COMPUTER ORGANIZATION & DESIGN
THE HARDWARE / SOFTWARE INTERFACE

DAVID A. PATTERSON · JOHN L. HENNESSY

7 **Large and Fast: Exploiting Memory Hierarchy**

While *caches*, *translation lookaside buffers*, and *virtual memory* appear different, they can be understood by examining how they deal with the four questions: (1) Where can a block be placed? (2) How is a block found? (3) What block is replaced on a miss? (4) How are writes handled? *(page 512)*

The challenge of memory hierarchies is that every change that improves the *miss* rate can also negatively affect performance: Increasing size decreases *capacity misses* but may also increase *access time*; increasing *associativity* decreases miss rate due to *conflict misses* but may also increase access time; and increasing *block size* may decrease miss rate yet also increase *miss penalty*. *(page 514)*

8 **Interfacing Processors and Peripherals**

Different bus characteristics allow the creation of buses optimized for a wide range of demands. In general, higher cost systems use wider and faster buses that are *synchronous*. In contrast, low-cost systems favor buses that are narrower, do not require intelligence among the devices, and are *asynchronous* so that low-speed devices can interface inexpensively. *(page 563)*

The performance of an I/O system, whether measured by bandwidth or latency, depends on all the elements in the path between the device and memory, including the operating system that generates the I/O commands. *(page 579)*

9 **Parallel Processors**

A key characteristic of programs for parallel machines is the frequency of *synchronization* and *communication*. Large-scale parallel machines have *distributed physical memory*; the higher bandwidth and lower overhead of local memory reward programs utilizing *locality*. *(page 634)*

APPENDICES

A **Assemblers, Linkers, and the SPIM Simulator**

Assembly language is a programming language. Its principal difference from high-level languages such as BASIC, Pascal, and C is that it provides only a few, simple types of data and control flow. Assembly language programs do not specify the type of value held in a variable, leaving the programmer to apply the appropriate operation. *(page A-13)*

B **The Basics of Logic Design**

C **Mapping Control to Hardware**

Independent of whether the control is represented as a finite state diagram or as a microprogram, the translation to hardware is similar: Each state or microinstruction asserts a set of control outputs and specifies how to choose the next state; next state function may be encoded in a finite state machine or with an explicit sequencer; control logic may be either ROMs or PLAs. *(page C-27)*

D **Introducing C to Pascal Programmers**

E **Another Approach to Instruction Set Architecture: VAX**

The differing goals for VAX and MIPS led to different architectures. The VAX goals, simple compilers and code density, led to powerful addressing modes, powerful instructions, and efficient instruction encoding. The MIPS goals were high performance via pipelining, ease of hardware implementation, and compatibility with optimizing compilers. These goals led to simple instructions, simple addressing modes, fixed-length instruction formats, and many registers. *(page E-4)*

Orthogonality is key to the VAX architecture; the opcode is independent of the addressing modes, which are independent of the data types and even the number of unique operands. Thus a few hundred operations expand to hundreds of thousands of instructions when accounting for the data types, operand counts, and addressing modes. *(page E-23)*

This page intentionally left blank

Computer Organization and Design

THE HARDWARE/SOFTWARE INTERFACE

TRADEMARKS

The following trademarks are the property of the following organizations:

TeX is a trademark of American Mathematical Society.

Apple II and Macintosh are trademarks of Apple Computer, Inc.

UNIX and UNIX F77 are trademarks of Novell.

The Cosmic Cube is a trademark of California Institute of Technology.

CP3100 is a trademark of Conner Peripherals.

CDC 6600, CDC 7600, CDC STAR-100, CYBER-180, CYBER-180/990, and CYBER-205 are trademarks of Control Data Corporation.

CRAY-1, CRAY-1S, CRAY-2, CRAY X-MP, CRAY X-MP/416, CRAY Y-MP, CFT-77 V3.0, CFT, CFT2 V1.3a, and T3D are trademarks of Cray Research.

Alpha, CVAX, DEC, DECsystem, DECstation, DECstation 3100, DEC system 10/20, fort, LP11, Massbus, MicroVAX-I, MicroVAX-II, PDP-8, PDP-10, PDP-11, RS-11M/IAS, Unibus, Ultrix, Ultrix 3.0, VAX, VAXstation, VAXstation 2000, VAXstation 3100, VAX-11, VAX 11/780, VAX-11/785, VAX Model 730, Model 750, Model 780, VAX 8600, VAX 8700, VAX 8800, VS FORTRAN V2.4, and VMS are trademarks of Digital Equipment Corporation.

Gnu C Compiler is a trademark of Free Software Foundation.

M2361A, Super Eagle, VP100, and VP200 are trademarks of Fujitsu Corporation.

Apollo DN 300, Apollo DN 10000, HP Precision Architecture, HPPA, HP 850, HP 3000, HP 3000/70, HP 9000, and Precision are trademarks of Hewlett-Packard Company.

432, 960 CA, 4004, 8008, 8080, 8086, 8087, 8088, 80186, 80286, 80386, 80486, Delta, iAPX 432, i860, iPSC, iPSC/2, Intel, Intel Hypercube, Multibus, Multibus II, and Paragon are trademarks of Intel Corporation.

Inmos and Transputer are trademarks of Inmos.

IBM, 360, 360/30, 360/40, 360/50, 360/65, 360/85, 360/91, 370, 370/135, 370/138, 370/145, 370/155, 370/158, 370/165, 370/168, 370-XA, ESA/370, System/360, System/370, 701, 704, 709, 801, 3033, 3080, 3080 series, 3080 VF, 3081, 3090, 3090/100, 3090/200, 3090/400, 3090/600, 3090/600S, 3090 VF, 3330, 3380, 3380D, 3380 Disk Model AK4, 3380J, 3390, 3880-23, 3990, 7030, 7090, 7094, IBM FORTRAN, ISAM, MVS, IBM PC, IBM PC-AT, PL.8, PowerPC, RT-PC, SAGE, Stretch, IBM SVS, Vector Facility, and VM are trademarks of International Business Machines Corporation.

FutureBus is a trademark of the Institute of Electrical and Electronic Engineers.

Goodyear MPP is a trademark of Goodyear Tire and Rubber Co, Inc.

ICL DAP is a trademark of International Computers Limited.

KSR-1 is a trademark of Kendall Square Research.

MASPAR MP-1 is a trademark of MasPar Corporation.

NuBus is a trademark of Massachusetts Institute of Technology.

MIPS, MIPS 120, MIPS/120A, M/500, M/1000, RC6230, RC6280, R2000, R2000A, R2010, R3000, R3010, and R4000 are trademarks of MIPS Technology, Inc.

Delta Series 8608, System V/88 R32V1, VME bus, 6809, 68000, 68010, 68020, 68030, 68882, 88000, 88000 1.8.4m14, 88100, and 88200 are trademarks of Motorola Corporation.

Ncube and nCube/ten are trademarks of Ncube Corporation.

Parsytec GC is a trademark of Parsytec, Inc.

Wren IV, Imprimis, Sabre, Sabre 97209, and IPI-2 are trademarks of Seagate Corporation.

Sequent, Balance 800, Balance 21000, and Symmetry are trademarks of Sequent Computers.

Silicon Graphics 4D/60, 4D/240, and Silicon Graphics 4D Series are trademarks of Silicon Graphics.

Connection Machine, CM-2, and CM-5 are trademarks of Thinking Machines.

Burroughs 6500, B5000, B5500, D-machine, UNIVAC, UNIVAC I, UNIVAC 1103 are trademarks of UNISYS.

Spice and 4.2 BSD UNIX are trademarks of University of California at Berkeley.

Alto, Ethernet, PARC, Palo Alto Research Center, Smalltalk, and Xerox are trademarks of Xerox Corporation.

Computer Organization and Design

THE HARDWARE/SOFTWARE INTERFACE

John L. Hennessy
Stanford University

David A. Patterson
University of California at Berkeley

With a contribution by
James R. Larus
University of Wisconsin

Morgan Kaufmann Publishers, Inc.
San Francisco, California

Senior Editor: Bruce M. Spatz
Production Manager: Yonie Overton
Editorial Coordinator: Douglas Sery
Copyediting: Steve Hiatt and Gary Morris
Text Design: Ross Carron Design
Illustration: Alexander Teshin Associates
Composition/Color Separation/Postscript Programming: Edward W. Sznyter, Babel Press
Cover Design: David Lance Goines
Additional Cover Mechanical Art: Patty King
Chapter Opener Illustrations: Jo Jackson
Indexing: Steve Rath
Proofreading: Gary Morris
Electronic Prepress: The Courier Connection
Printer: Courier Corporation

Morgan Kaufmann Publishers, Inc.
Editorial Office:
340 Pine Street, Sixth Floor
San Francisco, CA 94104

© 1994 by Morgan Kaufmann Publishers, Inc.
All rights reserved
Printed in the United States of America

97 5 4

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

Advice, Praise, and Errors: Any correspondence related to this publication or intended for the authors should be addressed to the editorial offices of Morgan Kaufmann Publishers, Inc., Dept. P&H APE. Information regarding error sightings is encouraged. Any error sightings that are accepted for correction in subsequent printings will be rewarded by the authors with a payment of $1.00 (U.S.) per correction at the time of their implementation in a reprint. Electronic mail can be sent to errors2@mkp.com.

