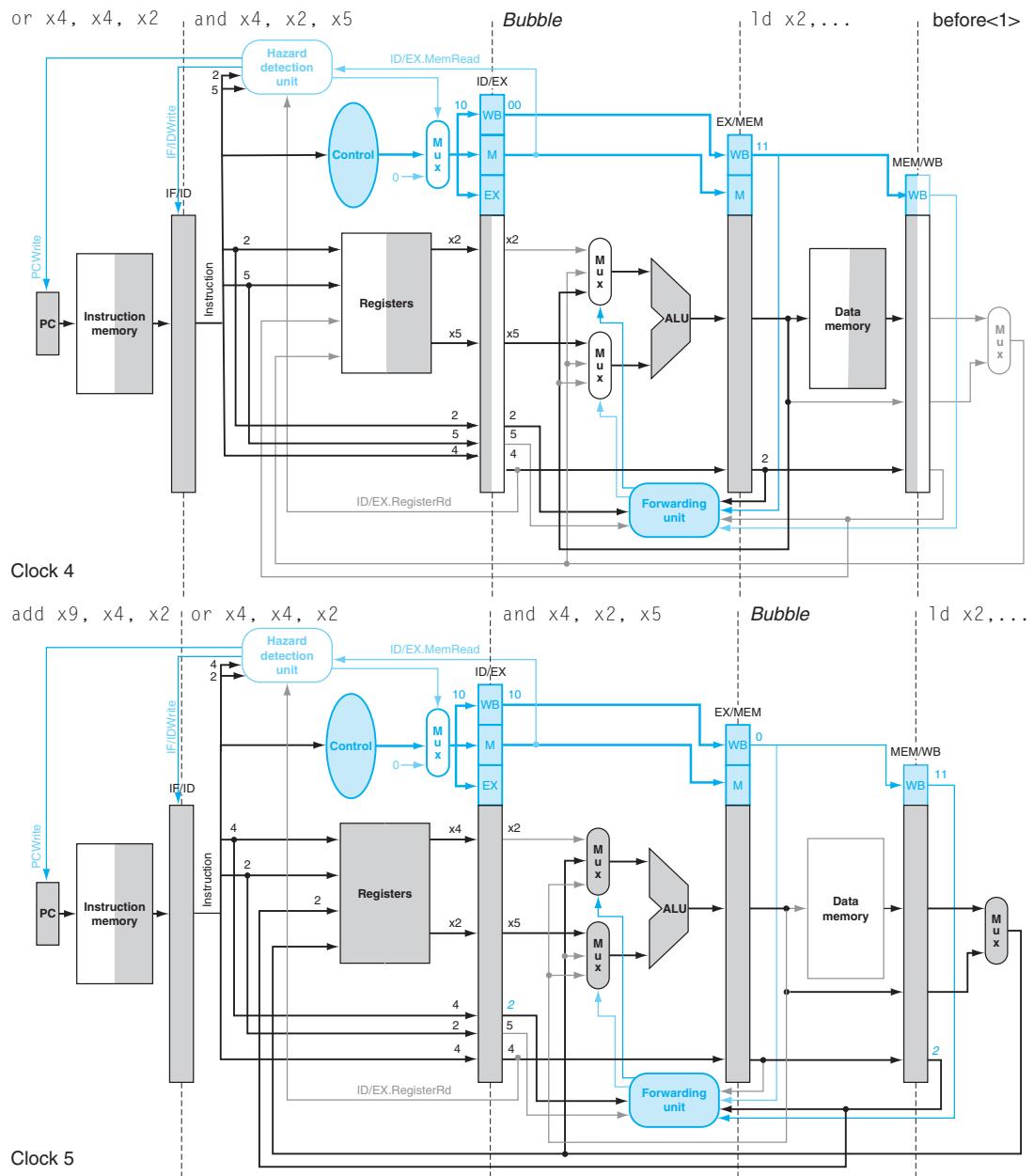
**FIGURE e4.13.16 Clock cycles 3 and 4 of the instruction sequence on page 366.e26.** The bold lines are those active in a clock cycle, and the italicized register numbers in color indicate a hazard. The forwarding unit is highlighted by shading it when it is forwarding data to the ALU. The instructions 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.



