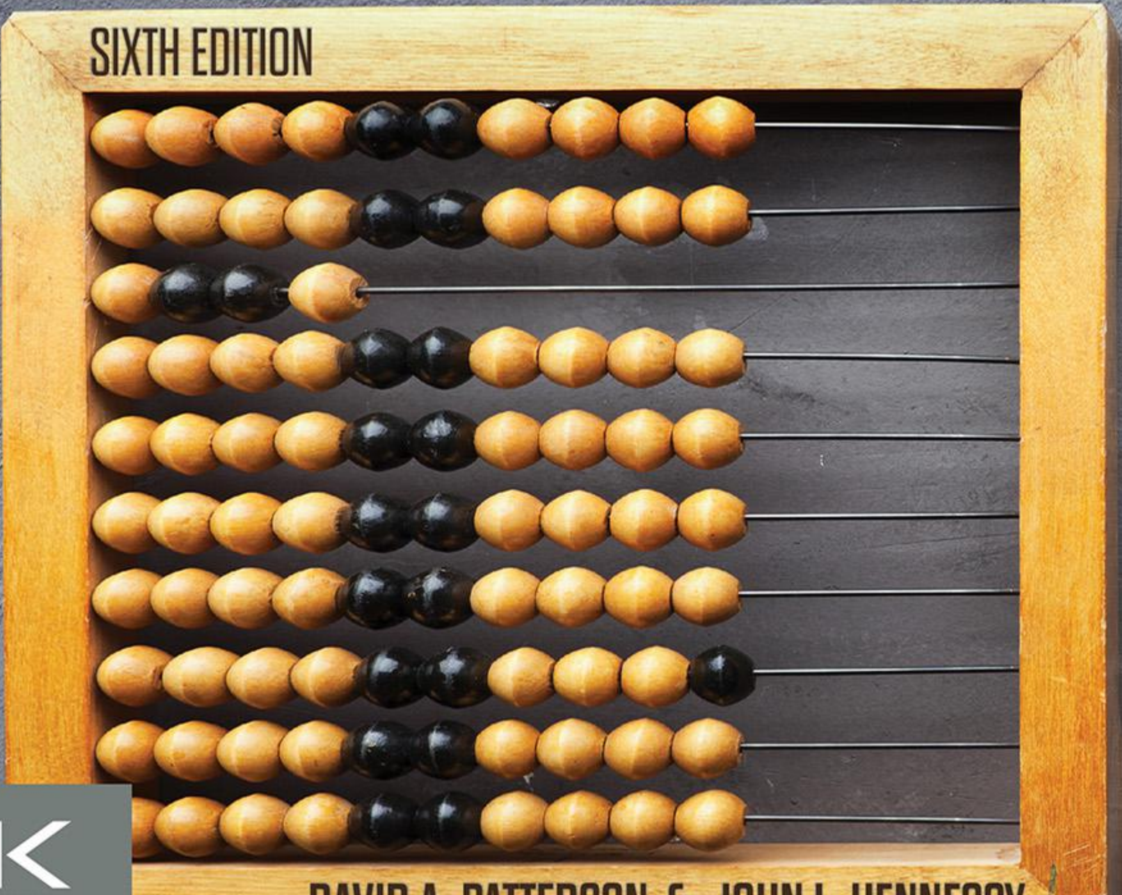
COMPUTER ORGANIZATION AND DESIGN MIPS EDITION

THE HARDWARE/SOFTWARE INTERFACE



SIXTH EDITION

MK MORGAN KAUFMANN

DAVID A. PATTERSON & JOHN L. HENNESSY

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

"With general purpose and specialized processors present in many aspects of everyday life, it is now more important than ever, that the next generation of computer engineers understand how computers compute and also the many tradeoffs and optimizations necessary to build fast energy-efficient machines. For several generations of students, *Computer Organization and Design* by Patterson and Hennesey has served as the gateway into the complex world of hardware/software interfaces; the memory hierarchy; and the benefits and hazards of pipelining (pun intended)."

—Mark Hempstead, *Tufts University*

"*Computer Organization and Design* is the computer architecture book for your (virtual) bookshelf. The book is both timeless and new, as it complements venerable principles—Moore's Law, abstraction, common case fast, redundancy, memory hierarchies, parallelism, and pipelining—with emerging trends from good (e.g., architectures targeting deep learning) to bad (processor core cyberattacks)."

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

"The new edition of *Computer Organization and Design* keeps pace with advances in emerging processor architectures and applications, where AI, security and virtualization will be supported on open instruction set architectures (e.g., RISC-V). 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 will power next generation secure, performant and efficient systems."

—Dave Kaeli, *Northeastern University*

"There are timeless principles in computer system design, which are essential to understand the organization and performance of any computer architecture. Based on these principles and using a unique pedagogical approach, Patterson and Hennessy present the evolution of computer architecture from uniprocessors to the latest innovations on domain-specific architectures. The inclusion of the Google TPU supercomputer as an example of DNN-DSA in this new edition, heralds the rise of a new generation of computer architects."

—Euripides Montagne, *University of Central Florida*

"*Computer Organization and Design* is the ultimate classic textbook that is still current and applicable. It is very readable, and provides valuable insight into the hardware/software interface, which is useful both for future hardware engineers and for software developers interested in improving performance and energy-efficiency. The MIPS architecture is ideal for teaching computer organization. It is straightforward, yet closely resembles both ARM and RISC-V."

—Tali Moreshet, *Boston University*

This page intentionally left blank

SIXTH EDITION

Computer Organization and Design

THE HARDWARE/SOFTWARE INTERFACE

David A. Patterson has been teaching computer architecture at the University of California, Berkeley, since joining the faculty in 1977, where he held the Pardee Chair of Computer Science. His teaching has been honored by the Distinguished Teaching Award from the University of California, the Karlstrom Award from ACM, and the Mulligan Education Medal and Undergraduate Teaching Award from IEEE. Patterson received the IEEE Technical Achievement Award and the ACM Eckert-Mauchly Award for contributions to RISC, and he shared the IEEE Johnson Information Storage Award for contributions to RAID. He also shared the IEEE John von Neumann Medal and the C & C Prize with John Hennessy. Like his co-author, Patterson is a Fellow of both AAAS organizaitons, 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 as chair of the CS division in the Berkeley EECS department, as chair of the Computing Research Association, and as President of ACM. This record led to Distinguished Service Awards from ACM and CRA. He received the Tapia Achievement Award for Civic Science and Diversifying Computing and shared the 2017 ACM A.M. Turing Award with Hennessy.