Instructor Support: For information on the SPIM software simulator and other instructor materials available to adoptors, please contact the editorial offices of Morgan Kaufmann Publishers, Inc.

Cataloging-in-Publication Data
Patterson, David A.
Computer organization and design: the hardware/software interface / David A. Patterson, John L. Hennessy.
p. cm.
Includes bibliographical references and index.
ISBN 1-55860-281-X
1. Computer organization. 2. Computers--Design and construction. 3. Computer interfaces. I. Hennessy, John L. II. Title
QA76.9.C643P37 1994
004.2'2--dc20 94-17639
CIP

TO LINDA AND ANDREA

Foreword

by Maurice V. Wilkes
Cambridge, England

This is an excellent time for a new book on computer design. During the last ten years the subject has undergone a marked renaissance. It has become less dependent on intuition and personal opinion, and more on measurement and rational analysis. The subtitle of the author's earlier book *Computer Architecture: A Quantitative Approach* makes this point.

Patterson and Hennessy played their part in these developments. Indeed, widespread public discussion of the new ideas may be said to have begun with the publication in 1980 of a paper by Patterson and Ditzel entitled "The Case for the Reduced Instruction Set Computer." The acronym RISC derives from this paper. Patterson proceeded to put RISC ideas into practice by developing the Berkeley RISC with the aid of a group of students. This design became the basis of the SPARC workstations. Hennessy applied his energies to the MIPS design project at Stanford and became one of the founders of the MIPS Computer company.

Like many seminal ideas, RISC is deceptively simple. The emphasis is not on the size of the instruction set, but on its nature; RISC instruction sets have made it possible to apply, in a single chip processor, subtle techniques for instruction level concurrency that were previously to be found only in large computers.

There is a lot more to computer design than is comprised in the RISC philosophy, as will be shown by a glance at the diagram entitled "The Five Classic Components of a Computer" which appears, with differing highlighting, at the head of various chapters of this book. But RISC has been a unifying influence. Another major unifying influence has been the need to work within the boundaries of a silicon chip, where there is never enough space and, if one thing goes in, another must go out. Performance depends critically on decisions taken at the chip level. No longer can system design be a subject divorced from computer implementation, a change which may have left some people high and dry, but which is nevertheless wholly to the good. Nor can system design be divorced from consideration of the software interface, a point which the authors bring out both in their subtitle and in their text.

The computer field, particularly on the software side, abounds with examples of new ideas that have found their way into industrial practice by being taught in universities and have thereby become part of the professional tool kit

which graduating students have carried with them into industry. In hardware, it is often the other way round; teaching follows practice. This is another reason for welcoming a book by two engineers who can write of current practice with authority.

As I followed the authors through their unhurried chapters, I was conscious of the all-seeing eye of the evaluator pictured at the chapter heads. Ever watchful, she—I think it is a female eye, but I cannot be sure—seemed to be saying that every argument has another side, and that every insight gains by being put into perspective.

I think students will enjoy learning from this book; at least, I hope so and I give them my good wishes.

Contents

Foreword vi
by Maurice V. Wilkes

Preface xiii

The SPIM Simulator for the MIPS R2000/R3000 xxiii
by James R. Larus, University of Wisconsin

C H A P T E R S

1 **Computer Abstractions and Technology** 2

1.1 Introduction 3
1.2 Below Your Program 5
1.3 Under the Covers 10
1.4 Integrated Circuits: Fueling Innovation 21
1.5 Fallacies and Pitfalls 26
1.6 Concluding Remarks 28
1.7 Historical Perspective and Further Reading 30
1.8 Exercises 41

2 **The Role of Performance** 46

2.1 Introduction 48
2.2 Measuring Performance 52
2.3 Relating the Metrics 54
2.4 Popular Performance Metrics 60
2.5 Choosing Programs to Evaluate Performance 66
2.6 Comparing and Summarizing Performance 68
2.7 Fallacies and Pitfalls 70
2.8 Concluding Remarks 76
2.9 Historical Perspective and Further Reading 77
2.10 Exercises 81

3 Instructions: Language of the Machine 92

3.1 Introduction 94
3.2 Operations of the Computer Hardware 95
3.3 Operands of the Computer Hardware 97
3.4 Representing Instructions in the Computer 103
3.5 Instructions for Making Decisions 110
3.6 Supporting Procedures in Computer Hardware 119
3.7 Other Styles of MIPS Addressing 124
3.8 Alternatives to the MIPS Approach 130
3.9 An Example to Put It All Together 135
3.10 A Longer Example 138
3.11 Arrays versus Pointers 143
3.12 Fallacies and Pitfalls 147
3.13 Concluding Remarks 148
3.14 Historical Perspective and Further Reading 150
3.15 Exercises 155

4 Arithmetic for Computers 166

4.1 Introduction 168
4.2 Negative Numbers 168
4.3 Addition and Subtraction 175
4.4 Logical Operations 179
4.5 Constructing an Arithmetic Logic Unit 182
4.6 Multiplication 198
4.7 Division 212
4.8 Floating Point 225
4.9 Fallacies and Pitfalls 244
4.10 Concluding Remarks 246
4.11 Historical Perspective and Further Reading 249
4.12 Exercises 258

5 The Processor: Datapath and Control 268

5.1 Introduction 270
5.2 Building a Datapath 276
5.3 A Simple Implementation Scheme 283
5.4 A Multiple Clock Cycle Implementation 312
5.5 Microprogramming: Simplifying Control Design 333
5.6 Exceptions 344
5.7 Fallacies and Pitfalls 350
5.8 Concluding Remarks 351
5.9 Historical Perspective and Further Reading 353
5.10 Exercises 357

6 Enhancing Performance with Pipelining 362

6.1 Introduction 364
6.2 A Pipelined Datapath 367
6.3 Pipelined Control 381
6.4 Data Hazards 390
6.5 Control for Data Hazards: Stalls 399
6.6 Reducing Data Hazards: Forwarding 412
6.7 Branch Hazards 424
6.8 Exceptions 430
6.9 Performance of Pipelined Systems 435
6.10 Fallacies and Pitfalls 436
6.11 Concluding Remarks 438
6.12 Historical Perspective and Further Reading 441
6.13 Exercises 445

7 Large and Fast: Exploiting Memory Hierarchy 452

7.1 Introduction 454
7.2 Caches 458
7.3 Virtual Memory 481
7.4 A Common Framework for Memory Hierarchies 501
7.5 Fallacies and Pitfalls 515
7.6 Concluding Remarks 519
7.7 Historical Perspective and Further Reading 521
7.8 Exercises 527

8 Interfacing Processors and Peripherals 532

8.1 Introduction 534
8.2 I/O Performance Measures: Some Examples from Disk and File Systems 537
8.3 Types and Characteristics of I/O Devices 539
8.4 Buses: Connecting I/O Devices to Processor and Memory 548
8.5 Interfacing I/O Devices to the Memory, Processor, and Operating System 565
8.6 Fallacies and Pitfalls 576
8.7 Concluding Remarks 578
8.8 Historical Perspective and Further Reading 581
8.9 Exercises 584

9 Parallel Processors 594

9.1 Introduction 596
9.2 SIMD Computers—Single Instruction Stream, Multiple Data Streams 596
9.3 MIMD Computers—Multiple Instruction Streams, Multiple Data Streams 602
9.4 Programming MIMDs 603
9.5 MIMDs Connected by a Single Bus 607
9.6 MIMDs Connected by a Network 619
9.7 Future Directions for Parallel Processors 630
9.8 Fallacies and Pitfalls 637
9.9 Concluding Remarks—Evolution versus Revolution in Computer Architecture 640
9.10 Historical Perspective and Further Reading 642
9.11 Exercises 646

APPENDICES

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

by James R. Larus, University of Wisconsin

A.1 Introduction A-3
A.2 Assemblers A-10
A.3 Linkers A-17
A.4 Loading A-19
A.5 Memory Usage A-19
A.6 Procedure Call Convention A-21
A.7 Exceptions and Interrupts A-30
A.8 Input and Output A-34
A.9 SPIM A-36
A.10 MIPS R2000 Assembly Language A-47
A.11 Concluding Remarks A-71
A.12 Exercises A-72

B The Basics of Logic Design B-2

B.1 Introduction B-3
B.2 Gates, Truth Tables, and Logic Equations B-4
B.3 Combinational Logic B-8
B.4 Clocks B-18
B.5 Memory Elements B-21
B.6 Finite State Machines B-35
B.7 Timing Methodologies B-39
B.8 Exercises B-45

C **Mapping Control to Hardware** C-2

C.1 Introduction C-3
C.2 Implementing Finite State Machine Control C-4
C.3 Implementing the Next-State Function with a Sequencer C-15
C.4 Translating a Microprogram to Hardware C-23
C.5 Concluding Remarks C-27
C.6 Exercises C-28

D **Introducing C to Pascal Programmers** D-2

D.1 Introduction D-3
D.2 Variable Declarations D-3
D.3 Assignment Statements D-4
D.4 Relational Expressions and Conditional Statements D-5
D.5 Loops D-6
D.6 Examples to Put it All Together D-7
D.7 Exercises D-8

E **Another Approach to Instruction Set Architecture—VAX** E-2

E.1 Introduction E-3
E.2 VAX Operands and Addressing Modes E-4
E.3 Encoding VAX Instructions E-7
E.4 VAX Operations E-9
E.5 An Example to Put It All Together: swap E-11
E.6 A Longer Example: sort E-15
E.7 Fallacies and Pitfalls E-19
E.8 Concluding Remarks E-22
E.9 Historical Perspective and Further Reading E-23
E.10 Exercises E-25

Index I-1

Preface

The most beautiful thing we can experience is the mysterious.
It is the source of all true art and science.

