

FOURTH EDITION

THE ARCHITECTURE OF COMPUTER HARDWARE, SYSTEM SOFTWARE, AND NETWORKING

AN INFORMATION TECHNOLOGY APPROACH

Irv Englander

Bentley University



WILEY

John Wiley & Sons, Inc.

Vice President & Executive Publisher *Don Fowley*
Executive Editor *Beth Lang Golub*
Marketing Manager *Christopher Ruel*
Marketing Assistant *Diana Smith*
Design Director *Harry Nolan*
Senior Designer *Kevin Murphy*
Senior Production Editor *Patricia McFadden*
Senior Media Editor *Lauren Sapira*
Editorial Assistant *Mike Berlin*
Production Management Services *Kate Boilard, Laserwords Maine*

This book was set in 10/12 Minion by Laserwords India and printed and bound by Courier/Westford. The cover was printed by Courier/Westford.

The book is printed on acid free paper. ☺

Copyright © 2009 John Wiley & Sons, Inc. All rights reserved. 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, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc. 222 Rosewood Drive, Danvers, MA 01923, website www.copyright.com. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030-5774, (201)748-6011, fax (201)748-6008, website <http://www.wiley.com/go/permissions>.

To order books or for customer service please, call 1-800-CALL WILEY (225-5945).

ISBN-13: 978-0471-71542-9

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

To four outstanding teachers and great human beings:
With your guidance, inspiration, and patience, you showed me
that everything is possible.

Dr. Sidney H. Englander (1900–1980)
and Mildred K. Englander (1906–2008),
in memoriam my father and mother

Albert L. Daugherty, in memoriam
teacher of Science in Cleveland Heights, Ohio
from 1927 to 1970

Edith B. Malin, in memoriam
teacher of English in Cleveland Heights, Ohio
from 1924 to 1958

BRIEF CONTENTS

PART ONE

AN OVERVIEW OF COMPUTER SYSTEMS 2

■ CHAPTER 1 Computers and Systems 4

■ CHAPTER 2 An Introduction to System Concepts and Systems Architecture 38

PART TWO

DATA IN THE COMPUTER 66

■ CHAPTER 3 Number Systems 68

■ CHAPTER 4 Data Formats 96

■ CHAPTER 5 Representing Numerical Data 136

PART THREE

COMPUTER ARCHITECTURE AND HARDWARE OPERATION 178

■ CHAPTER 6 The Little Man Computer 180

■ CHAPTER 7 The CPU and Memory 198

■ CHAPTER 8 CPU and Memory: Design, Enhancement, and Implementation 240

■ CHAPTER 9 Input/Output 276

■ CHAPTER 10 Computer Peripherals 306**■ CHAPTER 11 Modern Computer Systems 342****PART FOUR****NETWORKS AND DATA COMMUNICATIONS 368****■ CHAPTER 12 Networks and Data Communications 370****■ CHAPTER 13 Ethernet and TCP/IP Networking 422****■ CHAPTER 14 Communication Channel Technology 446****PART FIVE****THE SOFTWARE COMPONENT 476****■ CHAPTER 15 Operating Systems: An Overview 478****■ CHAPTER 16 The User View of Operating Systems 514****■ CHAPTER 17 File Management 548****■ CHAPTER 18 The Internal Operating System 592****SUPPLEMENTARY CHAPTERS**

On the Web at www.wiley.com/college/englander

■ SUPPLEMENTARY CHAPTER 1 An Introduction to Digital Computer Logic**■ SUPPLEMENTARY CHAPTER 2 System Examples****■ SUPPLEMENTARY CHAPTER 3 Instruction Addressing Modes****■ SUPPLEMENTARY CHAPTER 4 Programming Tools**

CONTENTS

Preface	xix
About the Author	xxv

PART ONE

AN OVERVIEW OF COMPUTER SYSTEMS 2

■ CHAPTER 1 Computers and Systems 4

1.0	Introduction	5
1.1	The Starting Point	9
1.2	Components of the Computer System	12
	The Hardware Component	13
	The Software Component	16
	The Communication Component	18
	The Computer System	18
1.3	The Concept of Virtualization	20
1.4	Protocols and Standards	20
1.5	Overview of This Book	22
1.6	A Brief Architectural History of the Computer	23
	Early Work	24
	Computer Hardware	25
	Operating Systems	28
	Communication, Networks, and the Internet	33
	Summary and Review	34
	For Further Reading	34
	Key Concepts and Terms	35
	Reading Review Questions	35
	Exercises	36

■ CHAPTER 2 An Introduction to System Concepts and Systems Architecture 38

2.0	Introduction	39
2.1	The General Concept of Systems	40
2.2	IT System Architectures	48
	Distributed Processing Systems	49

The Role of the System Architect	57
Google: A System Architecture Example	58
Summary and Review	62
For Further Reading	63
Key Concepts and Terms	63
Reading Review Questions	63
Exercises	64

PART TWO

DATA IN THE COMPUTER 66

■ CHAPTER 3 Number Systems 68

3.0	Introduction	69
3.1	Numbers as a Physical Representation	70
3.2	Counting in Different Bases	70
3.3	Performing Arithmetic in Different Number Bases	74
3.4	Numeric Conversion between Number Bases	77
	An Alternative Conversion Method	79
3.5	Hexadecimal Numbers and Arithmetic	81
3.6	A Special Conversion Case—Number Bases that are Related	81
3.7	Fractions	83
	Fractional Conversion Methods	86
3.8	Mixed Number Conversions	89
	Summary and Review	89
	For Further Reading	90
	Key Concepts and Terms	90
	Reading Review Questions	90
	Exercises	91

■ CHAPTER 4 Data Formats 96

4.0	Introduction	97
4.1	General Considerations	97
4.2	Alphanumeric Character Data	100
	Keyboard Input	106
	Alternative Sources of Alphanumeric Input	107
4.3	Image Data	109
	Bitmap Images	110
	Object Images	114
	Representing Characters as Images	117
	Video Images	117
	Image and Video Input	118
4.4	Audio Data	119
4.5	Data Compression	123

4.6	Page Description Languages	124
4.7	Internal Computer Data Format	125
	Numerical Character to Integer Conversion	127
	Summary and Review	128
	For Further Reading	129
	Key Concepts and Terms	130
	Reading Review Questions	130
	Exercises	131

■ CHAPTER 5 Representing Numerical Data 136

5.0	Introduction	137
5.1	Unsigned Binary and Binary-Coded Decimal Representations	138
5.2	Representations for Signed Integers	141
	Sign-and-magnitude Representation	142
	Nine's Decimal and 1's Binary Complementary Representations	143
	Ten's Complement and 2's Complement	150
	Overflow and Carry Conditions	153
	Other Bases	153
	Summary of Rules for Complementary Numbers	154
5.3	Real Numbers	155
	A Review of Exponential Notation	155
	Floating Point Format	157
	Normalization and Formatting of Floating Point Numbers	159
	A Programming Example	162
	Floating Point Calculations	163
	Floating Point in the Computer	165
	Conversion between Base 10 and Base 2	167
5.4	Programming Considerations	168
	Summary and Review	169
	For Further Reading	170
	Key Concepts and Terms	171
	Reading Review Questions	171
	Exercises	172

PART THREE

COMPUTER ARCHITECTURE AND HARDWARE OPERATION 178

■ CHAPTER 6 The Little Man Computer 180

6.0	Introduction	181
6.1	Layout of the Little Man Computer	181

6.2	Operation of the LMC	183
6.3	A Simple Program	185
6.4	An Extended Instruction Set	186
6.5	The Instruction Cycle	189
6.6	A Note Regarding Computer Architectures	192
Summary and Review	192	Key Concepts and Terms 193
Reading Review Questions	193	Exercises 194

■ CHAPTER 7 The CPU and Memory 198

7.0	Introduction	199
7.1	The Components of the CPU	200
7.2	The Concept of Registers	201
7.3	The Memory Unit	204
	The Operation of Memory	204
	Memory Capacity	208
	Primary Memory Characteristics and Implementation	209
7.4	The Fetch-Execute Instruction Cycle	211
7.5	Buses	214
	Bus Characteristics	214
7.6	Classification of Instructions	218
	Data Movement Instructions (LOAD, STORE, and Other Moves)	219
	Arithmetic Instructions	221
	Boolean Logic Instructions	222
	Single Operand Manipulation Instructions	222
	Bit Manipulation Instructions	222
	Shift and Rotate Instructions	223
	Program Control Instructions	224
	Stack Instructions	225
	Multiple Data Instructions	228
	Other Instructions	229
7.7	Instruction Word Formats	229
7.8	Instruction Word Requirements and Constraints	230
Summary and Review	234	For Further Reading 234
Key Concepts and Terms	235	Reading Review Questions 235
Exercises	237	

■ CHAPTER 8	CPU and Memory: Design, Enhancement, and Implementation	240
8.0	Introduction	241
8.1	CPU Architectures	242
	Overview	242
	Traditional Modern Architectures	243
	VLIW and EPIC Architectures	244
8.2	CPU Features and Enhancements	246
	Introduction	246
	Fetch-Execute Cycle Timing Issues	247
	A Model for Improved CPU Performance	249
	Scalar and Superscalar Processor Organization	253
8.3	Memory Enhancements	256
	Wide Path Memory Access	257
	Memory Interleaving	258
	Cache Memory	259
8.4	The Compleat Modern Superscalar CPU	263
8.5	Multiprocessing	265
8.6	A Few Comments on Implementation	269
	Summary and Review	269
	For Further Reading	270
	Key Concepts and Terms	271
	Reading Review Questions	271
	Exercises	272
 ■ CHAPTER 9	 Input/Output	 276
9.0	Introduction	277
9.1	Characteristics of Typical I/O Devices	278
9.2	Programmed I/O	284
9.3	Interrupts	285
	Servicing Interrupts	286
	The Uses of Interrupts	288
	Multiple Interrupts and Prioritization	293
9.4	Direct Memory Access	297
9.5	I/O Modules	300
	Summary and Review	302
	For Further Reading	303
	Key Concepts and Terms	303
	Reading Review Questions	304
	Exercises	304

■ CHAPTER 10 Computer Peripherals 306

- 10.0 Introduction 307**
- 10.1 The Hierarchy of Storage 308**
- 10.2 Solid State Memory 310**
- 10.3 Magnetic Disks 311**
 - Disk Arrays 317
- 10.4 Optical Disk Storage 318**
- 10.5 Magnetic Tape 321**
- 10.6 Displays 322**
 - Liquid Crystal Display Technology 328
 - CRT Display Technology 329
 - OLED Display Technology 330
- 10.7 Printers 330**
 - Laser Printers 331
 - Inkjet Printers 332
 - Thermal Wax Transfer and Dye Sublimation Printers 333
- 10.8 User Input Devices 333**
 - Keyboards and Pointing Devices 333
 - Scanners 335
 - Multimedia Devices 335
- 10.9 Network Communication Devices 335**
- Summary and Review 336 For Further Reading 337**
- Key Concepts and Terms 337 Reading Review Questions 338**
- Exercises 339**

■ CHAPTER 11 Modern Computer Systems 342

- 11.0 Introduction 343**
- 11.1 Putting All the Pieces Together 345**
- 11.2 Input/Output System Architectures 353**
 - I/O Bus Architecture 354
 - Channel Architecture 357
 - Blurring the Line 358
- 11.3 Computer Interconnection: A Brief Overview 359**
- 11.4 Clusters 360**
 - Overview 360
 - Classification and Configuration 360
 - Beowulf Clusters 362

11.5 High-Performance Computing	363
Grid Computing	364
Summary and Review	364
For Further Reading	365
Key Concepts and Terms	365
Reading Review Questions	366
Exercises	366

PART FOUR

NETWORKS AND DATA COMMUNICATIONS 368

■ CHAPTER 12 Networks and Data Communications 370

12.0 Introduction	371
12.1 The Impact of Networking on Business Processes and User Access to Knowledge and Services	372
12.2 A Simple View of Data Communications	373
12.3 Basic Data Communication Concepts	376
Messages	377
Packets	377
General Channel Characteristics	378
Packet Routing	382
12.4 TCP/IP, OSI, and Other Communication Models	386
Overview	386
The TCP/IP Network Model	387
The OSI Network Model	395
Addressing	396
12.5 Types of Networks	398
Network Topology	399
Local Area Networks	402
Backbone Networks	407
Metropolitan Area Networks	409
Wide Area Networks (WAN)	411
Internet Backbones and the Internet	412
Piconets	414
12.6 Standards	415
Summary and Review	416
For Further Reading	417
Key Concepts and Terms	417
Reading Review Questions	418
Exercises	419

■ CHAPTER 13	Ethernet and TCP/IP Networking	422
13.0	Introduction	423
13.1	Introducing The Process—The Application Layer	423
13.2	Domain Names and DNS Services	424
	Domain Name System Directory Services	424
13.3	Next Steps—TCP and the Transport Layer	429
13.4	The Network Layer, IP Addresses, and ARP	430
	IP Addresses	431
	Dynamic Host Configuration Protocol (DHCP)	433
	The Operation of IP	434
13.5	The Data Link Layer	435
	Hub-Based Ethernet	436
	Switched Ethernet	437
13.6	Quality of Service	437
13.7	Network Security	438
	Physical and Logical Access Restriction	439
	Encryption	440
13.8	Alternative Protocols	440
	MPLS	440
	ATM	441
	SONET/SDH	441
	Frame Relay	441
	Summary and Review	442
	For Further Reading	442
	Key Concepts and Terms	443
	Reading Review Questions	443
	Exercises	443
 ■ CHAPTER 14	 Communication Channel Technology	 446
14.0	Introduction	447
14.1	Communication Channel Technology	447
14.2	The Fundamentals of Signaling Technology	450
	Analog Signaling	451
	Digital Signaling	460
	Modems and Codecs	465
14.3	Transmission Media and Signaling Methods	466
14.4	Wireless Networking	468
	Wi-Fi	469

Summary and Review	471	For Further Reading	471
Key Concepts and Terms	471	Reading Review Questions	472
Exercises	473		

PART FIVE

THE SOFTWARE COMPONENT 476

■ CHAPTER 15 Operating Systems: An Overview 478

15.0	Introduction	479
15.1	The Barebones Computer System	480
15.2	The Operating Systems Concept: An Introduction	481
15.3	Services and Facilities	488
	User Interface and Command Execution Services	489
	File Management	490
	Input/Output Services	491
	Process Control Management	492
	Memory Management	493
	Scheduling and Dispatch	493
	Secondary Storage Management	496
	Network and Communications Support Services	496
	Security and Protection Services	497
	System Administration Support	498
15.4	Organization	502
15.5	Types of Computer Systems	505
	Summary and Review	509
	For Further Reading	510
	Key Concepts and Terms	510
	Reading Review Questions	510
	Exercises	511

■ CHAPTER 16 The User View of Operating Systems 514

16.0	Introduction	515
16.1	Purpose of the User Interface	516
16.2	User Functions and Program Services	518
	Program Execution	518
	File Commands	519
	Disk and Other I/O Device Commands	520
	Security and Data Integrity Protection	521
	Interuser Communication and Data Sharing Operations	521
	System Status Information	522
	Program Services	523

16.3	Types of User Interface	524
	The Command Line Interface	525
	Batch System Commands	527
	Graphical User Interfaces	528
	Trade-offs in the User Interface	533
	Software Considerations	535
16.4	X Window and Other Graphics Display Methodologies	536
16.5	Command and Scripting Languages	539
	The Elements of a Command Language	541
	The Command Language Start-up Sequence Files	542
16.6	Services to Programs	542
	Summary and Review	544
	For Further Reading	544
	Key Concepts and Terms	544
	Reading Review Questions	545
	Exercises	545

■ CHAPTER 17 File Management 548

17.0	Introduction	549
17.1	The Logical and Physical View of Files	549
17.2	The Role of the File Management System	554
17.3	Logical File Access Methods	560
	Sequential File Access	560
	Random Access	560
	Indexed Access	561
17.4	Physical File Storage	562
	Contiguous Storage Allocation	562
	Noncontiguous Storage Allocation	564
	Indexed Allocation	566
	Free Space Management	568
	Tape Allocation	569
	CD, DVD, and Flash Drive Allocation	570
17.5	File Systems, Volumes, Disks, Partitions, and Storage Pools	570
17.6	The Directory Structure	573
	Tree-Structured Directories	575
	Acyclic-Graph Directories	577
17.7	Network File Access	581
17.8	Storage Area Networks	582
17.9	File Protection	584
17.10	Journaling File Systems	585

Summary and Review	586	For Further Reading	586
Key Concepts and Terms	587	Reading Review Questions	587
Exercises	588		

■ CHAPTER 18 The Internal Operating System 592

18.0	Introduction	593
18.1	Fundamental OS Requirements	594
	Example: A Simple Multitasking Operating System	596
18.2	Starting the Computer System: The Bootstrap	599
18.3	Processes and Threads	601
	Process Creation	604
	Process States	605
	Threads	606
18.4	Basic Loading and Execution Operations	607
18.5	CPU Scheduling and Dispatching	608
	High-Level Scheduler	608
	Dispatching	610
	Nonpreemptive Dispatch Algorithms	612
	Preemptive Dispatch Algorithms	613
18.6	Memory Management	615
	Memory Partitioning	615
18.7	Virtual Storage	617
	Overview	617
	Pages and Frames	618
	The Concept of Virtual Storage	623
	Page Faults	624
	Working Sets and the Concept of Locality	626
	Page Sharing	627
	Page Replacement Algorithms	627
	Thrashing	629
	Page Table Implementation	630
	Segmentation	632
	Process Separation	633
18.8	Secondary Storage Scheduling	633
	First-Come, First-Served Scheduling	633
	Shortest Distance First Scheduling	634
	Scan Scheduling	634
	N-STEP C-SCAN Scheduling	635

18.9 Network Operating System Services	635
OS Protocol Support and Other Services	635
18.10 Other Operating System Issues	638
Deadlock	638
Other Issues	640
18.11 Virtual Machines	641
Summary and Review	643
For Further Reading	644
Key Concepts and Terms	644
Reading Review Questions	645
Exercises	647
 Bibliography	 653
Index	665

SUPPLEMENTARY CHAPTERS

On the Web at www.wiley.com/college/englander

■ SUPPLEMENTARY CHAPTER 1 An Introduction to Digital Computer Logic

S1.0 Introduction	
S1.1 Boolean Algebra	
S1.2 Gates and Combinatorial Logic	
S1.3 Sequential Logic Circuits	
Summary and Review	For Further Reading
Key Concepts and Terms	Reading Review Questions
Exercises	

■ SUPPLEMENTARY CHAPTER 2 System Examples

S2.0 Introduction
S2.1 Hardware Examples
The x86 Family
The POWER Family
The IBM System 360/370/390/zSeries Family
S2.2 Operating System Examples
The Microsoft Windows Family
UNIX and Linux
The IBM z/OS Operating System
S2.3 Networking Examples
Google

Summary and Review For Further Reading
Key Concepts and Terms Reading Review Questions
Exercises

■ SUPPLEMENTARY CHAPTER 3 Instruction Addressing Modes

S3.0 Introduction
S3.1 Register Addressing
S3.2 Alternatives to Absolute Addressing
S3.3 Alternatives to Direct Addressing
 Immediate Addressing
 Indirect Addressing
 Register Indirect Addressing
 Indexed Addressing
 Indirect Indexed and Indirect Indexed Addressing
Summary and Review For Further Reading
Key Concepts and Terms Reading Review Questions
Exercises

■ SUPPLEMENTARY CHAPTER 4 Programming Tools

S4.0 Introduction
S4.1 Program Editing and Entry
S4.2 The Concept of Program Translation
S4.3 Assembly Language and the Assembler
 Operation of the Assembler
 Assembly Language Formats
 Features and Extensions
 Relocatability
S4.4 Program Language Description and Rules
 A Description of Written English
 Programming Language Rules
 Computer Language Descriptions
 The Compilation Process
 Interpreters
S4.5 Linking and Loading
S4.6 Debuggers
Summary and Review For Further Reading
Key Concepts and Terms Reading Review Questions
Exercises

PREFACE

The modern world offers lots of readily available online resources for learning. Wikipedia, Google, news sources, millions of Web sites and blogs, even YouTube, offer access to information in nearly any subject that triggers your curiosity and interest. Nonetheless, I continue to believe that for deep understanding of something, nothing beats the integrated approach and focus of an old-fashioned printed-on-paper textbook.