At Berkeley, Patterson led the design and implementation of RISC I, likely the first VLSI reduced instruction set computer, and the foundation of the commercial SPARC architecture. He was a leader of the Redundant Arrays of Inexpensive Disks (RAID) project, which led to dependable storage systems from many companies. He was also involved in the Network of Workstations (NOW) project, which led to cluster technology used by Internet companies and later to cloud computing. These projects earned three dissertation awards from ACM. In 2016 he became Professor Emeritus at Berkeley and a Distinguished Engineer at Google, where he works on domain specific architectures for machine learning. He is also the Vice Chair of RISC-V International and the Director of the RISC-V International Open Source Laboratory.

John L. Hennessy was the tenth president of Stanford University, where he has been a member of the faculty since 1977 in the departments of electrical engineering and computer science. Hennessy is a Fellow of the IEEE and ACM; a member of the National Academy of Engineering, the National Academy of Science, and the American Philosophical Society; and a Fellow of the American Academy of Arts and Sciences. Among his many awards are the 2001 Eckert-Mauchly Award for his contributions to RISC technology, the 2001 Seymour Cray Computer Engineering Award, and the 2000 John von Neumann Award, which he shared with David Patterson. In 2017 they shared the ACM A.M. Turing Award. 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.

He is currently director of Knight-Hennessy Scholars and serves as non-executive chairman of Alphabet.

SIXTH EDITION

Computer Organization and Design

THE HARDWARE/SOFTWARE INTERFACE

David A. Patterson
University of California, Berkeley
Google, Inc.

John L. Hennessy
Stanford University

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

Senior Acquisitions Editor: Stephen R. Merken
Content Development Specialist: Beth LoGiudice
Project Manager: Beula Christopher
Designer: Patrick Ferguson

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

Library of Congress Cataloging-in-Publication Data

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

ISBN: 978-0-12-820109-1

Printed in the United States of America

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

Contents

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 switch from uniprocessor to multicore microprocessors and the recent emphasis on domain specific architectures 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. 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 modern computers.

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

About the Other Book

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

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

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

Changes for the Sixth Edition

There is arguably been more change in the technology and business of computer architecture since the fifth edition than there were for the first five:

- *The slowing of Moore's Law*. After 50 years of biannual doubling of the number of transistors per chip, Gordon Moore's prediction no longer holds. Semiconductor technology will still improve, but more slowly and less predictably than in the past.
- *The rise of Domain Specific Architectures (DSA)*. In part due to the slowing of Moore's Law and in part due to the end of Dennard Scaling, general purpose processors are only improving a few percent per year. Moreover, Amdahl's Law limits the practical benefit of increasing the number of processors per chip. In 2020, it is widely believed that the most promising path forward is DSA. It doesn't try to run everything well like general purpose processors, but focuses on running programs of one domain much better than conventional CPUs.
- *Microarchitecture as a security attack surface*. Spectre demonstrated that speculative out-of-order execution and hardware multithreading make

Chapter or Appendix	*Sections*	*Software focus*	*Hardware focus*
1. Computer Abstractions and Technology	1.1 to 1.12	🟦🟦	🟦🟦
	🌐 1.13 (History)	▨▨	▨▨
2. Instructions: Language of the Computer	2.1 to 2.14	🟦🟦	🟦⬜
	🌐 2.15 (Compilers & Java)	▨⬜	
	2.16 to 2.22	🟦🟦	🟦⬜
	🌐 2.23 (History)	▨▨	▨▨
E. RISC Instruction-Set Architectures	🌐 E.1 to E.6	▨⬜	
3. Arithmetic for Computers	3.1 to 3.5	🟦⬜	🟦⬜
	3.6 to 3.8 (Subword Parallelism)	🟦🟦	🟦🟦
	3.9 to 3.10 (Fallacies)	🟦⬜	🟦⬜
	🌐 3.11 (History)	▨▨	▨▨
B. The Basics of Logic Design	B.1 to B.13		🟦⬜
4. The Processor	4.1 (Overview)	🟦🟦	🟦🟦
	4.2 (Logic Conventions)		🟦🟦
	4.3 to 4.4 (Simple Implementation)	🟦⬜	🟦🟦
	4.5 (Multicycle Implementation)		▨⬜
	4.6 (Pipelining Overview)	🟦🟦	🟦🟦
	4.7 (Pipelined Datapath)	🟦⬜	🟦🟦
	4.8 to 4.10 (Hazards, Exceptions)		🟦🟦
	4.11 to 4.13 (Parallel, Real Stuff)	🟦🟦	🟦🟦
	🌐 4.14 (Verilog Pipeline Control)		▨⬜
	4.15 to 4.16 (Fallacies)	🟦🟦	🟦🟦
	🌐 4.17 (History)	▨▨	▨▨
D. Mapping Control to Hardware	🌐 D.1 to D.6		▨⬜
5. Large and Fast: Exploiting Memory Hierarchy	5.1 to 5.10	🟦🟦	🟦🟦
	🌐 5.11 (Redundant Arrays of Inexpensive Disks)	▨⬜	▨⬜
	🌐 5.12 (Verilog Cache Controller)		▨⬜
	5.13 to 5.16	🟦🟦	🟦🟦
	🌐 5.17 (History)	▨▨	▨▨
6. Parallel Process from Client to Cloud	6.1 to 6.9	🟦🟦	🟦🟦
	🌐 6.10 (Clusters)	▨⬜	▨⬜
	6.11 to 6.15	🟦🟦	🟦🟦
	🌐 6.16 (History)	▨▨	▨▨
A. Assemblers, Linkers, and the SPIM Simulator	A.1 to A.11	▨⬜	▨⬜
C. Graphics Processor Units	🌐 C.1 to C.11	⬜⬜	⬜⬜

Read carefully	🟦🟦	Read if have time	▨⬜	Reference	⬜⬜
Review or read	🟦⬜	Read for culture	▨▨		

timing based side-channel attacks practical. Moreover, these are not due to bugs that can be fixed, but a fundamental challenge to this style of processor design.

- *Open instruction sets and open source implementations*. The opportunities and impact of open source software have come to computer architecture. Open instruction sets like RISC-V enables organizations to build their own processors without first negotiating a license, which has enabled open-source implementations that are shared to freely download and use as well as proprietary implementations of RISC-V. Open source software and hardware are a boon to academic research and instruction, allowing students to see and enhance industrial strength technology.
- *The re-virticalization of the information technology industry*. Cloud computing has led to no more than a half-dozen companies that provide computing infrastructure for everyone to use. Much like IBM in the 1960s and 1970s, these companies determine both the software stack and the hardware that they deploy. The changes above have led to some of these "hyperscalers" developing their own DSA and RISC-V chips for deployment in their clouds.

The sixth edition of COD reflects these recent changes, updates all the examples and figures, responds to requests of instructors, plus adds a pedagogic improvement inspired by textbooks I used to help my grandchildren with their math classes.