Albert Einstein
What I Believe, 1930

About This Book

We believe that learning in computer science and engineering should reflect the current state of the field, as well as introduce the principles that are shaping computing. We also feel that readers in every specialty of computing need to appreciate the organizational paradigms that determine the capabilities, performance, and, ultimately, the success of computer systems.

Modern computer technology requires professionals of every computing specialty to understand both hardware and software. The interaction between hardware and software at a variety of levels also offers a framework for understanding the fundamentals of computing. Whether your primary interest is 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.

Traditionally, the competing influences of assembly language, organization, and design have encouraged books that consider each area as a distinct subset. In our view, such distinctions have increasingly lost meaning as computer technology has advanced. To truly understand the breadth of our field, it is important to understand the interdependencies among these topics.

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.

Relationship to CA:AQA

Many readers will be familiar with *Computer Architecture: A Quantitative Approach* (Morgan Kaufmann, 1990). Our motivation in writing that 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 scientific methodologies instead of a descriptive approach.

We've discovered that many people have used that book as a first introduction to the field. We've also learned that many institutions are now using this quantitative approach in more introductory computer organization courses. However, *Computer Architecture* was written at a more advanced level, for readers who already understood the basic principles.

A majority of the readers for *Computer Organization and Design: The Hardware/Software Interface* will not be or plan to become computer architects. However, the performance of future software systems will be dramatically affected 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.

Given these factors, we knew that this book had to be much more than a subset of the material in *Computer Architecture*. We've approached every topic in a new way. Topics shared between the books were written anew for this effort, while many other topics are presented here for the first time. To further ensure the uniqueness of *Computer Organization and Design*, we exchanged the writing responsibilities we assigned to ourselves for *Computer Architecture*. The topics that Hennessy covered in the first book were written by Patterson in this one, and vice versa. Several of our reviewers suggested that we call this book "Computer Organization: A Conceptual Approach" to emphasize the significant differences from our other book. It is our hope that the reader will find new insights in every section, as well as a more tractable introduction to the abstractions and principles at work in a modern computer.

Learning by Evolution

It is tempting for authors to present the latest version of a hardware concept and spend considerable time explaining how these often sophisticated ideas work. We decided instead to present each idea from its first principles, emphasizing the simplest version of an idea, how it works, and how it came to be. We believe that presenting the fundamental concepts first offers greater insight into why machines look the way they do today, as well as how they might evolve as technology changes.

To facilitate this approach, we have based the book upon the MIPS processor. It offers an easy to understand instruction set and can be implemented in a simple, straightforward manner. This allows readers to grasp an entire machine organization and to follow exactly how the machine implements its instructions. Throughout the text, we present the concepts before the details, building from simpler versions of ideas to more complex ones. Examples of this approach can be found in almost every chapter. Chapter 3 builds up to MIPS assembly language starting with one simple instruction type. The con-

cepts and algorithms used in modern computer arithmetic are built up starting from the familiar grammar school algorithms in Chapter 4. Chapters 5 and 6 start from the simplest possible implementation of a MIPS subset and build to a fully pipelined version. Chapter 7 illustrates the abstractions and concepts in memory hierarchies by starting with the simplest possible cache and introducing virtual memory and TLBs as an extension of the concepts.

This evolutionary process is used extensively in Chapters 5 and 6, where the complete datapath and control for a processor are presented. Since learning is a visual process, we have included sequences of figures that contain progressively more detail or show a sequence of events within the machine. We have also used a second color to help readers follow the figures and sequences of figures.

Learning from this Book

Our objective of demonstrating first principles through the interrelationship of hardware and software is enhanced by several features found in each chapter. The Hardware/Software Interface sections are used to highlight these relationships. We've also included Big Picture sections for each chapter to remind readers of the major insights. We hope that these elements reinforce our goal of making this book equally valuable as a foundation for further study in both hardware and software courses.

To illustrate the relationship between high-level language and machine language and to describe the hardware algorithms, we have chosen C. It is widely used in compiler and operating system courses, it is widely used by computer professionals, and several facilities in the language make it suitable for describing hardware algorithms. For those who are familiar with Pascal rather than C, Appendix D provides a quick introduction to C for Pascal programmers and should be sufficient to understand the code sequences in the text.

We have tried to manage the pace of the presentation for readers of varying experience. Ideas that are not essential to a newcomer, but which may be of interest to the more advanced reader, are presented as Elaborations. When appropriate, advanced concepts have been saved for the exercise sets and enhanced with additional discussion as In More Depth sections. In addition, we found that the extent of background that students have in logic design varies widely. Thus, Appendix B provides all the necessary background for those readers not versed in the basics of logic design, as well as some slightly more sophisticated material for the more advanced student. Within a course, this material can be used in supplementary lectures or incorporated into the mainstream of the course, depending on the background of the students and the goals of the instructor.

We have also found that readers enjoy learning the history of the field, so the Historical Perspective sections include many photographs of important machines and little known stories about the ideas behind them. We hope that the perspective offered by these anecdotes and photographs will add a new dimension for our readers.

Course Syllabi and this Book

One particularly difficult issue facing instructors is the balance of assembly language programming with computer organization. We have written this book so that readers will learn more about organization and design, while still providing a complete introduction to assembly language. By using a RISC architecture, students can learn the basics of an instruction set and assembly language programming in less time than is typically reserved in the curriculum for CISC based assembly courses. Many instructors have also found that using a simulator, rather than running in native mode on a real machine, provides the experience of assembly language programming in substantially less time (and with less pain for the student).

Instructors may contact the publisher regarding the SPIM simulator of the MIPS processor. The XSPIM simulator developed by James R. Larus is retrievable via ftp (see page xxiii). Adaptations are also available in Windows and Macintosh formats. Although not identical, they offer the same general functionality. We feel this will enhance student opportunities for learning about computer organization (see Appendix A). Finally, stepwise derivation of assembly from a high-level language takes less study time than learning it from the ground up. Chapter 3 and Appendix A may be used together or separately, depending upon the reader's background. Chapter 3 provides the basics and can be supplemented with additional detail from Appendix A for a complete introduction to modern assembly language programming, including assemblers, linkers, and loaders. In the end, we hope this approach offers a more efficient treatment of assembly for most readers, while being sufficiently broad to support detailed lecture or laboratory coverage if an instructor wants more emphasis on assembly language programming.

For those courses intended to expose students to the important principles of computer organization, the chapters from 4 to 9 explain the key ideas. Chapter 4 explains the idea of number representation for both integers and floating-point numbers and shows how arithmetic algorithms work. Chapters 5 and 6 introduce key ideas in control and pipelining and can be covered at several levels. Chapter 7 introduces the principles of memory hierarchies, unifying the ideas of caching and virtual memory. Chapter 8 shows how I/O systems are organized and controlled, explaining the cooperative relationship between the hardware and the operating system. Finally, Chapter 9 uses examples to introduce the key principles used in multiprocessors.

For readers who want a greater emphasis on computer design, Chapters 4 through 6, together with Appendices B and C, provide that opportunity. For example, Chapter 4 explains a number of techniques used by computer designers to speed up addition and multiplication. Chapters 5 and 6 derive complete implementations of a MIPS subset using the arithmetic elements from Chapter 4 and a number of common datapath elements (such as register files and memories) that are explained in detail in Appendix B. Chapter 5 starts with a very simple implementation; a complete datapath and control unit are constructed for this organization. The implementation is then modified to derive a faster version where each instruction can take differing numbers of clock

cycles. The control for this multicycle implementation is designed using two different methods in Chapter 5. Appendix C shows in detail how the control specifications are implemented using structured logic blocks. Chapter 6 builds on the single-clock cycle implementation created in Chapter 5 to show how pipelined machines are designed. The design is extended to show how hazards can be handled and how control for interrupts works. The student interested in computer design, is not only exposed to three different designs for the same instruction set, but can also see how these designs compare in terms of advantages and disadvantages.

Chapter Organization and Overview

Using these plans as the core, we developed the other chapters to introduce and support that core.