When I open a new book, in *any* subject, the first thing I want to know is what the book has to offer that makes it worth my while to read it. I would like to try to help you answer that question for the book that you're holding in your hand.

The information systems and technology fields are wonderfully exciting places to be! It seems as though every day brings new developments that alter the ways we create and work with information. Of course, with this excitement comes a challenge. To be a successful player in IS or IT we have to be adaptable and flexible.

Much of the change occurs around computer system technology. The computer is, after all, at the foundation of information systems. A deep understanding of computer systems is, therefore, an essential element of success. We must be able to understand each new development, assess its value, and place it in the context of our knowledge of computer systems.

The subject of this book is the architecture of computer systems. Computer architecture is about the structure and operation of digital computers. Computer architecture is concerned with the operational methods of the hardware; with the services provided by operating system software; with the acquisition, processing, storage, and output of data; and with the interaction between computers.

There is a tendency for people in information systems and technology to neglect a study of computer architecture. After all, the technology changes so rapidly—is it really worth trying to understand something that may be out of date by the time I finish this book? There is no question that computer technology has evolved rapidly. The computer in a personal computer, or even in a cell phone or MP3 player is far more powerful than the mainframe computer of twenty-five years ago, with memory, disk and flash storage capacity, display and multimedia capability, and ease of use that would have been unthinkable just a few years ago. Even more important, connecting systems to work together is now routine and simple.

Interestingly enough, however, as profound as advances in the technology have been, the concepts of computer architecture that really matter have changed only nominally over the last sixty years. The new technologies are based on a foundation of architectural concepts that were developed many years ago. The architecture of a modern computer system was developed in the 1940s. The instruction set in a modern personal computer

is nearly identical to that of computers built in the 1950s and 1960s. Modern operating system techniques were developed in the 1960s. The graphical user interface is based on a 1960s project. The Internet is built from concepts developed more than forty years ago.

So you see that an understanding of computer architecture makes it possible to “ride the wave” of technological change, secure in the feeling that you are equipped to deal with new developments as they occur, and to have fun doing so. When you are done reading this book you will have substantial knowledge about how a computer works and a good understanding of the operating concepts, the hardware, and system software that make up a computer. You will see the interaction between computers and between data and the computer. Plus, you will have learned lots of jargon that you can show off at parties and job interviews.

This textbook is designed for a wide range of readers, both undergraduate and graduate. The material is specifically directed toward IS and IT majors. There are no explicit prerequisites, although the book assumes that the student is familiar with a personal computer. It also assumes (but does not require) some basic programming skills: although there is no programming in the book, program code is occasionally used as an example to clarify an idea, and a knowledge of programming is helpful at understanding instruction set design and program execution concepts. The material in this textbook conforms to the criteria of the IT Infrastructure course as described in the December 2008 draft of the joint IS 2008 standard curriculum. Although the material in this book may be useful as background for system design and implementation project courses, the course can be placed anywhere in the curriculum.

Most instructors will not cover the entire textbook in a single semester. The organization of the book is designed to allow an instructor to cover the major topic areas in different levels of depth, depending on the experience and needs of the students. On the other hand, it is my intention that this book will serve a student as a useful reference long after the formal course is completed. It is designed for use as a book where a professional can look up the basic concepts that clarify new developments as they occur.

This text is the outgrowth of courses that I have taught to IS majors and minors at Bentley University at both the undergraduate and graduate level for thirty years. Student responses to the material and the approach have generally been very enthusiastic. Many students have returned after graduation to tell me that their knowledge in this area has directly contributed to their career development. Along the way, student comments have also been extremely helpful to me in the book’s continuing evolution.

Those familiar with previous editions will notice that the organization of the fourth edition has undergone substantial revision to reflect current technological practices and trends. In particular, it is no longer reasonable to discuss computers as individual units without also considering the networks that tie them together; computer networking is now covered thoroughly in its own section, and there is an increased emphasis on the integration and synergy of the various components of the computer system and on the system as a whole. Still, the basic philosophy, organization, and approach remain essentially similar to those of the first edition, reflecting the unchanging nature of the underlying principles.

ORGANIZATION OF THE FOURTH EDITION OF THE BOOK

The biggest challenge for me as the author of this book has been to preserve the guiding principles established in the first edition, while reflecting the major changes in the way computers are used, in the rapid deployment of new technology, and in the resulting evolution of IS/IT curriculum to reflect those changes. The fourth edition is the most substantial revision of this book to date, with a new title, a new chapter on systems, and significantly increased coverage of networking. The case study chapters have been updated and moved to the Web, along with the chapter on programming tools and the supplementary chapters on logic design and instruction addressing. Still, users of previous editions will find much that is familiar; after all, the way in which computers are used in IS/IT may have changed, but the basic guiding principles of computer architecture are essentially the same as they have been for many years.

The book is now organized into five parts totaling eighteen chapters, plus four additional supplementary chapters that are posted on the Web. The first part serves as an introduction and overview of the role of the computer in information systems; it introduces the concept of a system and provides a brief introduction to each of the components that make up a modern computer system. Each of the remaining four parts deals with a single architectural aspect of the computer system.

Part Two discusses the role and representation of data in the computer. Here we consider numbers, text, sound, images, video, and other data forms. Part Three presents the hardware architecture and operational concepts. It introduces the components of a computer and shows how they collaborate to execute computer instructions, discusses the nature of a computer instruction set, and explores the interaction between the CPU, memory, and I/O peripheral devices. Part Four presents a thorough introduction to the basics of computer networking. Part Five discusses the system software, the programs that function to make the resources of the computer system, and other interconnected computer systems and components, accessible to the user and to application programs.

The approach within each group of chapters is layered. Each new layer builds upon the previous material to add depth and understanding to the reader's knowledge. Each topic section consists of a short introduction that places the topic to be discussed into the context of the computer system as a whole and then lays out in detail the organization of the chapters within the section. Each topic area is introduced as gently as possible, using ideas and examples that are already familiar to the student. Successive material is progressive and accumulative. In addition to the numerous examples that are used throughout the text, the supplementary chapters offer substantial case studies that show application of the text material to current examples of importance. Overall, the approach is gentle, progressive, and accumulative. As much as possible, each section is self-contained.

An overview of the organization of each part follows. More details can be found in the introductions to each section.

Part One consists of two chapters that present a short overview of computing, placing architectural concepts into the context of information technology. Chapter 1 introduces the components of a computer system and shows the relationships among the components.

It also presents a simple model of computing and discusses the importance of standards and protocols in the development of computer systems. The chapter concludes with a short history of computers from the architectural point of view. Chapter 2 focuses on the concepts of systems, models, and system architectures, using various types of computer systems as examples.

Chapters 3 through 5 comprise Part Two. Chapter 3 introduces number systems and basic number system operations; it then explores the relationships between numbers in different number bases and the conversion techniques between the different representations. Chapter 4 investigates different types of data formats, including alphanumeric, image, video, and audio formats. It considers the relationship between numerical and character-based representations and briefly introduces various devices and data formats used for data input and output. Chapter 5 studies the various formats that are used to represent and to perform calculations on integer and floating point numbers.

Part Three discusses the hardware architecture and operational aspects of the computer. Chapter 6 begins the study with the introduction of the Little Man Computer, a simple model that provides a surprisingly accurate representation of the CPU and memory. The model is used to develop the concept of an instruction set and to explain the basic principles of the von Neumann architecture. Chapter 7 extends the discussion to a real computer. It introduces the components of the CPU and shows their relationship to the Little Man Computer model. It introduces the bus concept, explains the operation of memory, presents the instruction fetch-execute cycle, and discusses the instruction set. It identifies important classes of instructions and discusses the ways in which instructions can be categorized.

Chapter 8 expands the material in Chapter 7 to consider more advanced features of the CPU and memory. It offers an overview of various CPU architectures. It continues with a discussion of techniques for improving memory access, particularly cache memory, and an introduction to current CPU organization, design, and implementation techniques, including pipelining and superscalar processing. This chapter also introduces multiprocessing (or multicore, in current terminology) concepts.

Chapter 9 presents the principles of I/O operation, and Chapter 10 illustrates how I/O is performed in various I/O devices. Chapter 11 discusses the computer system as a whole. It discusses interconnection techniques and integration of the various hardware components. It also addresses the interconnection of computers to increase computer performance and reliability, with a specific focus on clustering and on grid computing.

Three supplementary chapters on the Web provide additional resources to support the chapters in Part Three. Supplementary Chapter 1 (SC1) offers an introduction to Boolean algebra, combinatorial logic, and sequential logic for those readers that would like a deeper understanding of the computer in its simplest and most elegant form. Supplementary Chapter 2 (SC2) offers three detailed case studies of important architectures: the Intel x86 family, including the Pentium IV architecture and Itanium extensions, the PowerPC, and the IBM zSystem. Supplementary Chapter 3 (SC3) discusses alternative approaches to instruction addressing.

Part Four presents a thorough introduction to networking. Chapter 12 introduces the major features and characteristics of networking, including a careful introduction to communication channels, a detailed discussion of layered network models, with particular emphasis on TCP/IP and Ethernet models, an introduction to network topologies, and finally, a discussion of the different types of networks in use, including LANs, MANs, WANs,

and the backbones that form the foundation of the Internet. Chapter 13 expands on the material in Chapter 12 to discuss specific details of various layers, including discussions of DNS, TCP connections, IP and physical address resolution, the operation of Ethernet, alternative protocols, and more. Chapter 14 focuses primarily on communication channel technology, including analog and digital signaling, modulation and data conversion techniques between analog and digital, the characteristics of transmission media, and wireless networking. A portion of Chapter 14 appeared in previous editions as a supplementary chapter.

Part Five is dedicated to a discussion of system software. Chapter 15 provides an overview of the operating system. It explains the different roles played by the operating system and introduces the facilities and services provided. Chapter 16 presents the role of the operating system from the viewpoint of the user of a system. Chapter 17 discusses the all-important topic of file systems. Chapter 18 discusses the operating system as a resource manager, with an in-depth discussion of memory management, scheduling, process control, network services, and other basic operating system services. Chapter 18 includes a detailed introduction to virtual memory technique, and also includes an introduction to virtual machines. In addition to its hardware discussions, Supplementary Chapter 3 also provides current Windows, UNIX/Linux, and z/OS case studies.

A fourth supplementary chapter provides an introduction to the system development software that is used for the preparation and execution of programs.

A detailed list of the changes between the second and third editions of the book can be found at the book Web site, www.wiley.com/college/englander.

This book has been a continuing labor of love. My primary goal has been to create and maintain a textbook that explains computer architecture in a way that conveys to you, the reader, the sense of excitement and fun that I believe makes a career in information systems and technology so satisfying. I hope that I have succeeded to some extent.

ADDITIONAL RESOURCES

Additional resources for students and instructors may be found at the textbook Web site, www.wiley.com/college/englander. I can also be reached directly by e-mail at ienglander@bentley.edu. Although I am happy to communicate with students, I am unable to supply tutorial help or answers to review questions and exercises in the book.

ACKNOWLEDGMENTS

I've discovered that a major, ongoing textbook project is a formidable task. Many individuals have helped to make the task manageable—and kept me going when, from time to time, I became convinced that textbooks really *do* appear by magic and are *not* written by humans. It is impossible to thank people adequately for all their help and support. First and foremost, a special thank you to my nearest and dearest friends, Wilson Wong, Rich Braun, Luis Fernandez, Jan Harrington, Ray Brackett, and Evan Horn. Their continuing backup through four editions has been amazing! I couldn't have asked for a better support team. The champagne is on ice. *Yet again!*

My continuing thanks, too, to Stuart Madnick. Stuart, your technical inspiration and personal encouragement was invaluable to me when I struggled to get the first edition

of this book going. You helped me to believe that this project was actually possible and worthwhile. That support has continued to inspire me through every subsequent edition.

Next, I thank the many colleagues at Bentley University who shared their ideas, experiences, and encouragement. Colleagues Wilson Wong, David Yates, Doug Robertson, Mary Ann Robbert, Lynn Senne, Jim Linderman, Kay Green, and Peggy Beranek have all offered contributions that have substantially improved the book over four editions. A special thank you, David, for your helpful technical discussions and reviews of the new data communications material in the fourth edition, and to you, Wilson, for serving as a technical reviewer for both the third and fourth editions, providing many comments, rewrites, and suggestions for clarification, and for creating many of the ancillary materials for the book. A special thank you also to Linda Cotroneo, our Bentley CIS Department Academic Administrative Assistant, who for the past twenty years has generously offered more empathy and support than any person deserves. From the bottom of my heart, Linda, thank you!

Thanks to the editors, production people, and marketing personnel at John Wiley & Sons and the editors and production people at Laserwords. You hassled me when I needed to be hassled and left me alone when I needed to be left alone. Incredible intuition, that! I consider myself fortunate to have worked with such wonderful people. Particular thanks to Beth Lang Golub, Trish McFadden, and Kate Boilard for your ongoing efforts to make this book perfect, even though we all know it's impossible!

I would like to acknowledge the reviewers who gave of their time and effort to assure that this book was as good as it could be: Dr. Stu Westin, The University of Rhode Island; Alan Pinck, Algonquin College; Mark Jacobi, Programme Director for Undergrad Computing at Sheffield Hallam University; Dr. Dave Protheroe, South Bank University, London; Julius Ilinskas, Kaunas University of Technology; Anthony Richardson, United States Army Informations Systems Engineering Command; Renee A. Weather, Old Dominion University; Jack Claff, Southern Cross University; Jan L. Harrington, Marist College; YoungJoon Byun, California State University, Monterey Bay; William Myers, Belmont Abbey College; Barbara T. Grabowski, Benedictine College; G.E. Strouse, York College of Pennsylvania; Martin J. Doyle, Temple University; Richard Socash, Metropolitan State College of Denver; Fred Cathers, Franklin University. Your comments, suggestions, and constructive criticism have made a real difference in the quality of this book. Thank you.

Many colleagues offered corrections to previous editions that have had important impact on the quality of the current edition. To each and everyone, your assistance in eliminating errors has been much appreciated. Among these, I especially wish to acknowledge David Feinstein and his crew at the University of South Alabama, Gordon Grimsey of AIT in Auckland, New Zealand, and Stu Westin of University of Rhode Island for efforts well above and beyond the call of duty. Stu has also generously made his excellent Little Man Simulator publicly available, for which I am truly grateful. Thanks for everything, Stu.

Numerous students, too many to name you all, also offered corrections, made suggestions, and provided ideas. Please accept my deepest appreciation and thanks.

I hope that I have not forgotten anyone. If I have, I apologize.

I have strived to make this book as technically accurate as is humanly possible. Nonetheless, I know that errors have a way of creeping in when one least expects them. I would greatly appreciate hearing from readers who find errors that need correction. Your comments and suggestions about the book are also welcome.

ABOUT THE AUTHOR

Dr. Irv Englander has been involved in many different aspects of the computing field for more than forty-five years. He has designed logic circuits, developed integrated circuits, developed computer architectures, designed computer-controlled systems, designed operating systems, developed application software, created the initial system design for a large water monitoring system, performed software auditing and verification of critical control software, and developed and specified hardware components and application software as a consultant for business systems large and small.

As an educator he has contributed papers and given workshops on end-user computing, e-commerce, and on computer architecture education in the IS curriculum. He was an invited contributor and reviewer for the IS-97 and IS-2002 information systems curricula, and continues to take an interest in the technical infrastructure components of the IS/IT curriculum. He is actively involved in the application of new technology to information systems.

Dr. Englander has a Ph.D. from MIT in Computer Science. His doctoral thesis was based on the design of a large image processing software laboratory. At MIT he won the Supervised Investors Award for outstanding teaching. He holds the rank of Professor of Computer Information Systems at Bentley University, where he has taught full-time for thirty years.