- The Going Faster section is now in every chapter. It starts with a Python version in Chapter 1, whose poor performance inspires learning C and then rewriting matrix multiply in C in Chapter 2. The remaining chapters accelerate matrix multiply by leveraging data level parallelism, instruction level parallelism, thread level parallelism, and by adjusting memory accesses to match the memory hierarchy of a modern server. This computer has 512-bit SIMD operations, speculative out-of-order execution, three levels of caches, and 48 cores. All four optimizations add only 21 lines of C code yet speedup matrix multiply by almost 50,000, cutting it from nearly 6 hours in Python to less than 1 second in optimized C. If I were a student again, this running example would inspire me to use C and learn the underlying hardware concepts of this book.
- With this edition, every chapter has a Self Study section that asks thought provoking questions and supplies the answers afterwards to help you evaluate if you follow the material on your own.
- Besides explaining that Moore's Law and Dennard Scaling no longer hold, we've de-emphasized Moore's Law as a change agent that was prominent in the fifth edition.
- Chapter 2 has more material to emphasize that binary data has no inherent meaning—the program determines the data type—not an easy concept for beginners to grasp.

- Chapter 2 also includes a short description of the RISC-V as a contrasting instruction set to MIPS alongside ARMv7, ARMv8, and x86. (There is also a companion version of this book based on RISC-V instead of MIPS, and we're updating that with the other changes as well.)
- The benchmark example of Chapter 2 is upgraded to SPEC2017 from SPEC2006.
- At instructors' request, we have restored the multi-cycle implementation of MIPS as an online section in Chapter 4 between the single-cycle implementation and the pipelined implementation. Some instructors find these three steps an easier path to teach pipelining.
- The Putting It All Together examples of Chapters 4 and 5 were updated to the recent ARM A53 microarchitecture and the Intel i7 6700 Skyelake microarchitecture.
- The Fallacies and Pitfalls Sections of Chapters 5 and 6 added pitfalls around hardware security attacks of Row Hammer and Spectre.
- Chapter 6 has a new section introducing DSAs using Google's Tensor Processing Unit (TPU) version 1. Chapter 6's Putting it All Together section is updated to compare Google's TPUv3 DSA supercomputer to a cluster of NVIDIA Volta GPUs.

Finally, we updated all the exercises in the book.

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

Instructor Support

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

https://textbooks.elsevier.com/web/manuals.aspx?isbn=9780128201091

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.

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 for the Sixth Edition

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

Special thanks goes to Dr. Rimas Avizenis, who developed the various versions of matrix multiply and supplied the performance numbers as well. I deeply appreciate his continued help after he has graduated from UC Berkeley. As I worked with his father while I was a graduate student at UCLA, it was a nice symmetry to work with Rimas when he was a graduate student at UC Berkeley.

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

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

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

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

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

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

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

We wish to thank the extended Morgan Kaufmann family for agreeing to publish this book again under the able leadership of **Steve Merken** and **Beth LoGiudice**: I certainly couldn't have completed the book without them. We also want to extend thanks to **Beula Christopher**, who managed the book production process, and **Patrick Ferguson**, who did the cover design.

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

David A. Patterson

1 Computer Abstractions and Technology

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

Alfred North Whitehead,
An Introduction to Mathematics, 1911

1.1 Introduction

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. This unusual industry embraces innovation at a breathtaking rate. In the last 40 years, there have been a number of new computers whose introduction appeared to revolutionize the computing industry; these revolutions were cut short only because someone else built an even better computer.

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

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

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

- *Computers in automobiles:* Until microprocessors improved dramatically in price and performance in the early 1980s, computer control of cars was ludicrous. Today, computers reduce pollution, improve fuel efficiency via engine controls, and increase safety through nearly automated driving and air bag inflation to protect occupants in a crash.
- *Cell phones:* Who would have dreamed that advances in computer systems would lead to more than half of the planet having mobile phones, allowing person-to-person communication to almost anyone anywhere in the world?
- *Human genome project:* The cost of computer equipment to map and analyze human DNA sequences was hundreds of millions of dollars. It's unlikely that anyone would have considered this project had the computer costs been 10 to 100 times higher, as they would have been 15 to 25 years earlier. Moreover, costs continue to drop; you will soon be able to acquire your own genome, allowing medical care to be tailored to you.
- *World Wide Web:* Not in existence at the time of the first edition of this book, the web has transformed our society. For many, the web has replaced libraries and newspapers.
- *Search engines:* As the content of the web grew in size and in value, finding relevant information became increasingly important. Today, many people rely on search engines for such a large part of their lives that it would be a hardship to go without them.

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

Classes of Computing Applications and Their Characteristics

Although a common set of hardware technologies (see Sections 1.4 and 1.5) is used in computers ranging from smart home appliances to cell phones to the largest supercomputers, these different applications have different design requirements and employ the core hardware technologies in different ways. Broadly speaking, computers are used in three different classes of applications.

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

Personal computers (PCs) in the form of laptops are possibly the best known form of computing, which readers of this book have likely used extensively. Personal computers emphasize delivery of good performance to single users at low cost and usually execute third-party software. This class of computing drove the evolution of many computing technologies, which is only about 40 years old!

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

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

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.

Servers span the widest range in cost and capability. At the low end, a server may be little more than a desktop computer without a screen or keyboard and cost a thousand dollars. These low-end servers are typically used for file storage, small business applications, or simple web serving (see Section 6.11). At the other extreme are **supercomputers**, which at the present consist of hundreds of thousands of processors and many **terabytes** of memory, and cost tens to hundreds of millions of dollars. Supercomputers are usually used for high-end scientific and engineering calculations, such as weather forecasting, oil exploration, protein structure determination, and other large-scale problems. Although such supercomputers represent the peak of computing capability, they represent a relatively small fraction of the servers and a relatively small fraction of the overall computer market in terms of total revenue.

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

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. A popular term today is Internet of Things (IoT), which suggests many small devices that all communicate wirelessly over the Internet. 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!

Decimal term	Abbreviation	Value	Binary term	Abbreviation	Value	% Larger
kilobyte	KB	1000^1	kibibyte	KiB	2^{10}	2%
megabyte	MB	1000^2	mebibyte	MiB	2^{20}	5%
gigabyte	GB	1000^3	gibibyte	GiB	2^{30}	7%
terabyte	TB	1000^4	tebibyte	TiB	2^{40}	10%
petabyte	PB	1000^5	pebibyte	PiB	2^{50}	13%
exabyte	EB	1000^6	exbibyte	EiB	2^{60}	15%
zettabyte	ZB	1000^7	zebibyte	ZiB	2^{70}	18%
yottabyte	YB	1000^8	yobibyte	YiB	2^{80}	21%
ronnabyte	RB	1000^9	robibyte	RiB	2^{90}	24%
queccabyte	QB	1000^{10}	quebibyte	QiB	2^{100}	27%

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. The society that runs the metric system created the decimal prefixes, with the last two proposed only in 2019 in anticipation of the global capacity of storage systems. All the names are derived from the entymology in Latin of the powers of 1000 that they represent.