Many students remarked that they appreciated learning about the continuing rapid change in speed and capacity of hardware, as well as some of the history of computer development. This material is the focus of Chapter 1. It provides a perspective on how software or hardware will need to scale during the coming decades. Chapter 1 also introduces topics to be covered in later chapters.

Chapter 2 shows that time is the only safe measure of computer performance. It also relates common measurements used by hardware and software designers to the reliable measurement of time. The material in this chapter motivates the techniques discussed in Chapters 5, 6, and 7 and provides a framework for evaluating them.

Chapter 3 builds on the knowledge of a programming language to derive an assembly language, offering several rules of thumb that guide the designer of the assembly language. We chose the instruction set of a real computer, in this case MIPS, so that real compilers could be used by students to see the code that would be generated. We hide the delayed branch and load until Chapter 6 for pedagogical reasons. Fortunately, the MIPS assembler schedules both delayed branches and loads so the assembly language programmer can ignore these complexities without danger. Readers interested in seeing a very different approach to instruction set design should read Appendix E, which gives a short introduction to the VAX architecture using the same major programming example as in Chapter 3.

Although there is no consensus on what should be covered or what should be skipped in learning about computer arithmetic, we couldn't write Chapter 4 without reaching some conclusions of our own! We understand that the topics and depth of coverage vary greatly from one course to another, sometimes within the same department, depending upon the taste and background of the individual instructor. For example, some instructors feel it's essential that everyone learn multibit Booth algorithms, while others will skip signed multiplication. Our solution is to introduce all the central ideas in the chapter and to provide additional background for more advanced topics in the exercises. This allows one instructor to cover more advanced topics and assign exercises based on them, while another instructor may skip the material.

Chapters 5 and 6 show a realistic example of a processor in detail. Most readers appreciate having a real example to study, and a complete example provides the insight needed to see how all the pieces of a processor fit together for a pipelined and nonpipelined machine. To facilitate skipping some details on hardware implementation of control, we have included much of this material in Appendix C.

Just as Chapters 2 through 6 provide important background for readers with an interest in compilers, Chapters 7 and 8 provide vital background to anyone pursuing further work in operating systems or databases. Chapter 7 describes the principles of memory hierarchies, focusing on the commonality between virtual memory and caching. Chapter 7 emphasizes the role of the operating system and its interaction with the memory system.

Topics as diverse as operating systems, databases, graphics, and networking require an understanding of I/O systems organization as well as the major technical characteristics of devices that influence this organization. Chapter 8 focuses on the topic of how I/O systems are organized starting with bus organizations, working up to communication between the processor and I/O device, and finally to the management role of the operating system. While we emphasize the interfacing issues, especially between hardware and software, several other important topics are introduced. Many of these topics are useful not only in computer organization but as background in other areas. For example, the handshaking protocol, used to interface asynchronous I/O devices, has applications in any distributed system.

For some readers, this book may be their only overview of computer systems, so we have included a survey of parallel processing. Rather than the traditional catalog of characteristics for many parallel machines, we have tried to describe the underlying principles that will drive the designs of parallel processors for the next decade. This section includes a small running example to show different versions of the same program for different parallel architectures.

Because the book is intended as an introduction for readers with a variety of interests, we tried to keep the presentation flexible. The appendices on assembly language and logic design are one of the principle vehicles to allow such flexibility, as these are easily skipped by more advanced readers. The presence of the appendices has made it possible to use this book in a course that mixes EE and CS majors with fairly different backgrounds in logic design and software.

Assembly language programming is best learned by doing and in many cases will be done with the use of the simulator available with this book. Because of this, we invited Jim Larus, the creator of the SPIM simulator, to join us as a contributor of Appendix A. Appendix A describes the SPIM simulator and provides further details of the MIPS assembly language. In addition, it describes assemblers and linkers, which handle the translation of assembly language programs to executable machine language.

The logic design appendix is intended as a supplement to the material on computer organization rather than a comprehensive introduction to logic design. While many EE students in a computer organization course will have al-

ready had a course on logic design or digital electronics, we have found that CS majors in many institutions have not had much exposure to this area. The first few sections of Appendix B provide the necessary background. We include some material, such as the organization of memories and finite state machine control of a processor, in the mainstream material, since it is crucial to understanding computer organization.

Selection of Material

If you had no prior background and wanted to read from cover-to-cover, the following order makes sense: Chapters 1 and 2, Appendix D (if needed), Chapter 3, Appendices A and E, Chapter 4, Appendix B, Chapter 5, Appendix C, Chapters 6, 7, 8, and 9. Clearly, most readers skip material. We have worked to provide readers with flexibility in their approach to the material, without making the discussions redundant. The chapters have been written as self-contained units with cross-references to other chapters when related text or figures should be considered. The book has been used successfully in a variety of Beta courses with different goals and student backgrounds. Specific choice of materials as well as the sequence of presentation varied significantly among the Beta sites. Table 1 samples some of these differences.

Students	EE/CS soph/jr	CS jr/sr	CS soph/jr	EE sr/gr	EE/CS jr/sr	EE/CS jr/sr
Prerequisites	HLL	Assembly	Assembly HLL	Assembly Some logic	Assembly Digital fund.	Assembly Digital design
Term (in weeks)	10	14	15	10	16	10
1 Introduction	1	—	2	1	Reference	1
2 Performance	2	—	3	2	Reference	2
3 Instructions	3 (p)	4 (p)	4	3	1	3 (p)
4 Arithmetic	4	2	6	4	2/6	4
5 Processor	5	3	7	5 (p)	3/5	5
6 Pipelining	6	5	8 (p)	6	7	6
7 Memory	7	6	9	7	8	7
8 I/O	8	7	10	8	9	8 (p)
9 Parallel		8	11	9 (p)		
A Assembly	3	Reference				
B Logic	Reference	1	5			
C Control	Reference				4	
Other topics	VAX (App E)		1 C language		RISC machines	

Table 1 (p) = partial coverage or cursory. Numbers refer to the sequence of chapter coverage. Numbers separated by / indicates chapter was covered in parts out of sequence.

Concluding Remarks

In our last book we alternated the gender of a pronoun chapter by chapter. In this book we believe we have removed all such pronouns, except of course for specific people.

If you read the following acknowledgement section, you will see that we went to great lengths to correct mistakes. Since a book goes through many printings, we have the opportunity to make even more corrections. If you uncover any remaining, resilient bugs, please contact the publisher by electronic mail at errors@cs.berkeley.edu or by low-tech mail using the address found on the copyright page. The first person to report a technical error will be awarded a $1.00 bounty upon its implementation in future printings of the book!

Finally, like the last book there is no strict ordering of the authors' names. About half the time you will see Hennessy and Patterson, both in this book and in advertisements, and half the time you will see Patterson and Hennessy. You'll even find it listed both ways in bibliographic publications such as *Books In Print*. This again reflects the true collaborative nature of this book: Together we brainstormed about the ideas and method of presentation, then individually wrote about one-half of the chapters and acted as reviewer for every draft of the other. The page count suggests we again wrote almost exactly the same number of pages. Thus, we equally share the blame for what you are about to read.

Acknowledgements

We wish first to acknowledge the encouragement and suggestions offered by the readers of *Computer Architecture: A Quantitative Approach* and the reviewers of the proposal originally produced for this book. We would not have written this book without their support and directions.

Before we started this book, we received valuable comments on an outline of our ideas from

Alan Berenbaum, AT&T; **Douglas W. Clark**, Digital Equipment Corporation/Princeton University; **David Culler**, University of California at Berkeley; **Stephen J. Hartley**, University of Texas at San Antonio; **Monica Lam**, Stanford; **Daniel McCrackin**, McMaster University; **William R. Michalson**, Worcester Polytechnic Institute; **Yuval Tamir**, University of California at Los Angeles; **Philip A. Wilsey**, University of Cincinnati

The early comments from these reviewers convinced us that there was a need for a book with the goals we have used for this effort.

We'd like to express our appreciation to **Jim Larus** for his willingness in contributing his expertise on assembly language programming, as well as for welcoming readers of this book to use the simulator he developed and maintains at the University of Wisconsin.

Thanks go to about 50 students at Berkeley taking CS 152 during Spring semester 1992 and about 80 students at Stanford taking CS 182 during Winter Quarter 1992 for debugging the alpha version of the text. Professors **John**

Wawrzynek and **John Hennessy** taught the two courses. **Jeff Kuskin**, who served as the teaching assistant at Stanford, provided valuable advice and generated the original versions of a number of exercises that appear in this book.