PART ONE

A computer-based information system is made up of a number of different elements:

- The *data* element. Data is the fundamental representation of facts and observations. Data is processed by a computer system to provide the information that is the very reason for the computer's existence. As you will see, data can take on a number of different forms.
- The *hardware* element. Computer hardware processes the data by executing instructions, storing data, and moving data and information between the various input and output devices that make the system and the information accessible to the users.
- The *software* element. Software consists of the system and application programs that define the instructions that are executed by the hardware. The software determines the work to be performed and controls operation of the system.
- The *communication* element. Modern computer information systems depend on the ability to share processing operations and data among different computers and users, located both locally and remotely. Data communication provides this capability.

The combination of hardware, software, communication, and data make up the *architecture* of a computer system. The architecture of computer systems is remarkably similar whether the system is a playstation, a personal computer that sits on your lap while you work, an embedded computer that controls the functions in your cell phone or in your car, or a large mainframe system that is never actually seen by the hundreds of users who access it every day.

Even more remarkably, the basic architecture of computer systems has changed surprisingly little over the last fifty-five years. The latest IBM mainframe computer executes essentially the same instruction set as the mainframe computer of 1965. The basic communication techniques used in today's systems were developed in the 1970s. As new as it might seem, the Internet will celebrate its fortieth anniversary in 2010. All of

AN OVERVIEW OF COMPUTER SYSTEMS

this is surprising considering the growth of computing, the rapid change of technology, and the increased performance, functionality, and ease of use of today's systems. This makes the study of computer architecture extremely valuable as a foundation upon which to understand new developments in computing as they occur.

Computer system architecture is the subject of this textbook. Each element of the system is addressed in its own section of the text, always with an eye to the system as a whole.

Part I is made up of two chapters that presents an overview of systems, and of the computer system in particular.

Chapter 1 addresses a number of issues, including

- The ways in which a knowledge of computer architecture enhances our abilities as computer users and professionals
- A simplified view of typical computer system architectures
- The basic components that make up a computer system
- The fundamental operations that are performed by computer systems

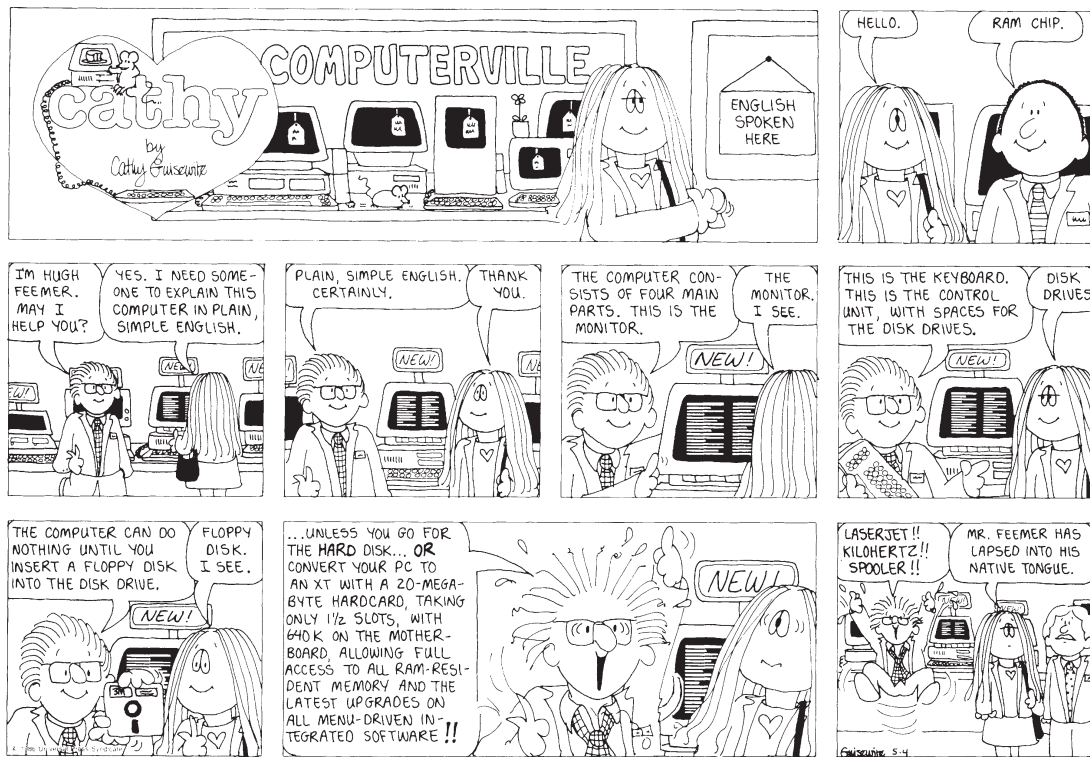
Chapter 1 concludes with a brief architectural history of the computer.

An encompassing theme throughout this text is that of systems and system architecture. The words “system” and “architecture” appear throughout this book: we talk about information systems, computer systems, operating systems, file systems, software architecture, I/O architecture, network architecture and more. You will probably take a course in *System Analysis and Design* sometime in your college career.

Although most people have an instinctive understanding of what a system is, it is more important for us as system professionals to understand the concepts of systems and system architecture at a deeper level than the average person. Chapter 2 offers careful definitions and examples of the concept of systems and system architecture, both generally and in the specific context of the computer systems that are the focus of this book.

CHAPTER I

COMPUTERS AND SYSTEMS



CATHY © 1986 Cathy Guisewite. Reprinted with permission of UNIVERSAL PRESS SYNDICATE. All rights reserved.

1.0 INTRODUCTION

It is nearly impossible today to escape the immediate reach of computers and computer-based systems. There is probably a cell phone in your pocket or on your desk and, perhaps, an iPod as well. For many of you, your laptop or desktop computer is sitting nearby as you read this paragraph. And that's not all. Your car probably has several embedded computers controlling various functions. Even your microwave oven and the machine that launders your clothes depend on computers to function properly. As you are probably aware, most of these machines can talk to each other, using the Internet or some other networking technology.

Indeed, the jargon of computers has become a part of common daily language. You can open a newspaper and find references to expressions such as “2 GB DDRAM” or “WXGA LCD display” or “2 MB level 2 cache” or “Wi-Fi” in articles and advertisements. (In a way, it's scary!) The ad in Figure 1.1, taken from a Sunday newspaper flier, is typical of recent computer ads.

You'll notice that this computer features a “Core 2 Duo Processor” CPU, 2 GB of DDR2 SDRAM memory, a 16× DVD ±RW Drive, and a 160 GB SATA hard drive. It also contains a 256 MB PCI Express graphics card among other things. But how good a system is this? Are these features important to the user? Is this the right combination of features that you need in your computer to have the computer perform the work that you wish to get done? Are there features missing that we need? Is a Core 2 Duo processor the best CPU for us? Perhaps we are paying too much for the performance that we need. Or maybe we need more. And what does “Core 2 Duo” mean, anyway? What I/O ports might you need to assure a satisfy long-term investment of computers for your organization? Is a 16× DVD ±RW drive adequate for your work? What if you have to burn a lot of disks? What other information about this system would allow you to make a more informed decision? (For example: Hey—where's the networking capability?)

Some of the expressions used in these articles and ads are obvious from the context. Other references may be more obscure. Presumably, everyone today knows what a “display monitor” is. But how many people know the meaning and significance of the terms “cache memory” or “multitasking” or “PCI Express bus”? Yet all these expressions have appeared recently in daily newspaper advertisements with the assumption that people would understand the meaning of the ad.

Despite the jargon, there is obviously no need to understand the inner workings of most modern computer-based systems to operate them adequately. Indeed, in many cases the presence of the computer is hidden from us, or **embedded**, and its operation invisible to us as users.

Even as experienced users, we can run standard software packages on a personal computer without understanding exactly how they work; we can program a computer in a high-level language without understanding the details of how the machine

FIGURE 1.1

A Typical Computer Ad



TECHNOLOGY CHOICES TO BUILD YOUR BUSINESS



New! Vostre 400 Mini Tower

Only Drex delivers this combination of performance and expandability in a system designed exclusively for Small Businesses—the Vostre 400.

- Intel Core 2 Duo Processor
- Genuine Home Basic Operating System
- 2GB DDR2 SDRAM
- 160GB SATA Hard Drive
- 16x DVD+/-RW Drive
- 256MB PCI Express Graphics Card
- 1-Yr Limited Warranty, Next Business Day On-Site Service, and Hardware Warranty Support
- 20" Widescreen Flat Panel Display

ON SALE NOW!

NEW! Lower-priced upgrades:
Upgrade to 4GB Memory, 250GB Hard Drive,
and 22" Widescreen Flat Panel Display for
only \$90!!

executes the individual instructions; we can design and implement Web pages without understanding how the Web browser gets its pages from a Web server or how the Web server creates those pages; we can purchase a computer system from a salesperson without understanding the specifications of the system.

And yet, there is something missing. Perhaps the package doesn't do exactly what we want, and we don't understand the machine well enough to risk fooling around with the package's options. Perhaps if we understood the system better we might have written and configured the program to be faster and more efficient. Perhaps we could create Web pages that load faster and work better. Perhaps the salesperson did not sell us the optimum system for our job. Or perhaps it's nothing more than a sense of excitement that's missing. But that's important, too!

You are reading this book because you are a student studying to become a computer professional, or maybe you are simply a user wanting a deeper understanding of what the computer is all about. In either case, you will most likely be interacting with computer systems for the rest of your life. It's nice (as well as useful) to know something about the tools of the trade. More important, understanding the computer system's operations has an immediate benefit: it will allow you to use the machine more effectively.

As a user, you will be aware of the capabilities, strengths, and limitations of the computer system. You will have a better understanding of the commands that you use. You will understand what is taking place during the operation of the programs that you use. You will be able to make informed decisions about your computer equipment and application programs. You will understand more clearly what an operating system is, and how to use it effectively and to your advantage. You will know when it is preferable to do a job manually, and when the computer should be used. You will understand the most efficient way to “go online,” and what benefits might be gained from a home network. You will improve your ability to communicate with system analysts, programmers, and other computer specialists.

As a programmer, it will allow you to write better programs. You will be able to use the characteristics of the machine to make your programs operate more effectively. For example, choosing the appropriate data type for a variable can result in significantly faster performance. Soon you will know why this is so, and how to make the appropriate choices.

Computers can perform integer calculations incorrectly if the integers exceed a certain size, but they do not necessarily warn the user of the error. You will learn how this can occur, and what can be done to assure that your programs generate correct results.

You will discover that some computers will process nested loops much more quickly if the index variables are reversed. A rather surprising idea, perhaps, and you’ll understand why this is true.

You will understand why programs written in a compiled language like C++ usually run much faster than those written in interpreted program languages like BASIC or scripting languages like JavaScript.

As a systems architect or system analyst, you will be responsible for the design and implementation of systems that meet an organization’s information technology (IT) needs, recognizing that the differences in the cost and capabilities of the components that you select may have significant impact on the organization. With the knowledge gained here you will be in a better position to determine and justify the set of computer system components and the system architecture that are appropriate for a particular job and to determine the tradeoffs with other possible system architectures.

You’ll be able to assist management in making intelligent decisions about system strategy: should the company adopt a large mainframe/virtual machine system approach for its Web servers, or would a system consisting of a network of off-the-shelf blade servers provide better performance at lower cost? You’ll be better prepared to analyze the best way to provide appropriate facilities to meet the needs of your users. In an era of fast-changing technology, you’ll be more able to differentiate between simple technological obsolescence that does not affect the organization’s requirements significantly and major advances that suggest a real need to replace older equipment.

When selecting computers, you would like to purchase the computer that best meets the needs of the organization’s applications and the users. You must be able to read and understand the technical specifications in order to compare different alternatives and to match the system to the users’ needs. This book will teach you what you need to know to specify and purchase a system intelligently. You’ll know the differences between various CPU technologies and the advantages and disadvantages of each. You will learn what peripheral hardware is appropriate for your organization’s files and the trade-offs between different file system formats, what is required to build an intranet, and what the speed and

size limitations of a particular system are. You'll be able to compare the features of Windows and Linux knowledgeably and decide which ones are important to you. You'll be able to apply your basic understanding of computers to new technologies such as virtual machines as they appear. You'll learn to understand the jargon used by computer salespeople and judge the validity of their sales claims.

As a system administrator or manager, your job is to maximize the availability and efficiency of your systems. You will need to understand the reports generated by your systems and be able to use the information in those reports to make changes to the systems that will optimize system performance. You will need to know when additional resources are required, and be able to specify appropriate choices. You will need to specify and configure operating system parameters, set up file systems, manage system and user PC upgrades in a fast-changing environment, reconfigure networks, provide and ensure the robustness of system security, and perform many other system management tasks. The configuration of large systems can be very challenging. This text will give you an understanding of operating system tools that is essential to the effective management of systems.

As a Web services designer, you will be able to make intelligent decisions to optimize your Web system configurations, page designs, data formatting and scripting language choices, and operating systems to optimize customer accessibility to your Web services.

In brief, when you complete this book, you will understand what computer hardware and software are and how programs and data interact with the computer system. You will understand the computer hardware, software, and communication components that are required to make up a computer system and what the role of each component in the system is.

You will have a better understanding of what is happening inside the computer when you interact with the computer as a user. You will be able to write programs that are more efficient. You will be able to understand the function of the different components of the computer system and to specify the computer system you need in a meaningful way. You will understand the options that you have as a system administrator or Web services designer.

In an era in which technology changes extremely rapidly, the architecture of the computer system rests on a solid foundation that has changed only slightly and gradually over the last sixty years. Understanding the foundations of computer system architecture makes it possible to flow comfortably with technological change and to understand changes in the context of the improvements that they make and the needs that they meet. In fact, interviews with former students and with IT executives and other IT professionals clearly indicate that a deep understanding of the basic concepts presented here is fundamental to long-term survival and growth in the field of information technology and IT management.

This type of understanding is at the very foundation of being a competent and successful system analyst, system architect, system administrator, or programmer. It may not be necessary to understand the workings of an automobile engine in order to drive a car, but you can bet that a top-notch race car driver knows his or her engine thoroughly and can use it to win races. Like the professional race car driver, it is our intention to help you to use your computer engine effectively to succeed in using your computer in a winning way. The typical end user might not care about how their computer system works, but you do.

... These are the goals of this book. So let's get started!

1.1 THE STARTING POINT

Before we begin our detailed study of the architecture of computer systems, let us briefly review some of the fundamental principles and requirements that guide computer system design and operation.

In a simple scenario, you use your laptop or desktop personal computer to word process a document. You probably use a mouse to move around the document and to control the features of the word processor software application, and you use the keyboard to enter and modify the document text data. The word processor application program, together with your document, appears on a screen. Ultimately, you might print the document on a printer. You store the document on a disk or some other storage device.

The fundamentals of a typical computer system are readily exposed in this simple example. Your mouse movements and clicks and your keyboard entry represent input to the system. The computer processes the input and provides output to the screen, and, perhaps, to a printer. The computer system also provides a storage medium of some sort, usually a hard disk, to store the text for future access. In simplest terms, your computer receives input from you, processes it, and outputs results to the screen. Your input takes the form of commands and data. The commands tell the computer how to process the data.

Now consider a second, slightly more complex example. Your task in this example is to access a Web page on the Internet. Again, your input to the computer is via mouse and keyboard. When you type the Web page URL, however, your computer sends a message to another computer that contains Web server software. That computer, in turn, sends a Web page file that is interpreted by the browser on your computer and presented on your screen. You are probably already aware that HyperText Transfer Protocol (HTTP) is used as a standard for Web message exchanges.

The elements of this example differ only slightly from the first example. Your command inputs tell a Web browser software application on your computer what processing is to take place; in this case, your desire to access a Web page. The output from your computer is a message to a Web server on the remote computer requesting the data that represents the Web page. Your computer receives the data as input from the network; the Web browser processes the data and presents the Web page output on the screen. Figure 1.2 illustrates the layout for this example.

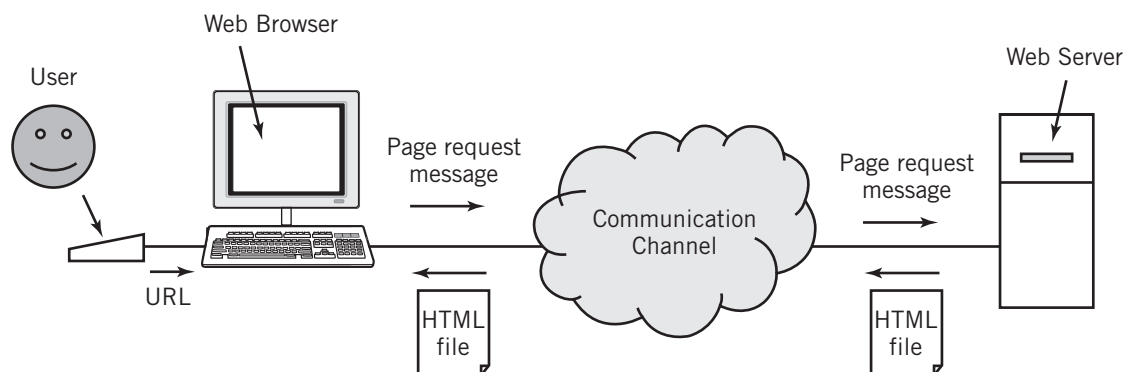
The major differences between this and the first example are the source of the input data and the fact that network connectivity is required between the two computers. Instead of the keyboard, the input data to be processed by your Web browser comes from a communication channel. (Note that the exact nature of the channel is not important for this discussion.) In both cases, your computer receives data input to process, and control input that determines how the data is to be processed, performs the processing, and provides output.

These two examples contain all of the key elements found in any IT system, large or small.

- An IT system consists of one or more computer systems; multiple computer systems are connected together using some type of network interconnectivity. As a matter of necessity, network interfaces must conform to standard agreements, known as **protocols**, for messages to be understood by both computers during a message exchange between a pair of computers. The network itself can take on a

FIGURE 1.2

Typical Web Browser Application Use



variety of forms, provided that the interface requirements are met, and are determined by such characteristics as performance, convenience, and cost.