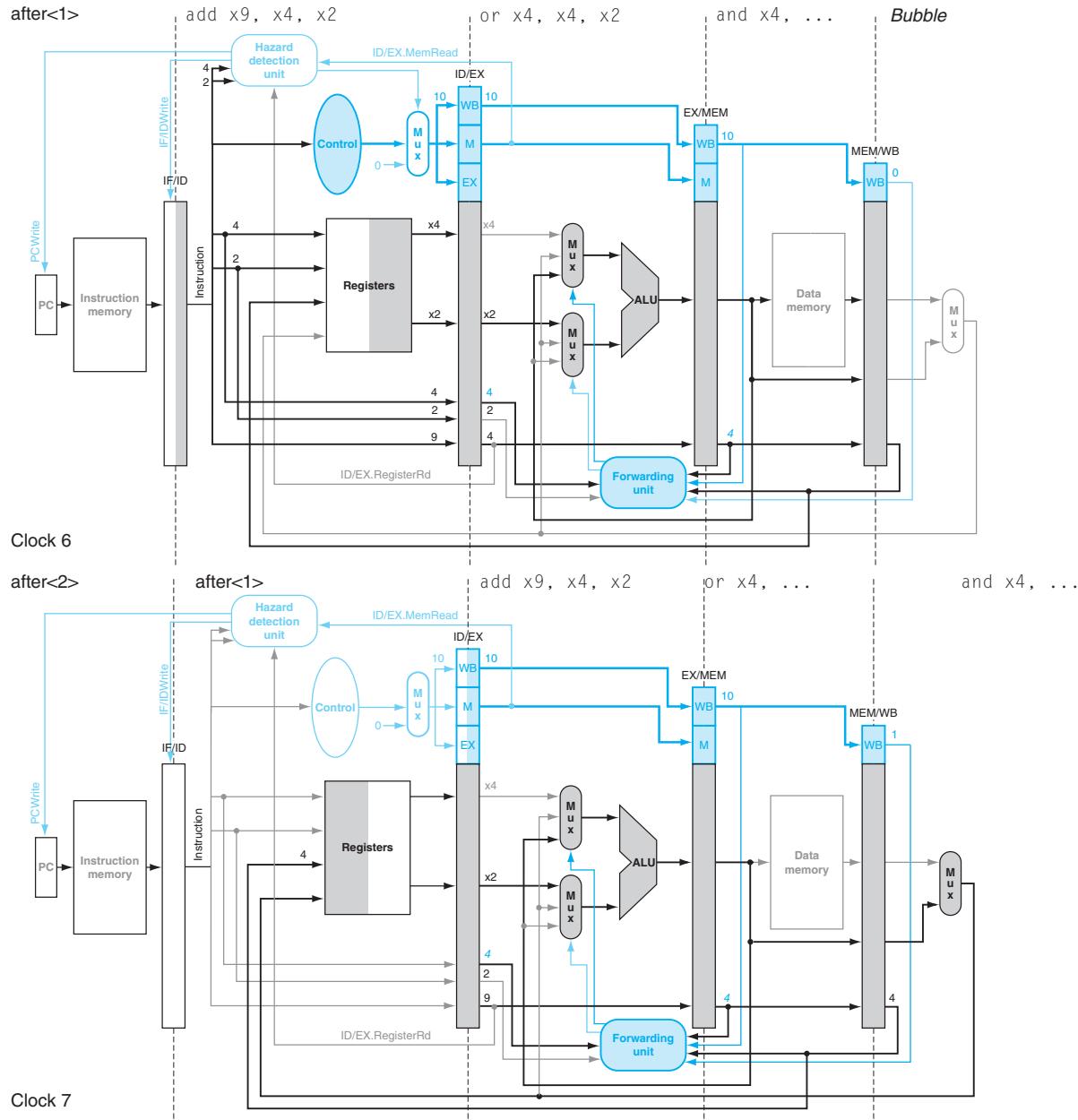
**FIGURE e4.13.17 Clock cycles 5 and 6 of the instruction sequence on page 366.e26.** The forwarding unit is highlighted when it is forwarding data to the ALU. The two instructions after add are shown as inactive just to emphasize what occurs for the four instructions in the example. The bold lines are 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 **1d** to the **ALU**. Note that in clock cycle 4, the forwarding unit forwards the address of the **1d** 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.