In addition to the student comments, we appreciate the feedback from these reviewers of the alpha version:

Alan Berenbaum, AT&T; **Douglas W. Clark**, Digital Equipment Corporation/Princeton University; **Rajan Chandra**, California State Polytechnic University at Pomona; **Edward W. Czeck**, Northeastern University; **Chris Edmondson-Yurkanan**, University of Texas at Austin; **Robert Fowler**, University of Rochester; **Gideon Frieder**, George Washington University (Chapter 1); **Mark Hill**, University of Wisconsin at Madison; **Kai Li**, Princeton University; **Bart Locanthi**, AT&T (Chapter 8); **David Meyer**, Purdue University; **William R. Michalson**, Worcester Polytechnic Institute; **Mark Smotherman**, Clemson University ; **Evan Tick**, University of Oregon (careful review of figures); **Shlomo Weiss**, Tel Aviv University (Chapter 8); **Alan Zaring**, Ohio Wesleyan University (Chapter 9 and Appendix B)

Mark Smotherman's comments on the role of assembly language were especially helpful in deciding how to deal with this topic. Many of the reviewers provided helpful suggestions for exercises. **William Kahn** of UC Berkeley provided the material for the history section for the computer arithmetic chapter.

The Beta reviewers included the following:

David Douglas, Thinking Machines (Chapter 9); **Alan Fekete**, University of Sydney; **Corinna Lee**, University of Toronto; **William R. Michalson** Worcester Polytechnic Institute; **Ned Okie**, Radford University; **Klaus Erik Schauser**, University of California at Berkeley; **Guri Sohi**, University of Wisconsin at Madison (Chapter 9); **Arun Somani**, University of Washington; **Philp Tromovitch**, SUNY at Stony Brook; **David Ward**, Brigham Young University; **James Van Orman**, Brigham Young University (provided extensive figure review for both the Beta and the final edition)

Special thanks go to **Doug Clark** for his inputs on both the Alpha and Beta versions. As with *Computer Architecture*, Doug provided a wealth of comments to us. His insights, as well as his persistence in urging us to simplify and improve the pedagogy, are deeply appreciated.

The Beta Edition was released for class testing in the fall of 1992 by the following instructors and institutions:

Rajendra Boppana, University of Texas at San Antonio; **Barry S. Fagin,** Dartmouth; **Michael Faiman**, University of Illinois, Urbana–Champaign; **Mark A. Friedman,** Trinity College; **Anoop Gupta,** Stanford University; **Brian Harvey,** University of California at Berkeley; **Roy Jenevein**, University of Texas at Austin; **Corinna Lee,** University of Toronto; **Ned Okie,** Radford University; **Parameswaran Ramanathan,** University of Wisconsin at Madison; **Arun K. Somani,** University of Washington; **David M. Ward,** Brigham Young University; **John Wawrzynek,** University of California at Berkeley

We appreciate their adventurous spirit and thank them for their comments which helped improve the final text. We are especially appreciative to **Brian Harvey**, **Corinna Lee**, and **Ned Okie** for their extensive comments and to the students at Berkeley, Brigham Young, Radford, and Toronto for being especially diligent in completing their surveys. The surveys had an enormous impact on the first edition of this book. The teaching assistants at these institutions played a valuable role by collecting and forwarding surveys, as well as by providing feedback on sections or exercises that proved difficult for their classes. **Scott Bevan** at Brigham Young was especially helpful in getting comments and surveys back.

Many students reported bugs in the Beta Edition. Their comments were especially helpful. We would like to thank a group of students who were very diligent in finding and rapidly reporting bugs. These students were the first to report the greatest number of bugs:

Ernest Bailey, Brigham Young University; **Wallace Chan**, University of Toronto; **Isaac Cheng**, University of California at Berkeley; **Jeff Clark**, Radford University; **Moored Fahim**, Radford University; **Xilin Jai**, Dartmouth College; **Guy Lemieux**, University of Toronto; **Cameron McNairy**, Brigham Young University; **Jose L. Urrusti**, University of Wisconsin; **James Van Orman**, Brigham Young University (who was the first reporter of 34 different bugs!); **John Yen**, University of California at Berkeley

We started writing this book in the Fall of 1991 and raced to stay ahead of the classes at Berkeley and Stanford that began in January. We completed the final chapter of the alpha edition in April 1992. After taking time to catch up with our postponed obligations, and to attend the Computer Architecture conference in Australia, we started the Beta Edition in late May, completing our revisions in July for the September printing. Inspired by comments on the Beta Edition from the classroom, we started writing again in October and finished in January 1993.

We wish to thank the extended Morgan Kaufmann family for agreeing to publish this book twice, under the able leadership of **Bruce Spatz** and the watchful eye of **Yonie Overton**. Composition, color separation, and postscript programming were provided by **Ed Sznyter** of Babel Press. **Ross Carron** designed the text. **Alexander Teshin Associates** served as the art source. **Gary Morris** was copyeditor and **Steve Rath** compiled the index. **David Lance Goines** joined us once more as the cover designer. We would also like to thank **Steve Hiatt**, **Sandra Popovich**, and **Sharilyn Hovind** for their contributions to the Beta Edition of the book.

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

January 1993

The SPIM Simulator for the MIPS R2000/R3000

James R. Larus
University of Wisconsin

The SPIM S20 is a software simulator that runs assembly language programs for the MIPS R2000/R3000 RISC computers. SPIM can run files containing assembly language statements and read and run MIPS a.out files (when compiled and running on a system containing a MIPS processor). It is a self-contained system for running these programs and contains a debugger and interface to the operating system. SPIM is portable; it has run on a DECStation 3100/5100, Sun 3, Sun 4, PC/RT, IBM RS/6000, HP Bobcat, HP Snake, and Sequent. Students can generate code for a simple, clean, orthogonal computer, regardless of the machine used. SPIM comes with complete source code and documentation of all instructions.

SPIM has a simple terminal style and a flashy X-windows interface. SPIM also includes an optional extension by Anne Rogers and Scott Rosenberg of Princeton that performs a cycle-by-cycle MIPS simulation that exposes the hardware pipeline. UNIX, Windows, and Macintosh formats are available via anonymous FTP from the University of Wisconsin. See instructions below.

Retrieval of the SPIM by FTP

spim and xspim are available for anonymous ftp from ftp.cs.wisc.edu in the file pub/spim/spim.tar.Z (which is a compressed tar file).

For those unfamiliar with anonymous ftp, here are the steps to follow to get a copy of spim and xspim:

1. ftp to ftp.cs.wisc.edu from your computer:

   % ftp ftp.cs.wisc.edu

2. The ftp server will respond and ask you to login. login as anonymous and use your email address as a password:

   Name (ftp.cs.wisc.edu:larus): anonymous
   331 Guest login ok, send login or email address as password
   Password:

3. The server will then print a welcome message. Select the directory containing the spim format that you wish to retrieve:

```
ftp> cd pub/spim
```

4. Set binary mode for the transfer (since the file is compressed):

```
ftp> binary
```

5. Copy the file to your machine:

```
ftp> get spim.tar.Z          (UNIX)
ftp> get spim.zip            (Windows 3.1)
ftp> get SPIM.sit.bin        (Mac)
ftp> get SPIM.sit.Hqx.txt    (Mac)
```

6. Exit the ftp program:

```
ftp> quit
```

7. Uncompress and untar the file:

```
% uncompress spim.tar.Z
% tar xvf spim.tar
```

If the uncompression fails, you probably forgot to set binary (step 4). Try again. Read the directions in the file README.

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

This page intentionally left blank

1.1 **Introduction** 3

1.2 **Below Your Program** 5

1.3 **Under the Covers** 10

1.4 **Integrated Circuits: Fueling Innovation** 21

1.5 **Fallacies and Pitfalls** 26

1.6 **Concluding Remarks** 28

1.7 **Historical Perspective and Further Reading** 30

1.8 **Exercises** 41

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! Computer systems have a vital and synergistic relationship to an important industry—responsible for 5% to 10% of the gross national product of the United States—and this unusual industry embraces innovation at a breathtaking rate. In the last decade there have been a half-dozen new machines 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 computing's inception in the late 1940s. Had the transportation industry kept pace with the computer industry, for example, today we could travel coast to coast in 30 seconds for 50 cents. Take just a moment to contemplate how such an improvement would change society—living in Tahiti while working in San Francisco, going to Moscow for an evening at the Bolshoi ballet—and you can appreciate the implications of such a change.

Computers have led to a third revolution for civilization, with the information revolution taking its place alongside the agricultural and the industrial

revolutions. The resulting multiplication of humankind's intellectual strength and reach naturally has affected the sciences as well. There is now a new vein of scientific investigation, with computational scientists joining theoretical and experimental scientists in the exploration of new frontiers in astronomy, biology, chemistry, physics, ...

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

- *Automatic teller machines:* A computer placed in the wall of banks to distribute and collect cash was a ridiculous concept in the 1950s, when the cheapest computer cost at least $500,000 and was the size of a car.
- *Computers in automobiles:* Until microprocessors improved dramatically in price and performance in the early 1980s, computer control of cars was ludicrous. Today, computers reduce pollution and improve fuel efficiency via engine controls and increase safety through the prevention of dangerous skids and through the inflation of air bags to protect occupants in a crash.
- *Laptop computers:* Who would have dreamed that advances in computer systems would lead to laptop computers, allowing students to bring computers to coffeehouses and on airplanes?
- *Human genome project:* The cost of computer equipment to map human DNA sequences in the 1990s will be hundreds of millions of dollars. It's unlikely that anyone would have considered this project had the computer costs been 10 to 100 times higher, as they would have been 10 to 20 years ago.