- The work performed by an individual computer system within the IT system can be characterized by input, processing, and output. This characterization is often represented by the **Input-Process-Output (IPO) model** shown in Figure 1.3. Storage is also represented within this model. Alternatively, storage can be interpreted as output to be saved for use as future input. Storage is also used to hold the software programs that determine the processing operations to be performed. The ability to store programs and data on a temporary, short-term, or long-term basis is fundamental to the system. In Chapter 2, Section 2.2, we will show that all IT systems can ultimately be characterized by the same basic IPO model at all levels, from a single computer to a complex aggregation of computers, although the complexity of large systems may obscure the model and make it more difficult to determine the actual inputs, outputs, and processing

FIGURE 1.3

A Computer Process

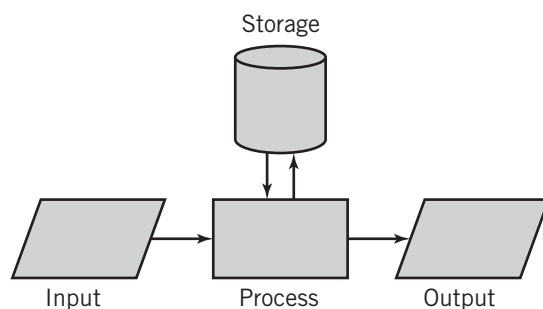
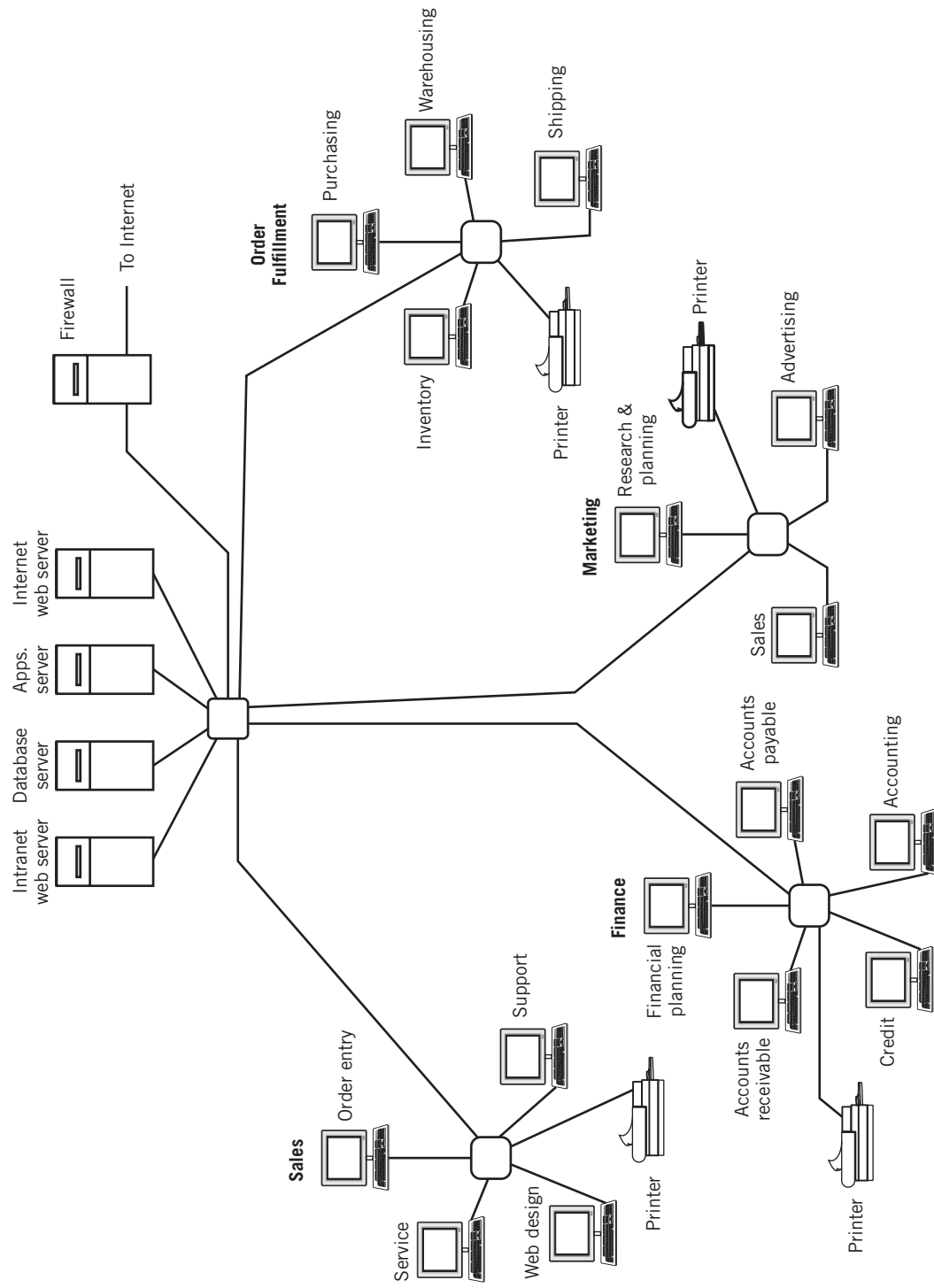


FIGURE 1.4

A simplified IT Computer System Layout



operations. The IPO model provides an important basic tool for system analysis and design practices.

- The components of an individual computer system consist of processing hardware, input devices, output devices, storage devices, application software, and operating system software. The task of the operating system software is to provide overall control of the individual system, including management of input, output, and file storage functions. The medium of exchange, both with users and between computers within a larger system, is data. (Note that the messages between computers in the second example are a form of data.) Figure 1.4 is a simple illustration of computer systems embedded in a larger IT system.

Figure 1.5 summarizes the basic operations that are performed during computer processing. These operations, in turn, can be reduced to the primitive operations that are also familiar to you from your understanding of programming languages. The primitive processing operations common to high-level programming languages are shown in Figure 1.6.

1.2 COMPONENTS OF THE COMPUTER SYSTEM

As noted in the previous section, there are three components required for the implementation of a computerized input-process-output model:

1. The computer hardware, which provides the physical mechanisms to input and output data, to manipulate and process data, and to electronically control the various input, output, and storage components.
2. The software, both application and system, which provides instructions that tell the hardware exactly what tasks are to be performed and in what order.
3. The data that is being manipulated and processed. This data may be numeric, it may be alphanumeric, it may be graphic, or it may take some other form, but in all cases it must be representable in a form that the computer can manipulate.

In modern systems, input entry, output display, and storage of the data and software used for processing often take place at a location different from the computer where the

FIGURE 1.5

Basic Computer Operations

- Input/output
- Basic arithmetic and logical calculations
- Data transformation or translation (e.g., program compilation, foreign language translation, file updating)
- Data sorting
- Searching for data matches
- Data storage and retrieval
- Data movement (e.g., movement of text or file data to make room for insertion of additional data)

FIGURE 1.6

Basic High-Level Language Constructs

- Input/output (including file storage and retrieval)
- Arithmetic and logical assignment statements
- True/false decision branching (IF-THEN-ELSE or IF-GOTO)
- Loops and/or unconditional branching (WHILE-DO, REPEAT-UNTIL, FOR, GOTO)

actual processing occurs. In many installations, actual processing is distributed among computer systems, with particular results passed to the individual systems that require them. Therefore, we must also consider a fourth component:

4. The communication component, which consists of hardware and software that transport programs and data between interconnected computer systems.

The hardware and system software components make up the architecture of the computer system. The communication component connects individual computer systems together. The data component, and also the application software, while fundamental to the operation of the computer system, are supplied to the computer system by the user, rather than being a part of the architecture of the computer system itself. (It is useful to note, however, that application software and data structure *are* often considered as part of the overall system architecture when one considers the architecture from the perspective of the organization. We explore this issue briefly in Chapter 2. Note, however, that the focus of this book is primarily on *computer* system architecture, rather than on *organizational* system architecture.)

The Hardware Component

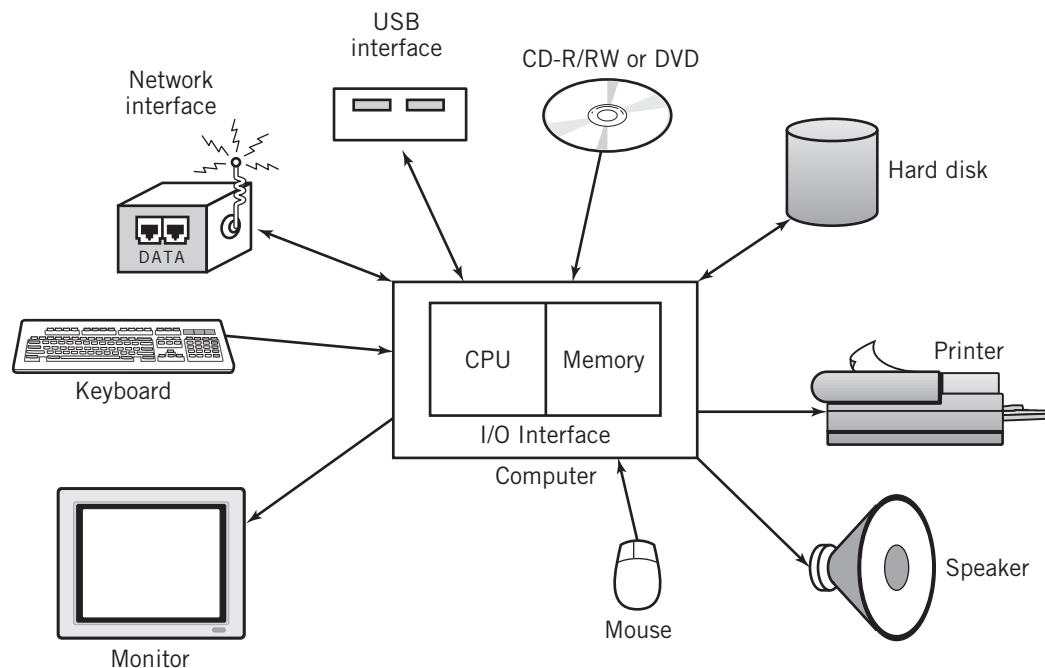
The most visible part of the computer system is obviously the hardware that makes up the system. Consider the computer system upon which you write and execute your programs. You use a keyboard and mouse to provide **input** of your program text and data, as well as for commands to the computer. A display screen is commonly used to observe **output**. A printer is frequently available as an alternative output to the screen. These are all physical components.

Calculations and other operations in your program are performed by a **central processing unit (CPU)** inside the computer. **Memory** is provided to hold your programs and data while processing is taking place. Other input and output devices, such as a disk and SD plug-in cards, are used to provide long-term storage of your program and data files. Data and programs are transferred between the various input/output devices and memory for the CPU to use.

The CPU, memory, and all the input, output, and storage devices form the **hardware** part of a computer system. The hardware forms the tangible part of the system. It is physical—you can touch it, which is what the word “tangible” means. A typical hardware

FIGURE 1.7

A Typical Personal Computer System



block diagram for a computer is seen in Figure 1.7. In addition to the input and output devices shown in this diagram, Figure 1.8 lists some other input and output devices that are frequently seen as part of computer systems. The diagram in Figure 1.7 actually applies equally well to large mainframe computers, small personal computers, and even devices with computers embedded in them, such as PDAs, iPods, GPSs, and cell phones. Large and small computers differ primarily in speed, capacity, and the selection of peripheral devices provided. The basic hardware components and design are very similar.

Conceptually, the CPU itself is often viewed as a composition of three primary subunits:

1. The **arithmetic/logic unit (ALU)** where arithmetic and Boolean logical calculations are performed.
2. The **control unit (CU)**, which controls the processing of instructions and the movement of internal CPU data from one part of the CPU to another.
3. The **interface unit**, which moves program instructions and data between the CPU and other hardware components.

(In modern CPUs, the actual implementation is usually modified somewhat to achieve higher performance, although the basic concepts are carefully preserved. More about that later, in Chapter 8.)

FIGURE 1.8

Other Common Input/Output Devices

- Bar Code Scanners
- Optical Character Recognition Scanners
- Image Scanners
- RFID Readers
- Video and Audio Capture Devices
- TV Tuners
- Video Cameras
- SD, SmartCard, etc. Card Readers
- Fingerprint and Face Readers
- Touch Screens
- Graphics Tablets
- X-Y Plotters

The interface unit interconnects the CPU with memory and also with the various I/O (input/output) modules. It can also be used to connect multiple CPUs together. In many computer systems, a bus interconnects the CPU, memory, and all of the I/O components. A **bus** is simply a bundle of wires that carry signals and power between different components. In other systems, the I/O modules are connected to the CPU through one or more separate processors known as **channels**.

The main memory, often known as primary storage, working storage, or **RAM** (for **random access memory**), holds programs and data for access by the CPU. **Primary storage** is made up of a large number of cells, each numbered and individually addressable. Each cell holds a single binary number representing part of a data value or part of an instruction. The smallest addressable size of the cell in most current computers is 8 bits, known as a **byte** of memory. Eight bits of memory can only hold 256 different patterns, so neighboring cells in memory are nearly always combined

to form groupings with a larger number of bits. In many systems, for example, 4 bytes of memory combine to form a 32-bit **word**. Modern computers address memory at least 4 bytes (a “32-bit” computer) or 8 bytes (a “64-bit” computer) at a time to take advantage of larger instruction and data groupings.

The amount of primary storage determines the maximum number of instructions and data words that can be loaded into memory from a peripheral device at one time. For example, a computer with 2 gigabytes (GB), actually 2,147,483,648 bytes¹, of memory would not be able to execute a program that requires 2.7 GB for its instructions and data unless some means is provided within the computer to load the program in sections as each section of the program is needed.

The amount of primary storage provided in a typical computer has increased rapidly as computer technology improves. Whereas 64 *kilobytes* (KB) of memory was considered a large amount in 1980, even the least expensive personal computers today usually have 2 *gigabytes* (GB) of memory or more. Large computers may provide many gigabytes of primary storage. There are programs on the market that require hundreds of megabytes (MB) of memory to execute. The inexpensive availability of increased amounts of memory have allowed the design of very sophisticated programs that would not have been possible just a few years ago.

The same is true for secondary storage. Even small personal computers provide hard disks with storage measured in tens or hundreds of gigabytes. The storage of images and video, in particular, requires tremendous amounts of storage capacity. It is not uncommon to see arrays of hard disks, even on some personal computers, providing trillions of bytes (specified as *terabytes*) of long-term storage.

¹1 Kilobyte actually equals 1024 bytes. Thus, 1 MB = 1024 × 1024 = 1,048,576 bytes × 2048 = 2,147,483,648 bytes.

The instructions that form a particular program are stored within the primary storage, then brought into the central processing unit and executed. Conceptually, instructions are brought in and executed one at a time, although modern systems overlap the execution of instructions to some extent. Instructions must be in primary storage in order to be executed. The control unit interprets each instruction and determines the appropriate course of action.

Each instruction is designed to perform a simple task. Instructions exist to perform basic arithmetic, to move data from one place in the computer to another, to perform I/O, and to accomplish many other tasks. The computer's power comes from the ability to execute these simple instructions at extremely high speeds, measured in millions or billions or trillions of instructions executed per second. As you are already aware, it is necessary to translate high-level language programs into the language of the machine for execution of the program to take place. It may require tens or even hundreds of individual machine instructions to form the machine language equivalent of a single high-level language statement. Program instructions are normally executed sequentially, unless an instruction itself tells the computer to change the order of processing. The instruction set used with a particular CPU is part of the design of the CPU and cannot normally be executed on a different type of CPU unless the different CPU was designed to be instruction set compatible. However, as you shall see, most instruction sets perform similar types of operations. As a result, it is possible to write programs that will emulate the instruction set from one computer on a computer with a different instruction set, although a program written for the original machine may execute slowly on the machine with the emulator.

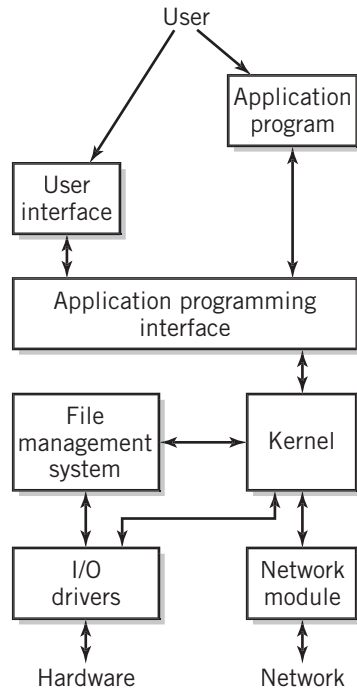
The data that is manipulated by these instructions is also stored in memory while being processed. The idea that the program instructions and data are both stored in memory while being processed is known as the **stored program concept**. This important concept is attributed primarily to John von Neumann, a famous computer scientist. It forms the basis for the computer architecture that is standard to nearly every existing computer.

The Software Component

In addition to the hardware requirement, your computer system also requires **software**. Software consists of the programs that tell the computer what to do. To do useful work, your system must execute instructions from some program.

There are two major categories of software: system software and application software. System software helps you to manage your files, to load and execute programs, and to accept your commands from the mouse and keyboard. The system software programs that manage the computer are collectively known as an **operating system**, and differ from the application programs, such as Microsoft Word, or Firefox, or the programs that you write, that you normally run to get your work done. Windows and Linux are the best known examples of an operating system. Others include Unix, Mac OS X, Sun Solaris, and IBM z/OS.

The operating system is an essential part of the computer system. Like the hardware, it is made up of many components. A simplified representation of an operating system is shown in Figure 1.9. The most obvious element is the user interface that allows

FIGURE 1.9Simplified Operating System
Block Diagram

you to execute programs, enter commands, and manipulate files. The user interface accepts input from a keyboard and, in most modern systems, a mouse, touch screen, or other pointing device. The user interface also does output presentation on the display. On some systems, the output display might be simple text, but more likely the display includes a graphical user interface with a windowing system, and various gadgets for manipulating the windows.

The operating system's **application program interface (API)**, acts as an interface for application programs and utilities to access the internal services provided by the operating system. These include file services, I/O services, data communication services, user interface services, program execution services, and more.²

Many of the internal services are provided by the **kernel** module, which contains the most important operating system processing functions. The remaining services are provided by other modules that are controlled by the kernel. The kernel manages memory by locating and allocating space to programs that need it, schedules time for each application to execute, provides communication between programs that are being executed, manages and arranges services and resources that are provided by other modules, and provides security.