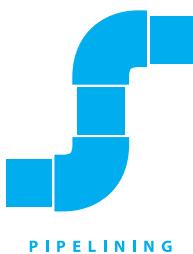
**FIGURE e4.13.20** Clock cycles 6 and 7 of the instruction sequence on page 366.e26 with a load replacing sub. Note that unlike in Figure e4.13.17, the stall allows the 1 d to complete, and so there is no forwarding from MEM/WB in clock cycle 6. Register x4 for the add in the EX stage still depends 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



**instruction latency** The inherent execution time for an instruction.

## 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.

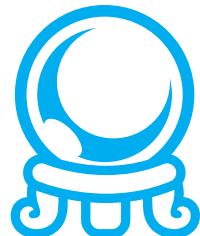
Pipelining improves throughput but not the inherent execution time, or **instruction latency**, of instructions; for some instructions, the latency is similar in length to the single-cycle approach. Multiple instruction issue adds additional datapath hardware to allow multiple instructions to begin every clock cycle, but at an increase in effective latency. Pipelining was presented as reducing the clock cycle time of the simple single-cycle datapath. Multiple instruction issue, in comparison, clearly focuses on reducing *clock cycles per instruction* (CPI).

Pipelining and multiple issue both attempt to exploit instruction-level parallelism. The presence of data and control dependences, which can become hazards, are the primary limitations on how much parallelism can be exploited. Scheduling and speculation via **prediction**, both in hardware and in software, are the primary techniques used to reduce the performance impact of dependences.

We showed that unrolling the DGEMM loop four times exposed more instructions that could take advantage of the out-of-order execution engine of the Core i7 to more than double performance.

The switch to longer pipelines, multiple instruction issue, and dynamic scheduling in the mid-1990s 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**.



PREDICTION



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:  $\text{Reg}[rd] = \text{Reg}[rs1] \text{ AND } \text{Reg}[rs2]$

**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

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.

**supercomputer:** *Any machine still on the drawing board.*

Stan Kelly-Bootle, *The Devil's DP Dictionary*, 1981



**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 Multi-flow 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-Id)	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 300ps 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 150ps to 160ps 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` (“Load With Increment”) instruction to RISC-V.

Interpretation:  $\text{Reg}[rd] = \text{Mem}[\text{Reg}[rs1] + \text{Reg}[rs2]]$

**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 rs<sub>1</sub>, rs<sub>2</sub> instruction to RISC-V.

Interpretation: Reg[rs<sub>2</sub>]=Reg[rs<sub>1</sub>]; Reg[rs<sub>1</sub>]=Reg[rs<sub>2</sub>]

**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 rs<sub>1</sub>, rs<sub>2</sub>, imm (Store Sum) instruction to RISC-V.

Interpretation: Mem[Reg[rs<sub>1</sub>]]=Reg[rs<sub>2</sub>]+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 rs<sub>1</sub> 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  $x_{11}$  is initialized to 11 and  $x_{12}$  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  $x_{13}$  and  $x_{14}$  be?

```
addi  x11, x12, 5
add   x13, x11, x12
addi  x14, x11, 15
```

**4.19** [10] <§4.5> Assume that  $x_{11}$  is initialized to 11 and  $x_{12}$  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  $x_{15}$  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  $.075 \times n$  NOPs possibly run faster on the pipeline with forwarding? Explain why or why not.

**4.21.5** [10] <§4.5> At minimum, how many NOPs (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 where the code above will stall.

**4.22.2** [5] <§4.5> In general, is it possible to reduce the number of stalls/NOPs 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 NOPs 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 1 <sup>st</sup> Only	MEM to 1 <sup>st</sup> Only	EX to 2 <sup>nd</sup> Only	MEM to 2 <sup>nd</sup> Only	EX to 1 <sup>st</sup> and EX to 2 <sup>nd</sup>
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 NOPs 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 NOPs could be inserted.

**4.26.3** [10] <§4.7> Analyzing each instruction independently will over-count the number of NOPs 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 1<sup>st</sup>” and “EX to 1<sup>st</sup> and EX to 2<sup>nd</sup>”. Why is there no entry for “MEM to 1<sup>st</sup> and MEM to 2<sup>nd</sup>”?

**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 NOPs to ensure correct execution.

**4.27.2** [10] <§4.7> Now, change and/or rearrange the code to minimize the number of NOPs 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	Id	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.

§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  $l_d$  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.

## Answers to Check Yourself