Such hardware advances have allowed programmers to create infinitely useful software, and explain why computers are omnipresent. Today's science fiction computer applications include electronic libraries, the cashless society, automated intelligent highways, and genuinely ubiquitous computing—a pervasiveness which precludes the need to carry computers because they will be everywhere. Clearly, advances in this technology now affect almost every aspect of our society.

Successful programmers have always been concerned about the performance of their programs, because getting results to the user quickly is critical in creating successful software. In the 1960s and 1970s, a primary constraint on computer performance was the size of the computer's memory. Thus programmers often followed a simple credo: Minimize memory space to make programs fast. In the last decade advances in computer design and memory technology have greatly reduced the importance of small memory size. Programmers interested in performance now need to understand the issues that

have replaced the simple memory model of the 1960s: the hierarchical nature of memories and the parallel nature of processors. Programmers who seek to build competitive versions of compilers, operating systems, databases, and even applications will therefore need to increase their knowledge of computer organization.

We are honored to have the opportunity to explain what's inside this revolutionary machine, unraveling the software below your program and the hardware under the covers of your computer. By the time you finish this book, you will understand the secrets of programming a computer in its native tongue, the internal organization of computers and how it affects performance of your programs, and even how you would go about designing a computer of your own.

This first chapter lays the foundation for the rest of the book. It introduces the basic ideas and definitions, places the major components of software and hardware in perspective, and introduces integrated circuits, the technology that fuels the computer revolution.

1.2 Below Your Program

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

Mark Twain, *The Innocents Abroad,* 1869

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

```
1000110010100000
```

tell one computer to add two numbers. Chapter 3 explains why we use numbers for instructions *and* data; we don't want to steal that chapter's thunder, but using numbers for both instructions and data is a foundation of computing.

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

```
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*.

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

Such low-level thinking inspired a simple question: If we can write a program to translate from assembly language to binary instructions to simplify programming, what prevents us from writing a program that translates from some higher level notation down to assembly language?

The answer was: nothing. Although more challenging to create than an assembler, this higher level translator was plausible.

Programmers today owe their productivity, and their sanity, to this observation. Programs that accept this more natural notation are called *compilers*, and the languages they *compile* are called *high-level programming languages*. They enable a programmer to write this high-level language statement:

```
A + B
```

The compiler would compile it into this assembly language statement:

```
add A,B
```

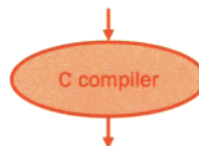
The assembler would translate this statement into the binary instruction that tells the computer to add the two numbers A and B:

```
1000110010100000
```

Figure 1.1 shows the relationships among these programs and languages.

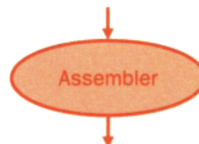
High-level
language
program
(in C)

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



Assembly
language
program
(for MIPS)

```
swap:
      muli $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
```



Binary machine
language
program
(for MIPS)

```
00000000101000010000000000011000
00000000100011100001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

FIGURE 1.1 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 3.

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 1.1). 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. The second advantage of programming languages is improved programmer productivity. One of the few areas of widespread agreement in software development is that it takes less time to develop programs when they are written in languages that require fewer lines to express an idea. Conciseness is a clear advantage of high-level languages over assembly language. The final advantage is that programming languages allow programs to be independent of the computer on which they were developed, since compilers and assemblers can translate high-level language programs to the binary instructions of any machine. These advantages are so strong that today little programming is done in assembly language.

As programming matured, many of its practitioners saw that reusing programs was much more efficient than writing everything from scratch. Hence programmers began to pool potentially widely used routines into libraries. One of the first of these *subroutine libraries* was for inputting and outputting data, which included, for example, routines to control printers, such as ensuring paper is in the printer before printing can begin. Such software controlled other input/output devices, such as magnetic disks, magnetic tapes, and displays. It soon became apparent that a set of programs could be run more efficiently if there was a separate program that supervised running those programs. As soon as one program completed, the supervising program would start the next program in the queue, thereby avoiding delays. These supervising programs, which soon included the input/output subroutine libraries, are the basis for what we call *operating systems* today. Operating systems are programs that manage the resources of a computer for the benefit of the programs that run on that machine.

Software came to be categorized by its use. Software that provides services that are commonly useful is called *systems software.* Operating systems, compilers, and assemblers are examples of systems software. In contrast to programs aimed at programmers, *applications software* or just *applications* is the name given to programs aimed at computer users, such as spreadsheets or text editors. Figure 1.2 shows the classical drawing mapping the hierarchical layers of software and hardware.

This simplified view has some problems. Should we really place compilers in the systems software level in Figure 1.2? Compilers produce programs at both the applications *and* the systems level, and applications programs don't normally call on the compiler while they are running. A more realistic view of the nature of systems appears in Figure 1.3. It shows that software does not

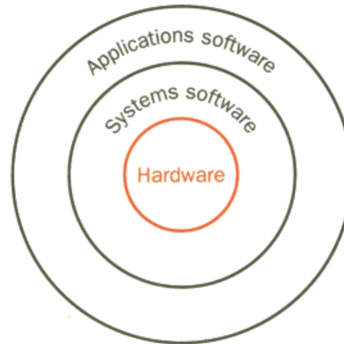


FIGURE 1.2 A simplified view of hardware and software as hierarchical layers, classically shown as concentric rings building up from the core of hardware to the software closest to the user.

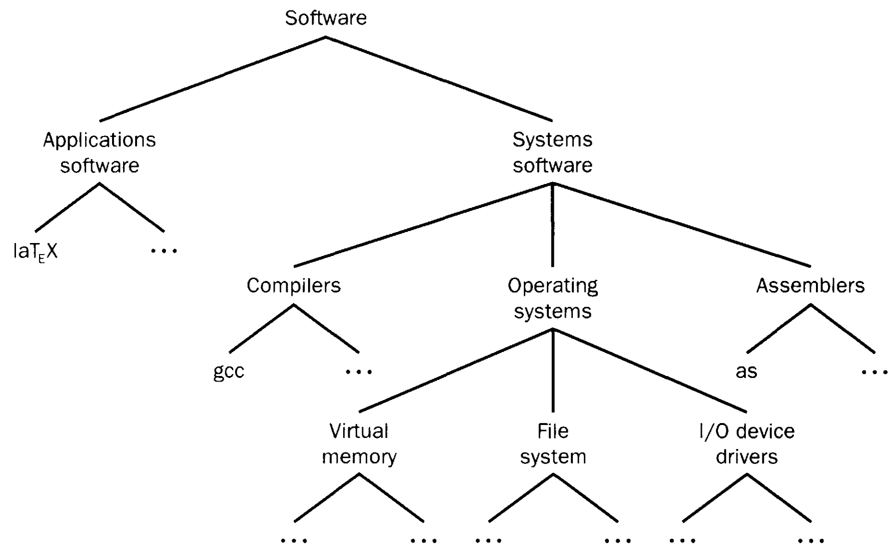


FIGURE 1.3 An example of the decomposability of computer systems. The terms in the middle of the chart, such as LaT$_E$X and gcc, are examples of Unix programs. The terms lower in the chart, such as virtual memory, will be introduced in Chapters 7 and 8.

consist of monolithic layers, but is composed of many programs that build on one another. Like the strands of a thick rope, each time you look carefully at what appears to be a single strand you find it is really composed of many finer components.

1.3 Under the Covers

Now that we have looked below your program to uncover the underlying software, let's open the covers of the computer to learn about the underlying hardware.

Figure 1.4 shows a typical workstation with keyboard, mouse, screen, and a box containing even more hardware. What is not visible in the photograph is a network that connects the workstation to printers and disks. This photograph reveals two of the key components of computers: *input devices*, such as the keyboard and mouse, and *output devices*, such as the screen and printers. As the names suggest, input feeds the computer and output is the result of computation sent to the user. Some devices, such as networks and disks, provide both input and output to the computer.

Chapter 8 describes input/output (I/O) devices in more detail, but let's take an introductory tour through the computer hardware, starting with the external I/O devices.

Anatomy of a Mouse

> *I got the idea for the mouse while attending a talk at a computer conference. The speaker was so boring that I started daydreaming and hit upon the idea.*
>
> Doug Engelbart

Although many users now take mice for granted, the idea of a pointing device such as a mouse is less than 30 years old. Engelbart showed the first demonstration of a system with a mouse on a research prototype in 1967. The Alto, which was the inspiration for all workstations as well as for the Macintosh, included a mouse as its pointing device in 1973. By the 1980s, all workstations and many personal computers included this device, and new user interfaces based on graphics displays and mice became popular. The mouse is actually quite simple, as the photograph in Figure 1.5 shows.

The mechanical version consists of a large ball that is mounted in such a way that it makes contact with a pair of wheels, one positioned on the x-axis and the other on the y-axis. These wheels either turn mechanical counters or turn a slotted wheel, through which a light-emitting diode (LED) shines on a

FIGURE 1.4 Photograph of a workstation. The cathode ray tube (CRT) screen is the primary output device, and the keyboard and mouse are the primary input devices. Photo courtesy of Silicon Graphics.