The file management system allocates and manages secondary storage space and translates file requests from their name-based form into specific I/O requests. The actual storage and retrieval of the files is performed by the I/O drivers that comprise the I/O component. Each I/O driver controls one or more hardware devices of similar type.

The network module controls interactions between the computer system and the network(s) to which it is attached.

The operating system software has nearly always been stored on a hard disk, but on some smaller systems, especially lightweight laptops and embedded systems such as cell phones and iPods, a solid state disk or SD card may be used instead. On a few systems the operating system is actually provided as a network service when the system is turned on. In either case, the bootstrap or IPL (Initial Program Load) program in the operating system is stored within the computer using a type of memory known as **ROM**, or **read-only memory**. The bootstrap program provides the tools to test the system and to load the remainder of the operating system from the disk or network. Although the physical medium where the software is stored can be touched, the software itself is considered intangible.

Together, the hardware and system software provide a working computer system environment. Application software, communication support, and user data complete the picture.

²The same term (API) is also sometimes used to describe the services provided by one application to another. For example, Amazon and Google are among many companies whose application software provides API tools to allow users to extend the functionality of the original software.

The Communication Component

Very few modern computers or computer-based devices (which includes cell phones, iPods, and automobile computers, to name just a few possibilities) operate independently. Instead, they are tied to other computers directly, by modem, or through a network connection of some sort. The computers may be located physically close to each other, or they may be separated, even by thousands of miles. To work together, computers must have means to communicate with each other. The communication component requires both hardware and software to achieve this goal. Additional hardware components physically connect computers together into multiprocessing systems, or clusters, or networks, or, via telephone, satellite, or microwave, to computers at other remote locations. A **communication channel** provides the connection between computers. The channel may be a wire cable, a fiber-optic cable, a telephone line, or a wireless technology, such as infrared light, cellular phone, or radio-based technology such as Wi-Fi or Bluetooth. Special I/O hardware, consisting of a device such as a **modem** or **network interface card (NIC)** within the computer, serves as an interface between the computer and the communication channel. There may be additional hardware within the channel itself.

The communication component also requires additional software within the operating system of each computer to make it possible for each computer to understand what the other computers that they are connected with are saying. This software establishes the connections, controls the flow of data, and directs the data to the proper applications for use.

The Computer System

Our general description of the computer is valid for all general-purpose computer systems, and also for most devices with computers embedded in them, regardless of brand name or size. In more general terms, every computer system consists of a CPU, or central processing unit, where all the processing takes place; memory to hold the programs and data while they are being processed; and some form of input and output, usually one or more keyboards and flat-screen display devices plus one or more forms of long-term storage, usually disks, CDs or DVDs, and USB or SD plug-in memory. Most modern computer systems provide more than one CPU (or “core”) within the computer system. A single CPU can process only one instruction at a time; the use of multiple CPUs can increase processing speed by allowing instructions that do not affect each other to be executed in parallel.

The validity of our general description is true regardless of how complex or simple the computer system may seem.

As a specific example, the large z10 EC IBM mainframe computer shown in Figure 1.10 can provide complex Web services to thousands of users at a time. IBM mainframes can have dozens of CPUs working together, with up to 1.52 terabytes (TB) of primary storage. They are capable of executing instructions at a rate of tens of billions of instructions per second! The powerful z/OS operating system can keep track of hundreds or thousands of simultaneous users and divides the time among them to satisfy their differing requirements. Even in its smallest configuration, the z10 EC Model S64 system, which is the largest current model at this writing, provides at least 16 GB of memory and processes instructions at

FIGURE 1.10

IBM System z10 EC Mainframe Computer



Courtesy of International Business Machines Corporation.
Unauthorized use not permitted.

the rate of several billion instructions per second. In addition to the CPU, there are many large I/O devices—including tape drives and high speed printers—and disks that store many billions or trillions of characters. The computer alone weighs over 5000 pounds/2200 kilograms!

In contrast, the laptop PC shown in Figure 1.11 is designed for personal use. Everything is self-contained in one package. This system only has 2 GB of primary RAM storage and operates at a small fraction of the speed of the z10 EC. A hard drive is one of many storage options. The entire system, complete with display screen, built-in webcam, multiple network connections, and battery, weighs about three pounds (1.4 kilograms, if you prefer).

Although these two systems seem very different, the difference is actually one of magnitude, not of concept. The large system operates much faster, can support much more memory, and handles more input and output much faster. It has operating system software that allows many users to share this larger resource. Nonetheless, the fundamental system architecture is remarkably similar in both cases. Even the actual processing performed by the CPU is similar.

In fact, today's CPU operates in the same fundamental way as its CPU counterpart of fifty-five years ago, even though the construction is very different. Since computers all operate so similarly, regardless of size or type, it is not difficult today to transfer data between these different systems, allowing each system to do part of the processing for higher overall efficiency. This concept is known as **distributed computing**. The fact that different types of computers can work together, share files, and communicate successfully is known as **open computing**. Communication technology fulfills the requirements that make open and distributed computing possible.

Computers are sometimes divided into categories: mainframe computers, minicomputers, workstations, and personal computers, but these categories are less significant than they once were. The capability of today's personal computer far exceeds the capabilities of a mainframe computer of just a few years ago.

FIGURE 1.11

A Laptop Computer



© 2007 Hewlett-Packard Company.

The Sun Ultra 40 computer is an example of a workstation that is frequently used as though it were a minicomputer, or even a small mainframe. Rather than attempting to categorize a particular computer, it is usually more productive to describe its capabilities in comparison to other systems being discussed or considered.

1.3 THE CONCEPT OF VIRTUALIZATION

The word “**virtual**” appears frequently throughout the computer literature in many different contexts. To name a few applications of the word that appear in this text, there are *virtual* computers, a Java *Virtual Machine*, *virtual* memory, and *virtual* networks. Sometimes, a synonymous word, *logical*, is used instead: in networking we have *logical* connections. *Virtual* storage consists of a relationship between *logical* memory and physical memory.

It is not important at this point that you understand any of the specific concepts mentioned above. (In fact, we realize that you probably don’t.) Since the words *virtual* and *logical* represent a number of important concepts, however, we introduce them here.

In optics, a virtual image is the reflection that you see when you stand in front of a regular mirror. (See, for example, the cartoon at the beginning of Chapter 18.) You know that the image isn’t real. For one thing, it’s behind the wall that the mirror is mounted on. For another, you can’t touch it. In early, time-shared computing, a large central computer commonly supplied computing services to users at terminals located remotely from the computer. In a sense, it seemed as though the user had access to a computer that was all her own. Starting in the early 1970s, IBM offered the VM (*virtual machine*) operating system to support this concept.

The *American Heritage Dictionary* offers two applicable definitions of *virtual* that together describe the usage of the word in modern computing:

- Existing or resulting in essence or effect though not in actual fact, form, or name
- Created, simulated, or carried on by means of a computer or computer network

Wikipedia defines *virtualization* as “a broad term that refers to the abstraction of computer resources”.

In essence, *virtual* and *logical* are used to refer to something that appears as though it is something different. Thus, the *Java Virtual Machine (JVM)* uses software to simulate a real computer that works well with the Java programming language, even though the actual computer executes a different set of instructions than the JVM. A *logical connection* in networking offers the appearance of a direct communication link for passing data between two computers, even though the actual connection might involve a complex series of interconnections involving many computers and other devices and a variety of software to make it all look simple. The *virtualization* of a computer allows a single computer to appear as a multiplicity of computers, each with its own operating system and hardware resources.

1.4 PROTOCOLS AND STANDARDS

Standards and protocols are of great importance in computer systems. **Standards** are agreements among interested parties, often manufacturers, to assure that various system components will work together interchangeably. Standards make it possible to build

a computer with components from different suppliers, for example, knowing that a graphics card will plug properly into a connector on a motherboard and that the image representations will be consistent between the connector, the CPU, memory, and the display monitor.

Standards apply to every aspect of computing: hardware, software, data, and communications, the voltage of a power supply, the physical spacing of pins on a connector, the format of a file, the pulses generated by a mouse. Computer language standards, such as Java and SQL, allow programs written on one type of computer to execute properly and consistently on another, and also make it possible for programmers to work together to create and maintain programs.

Similarly, data format and data presentation standards, such as the GIF and JPEG image format standard, the Unicode text format standard, and the HTML and XML Web presentation standards allow different systems to manipulate and display data in a consistent manner.

Standards can arise in many different ways. Many standards occur naturally: a proprietary data format (PDF) belonging to a single vendor becomes a *de facto* standard due to the popularity of the product. The PDF print description language is an example of such a standard. The format was designed by Adobe Corporation to provide a way of communicating high-quality printed output between computers and printers. Other standards are created because of a perceived need in an area where no standard exists.

Often a committee will form to investigate the requirements and create the standard. The MPEG-2 and MPEG-4 standards, which establish the means for the transmission and processing of digital video images, occurred in this way. The committee that designed the standard, made up primarily of motion picture engineers and video researchers, continues to develop the standard as improved techniques evolve. The JPEG photographic standard and MP3 sound standard are other examples of standards that were developed formally. Similarly, each version of HTTP has been formalized after many years of discussion by parties interested in Web communication. A nonstandard protocol or data format is limited in use to its supporters and may or may not become a standard, depending on its general acceptance. For example, DVD videos encoded in the proprietary DivX format will play on some DVD players, but not on others.

Protocols define the specific agreed-upon sets of ground rules that make it possible for a communication to take place. Except for special applications, most computers perform their operations such that each hardware or software computer unit will understand what other computer units that they are connected with are saying. Protocols exist for communications between computers, for the communications between various I/O devices and a computer, and for communications between many software programs. A protocol specification defines such communication features as data representation, signaling characteristics, message format, meanings of messages, identification and authentication, and error detection. Protocols in a client-server system assure that requests are understood and fulfilled and that responses are interpreted correctly.

Since the use of a proprietary protocol would be limited to those with permission to use it, protocols are almost always eventually standardized. Although not always the case, protocols that are not standardized tend to die out from lack of use. In fact, international standards are often created to ensure that the protocols are universally compatible. As an example, HTTP, HyperText Transfer Protocol, guides communication between Web

servers and Web browsers on the Internet. The movement of data through the Internet is controlled by a **suite** of protocols called TCP/IP (Transmission Control Protocol/Internet Protocol). Storage devices communicate with a computer using a protocol called SATA. There are thousands of such protocols.

New protocols and other standards are proposed and created and standardized as the need arises. XML, RSS, and SIP are all examples of protocols developed recently to meet new demands. Satellite telecasting, near-universal telephone communication, wireless communications, and the Internet all demonstrate powerful and useful technologies made possible by protocols and standards. Indeed, the Internet is a measure of the success to which protocols that govern intercommunication between computer hardware and software have been standardized throughout the world. Discussions of various protocols and standards will occur regularly throughout this book.

1.5 OVERVIEW OF THIS BOOK

The focus of this book is upon the architecture and organization of computers, computer systems, and computer-based IT systems. Technically, there is a slight difference in definition between the terms “computer architecture” and “computer organization.” In this book we will usually not attempt to differentiate these terms and will use them interchangeably.

In this book we will be concerned with all four components of computer systems: hardware, software, data, and interconnectivity, and with the interactions between each component. We will also look initially at the larger picture: the organization of computer systems as components, themselves, to form enterprise IT systems. Chapter 2 of this first part is concerned with the system as a whole. The remainder of this book is divided into four additional parts, consisting of discussions of number systems and the representation of data in the computer, the hardware that makes up the computer, the software that the computer uses, and the networks that interconnect computers.

Our first step will be to examine the concept of systems in general. We will look at the characteristics and qualities that define a system. We will then use that basic understanding to look at the characteristics of computer-based IT systems and show how the various elements and requirements of computer systems fit into the system concept. Part 1 illustrates fundamental IT architecture concepts with several examples of IT system architectures.

In Part 2, we will look at the different forms the input data may take, and we will consider the translation processes required to convert data into forms that the computer hardware and software can process. You will see how the various data types that are familiar to you from programming languages are stored and manipulated inside the computer. You’ll learn the many different ways in which math calculations can be performed, and the advantages and disadvantages of each. You will see the difference between a number and the alphanumeric representation of a number, and understand why that difference can be critical in whether a program works or not. You will be able to relate the size of a word processing text to the storage capability of the computer’s disk.

In Part 3, we will take a detailed look at the various components of the hardware and how they fit together. You will learn how the CPU works, how different I/O devices work, and even how text and graphics manage to appear, seemingly by magic, on the display

screen. You will learn what makes some computers faster and more powerful than others, and what that means. You will learn about different ways of connecting I/O devices to the computer and see why you get a fast response from some devices, a slow response from others. You'll learn the difference between a serial port, a USB port, and a parallel port. We'll even explain the difference between PCI and PCI Express buses.

Most important, you will have the opportunity to see what a simple, program-obedient machine the computer really is. You will learn about the limitations of a computer. We all tend to think of the computer as a resource of infinite capacity, speed, and perhaps even intelligence, but of course that's not true. We will consider how these limitations affect your work as a user, and as a means of specifying a system that will meet your needs and requirements.

Part 4 will provide a careful introduction to the foundational principles of communication and networking. We will consider the basic communication technologies, networking hardware, software, channels and channel media, protocols, and methodologies that are required to support communication between computer systems in an IT system environment.

In the final part, we will consider the software that is used to control the computer's basic processing capabilities. Although computer software falls into two categories, operating system software and application software, we will focus exclusively on the system software. We will be concerned with control and efficient use of the computer hardware, fair and effective allocation of computer resources to different programs, security, storage management and file system structure, system administration, security, user interfaces, and more.

There are also four supplementary chapters covering topics that are somewhat outside the scope of the text, but important and interesting nonetheless. The first supplementary chapter introduces the fundamental logic that makes up a computer. The second supplementary chapter provides case studies that describe the hardware and system software of important real-world computer systems. These examples include the x86 family of PC hardware, the Microsoft Windows family of operating systems, Linux operating systems, and IBM mainframe hardware and software. The remaining two supplementary chapters, on CPU instruction addressing modes and on programming tools, have been maintained and updated from the 3rd edition. The supplementary chapters can be found on the book's website, www.wiley.com/college/englander.

Additional related topics of current interest may also be found on the book's website. The website also contains numerous links to reference materials, both general to computing as well as specific to individual topics discussed within the book.

1.6 A BRIEF ARCHITECTURAL HISTORY OF THE COMPUTER

Although a study of the history of computing is generally outside the scope of this book, a brief introduction is useful in showing the wide-ranging and quirky path by which IT has arrived to its present position. It is of particular interest to note that nearly all of the revolutionary concepts that define computer systems today were developed between thirty and sixty years ago; today's advances are more evolutionary and incremental in nature. This suggests that an understanding of the basic concepts that we are presenting in this

book should serve you, the reader, well in your ability to understand the importance and significance of future developments as they occur.

Early Work

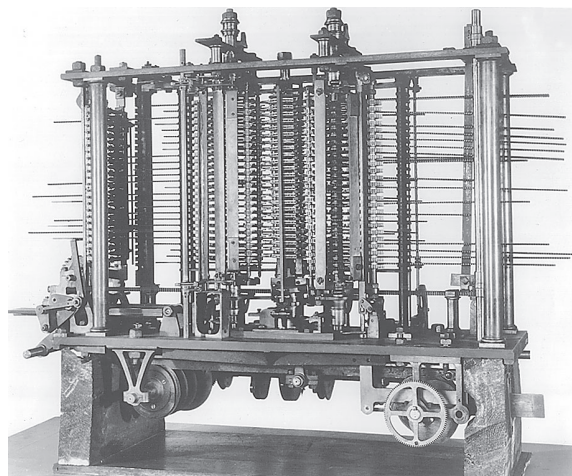
It is not possible, nor particularly useful, to identify the date of the “invention” of the computer. Indeed it has always been the aspiration of humankind to create devices that would simplify people’s work. Thus, it is not surprising that people were envisioning mechanical devices to simplify the jobs of routine data processing and calculation even in ancient times. In fact, there is recent evidence of the existence of an ancient computing device used for astronomical calculations. Instead, this discussion covers just a few of the major developments related to computer architecture.

In this context, one could consider the abacus, already in use as early as 500 BC by the ancient Greeks and Romans, to be an early predecessor of the computer. Certainly, the abacus was capable of performing calculations and storing data. Actually, if one were to build a binary numbered abacus, its calculations would very closely resemble those of the computer.

The abacus remained in common use until the 1500s and, in fact, is still considered an effective calculating tool in some cultures today. In the late 1500s, though, European inventors again began to put their minds to the problem of automatic calculation. Blaise Pascal, a noted French mathematician of the 1600s, invented a calculating machine in 1642 at the age of nineteen, although he was never able to construct the machine. In 1801, Joseph Marie Jacquard invented a loom that used punched cards to control the patterns woven into cloth. The program provided by the punched cards controlled rods that raised and lowered different threads in the correct sequence to print a particular pattern. This is the first documented application of the use of some form of storage to hold a program for the use of a semiautomated, programmable machine.

FIGURE 1.12

Babbage’s Analytical Engine

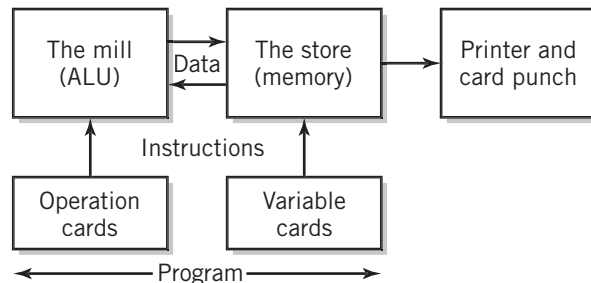


Courtesy of International Business Machines Corporation. Unauthorized use not permitted.

Charles Babbage, an English mathematician who lived in the early 1800s, spent much of his own personal fortune attempting to build a mechanical calculating machine that he called an “analytical engine.” The analytical engine resembles the modern computer in many conceptual ways. A photo of an early version of the analytical engine is shown in Figure 1.12. Babbage’s machine envisioned the use of Jacquard’s punched cards for input data and for the program, provided memory for internal storage, performed calculations as specified by the program using a central processing unit known as a “mill,” and printed output. Augusta Ada Byron, Countess of Lovelace and the daughter of the poet Lord Byron, worked closely with Babbage and developed many of the fundamental ideas of programming and program design, including the concepts of branches and loops.