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

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

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

Welcome to the PostPC Era

The continuing march of technology brings about generational changes in computer hardware that shake up the entire information technology industry. Since the fourth edition of the book we have undergone such a change, as significant in the

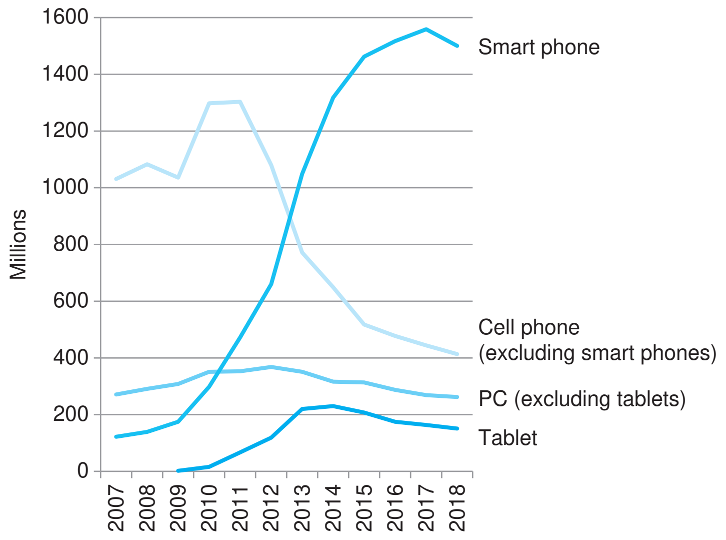


FIGURE 1.2 The number manufactured per year of tablets and smart phones, which reflect the PostPC era, versus personal computers and traditional cell phones. Smart phones represent the recent growth in the cell phone industry, and they passed PCs in 2011. PCs, tablets, and traditional cell phone categories are declining. The peak volume years text are 2011 for cell phones, 2013 for PCs, and 2014 for tablets. PCs fell from 20% of total units shipped in 2007 to 10% in 2018.

past as the switch starting 40 years ago to personal computers. Replacing the PC is the **personal mobile device (PMD)**. PMDs are battery operated with wireless connectivity to the Internet and typically cost hundreds of dollars, and, like PCs, users can download software ("apps") to run on them. Unlike PCs, they no longer have a keyboard and mouse, and are more likely to rely on a touch-sensitive screen or even speech input. Today's PMD is a smart phone or a tablet computer, but tomorrow it may include electronic glasses. Figure 1.2 shows the rapid growth time of tablets and smart phones versus that of PCs and traditional cell phones.

Taking over from the traditional server is **Cloud Computing**, which relies upon giant datacenters that are now known as *Warehouse Scale Computers* (WSCs). Companies like Amazon and Google build these WSCs containing 50,000 servers and then let companies rent portions of them so that they can provide software services to PMDs without having to build WSCs of their own. Indeed, **Software as a Service (SaaS)** deployed via the cloud is revolutionizing the software industry just as PMDs and WSCs are revolutionizing the hardware industry. Today's software developers will often have a portion of their application that runs on the PMD and a portion that runs in the Cloud.

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

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

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

What You Can Learn in This Book

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

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

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

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

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

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

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

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

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

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

Understanding Program Performance

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

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

Check Yourself

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

1. The number of embedded processors sold every year greatly outnumbers the number of PC and even PostPC processors. Can you confirm or deny this insight based on your own experience? Try to count the number of embedded processors in your home. How does it compare with the number of conventional computers in your home?
2. As mentioned earlier, both the software and hardware affect the performance of a program. Can you think of examples where each of the following is the right place to look for a performance bottleneck?
   - The algorithm chosen
   - The programming language or compiler
   - The operating system
   - The processor
   - The I/O system and devices

1.2 Seven Great Ideas in Computer Architecture

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

Use Abstraction to Simplify Design

ABSTRACTION

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. A major productivity technique for hardware and software is to use **abstractions** to represent the design at different levels of representation; lower-level details are hidden to offer a simpler model at higher levels. We'll use the abstract painting icon to represent this second great idea.

Make the Common Case Fast

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

COMMON CASE FAST

Performance via Parallelism

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

PARALLELISM

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.

PIPELINING

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.

PREDICTION

Hierarchy of Memories

Programmers want memory to be fast, large, and cheap, as memory speed often shapes performance, capacity limits the size of problems that can be solved, and the cost of memory today is often the majority of computer cost. Architects have found that they can address these conflicting demands with a **hierarchy of memories**, with

HIERARCHY

the fastest, smallest, and most expensive memory per bit at the top of the hierarchy and the slowest, largest, and cheapest per bit at the bottom. As we shall see in Chapter 5, caches give the programmer the illusion that main memory is nearly as fast as the top of the hierarchy and nearly as big and cheap as the bottom of the hierarchy. We use a layered triangle icon to represent the memory hierarchy. The shape indicates speed, cost, and size: the closer to the top, the faster and more expensive per bit the memory; the wider the base of the layer, the bigger the memory.

Dependability via Redundancy

DEPENDABILITY

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!)

In the prior edition, we listed an eighth great idea, which was "Designing for Moore's Law." Gordon Moore, one of the founders of Intel, made a remarkable prediction in 1965: integrated circuit resources would double every year. A decade later he amended his prediction to doubling every 2 years.

His prediction was accurate, and for 50 years, Moore's Law shaped computer architecture. As computer designs can take years, the resources available per chip ("transistors"; see page 24) could easily double or triple between the start and finish of the project. Like a skeet shooter, computer architects had to anticipate where the technology would be when the design finishes rather than design for when it starts.

Alas, no exponential growth can last forever, and Moore's Law is no longer accurate. The slowing of Moore's Law is shocking for computer designers who have long leveraged it. Some do not want to believe it is over, despite the substantial evidence to the contrary. Part of the reason is confusion between saying that Moore's prediction of the biannual doubling rate is now incorrect and claiming that semiconductors will no longer improve. Semiconductor technology *will* continue to improve, but more slowly than in the past. Starting with this edition, we will discuss the implications of the slowing of Moore's Law, especially in Chapter 6.

Elaboration: During the heydays of Moore's Law, the cost per chip resource would drop with each new technology generation. In the latest technologies, the cost per resource may be flat or even *rising* with each new generation, due to the cost of the new equipment, the elaborate processes invented to make chips work at these finer feature sizes, and the reduction of the *number* of companies who are investing in these new technologies to push the state-of-the-art. Less competition naturally leads to higher prices.

1.3 Below Your Program

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

Mark Twain, *The Innocents Abroad*, 1869

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

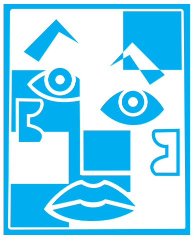

ABSTRACTION

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

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

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

- 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, Android, and Windows.