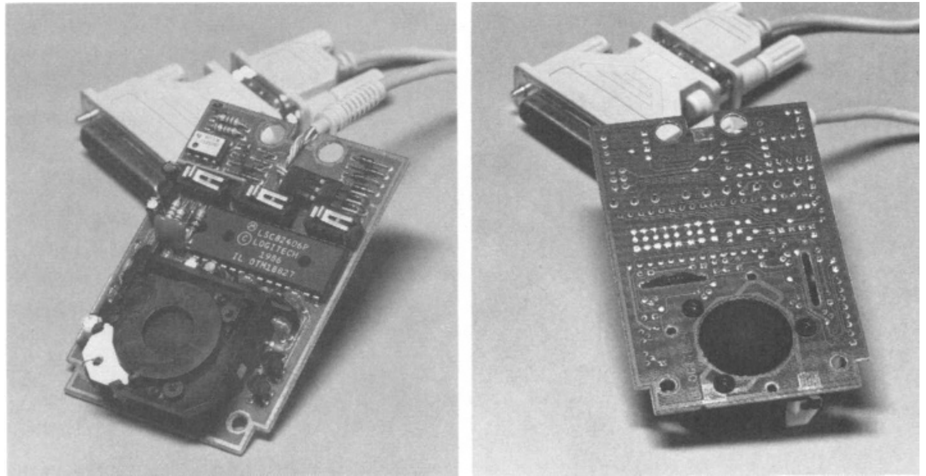


FIGURE 1.5 Photograph of the inside of a mechanical mouse. Mouse courtesy of Logitech.

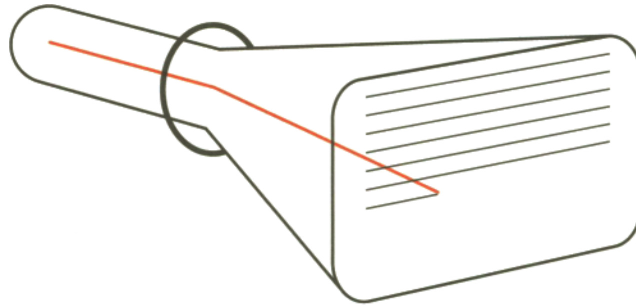


FIGURE 1.6 A CRT display. A beam is shot by an electronic gun through the vacuum onto a phosphor-coated screen. The steering coil at the neck of the CRT aims the gun. Raster scan systems, used in television and in almost all computers, paint the screen a line at a time as a series of dots, or pixels. The screen is refreshed 30 to 60 times per second.

photosensor. In either scheme, moving the mouse rolls the large ball, which turns the x-wheel or the y-wheel or both, depending on whether the mouse is moved in the vertical, horizontal, or diagonal direction. Although there are many styles of interfaces for these pointing devices, moving each wheel essentially increments or decrements counters somewhere in the system. The counters serve to record how far the mouse has moved and in which direction.

Through the Looking Glass

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

Ivan Sutherland, the "father" of computer graphics, quoted in "Computer Software for Graphics," *Scientific American*, 1984

The most fascinating I/O device is probably the graphics display. Based on television technology, a *raster cathode ray tube* (CRT) *display* scans an image one line at a time, 30 to 60 times per second (Figure 1.6). At this *refresh rate*, few people notice a flicker on the screen. The image is composed of a matrix of picture elements, or *pixels*, which can be represented as a matrix of bits, called a *bit map*. Depending on the size of screen and resolution, the display matrix ranges in size from 512×340 to 1560×1280 pixels. The simplest display has 1 bit per pixel, allowing it to be black or white. For displays that support over 100 different shades of black and white, sometimes called *gray-scale* displays, 8 bits per pixel are required. A color display might use 8 bits for

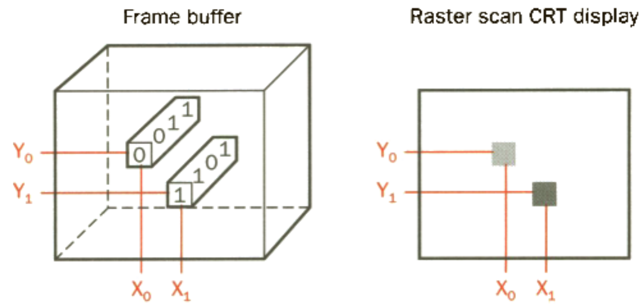


FIGURE 1.7 Each coordinate in the frame buffer on the left determines the shade of the corresponding coordinate for the raster scan CRT display on the right. Pixel (X_0,Y_0) contains the bit pattern 0011, which is a lighter shade of gray on the screen than the bit pattern 1101 in pixel (X_1,Y_1).

each of the three primary colors (red, blue, and green), for 24 bits per pixel, permitting millions of different colors to be displayed.

The hardware support for graphics consists mainly of a *raster refresh buffer*, or *frame buffer*, to store the bit map. The image to be represented on-screen is stored in the frame buffer, and the bit pattern per pixel is read out to the graphics display at the refresh rate. Figure 1.7 shows a frame buffer with 4 bits per pixel.

The goal of the bit map is to faithfully represent what is on the screen. The challenges in graphics systems arise because the human eye is very good at detecting even subtle changes on the screen. For example, when the screen is being updated, the eye can detect the inconsistency between the portion of the screen that has changed and that which hasn't.

Opening the Box

If we open the box containing the computer, we see a fascinating board of thin green plastic, covered with dozens of small gray or black rectangles. Figure 1.8 shows the contents of the workstation in Figure 1.4. The board is shown vertically on the left, with a tape reader and floppy disk drive shown on the right. The small rectangles on the board contain the devices that drive our advancing technology, *integrated circuits* or *chips*. The board is composed of three pieces: the piece connecting to the I/O devices mentioned above, the memory, and the processor. The *memory* is where the programs are kept when they are running; it also contains the data needed by the running programs. In Figure 1.8, memory is found on the eight small boards that are attached

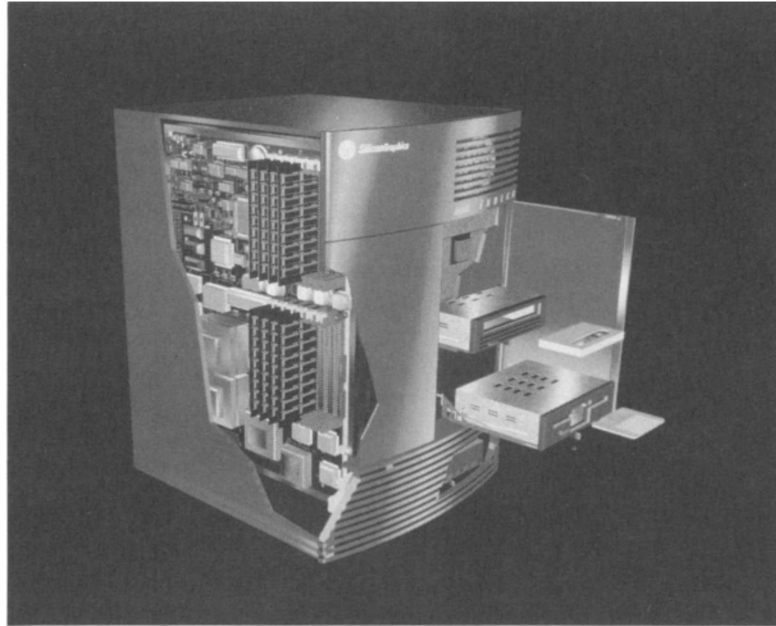


FIGURE 1.8 Inside a workstation. An exploded view of a workstation. The vertical board on the left is a printed circuit board (PC board) that contains most of the electronics of the computer; Figure 1.11 is an overhead photograph of that board, rotated 90 degrees. The eight small boards attached to the main board contain the memory chips. The processor is below the memory boards; Figure 1.18 is a photograph of the processor. To the right of the PC board in this workstation is a tape reader and a floppy disk drive. Photo courtesy of Silicon Graphics.

perpendicularly toward the front of the large board. Each small memory board contains 18 integrated circuits. The *processor* is the active part of the board, following the instructions of the programs 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*. The processor is the large square below the bottom memory boards and to the left in Figure 1.8.

Descending even lower into the hardware, Figure 1.9 reveals details of the processor in Figure 1.8. The processor comprises two main components: datapath and control, the respective brawn and brain of the processor. The *datapath* performs the arithmetic operations, and *control* tells the datapath, memory, and I/O devices what to do according to the wishes of the instructions of the program. Chapters 4 and 5 explain the datapath and control for a straightforward implementation, and Chapter 6 describes the changes needed for a higher performance design.