FIGURE 1.13

Block Diagram of Babbage's Analytical Engine



Source: From *Computer Architecture and Organization*, 2nd ed., J. Hayes, copyright © 1988, by McGraw-Hill Companies, pg. 14. Reprinted by permission.

A block diagram of the Babbage analytical engine is shown in Figure 1.13. The mill was capable of selecting one of four arithmetic operations, and of testing the sign of a number with a different program branch specified for each result. The sequence of operation was specified by instructions on the operation cards. The operation cards could be advanced or reversed as a means of implementing a sort of “goto” instruction. The second set of cards, known as variable cards, were to be used to specify particular memory locations for the data involved in the calculations.

Babbage envisioned a memory of one thousand 50-digit decimal numbers. Each digit was to be stored using a ten-toothed gear known as

a counter wheel. Although the analytical engine was never completed, it should be apparent to you that it contains all the essential elements of today's computers. At approximately the same time, another English mathematician, George Boole, developed the binary theory of logic that bears his name, Boolean logic. He also recognized the relationship between binary arithmetic and Boolean logic that makes possible the circuitry that implements the modern electronic computer.

Computer Hardware

In the late 1930s and early 1940s, several different groups of researchers independently developed versions of the modern electronic computer. The Mark I, built in 1937 by Howard H. Aiken and associates at Harvard University with help and funding from IBM, used thousands of relays; relays are mechanical binary switches controlled by electrical currents, familiar to you perhaps as the clicking devices that control operations in tape cassette players and telephone answering machines. Although binary relays were used for computation, the fundamental design was decimal. Storage consisted of seventy-two 23-digit decimal numbers, stored on counter wheels. An additional counter wheel digit held the sign, using the digit 0 for plus and 9 for minus. The design appears to be based directly on Babbage's original concepts and use of mechanical calculator parts from IBM accounting machines. A similar electromechanical computer was designed and built by Conrad Zuse in Germany at about the same time.

The first totally electronic digital computer was apparently devised by John V. Atanasoff, a physicist at Iowa State College, in 1937. The machine was built in 1939 by Atanasoff and a graduate student, Clifford Berry, using electronic vacuum tubes as the switching components. The machine was known as ABC, for Atanasoff-Berry Computer. It is claimed that Atanasoff worked out the original details as he drove restlessly late one winter night from his house in Iowa to a bar in neighboring Illinois. The machine was not intended as a general-purpose computer, but was built to solve physics equations that Atanasoff was working on at the time. There is some doubt as to whether the machine ever worked completely.

ABC was a binary-based machine, just like today's computers. It consisted of an arithmetic/logic unit with thirty units that could do addition and subtraction, a rotating drum memory that held thirty binary numbers of 50 digits each, and punched card input. Each punched card held five 15-digit decimal numbers. These numbers were converted to binary as they entered the machine. Despite its limitations, ABC was an important pathmark that led to later significant advances in computer design. It is only recently that Atanasoff has begun to receive recognition for his achievement.

Much of the effort that culminated in a successful general-purpose computer architecture resulted from a wartime need for the solution to difficult mathematical formulas related to ballistic missile trajectories and other World War II research. The ENIAC (for Electronic Numerical Integrator and Computer, believe it or not) is generally considered to be the first all-electronic digital computer. It was designed and built between 1943 and 1946 by John W. Mauchly and J. Presper Eckert at the University of Pennsylvania, using the concepts that Mauchly had seen in Atanasoff's machine, although this was not publicly known at the time.

ENIAC had very limited storage capability, with only twenty locations each capable of holding a 10-digit decimal number. An additional one hundred numbers could be stored in read-only memory. Calculations were performed using decimal arithmetic. Ten electronic vacuum tube binary switches were used for each digit, with only one switch in the "ON" position to represent the value of the digit. Input and output used punched cards. The system could also provide printed output.

Programs could not be stored internally, but were hard wired with external "patch panels" and toggle switches. It took many hours to change programs, and, of course, debugging was a nightmare. Nonetheless, ENIAC was an important machine, some say the most important machine, especially since it led directly to the development of the UNIVAC I, the first commercially available computer, in 1951.

ENIAC contained eighteen thousand vacuum tubes, occupied a floor space of more than fifteen thousand square feet, and weighed more than thirty tons. A photograph of ENIAC, taken from *The New York Times* of February 15, 1946, is shown in Figure 1.14. Even in its day, ENIAC was recognized as an important achievement. ENIAC operated successfully until 1955, when it was dismantled, but not destroyed. Parts of the computer can be seen at the Smithsonian Institute, at the U.S. Military Academy at West Point, at the Moore School of the University of Pennsylvania, and at the University of Michigan.

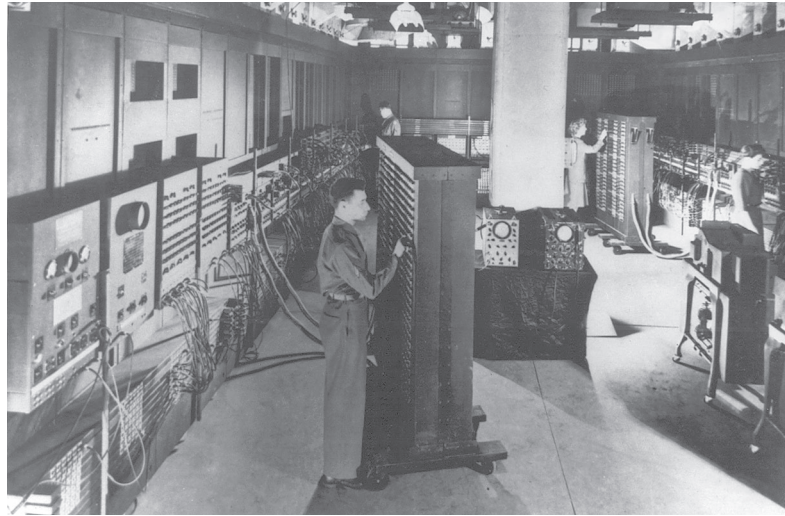
In 1945, John von Neumann, a consultant on the ENIAC project, proposed a computer that included a number of significant improvements over the ENIAC design. The most important of these were

1. A memory that would hold both programs and data, the so-called stored program concept. This solved the difficult problem of rewiring the control panels for changing programs on the ENIAC.
2. Binary processing of data. This simplified the design of the computer and allowed the use of binary memory for both instructions and data. It also recognized the natural relationship between the ON/OFF nature of switches and calculation in the binary number system, using Boolean logic.

The CPU was to include ALU, memory, and CU components. The control unit read instructions from memory and executed them. A method of handling input/output through

FIGURE 1.14

The ENIAC



Courtesy Sperry Univac, Div. of Sperry Corporation.

the control unit was also established. The instruction set contained instructions representing all the essential features of a modern computer. In other words, von Neumann's machine contained every major feature considered essential to modern computer architecture. Modern computer architecture is still referred to as **von Neumann architecture**.

Due to political intrigue and controversy, two different versions of von Neumann's architecture were designed and built, EDVAC at the University of Pennsylvania and IAS at the Princeton University Institute for Advanced Studies (hence the unusual name). Both machines were completed in 1951–1952. The success of EDVAC and IAS led to the development of many offspring, mostly with odd names, and to several commercial computers, including the first IBM computers.

At this point, von Neumann's architecture was firmly established. It remains the prevalent standard to this day and provides the foundation for the remainder of the material in this book. Although there have been significant advances in technology, and improvements in design that have resulted, today's designs still reflect the work done prior to 1951 on ABC, ENIAC, EDVAC, and IAS.

All of these early electronic computers relied on the electronic vacuum tube for their operation. Vacuum tubes were bulky, made of glass, fragile, short-lived, and required large amounts of power to operate. Vacuum tubes require an internal electric heater to function, and the heaters tend to fail quickly, resulting in what was known as a "burned out" tube. Furthermore, the heat generated by the large number of tubes used in a computer required a massive forced-air or water-cooling system. A report reprinted by computer historian James Cortada [CORT87] states that the average error-free operating time for ENIAC was only 5.6 hours. Such bulky, maintenance-requiring systems could not have attained the prevalence that computers have in our society. The technological breakthrough that made possible today's small, sophisticated computers was the invention of the transistor

and, subsequently, the integration of transistors and other electronic components with the development of the integrated circuit.

The invention of the integrated circuit led to smaller, faster, more powerful computers as well as a new, compact, inexpensive form of memory, RAM. Although many of these computers played an important role in the evolution of today's computers, two specific developments stand out from the rest: (1) development of the first widely accepted personal computer, by IBM in 1981, and (2) design of the Intel 8008 microprocessor, predecessor to the x86 CPU family, in 1972. The impact of these two developments is felt to this day.

Companies have developed better ways of moving data between different parts of the computer, better ways of handling memory, and methods for increasing the speed of instruction execution. There is a lot more processing power in today's personal computer than there was in the largest mainframe computer in the 1970s. Nonetheless, the basic architecture of today's machines is remarkably similar to that developed in the 1940s.

Operating Systems

Given how easy it is to communicate with computers today, it is hard to picture a time when the user had to do everything by hand, one step at a time. We take it for granted that we can type commands at a keyboard or move a mouse and launch programs, copy files, send text to a printer, and perform myriad other computer tasks. We power up and bootstrap our systems by pressing a switch.

It was not always this way. Early computers had no operating systems. The user, who was also the programmer, entered a program by setting it, one word at a time, with switches on the front panel, one switch per bit, or by plugging wires into a patch panel that resembled a cribbage board. Not a pleasant operation! Needless to say, early computers were single-user systems. Much of the computer's time was tied up with this primitive form of program and data entry. In fact, as late as the mid-1970s, there were still vendors producing computer systems with no operating system and computer hardware that was still bootstrapped by entering the bootstrap program one instruction at a time into switches on the front panel of the computer.

The history of system software, particularly operating systems, is much less well defined than that of hardware. According to Cortada [CORT87],

Without more sophisticated operating systems, scientists would not have been able to take full advantage of the power of the transistor and later of the [microprocessor] chip in building the computers known today. Yet their contribution to the overall evolution of digital computers has been overlooked by historians of data processing.

Part of the reason, undoubtedly, is that software evolved gradually, rather than as a series of important individually identifiable steps. The first operating systems and high-level programming languages appeared in the early 1950s, particularly associated with IBM and MIT, but with only a few exceptions, these efforts have not been associated with individual people or projects.

The need for operating system software came from the increasing computer power that resulted from the rapid development of new computers in the 1950s. Although the

hardware architecture has not changed substantially since that time, improved technology has resulted in a continuum of ever-increasing computer capability that continues to this day. It has been necessary to continually modify and improve operating system architecture to take advantage of that power and make it available to the user. Computing has changed from single-user batch processing (where only a single user, with a single program, could access the machine at one time), to multiple-user batch job submission (where each user's "job" was submitted to the computer by an operator for sequential runs), to multiuser batch job execution (where the computer executed several jobs simultaneously, thereby keeping the CPU busy while I/O took place on another user's job), to multiuser online computing (where each user had direct access to the computer), to single-user interactive personal computing, to today's powerful interactive networked systems, with multitasking, easy-to-use graphical interfaces, the ability to move data between applications, and near-instant access to other computers all over the world.

Each of these developments, plus various hardware developments—minicomputers, PCs, new I/O devices, multimedia—have required additional operating system sophistication; in each case, designers have responded to the need.

The early computers were used primarily by scientists and engineers to solve technical problems. The next generation of computers, in the late 1950s, provided a punched card reader for input and a printer for output. Soon after, magnetic tape systems became available. The first "high-level" languages, primarily assembly language, then FORTRAN, made it possible to write programs in a language other than binary, and offline card punch machines allowed programmers to prepare their programs for entry without tying up the machine. Algol, COBOL, and Lisp followed shortly after. New technology improved the reliability of the computers. All these advances combined to make the computer system practical for business commercial use, especially for large businesses.

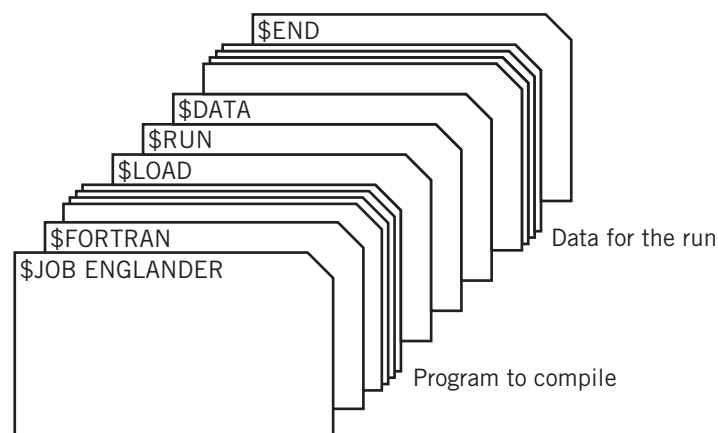
Still, these computers were single-user batch systems. Initially, users **submitted** the cards that they had prepared to the computer for execution. Later, separate, offline systems were developed that allowed the cards to be grouped together onto a magnetic tape for processing together. Programs were then submitted to the computer room in the form of *jobs*. A job consisted of one or more program card **decks**, together with the required **data decks** for each program. An output tape could also be used to support printing offline. As an example, Figure 1.15 shows a job that compiles and executes a FORTRAN program.

I/O routines were needed to operate the card readers, tape drives, and printers. The earliest operating systems consisted of just these I/O routines, but gradually operating systems evolved to perform other services. Computer time was very expensive, hundreds of dollars per minute, and in growing demand. To increase availability, control of the computer was placed in the hands of an operator, who fed the punched cards, mounted tapes, and generally tried to keep the system busy and efficient. The operating system provided a monitor that fed jobs to the system and supported the operator by notifying him or her of necessary actions, such as loading a new tape, setting switches on the panel, removing printout, and so on. As system demand increased, the monitor expanded to include accounting and simple, priority-based scheduling of jobs.

It is generally accepted that the first operating system was built by General Motors Research Laboratories in 1953–1954 for their IBM 701 computer. Other early systems included the FORTRAN Monitor System (FMS), IBSYS, and Share Operating System

FIGURE 1.15

Job Card Deck Used to Compile and Execute a FORTRAN Program



(SOS).³ Many important breakthroughs in operating system design occurred in the early 1960s. These breakthroughs laid the groundwork for the operating system as we know it today.

- In 1963, Burroughs released its Master Control Program (MCP). MCP contained many of the features of modern systems, including high-level language facilities and support for multiprocessing (with two identical CPUs). Most important, MCP supported virtual storage, as well as powerful multitasking capabilities.
- IBM introduced OS/360 as the operating system for its new System/360 in 1964. OS/360 provided a powerful language to expedite batch processing, JCL, or Job Control Language, and a simple form of multiprogramming that made it possible to load several jobs into memory, so that other jobs could use the CPU when one job was busy with input/output. By this time, disks were also becoming available, and the system was capable of reading cards onto disk while the CPU executed its jobs; thus, when a job completed, the operating system could load another job from disk into memory, ready to run. This improved the OS scheduling capability. JCL is still used for batch processing! The enormous success of the IBM OS/360 and its successors firmly established the basis of an operating system as a fundamental part of the computer.
- In 1962, a group at MIT known as Project MAC introduced the concept of time-sharing with an experimental operating system called CTSS. Project MAC was one of the seminal centers for the development of computer science. Shortly thereafter, MIT, Bell Labs, and GE formed a partnership to develop a major time-sharing system. The system was called MULTICS (Multiplexed

³Share was a consortium of system programmers who used IBM systems and who met to discuss problems and develop solutions. SOS was produced by a team of consortium members.

Information and Computing Service), and although MULTICS never fully realized its dream of becoming a major computer utility, many of the most important multitasking concepts and algorithms were developed by the MULTICS team. It was supplied for many years as the operating system for Honeywell computer systems.

- When Bell Labs withdrew from the MULTICS project, Ken Thompson, a MULTICS researcher, turned to the development of a small personal operating system, which he called Unics, later UNIX, to contrast it from MULTICS. He was later joined by Dennis Ritchie. The original UNIX development was performed on a Digital PDP-7 minicomputer and later moved to a PDP-11 minicomputer, the forerunner of the VAX computer. Originally, the system was written in assembly language, but Ritchie developed a new high-level language, which he called C, and the operating system was largely rewritten in C.

UNIX introduced many important OS concepts that are standard today, including the hierarchical file system, the shell concept, redirection, piping, and the use of simple commands that can be combined to perform powerful operations. Thompson and Ritchie included facilities for document production and formatting, including such novelties as a spell checker and a grammar checker. They created many inventive algorithms to improve operating system performance, developed techniques for interprocess communication, and even provided tools for networked and distributed processing. Many facets of operating systems that are taken for granted today were originated in UNIX development.

UNIX earned a reputation for power and flexibility. Because it was written in C, it was also easy to **port** it, that is, convert it for use, to other computers. As a result of these factors, UNIX became an important operating system for universities and was ultimately adopted, in many versions, by the commercial marketplace as well. UNIX continues to be of great importance, particularly due to its flexibility in the area of networks and distributed systems.

- Another important innovation, some would say the most important development in making the computer accessible to nontechnical users, was the development of the concept of **graphical user interfaces**. Most historians would credit the invention of the windows and mouse interface to Doug Englebart. This work was done, amazingly enough, in the 1960s, at Stanford Research Institute. A practical windowing system was built in the 1970s by Alan Kay and others at Xerox PARC (Palo Alto Research Center), as part of a visionary computer concept known as the Dynabook project. The original intention of Dynabook was to develop a book-sized personal computer with a high-resolution color display and wireless communication that would provide computer capabilities (particularly secretarial), games, e-mail, and a reference library. Although the technology of the time was not sufficient to bring the Dynabook as an entirety to fruition, the engineers at Xerox in the late 1970s built a personal computer workstation with a graphical user interface known as Star. It is believed that a visit to Xerox PARC by Steve Jobs, the founder of Apple, in 1979, inspired the development of the Apple Lisa and, subsequently, the Apple Macintosh.