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

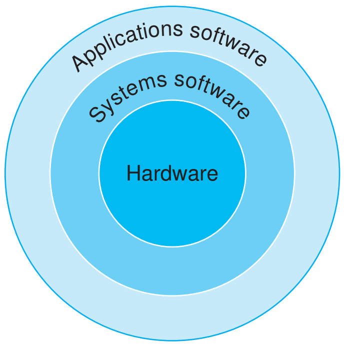


FIGURE 1.3 A simplified view of hardware and software as hierarchical layers, shown as concentric circles with hardware in the center and applications software outermost. In complex applications, there are often multiple layers of application software as well. For example, a database system may run on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

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

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

From a High-Level Language to the Language of Hardware

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

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.

```
1000110010100000
```

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

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

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

```
add A,B
```

and the assembler would translate this notation into

```
1000110010100000
```

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

assembly language A symbolic representation of machine instructions.

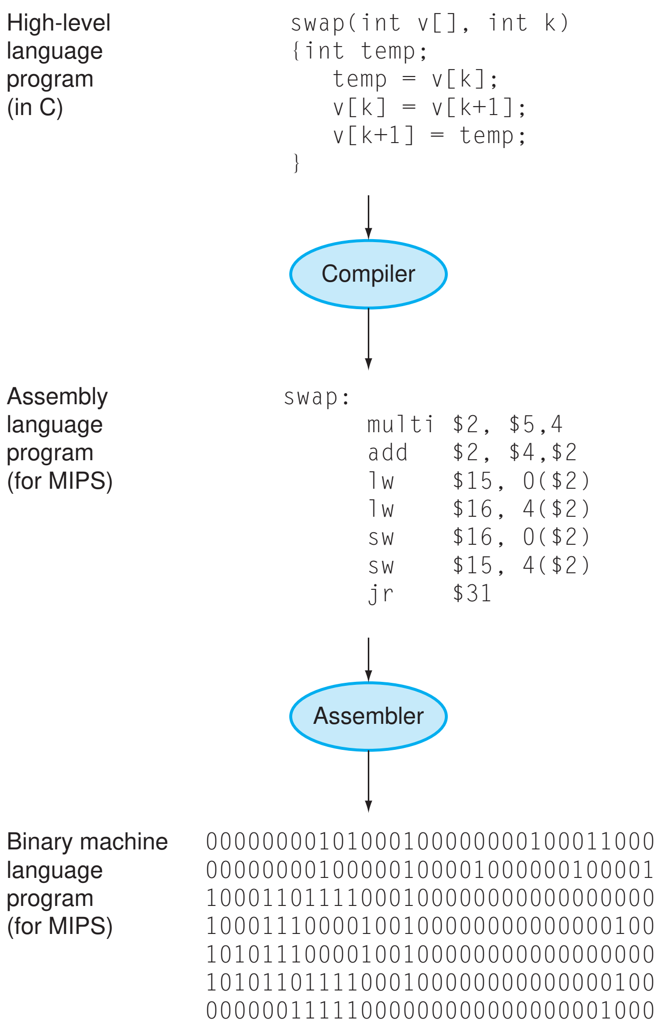
machine language A binary representation of machine instructions.

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

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


High-level language program (in C)
Compiler
Assembly language program (for MIPS)
Assembler
Binary machine language program (for MIPS)

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

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

```
00000000101000100000000100011000
00000000100000100001000000100001
10001101111000100000000000000000
10001110000100100000000000000100
10101110000100100000000000000000
10101101111000100000000000000100
00000011111000000000000000001000
```

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 machine learning, for example.

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

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

1.4 Under the Covers

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

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

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

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

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

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

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

The 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.

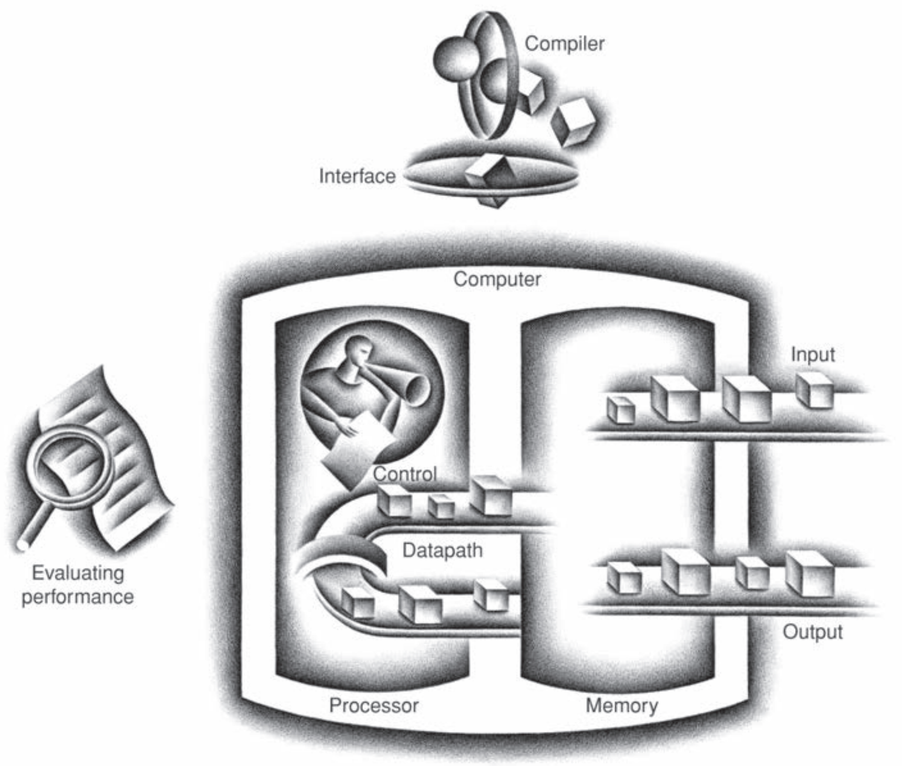


FIGURE 1.5 The organization of a computer, showing the five classic components. The processor gets instructions and data from memory. Input writes data to memory, and output reads data from memory. Control sends the signals that determine the operations of the datapath, memory, input, and output.

Through the Looking Glass

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

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

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

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

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

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

The image is composed of a matrix of picture elements, or **pixels**, which can be represented as a matrix of bits, called a *bit map*. Depending on the size of the screen and the resolution, the display matrix in a typical tablet ranges in size from 1024 X 768 to 2048 X 1536. A color display might use 8 bits for each of the three colors (red, blue, and green), for 24 bits per pixel, permitting millions of different colors to be displayed.