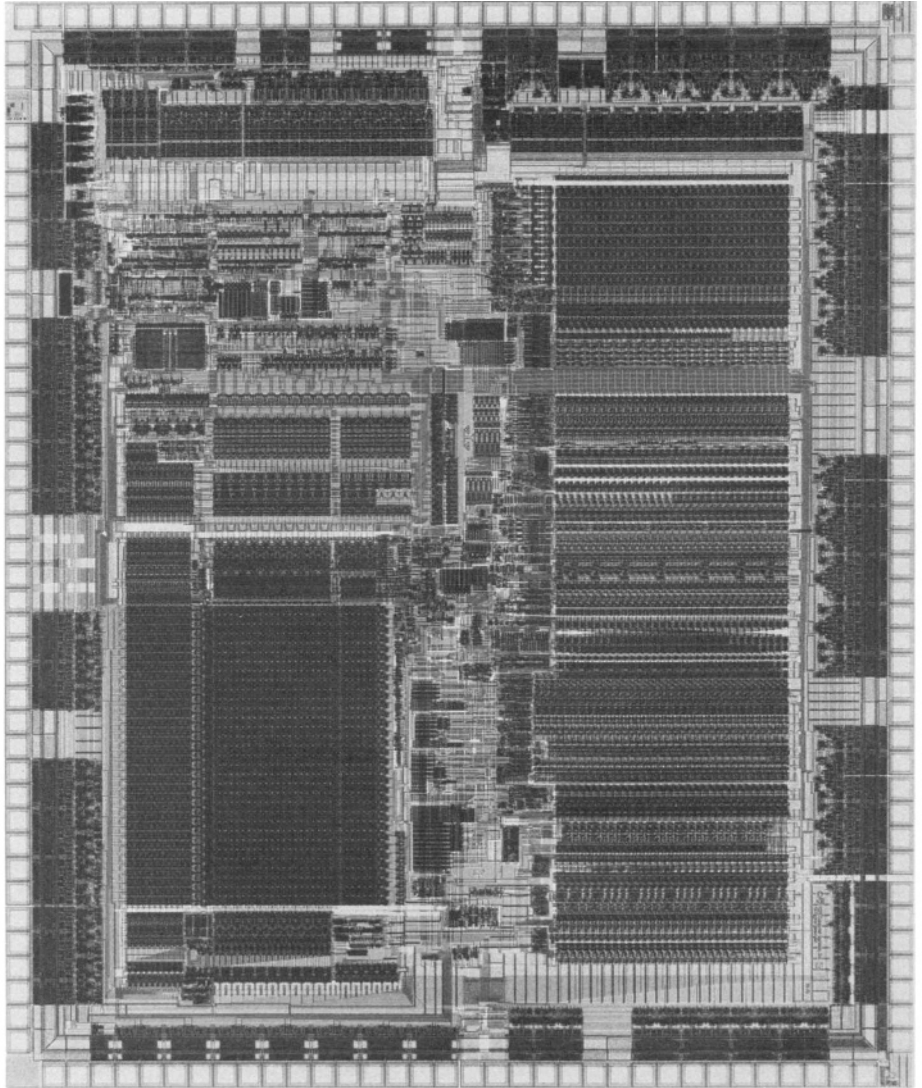


FIGURE 1.9 Inside the processor chip used on the board shown in Figure 1.8. The right-hand side of the chip is the datapath. The upper-left-hand side is the control unit. The lower left contains the portion of the memory system called the Translation Lookaside Buffer, which we discuss in Chapter 7. This chip is called the MIPS R3000. Photo courtesy of MIPS Technology, Inc.

We have now identified the major components of any computer. When we come to an important point in this book, a point so important that we hope you will remember it forever, we emphasize it by identifying it as a "Big Picture" item. We have about a dozen Big Pictures in this book, with the first being the five components of a computer.

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 1.10 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.

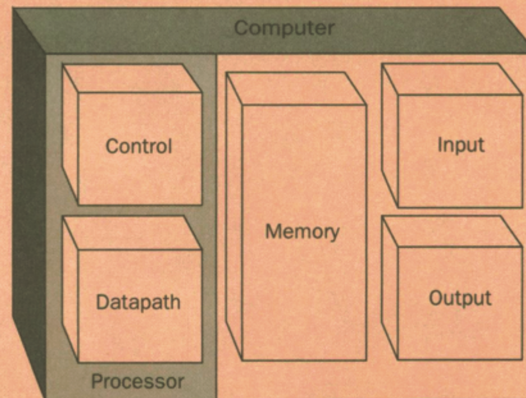


FIGURE 1.10 The organization of a computer, showing the five classic components. The processor gets instructions and data from memory; input writes data to memory, and output reads data from memory. Control sends the signals that determine the operations of the datapath, memory, input, and output.

Descending into the depths of any component of the hardware reveals insights into the machine. We have done this for the processor, so let's try memory. The board in Figure 1.11 contains two kinds of memories: DRAM and cache. *DRAM* stands for *dynamic random access memory.* Several DRAMs are used together to contain the instructions and data of a program. In contrast to sequential access memories such as magnetic tapes, the *RAM* portion of the term DRAM means that memory accesses take the same amount of time no matter what portion of the memory is read. *Cache* memory consists of a small,

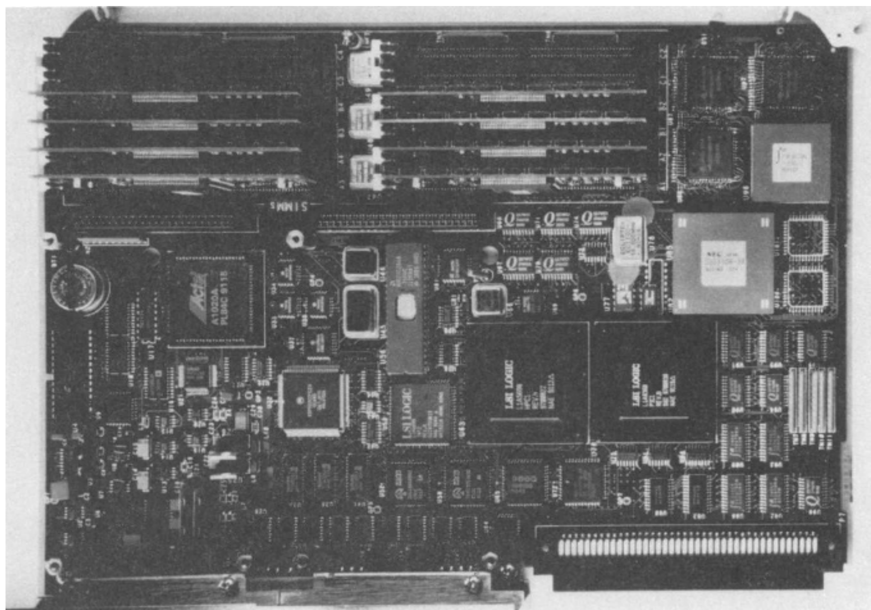


FIGURE 1.11 Close-up of workstation processor board. This board uses the MIPS R4000 processor, which is located on the right edge of the board in the middle. The R4000 contains high-speed cache memories on the processor chip. The main memory is contained on the small boards that are perpendicular to the motherboard in the upper left corner. The DRAM chips are mounted on these boards (called *SIMMs* for Single In-line Memory Module) and then plugged into the connectors. The connectors at the bottom of the photograph are for external I/O devices, such as the network (Ethernet), keyboard, and CRT display. Photo courtesy of Silicon Graphics.

fast memory that acts as a buffer for the DRAM memory. (The nontechnical definition of *cache* is a safe place for hiding things.)

The careful reader may have noticed a common theme in both the software and the hardware descriptions: delving into the depths of hardware or software reveals more information or, conversely, lower level details are hidden to offer a simpler model at higher levels. The use of such layers or *abstractions* is a principal technique for designing very sophisticated computer systems.

One of the most important abstractions is the interface between the hardware and the lowest level software. Because of its importance, it is given a special name: the *instruction set architecture*, or simply *architecture*, of a machine. The instruction set architecture includes anything programmers need to know to make a binary machine language program work correctly, including instructions, I/O devices, and so on. (The components of an architecture are discussed in Chapters 3, 4, 7, and 8.)

This standardized interface 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, with each lower layer hiding details from the level above. This principle of *abstraction* is the way both hardware designers and software designers cope with the complexity of computer systems. One key interface between the levels of abstraction is the *instruction set architecture*: the interface between the hardware and low-level software. This abstract interface enables many *implementations* of varying cost and performance to run identical software.

A Safe Place for Data

I think Silicon Valley was misnamed. If you look back at the dollars shipped in products in the last decade, there has been more revenue from magnetic disks than from silicon. They ought to rename the place Iron Oxide Valley.

Al Hoagland, one of the pioneers of magnetic disks, 1982

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, it forgets when it loses power. In contrast, a cassette tape for a stereo doesn't forget the recorded music when you turn off the power. This is because the tape is magnetic and is thus a *nonvolatile* memory technology. To distinguish between the memory used to hold programs while they are running and this nonvolatile memory used to store programs between runs, the term *primary memory* or *main memory* is used for the former and *secondary memory* for the latter. The DRAMs of Figure 1.11 are the main memory of that computer.