The next important breakthrough in computer use occurred in 1982, with the introduction of the IBM personal computer. The IBM PC was designed as a stand-alone, single-user computer for the mass market. The IBM PC was supplied with a reasonably easy-to-use operating system, PC-DOS, which was developed and also later marketed by Microsoft as MS-DOS. PC-DOS was actually derived from an earlier personal computer operating system, CP/M (Control Program for Microcomputers), but is important because of the tremendous success of the IBM PC and its derivatives. Gradually, PC-DOS and MS-DOS became the prevalent operating system of the era. With later versions, Microsoft made many improvements, including hierarchical directory file storage, file redirection, better memory management, and an improved and expanded command set. Many of these improvements were derived from UNIX innovations. With the addition of Englebart and Kay's user interface innovations, MS-DOS has gradually evolved into Windows XP and Windows Vista, and most recently, Windows 7.

Even with all these earlier innovations, there continue to be tremendous advances in operating system software. Today's systems, such as Windows XP and Vista, Linux, and Macintosh OS X, combine much more power on one hand with improved user friendliness and ease of use on the other. There are several reasons for this:

- There has been a great increase in computer speed and power. More powerful integrated circuits have allowed the design of faster computers using faster clocks and larger internal data paths, together with techniques for speeding up instruction execution. Even small personal computers can support tens of megabytes of memory and many gigabytes of disk storage. A modern PC may contain as much as one thousand times the memory and execute instructions one thousand times as fast as the 1965 IBM OS/360 mainframe computer. Thus, more capability can be built into the operating system without sacrificing performance.
- There have been fundamental improvements in computer hardware design. Many modern computers are designed as an integrated unit, hardware and operating system software together. Most computer hardware contains special features intended to support a powerful operating system. Such features as special graphics, cache memory, vector processing, and virtual storage memory management hardware are intended primarily for use by the operating system. These features used to be available only on large mainframes. A protected mode of hardware instructions, accessible only to the operating system, provides security and protection to the operating system and allows the operating system to protect the system's resources and users.
- There have been fundamental improvements in operating system software design. Operating system programs have grown in size and complexity. Increased memory capacity has made a larger operating system feasible. Increased speed has made it practical. Gradually, innovative operating system techniques from large computers have drifted down to the PC level. In addition, program design itself has helped the process. New languages, well designed for system programming, and better programming methods such as object-oriented programming have also contributed to the process.
- There has been a shift in focus to creating operating systems that better serve the end user. This has resulted in much current research on human-computer

interfaces, and on the ways in which humans work and use the computer. New work paradigms, based on object-oriented programming and communication technologies, and new interfaces continue to extend the role of the operating system. There is a new willingness to include features that were not a part of earlier operating systems and to modularize the operating system in different ways to improve the delivery of services to the user and to the user's application programs.

- Networking has provided the opportunity for innovative research and development in distributed computing, including client-server technology, shared processing, and grid computing. There is a continuing progression of new operating system techniques, developed in response to the changing requirements of modern distributed systems.
- The rapid growth of the Internet, and of e-mail use, the Web, and multimedia in particular, has created opportunities and the need for better methods of accessing, retrieving, and sharing information between different systems. The results have impacted network design, user interface design, distributed processing technology, and open system standardization with corresponding effects in operating system design.

Although today's operating systems are highly complex and sophisticated, with many capabilities made possible by modern technology, particularly fast processors, large amounts of memory, and improved graphical I/O design, it is interesting to note that the major operating system features that we take for granted today are all evolutions based on innovations of more than thirty years ago.

Communication, Networks, and the Internet

With the development of large, multiterminal computer systems in the 1960s and 1970s, it was natural that users would want to use the computer to communicate with each other and to work collaboratively. Data was centrally stored in storage that was available to all, so it was easily shared among users on the same system. It soon occurred to software developers that it would be desirable to allow direct discussion among the users, both in real time and in the form of messages that could be stored on the system and made available to users when they logged in. Since data was centrally stored, the addition of message storage was a minor enhancement. "Talk" facilities that allowed users to communicate in real time were added later. These were similar to today's text messaging, although some had split-screen capability that allowed two users to send messages simultaneously. By 1965, some of these systems supported e-mail, and in 1971, Ray Tomlinson created the standard *username@hostname* format that is still in use today. As modems became available for users to log into their office systems from home and computers became more affordable, software innovators developed bulletin board systems, newsgroups, and discussion boards, where users could dial in and leave and retrieve messages. Gradually, it became possible to support modems on multiple lines, and affordable real-time "chat rooms" became possible.

During the same period, various developments occurred that made it possible to connect different computers together into simple networks. Some were based on direct links between modems on each computer. Others were based on early protocols, notably

X.25, a packet-switching protocol using phone lines. By 1980, these various innovations had evolved into a number of international networks, as well as three companies, Compuserve, AOL, and Prodigy, who offered e-mail, Usenet news, chat rooms, and other services to personal computer users.

All of this activity was, of course, a precursor to the Internet. Much of the modern history of networking and communication can be traced back to two specific developments: (1) a research project, ARPANET, whose goal was to connect computers at various universities and research centers, funded starting in 1969 by the U.S. Defense Department and later by the National Science Foundation and other groups, and (2) the development of the Ethernet by Robert Metcalfe, David Boggs, and others, which started at Xerox PARC in 1973. The ARPANET project was responsible for the design of TCP/IP, which was first tested in 1974, and issued as an international standard in 1981.

Because ARPANET and its successors, CSNet and NSFNet, were funded by the U.S. government, its use was initially limited to noncommercial activities. Gradually, other networks, some of them commercial, joined the network in order to exchange e-mails and other data, while the administrators of NSFNet chose to “look the other way”. Ultimately, the government turned over its Internet resources to private interests in 1995; at that point the Internet became commercial and expanded rapidly into the form that we know today.

Although it is only marginally related to the major issues addressed in this book, we would be remiss if we did not complete this discussion with a mention of Sir Tim Berners-Lee, of CERN, the European organization for nuclear research, who in 1991 developed the concepts that became the World Wide Web and Max Andreessen of the University of Illinois, who, in 1993, developed Mosaic, the first graphical Web browser.

SUMMARY AND REVIEW

This chapter has presented a brief review of the basics of computing. We began by recalling the input-process-output model for computing. Next we demonstrated the connection between that model and the components of the computer system. We noted that implementation of the model requires four components: hardware, software, communication, and data. The architecture of the computer system is made up of the hardware and system software. In addition, a communication component exists to enable interconnecting systems. We discussed the general architecture of a computer and noted that the same description applies to CPUs both modern and ancient, both large and small. We introduced the important concepts of virtualization, standards and protocols, noting that these ideas will appear throughout the book. The chapter concluded with a brief history of the computer from an architectural perspective.

FOR FURTHER READING

There are many good general introductory computer texts available for review if you feel you need one. New books appear so rapidly that we are reluctant to recommend any particular one. For alternative coverage of material in this book, you may find recent editions of various books by Stallings [e.g., STAL05] or Tanenbaum [e.g., TANE07] to be useful. Various chapters offer additional suggestions that are specifically applicable to the material in those chapters. The Web is also a rich source of knowledge. Two websites that

we have found particularly useful are wikipedia.org and howstuffworks.org. In addition to a wide range of material, these websites also offer numerous references to facilitate further study. Other useful websites include arstechnica.com and realworldtech.com.

The book by Rochester and Gantz [ROCH83] is a fun way to explore the history of computing. Historical facts are blended with other facts, anecdotes, humor, and miscellany about computers. Although the book is (sadly) out of print, it is available in many libraries. You can learn, in this book, about von Neumann's party habits, about movies that became video games, about computer scams and ripoffs, and lots of other interesting stuff. Perhaps the most thorough discussion of computer history is found in the three-volume dictionary by Cortada [CORT87]. Although Cortada is not really designed for casual reading, it provides ready access and solid information on particular topics of interest. Much of the historical discussion in this chapter was obtained from the Cortada volumes.

If you live or vacation in a city with a computer museum, you can enjoy another approach to computer history. Computer museums even allow you to play with some of the older computers. Well-known museums can be found in Washington, D.C., and within the Science Museum in Boston.

KEY CONCEPTS AND TERMS

application programming interface (API)	hardware input	port (from one computer to another)
arithmetic/logic unit (ALU)	input-process-output (IPO) model	primary storage
bus	interface unit	protocol
byte	kernel	random access memory (RAM)
central processing unit (CPU)	logical memory	read-only memory (ROM)
channel (I/O)	modem	software
communication channel	network interface card (NIC)	standards
control unit (CU)	open computing	stored program concept
data deck	operating system	submit (a job)
deck (program)	output	suite (protocol)
distributed computing		virtual
embedded computer		von Neumann architecture
graphical user interface		word

READING REVIEW QUESTIONS

- 1.1 Any computer system, large or small, can be represented by the four elements of an IPO model. Draw an IPO model; clearly label each of the four elements in your drawing.
- 1.2 One way to view an information technology system is to consider an IT system as consisting of four major components or building blocks. This book takes this approach by dividing the remainder of the book into parts, with a part devoted to each major type of component. What are the four components of an IT system that you will study in this book?

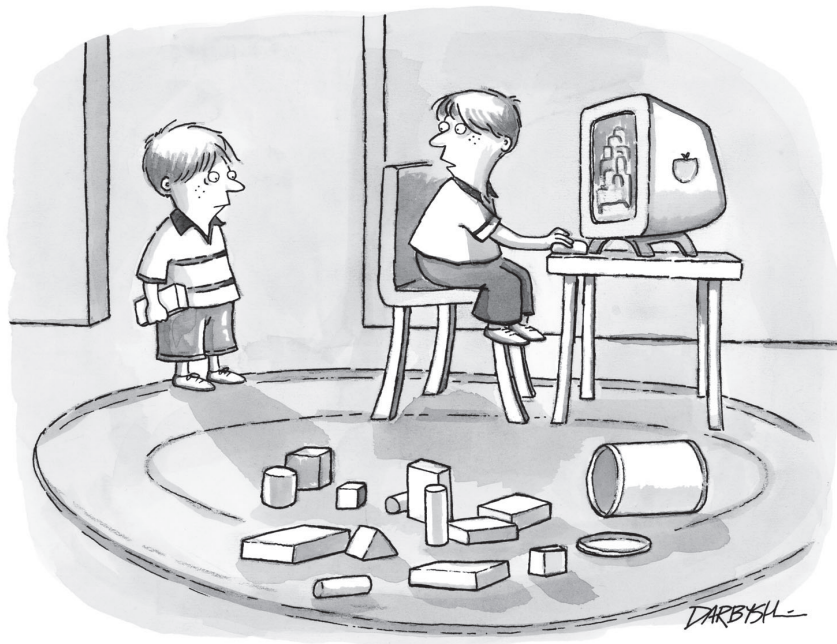
- 1.3 Explain the differences between primary storage and secondary storage. What is each type used for?
- 1.4 The book divides the software component of a computer system into two major categories. Identify each category and give an example of each that you are already familiar with. Briefly explain the role of each category.
- 1.5 The book compares a large mainframe computer to a small laptop computer or PDA, and states that the difference between them is one of magnitude, not of concept. Explain the meaning of that statement.
- 1.6 Virtualization is a concept that has taken on major importance in the early twenty-first century. Explain what is meant by virtualization.
- 1.7 What is a protocol? What is a standard? Do all protocols have to be standards? Explain. Are all standards protocols? Explain.

EXERCISES

- 1.1 Look at the computer ads on the business pages of a large daily newspaper and make a list of all the terms used that you don't understand. Save this list and check it from time to time during the semester. Cross out the items that you now understand and look up the items that have been covered but which you still don't understand.
- 1.2 For the computer that you normally use, identify which pieces constitute the hardware and which pieces constitute the system software. Now think about the file system of your computer. What part of the file system is hardware, what part software, and what part data?
- 1.3 Suppose you would like to buy a computer for your own needs. What are the major considerations and factors that would be important in your decision? What technical factors would influence your decision? Now try to lay out a specification for your machine. Consider and justify the features and options that you would like your machine to have.
- 1.4 Write a small program in your favorite high-level language. Compile your program. What is the ratio of high-level language statements to machine language statements? As a rough estimate, assume that each machine language statement requires approximately four bytes of file storage. Add various statements one at a time to your program and note the change in size of the corresponding machine language program.
- 1.5 Locate a current reference that lists the important protocols that are members of the TCP/IP protocol suite. Explain how each protocol contributes to the operation and use of the Internet.
- 1.6 Protocols and standards are an important feature of networks. Why is this so?
- 1.7 Although there is substantial overlap between protocols and standards there are protocols that are not standards and standards that are not protocols. With the help of a dictionary, identify the differences between the definition of protocol and the definition of standard; then, identify a specific example of a standard that is not a protocol; identify a specific example of a protocol that is not a standard.

CHAPTER 2

AN INTRODUCTION TO SYSTEM CONCEPTS AND SYSTEMS ARCHITECTURE



*"Now, this is just a simulation of what the blocks
will look like once they're assembled."*

C. Covert Darbyshire/The Cartoon Bank

2.0 INTRODUCTION

In this book we discuss systems: computer systems, operating systems, file systems, I/O (sub)systems, network systems, and more. Each of these same systems is also an element with a major role in the information technology systems that form the backbone of modern organizations. Indeed, these elements—computer hardware, software, data, and communication—together represent the infrastructure of every IT system. If we are to understand the various types of systems that are the focus of this book, it is important that we first understand the concept of “system” itself, and, then, equally important, the basic architectures of the IT systems that use these elements. Only then is it possible to see clearly the role of the various system elements in the larger IT picture as we visit each in turn.

Use of the word “system” is obviously not unique to IT. In our daily lives, too, we often use the word “system” to describe things in everyday language. Our homes have electrical systems, plumbing systems, heating and air conditioning systems, and maybe for some, even, home theatre systems. There are ignition, braking, fuel, exhaust, and electrical systems in our cars. Our cities have water systems, sewer systems, and transportation systems, to name a few. Philosophers and social scientists talk about social systems and linguistic systems. The economy deals with banking systems, financial systems and trading systems, and, for that matter, economic systems. The word “system” even appears in the names of thousands of companies.

So it seems as though everyone knows what a system is, but what is a system? We use the word “system” intuitively, without thinking about the meaning of the word, so we obviously have an intuitive understanding of what a system is. IT professionals, however, spend their careers analyzing, designing, developing, implementing, upgrading, maintaining, administering, and using systems everyday. It is therefore important that we have a deeper, more formal understanding of system concepts.

In this chapter, we consider the concept of a system from an IT perspective. We investigate the characteristics and composition of systems, explain the meaning of system architecture, and show the fundamental role of systems, particularly various types of IT systems, in business. We offer examples of different types of IT systems, and show how IT systems work together to accomplish tasks and solve problems. We show how systems can themselves be composed of subsystems, where the subsystems also fit the definition of systems.

After you have studied this chapter, you should have a clear understanding of what a system is, what kinds of systems are used in IT, the purpose and goals for each of these systems, and how these systems fit together and interact with each other and with their environment. You’ll understand the concept of system architecture. This discussion will set the stage for the remainder of the book, which considers individually and collectively the specific computer-based systems and subsystems that constitute the primary tools and components of business information technology.

2.1 THE GENERAL CONCEPT OF SYSTEMS

The most important characteristic that is shared by all of the systems mentioned above, and, indeed, by all systems, is that each is built up from a set of components that are linked together to form what we think of as a single unit. The house plumbing system, for example, consists of sinks, faucets, toilets, a hot water heater, bathtubs or showers, valves, and more, all connected together by pipes. An IT system consists of groups of computer hardware, various I/O devices, and application and system software, connected together by networks.

Often, the system is intended to serve a purpose or to produce results. The purpose of the house plumbing system is to allow the residents of the home access to water to wash, bathe, and drink. The purpose of an IT system is to allow organizations to process, access, and share information. The results of a successful IT system are documents, information, improved business processes and productivity, profits, strategic plans, and the like. This is, in fact, the “output” of the IPO model described in Chapter 1. In general, though, there is no requirement that a system serve a specific, definable purpose. The fact that the set of components may be considered as a single unit is sufficient to satisfy the concept of a system. The solar system is an example of a system where the purpose is unspecified.

There is also no requirement that the components of a system be physical. The links between components can also be physical or conceptual. In fact, the system itself may be conceptual, rather than physical. The number system is an example of a conceptual system. Computer operating systems are also conceptual, rather than physical. Business systems are also conceptual, although some of the components that they contain may be physical. The words *tangible* and *intangible* are sometimes used in place of physical and conceptual, respectively. Intangible or conceptual components and systems include ideas, methods, principles and policies, processes, software, and other abstractions. If, for example, the components in a system represent steps (intangible) in a multistep process, the links may represent the need to complete one step before the next is begun (also intangible).

Figure 2.1 illustrates a number of different systems to show you some of the possibilities. Figure 2.1(a) is a model of a home plumbing system. This is a physical system. The components are plumbing fixtures, linked by pipes. Figure 2.1(b) is a simplified representation of the solar system. The sun and planets are physical; the links in this system are conceptual, specifically, the distance of each planet from the sun, interplanetary and solar gravity, orbital relationships, the distances between planets at a particular point in time, and other attributes. Figure 2.1(c) is a diagram of a home networking system. The links in this case are a mixture of physical wires and (intangible) wireless connections. Sometimes the nature of the links is important only in terms of providing the proper interface connections to the components. Figure 2.1(d) is a simplified diagram of part of the inventory control portion of a sales system. The relationships between the components in this case are temporal (i.e., related to time). For example, inventory from a previous sale must be deducted from stock before we process the next order; otherwise we can’t promise delivery of goods on the new order because we don’t know if we still have sufficient goods in stock to fill the order.

With these pictures and ideas about systems in mind, we will define a **system** as follows:

A system is a collection of components linked together and organized in such a way as to be recognizable as a single unit.