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

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

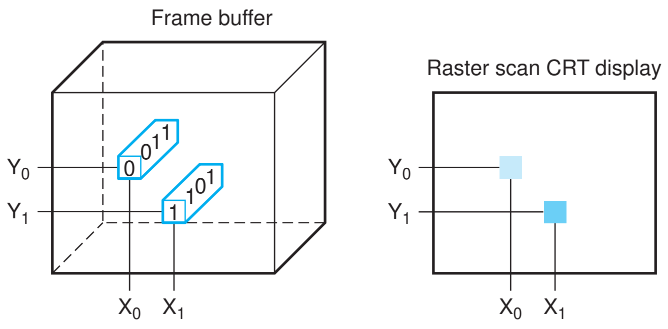


FIGURE 1.6 Each coordinate in the frame buffer on the left determines the shade of the corresponding coordinate for the raster scan CRT display on the right. Pixel (X_0, Y_0) contains the bit pattern 0011, which is a lighter shade on the screen than the bit pattern 1101 in pixel (X_1, Y_1).

Touchscreen

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

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

Opening the Box

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

The small rectangles in Figure 1.8 contain the devices that drive our advancing technology, called **integrated circuits** and nicknamed **chips**. The A12 package seen in the middle of in Figure 1.8 contains two large ARM processors and four little ARM processors that operate with a clock rate of 2.5 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 iPhone Xs Max package in Figure 1.8 also includes a memory chip with 32 gibibits or 2 GiB of capacity. 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 a DRAM chip. *DRAM* stands for **dynamic random access memory**. 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.

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 2020 was $3 to $6.

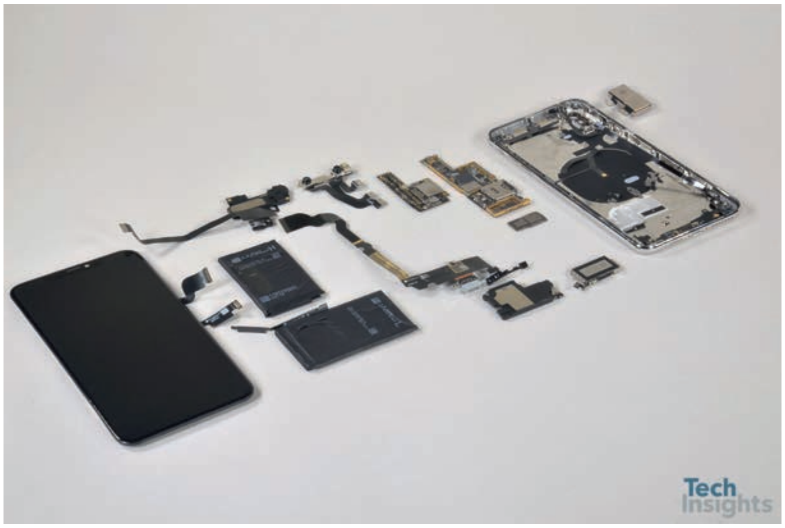


FIGURE 1.7 Components of the Apple iPhone Xs Max cell phone. At the left is the capacitive multitouch screen and LCD display. Next to it is the battery. To the far right is the metal frame that attaches the LCD to the back of the iPhone. The small components surrounding in the center are what we think of as the computer; they are not simple rectangles to fit compactly inside the case next to the battery. Figure 1.8 shows a close-up of the board to the left of the metal case, which is the logic printed circuit board that contains the processor and the memory (Courtesy TechInsights, www.techIngishts.com).

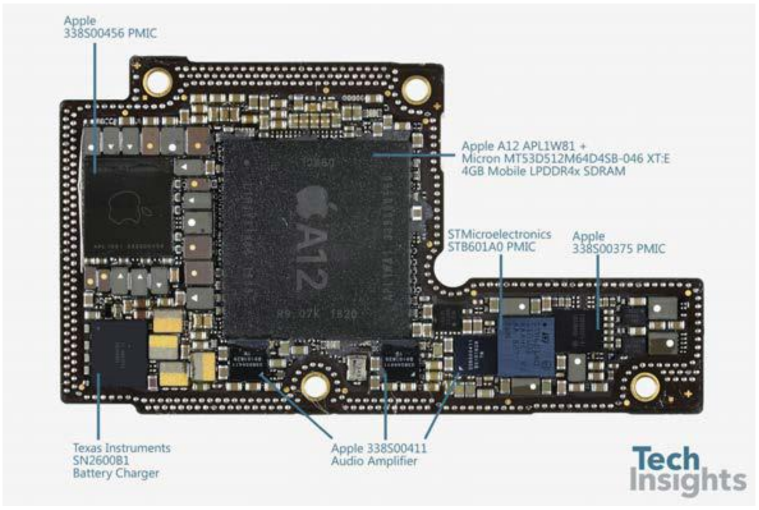


FIGURE 1.8 The logic board of Apple iPhone Xs Max in Figure 1.7. The large integrated circuit in the middle is the Apple A12 chip, which contains two large ARM processor cores and four little ARM processor cores that run at 2.5 GHz, as well as 2 GiB of main memory inside the package. Figure 1.9 shows a photograph of the processor chip inside the A12 package. A similar-sized chip on a symmetric board attached to the back is the 64 GiB flash memory chip for nonvolatile storage. The other chips on the board include power management integrated controller and audio amplifier chips (Courtesy TechInsights, www.techIngishts.com).

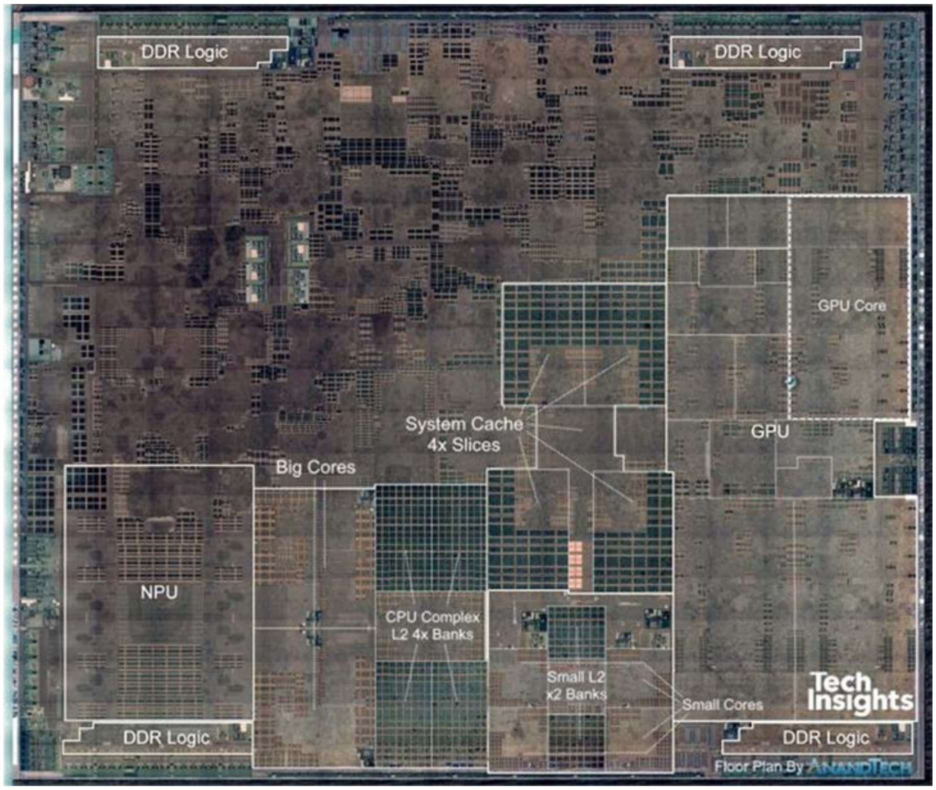


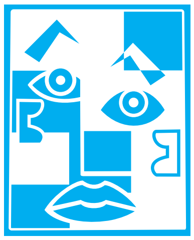
FIGURE 1.9 The processor integrated circuit inside the A12 package. The size of chip is 8.4 by 9.91 mm, and it was manufactured originally in a 7-nm process (see Section 1.5). It has two identical ARM big processors or cores in the lower middle of the chip, four small cores on the lower right of the chip, a graphical processor unit (GPU) on the far right (see Section 6.6), and a domain-specific accelerator for neural networks (see Section 6.7), called the NPU, on the far left. In the middle are second-level cache memories (L2) for the big and small cores (see Chapter 5). At the top and bottom of the chip are interfaces to main memory (DDR DRAM) (Courtesy TechInsights, www.techinsights.com, and AnandTech, www.anandtech.com).

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

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

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

HIERARCHY



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

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

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

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

The BIG Picture

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

implementation Hardware that obeys the architecture abstraction.

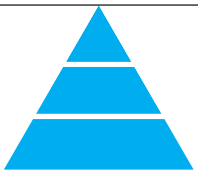
A Safe Place for Data

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

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

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

To distinguish between the volatile memory used to hold data and programs while they are running and this nonvolatile memory used to store data and programs between runs, the term **main memory** or **primary memory** is used for the former, and **secondary memory** for the latter. Secondary memory forms the next lower layer of the **memory hierarchy**. DRAMs have dominated main memory since 1975, but **magnetic disks** dominated secondary memory starting even earlier. Because of their size and form factor, personal Mobile Devices use **flash memory**, a nonvolatile semiconductor memory, instead of disks. Figure 1.8 shows the chip containing the 64 GiB flash memory of the iPhone Xs. 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.



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 2020 was $0.01 to $0.02.

flash memory A nonvolatile semi-conductor 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 2020 was $0.06 to $0.12.

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

Networks have changed the face of computing in the last 40 years, both by becoming much more ubiquitous and by making dramatic increases in performance.

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.

In the 1970s, very few individuals had access to electronic mail, the Internet and web did not exist, and physically mailing magnetic tapes was the primary way to transfer large amounts of data between two locations. Local area networks were almost nonexistent, and the few existing wide area networks had limited capacity and restricted access.

As networking technology improved, it became much cheaper and had a much higher capacity. For example, the first standardized local area network technology, developed about 40 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 100 gigabits per second, usually shared by at most a few computers. Optical communications technology has allowed similar growth in the capacity of wide area networks, from hundreds of kilobits to gigabits and from hundreds of computers connected to a worldwide network to millions of computers connected. This combination of dramatic rise in deployment of networking combined with increases in capacity have made network technology central to the information revolution of the last 30 years.

For the last 15 years another innovation in networking is reshaping the way computers communicate. Wireless technology is widespread, which enabled the PostPC Era. The ability to make a radio in the same low-cost semiconductor technology (CMOS) used for memory and microprocessors enabled a significant improvement in price, leading to an explosion in deployment. Currently available wireless technologies, called by the IEEE standard name 802.11ac, allow for transmission rates from 1 to 1300 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 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.

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

A **transistor** is simply an on/off switch controlled by electricity. The *integrated circuit* (IC) combined dozens to hundreds of transistors into a single chip. When Gordon Moore predicted the continuous doubling of resources, he was predicting

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
2020	Ultra large-scale integrated circuit	500,000,000,000

FIGURE 1.10 Relative performance per unit cost of technologies used in computers over time. Source: Computer Museum, Boston, with 2020 extrapolated by the authors. See Section 1.13.

the growth rate of the number of transistors per chip. To describe the tremendous increase in the number of transistors from hundreds to millions, the adjective *very large scale* is added to the term, creating the abbreviation *VLSI*, for **very large-scale integrated circuit**.

This rate of increasing integration has been remarkably stable. Figure 1.11 shows the growth in DRAM capacity since 1977. For decades, the industry has consistently quadrupled capacity every 3 years, resulting in an increase in excess of 16,000 times! Figure 1.11 also shows the slowdown due to the slowing of Moore's Law; quadrupuling capacity has taken 6 years recently.

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

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

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

silicon A natural element that is a semiconductor.

semiconductor A substance that does not conduct electricity well.

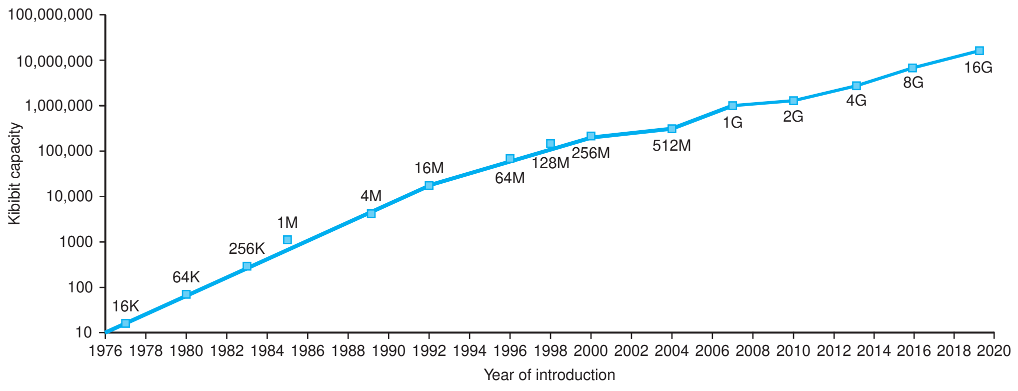


FIGURE 1.11 Growth of capacity per DRAM chip over time. The y-axis is measured in kibibits (2^{10} bits). The DRAM industry quadrupled capacity almost every three years, a 60% increase per year, for 20 years. In recent years, the rate has slowed down and is somewhat closer to doubling every three years. With the slowing of Moore's Law and difficulties in reliable manufacturing of smaller DRAM cells given the challenging aspect ratios of their three-dimensional structure.

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

The manufacturing process for integrated circuits is critical to the cost of the chips and hence important to computer designers. Figure 1.12 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.

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.

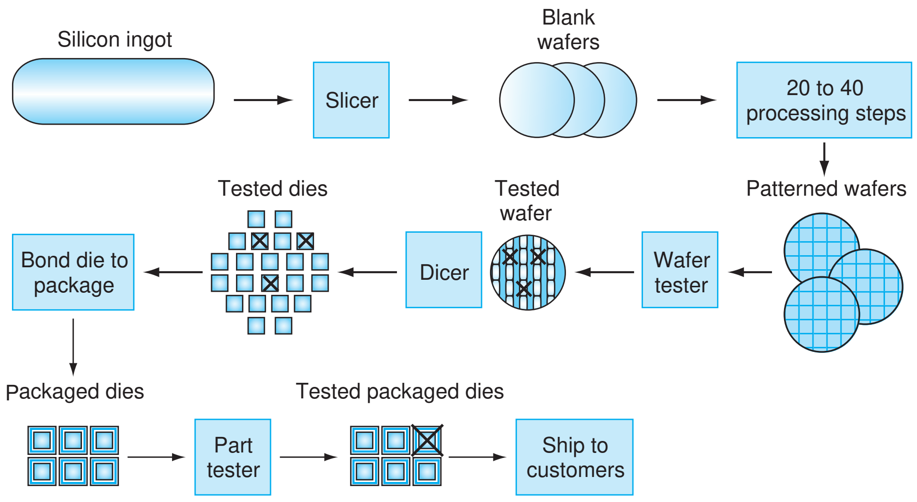


FIGURE 1.12 The chip manufacturing process. After being sliced from the silicon ingot, blank wafers are put through 20 to 40 steps to create patterned wafers (see Figure 1.13). These patterned wafers are then tested with a wafer tester, and a map of the good parts is made. Then, the wafers are diced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

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, 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.

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

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

Dicing enables you to discard only those dies that were unlucky enough to contain the flaws, rather than the whole wafer. This concept is quantified by the

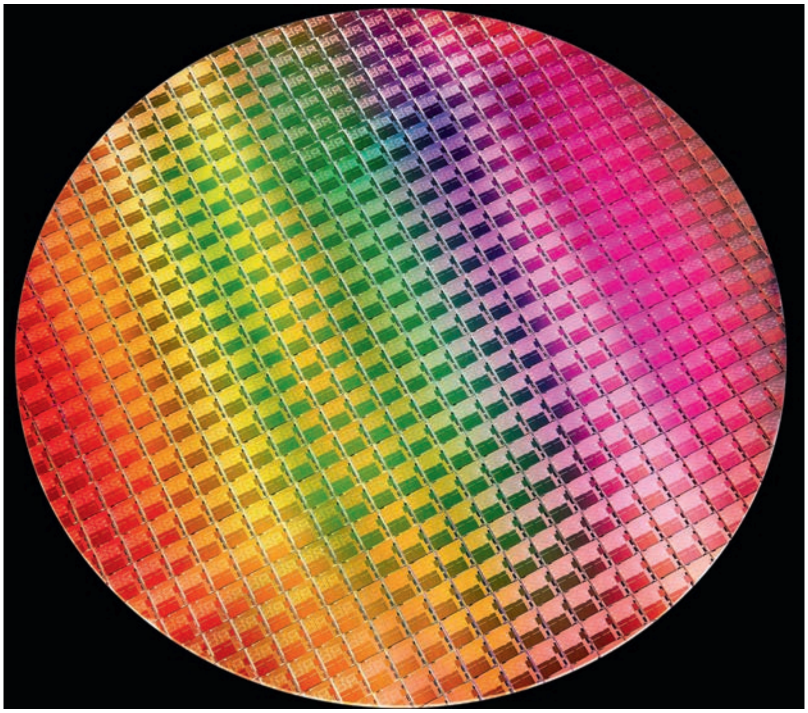


FIGURE 1.13 A 12-inch (300-mm) wafer this 10nm wafer contains 10th Gen Intel® Core™ processors, code-named "Ice Lake" (Courtesy Intel). The number of dies on this 300-mm (12-inch) wafer at 100% yield is 506. According to AnandTech,[1] each Ice Lake die is 11.4 by 10.7 mm. The several dozen partially rounded chips at the boundaries of the wafer are useless; they are included because it is easier to create the masks used to pattern the silicon. This die uses a 10-nm technology, which means that the smallest features are approximately 10 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.

yield of a process, which is defined as the percentage of good dies from the total number of dies on the wafer.

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

The cost of an integrated circuit rises quickly as the die size increases, due both to the lower yield and the smaller number of 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 7-nanometer (nm) process was state-of-the-art in 2020, which means essentially that the smallest feature size on the die is 7 nm.

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.

While we have talked about the cost of chips, there is a difference between cost and price. Companies charge as much as the market will bear to maximize the

[1]Ian Cutress, "I Ran Off with Intel's Tiger Lake Wafer. Who Wants a Die Shot?" January 13, 2020, https://www.anandtech.com/show/15380/i-ran-off-with-intels-tiger-lake-wafer-who-wants-a-die-shot

return on their investment, which must cover costs like a company's research and development (R&D), marketing, sales, manufacturing equipment maintenance, building rental, cost of financing, pretax profits, and taxes. Margins can be higher on unique chips that come from only one company, like microprocessors, versus chips that are commodities supplied by several companies, like DRAMs. The price fluctuates based on the ratio of supply and demand, and it is easy for multiple companies to build more chips than the market demands.

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

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

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

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

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

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

Check Yourself

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

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

1.6 Performance

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

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

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

Defining Performance

When we say one computer has better performance than another, what do we mean? Although this question might seem simple, an analogy with passenger airplanes shows how subtle the question of performance can be. Figure 1.14 lists some typical passenger airplanes, together with their cruising speed, range, and capacity. If we wanted to know which of the planes in this table had the best performance, we would first need to define performance. For example, considering different measures of performance, we see that the plane with the highest cruising speed was the Concorde (retired from service in 2003), the plane with the longest range is the Boeing 777-200LR, and the plane with the largest capacity is the Airbus A380-800.

Airplane	Passenger capacity	Cruising range (miles)	Cruising speed (m.p.h.)	Passenger throughput (passengers × m.p.h.)
Boeing 737	240	3000	564	135,360
BAC/Sud Concorde	132	4000	1350	178,200
Boeing 777-200LR	301	9395	554	166,761
Airbus A380-800	853	8477	587	500,711

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 500 passengers from one point to another, however, the Airbus A380-800 would clearly be the fastest, as the last column of the figure shows. Similarly, we can define computer performance in several different ways.

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.

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

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

Throughput and Response Time

EXAMPLE

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

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

ANSWER

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

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

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

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

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

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

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

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

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

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

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

Relative Performance

EXAMPLE

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

ANSWER

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

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

Thus the performance ratio is

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

and A is therefore 1.5 times as fast as B.

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

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

means that

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

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

Measuring Performance

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

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

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

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

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

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

Understanding Program Performance

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