FIGURE 2.1(a)

Plumbing System Diagram

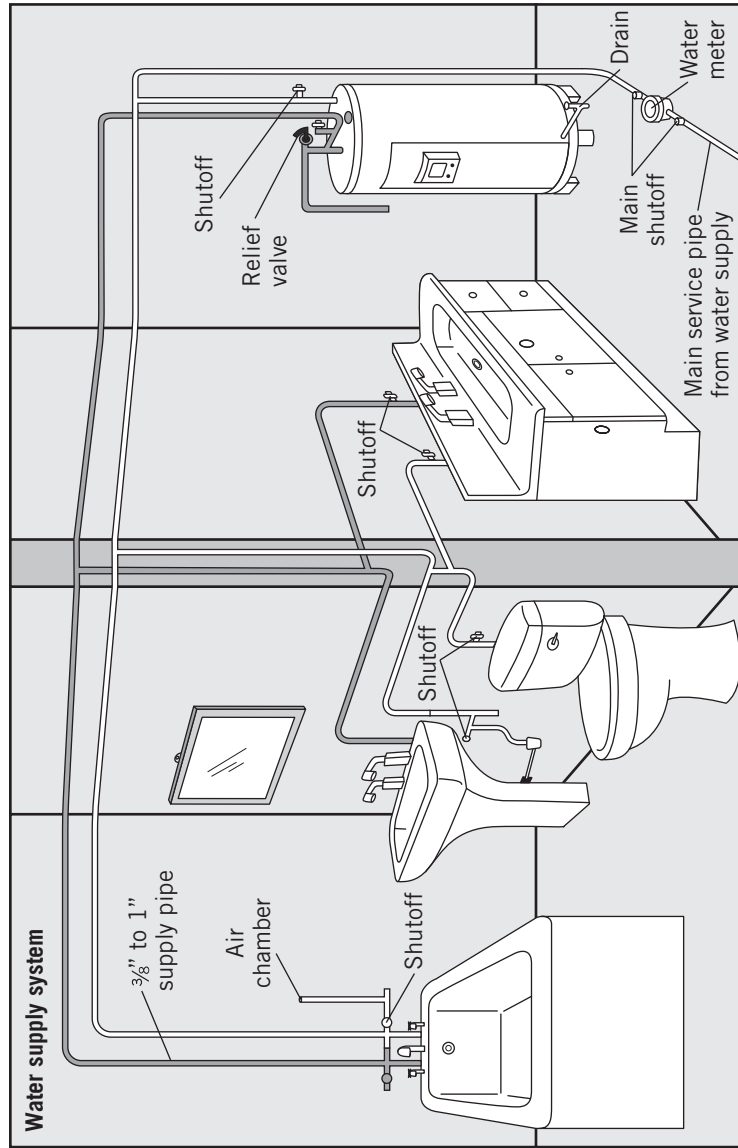


FIGURE 2.1(b)

The Solar System

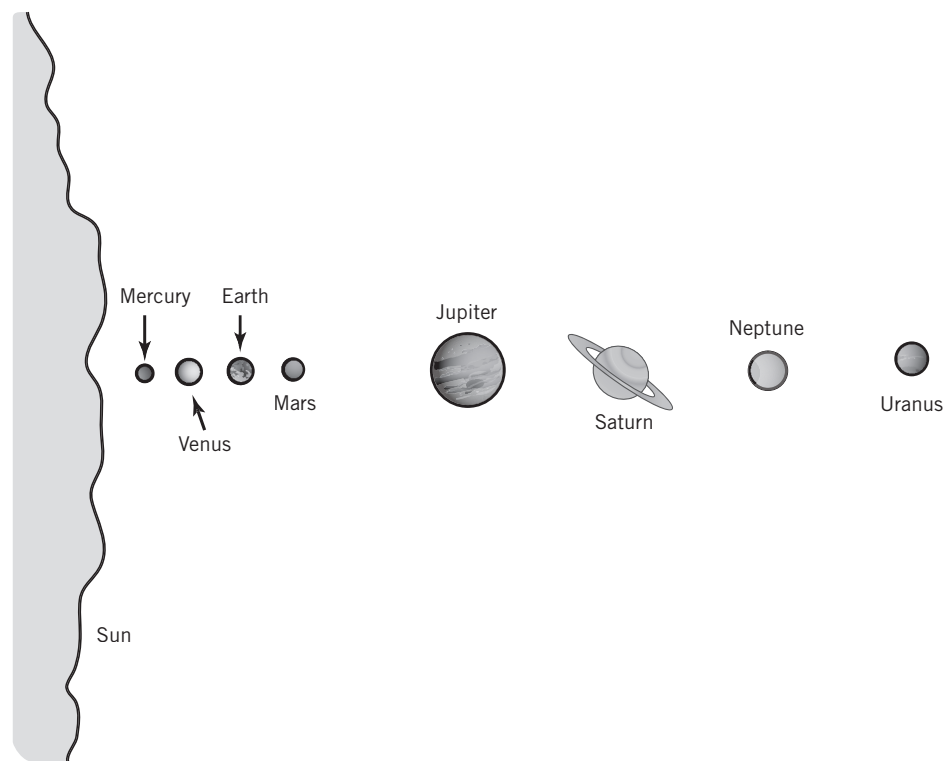


FIGURE 2.1(c)

A Typical Home Network System

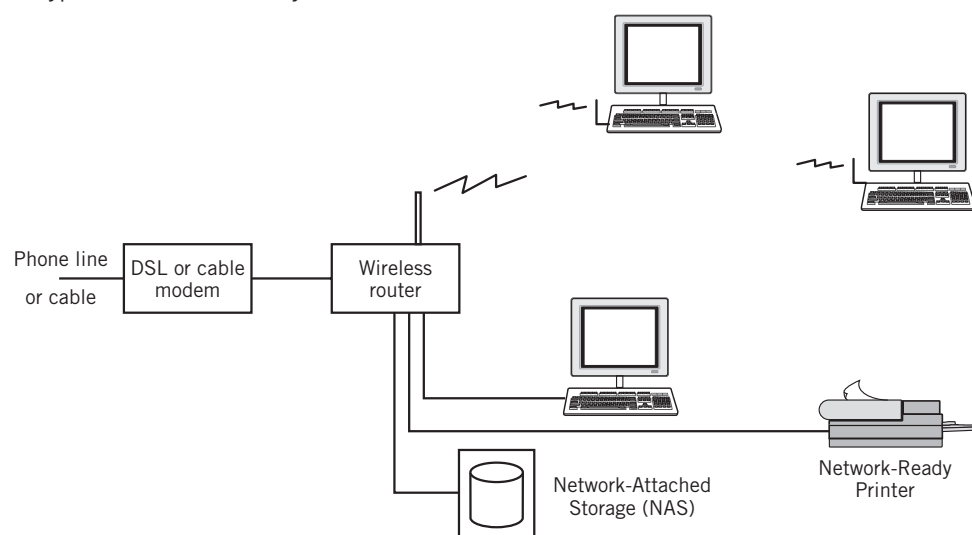
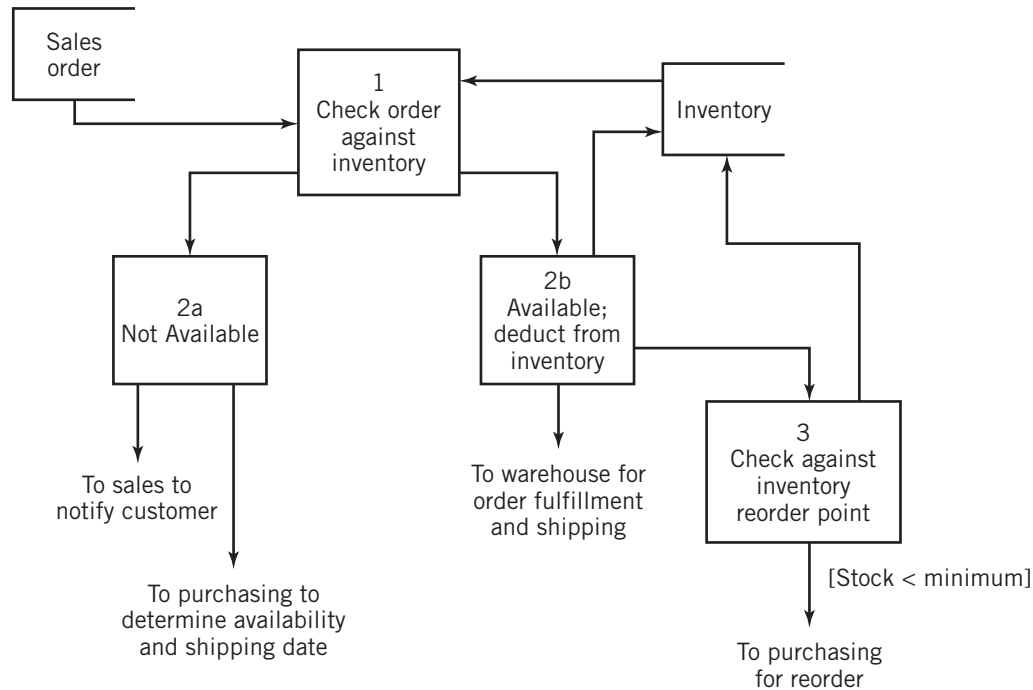


FIGURE 2.1(d)

Flow Diagram for Part of an Inventory Control System



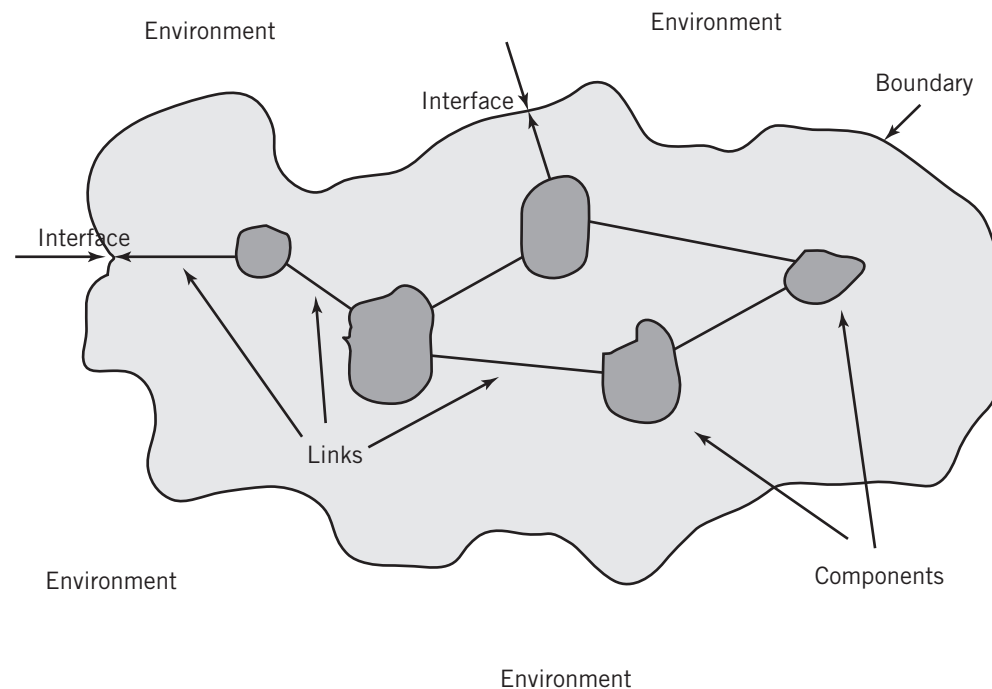
A general representation of a system is shown in Figure 2.2.

The linked components that constitute a system also define a boundary for the system. Anything outside the boundary represents the **environment** that the system operates or presents itself within. The environment may interact with and affect the system in various ways. The reverse is also true. The **interface** between the system and its environment is an important characteristic of the system. If the interface is well-defined, it is often possible to replace the existing system with a different system, as long as the interface between the system and the environment remains constant. This idea can have important implications when designing IT systems. For example, in a particular IT installation, a single large computer may be functionally the same as a network of small computers. When we define inputs and outputs for a system, the environment is the source of the input and also the receiver of the output.

As an example of the relationship between a system and its environment, consider the rather simplistic view of an e-business system illustrated in Figure 2.3. The organization represented by this illustration purchases goods from suppliers and makes them available for sale. (The value-adding component in the figure consists of various operations that make it worthwhile to buy from this organization, rather than directly from the supplier. For example, Amazon.com makes it possible to buy a wide variety of books from one source, rather than having to place separate orders from a number of different suppliers.) The environment for this system consists of customers who purchase from the system, suppliers

FIGURE 2.2

General Representation of a System

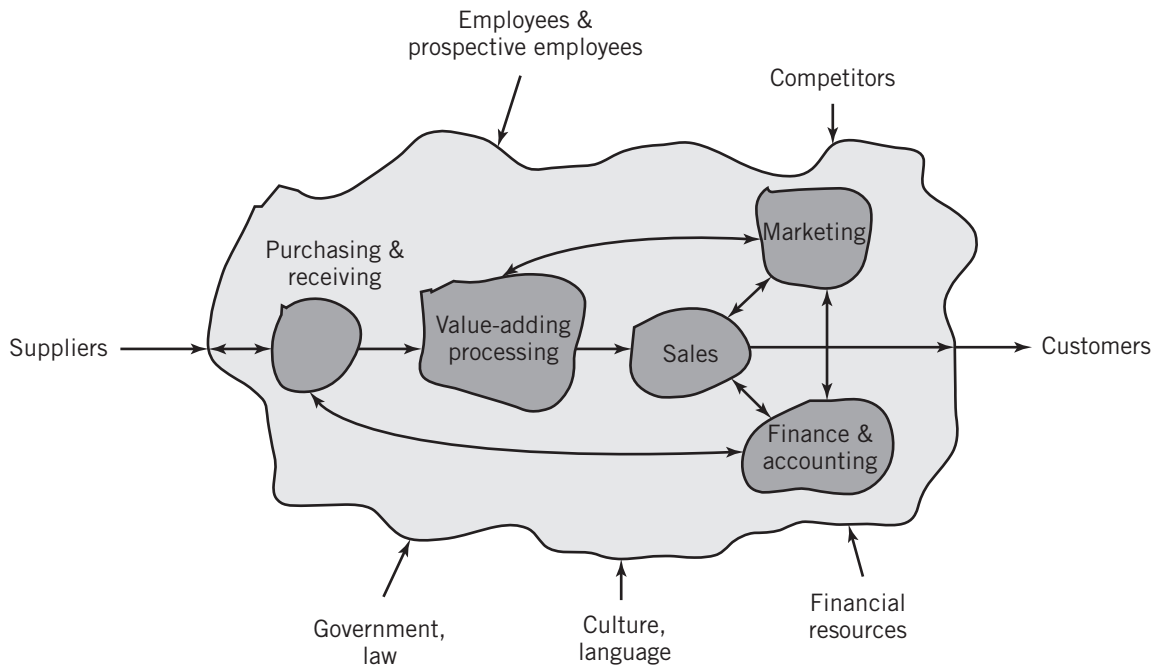


to the system, governments who control the legal aspects of the business and collect taxes, employees and prospective employees, external support personnel (such as repair people), financial resources, and others. The primary interfaces for this system are system input from suppliers and system output to purchasers; however, there are additional, more subtle interfaces to be considered, including legal, cultural, and financial interactions with the system. For example, sensitive cultural and language issues that offend potential customers on a website might have an important impact on an organization's sales.

When analyzing a system, the components of the system may be treated as irreducible or they may themselves be representable as systems. When considered in the context of a particular system, these components would be viewed more accurately as **subsystems**. A business IT system, for example, might have marketing, manufacturing, purchasing, inventory, finance, and accounting subsystems, among others. Even these components might be expanded. The marketing subsystem might be further broken down into sales, development, and advertising components, as one possibility. The level of detail to be considered depends on the context in which the system is being considered, discussed, evaluated, or used. The division of a system or subsystem into its components and linkages is called **decomposition**. Decomposition is inherently hierarchical. The ability to decompose a system hierarchically into subsequent sets of components and subsystems is an important property of systems.

FIGURE 2.3

A Simple E-Business System



The fundamental properties, and the patterns of relationships, connections, constraints, and linkages among the components and between the system and its environment are known collectively as the **architecture** of the system. Some people choose to differentiate the *architecture* of a system from the *organization* of a system. The assumption is that the architecture is fundamental to the meaning and value of the system whereas the organization is one of possibly many combinations of components and linkages that meets the requirements of the architecture. The difference is subtle and often unimportant.

It is common to represent systems and their components by models or drawings on paper or objects within a computer program. These representations are **abstractions**. They *represent* the real system but are *not actually* the real system. (For example, the solar system does not fit conveniently inside a computer!) It should be obvious to you that all of the illustrations of systems in Figures 2.1, 2.2, and 2.3 are abstractions.

The primary reason for humans to group components into systems and to represent them as abstractions is to simplify understanding and analysis, particularly if the individual components are numerous and complex. We can study the relationships between the different components without the distraction created by the details of individual components. We can decompose, isolate and study individual components when required. We can study the interactions between the environment and the system as a whole. Effectively, our analyses are simplified by eliminating factors that are not relevant in the context of our interests. In a large network of computers, for example, we may be concerned primarily

with the flow of data between computers. The details of the individual computers are unimportant. In general, dealing with models at the system level allows us to isolate and focus on the specific elements of interest more easily, by treating other elements collectively.

To escape our fixation on information technology systems for an instant, consider, just for fun, the solar system that we've used previously as an example. If we are studying the Milky Way galaxy, it is convenient and sufficient to treat the solar system as a single irreducible component in the galaxy. We might be interested in the location and movement of our Sun in the galaxy, for example, but the structure of the planets is irrelevant to our study in this case. On the other hand, if we are interested in studying the effects of the tides on a sea shore where we are planning to vacation, we will have to expand the "Earth" component and look at the specific effects of the moon and other nearby objects as part of our analysis.

Consider, too, the role played by decomposition and the ability to isolate and study individual components. A complex system may be divided into relatively independent components and analyzed by different individuals, each a specialist in their own area. Thus a plumber can create a home water system component without concern for the details of the electrician's efforts. They can work together on the linkages that concern both of them, for example, the wiring for the boiler in a hot water heating system. The system architect coordinates the different efforts. The role of an IT system architect is similar: to work with finance experts on the finance component, marketing experts on the marketing component, and so forth.

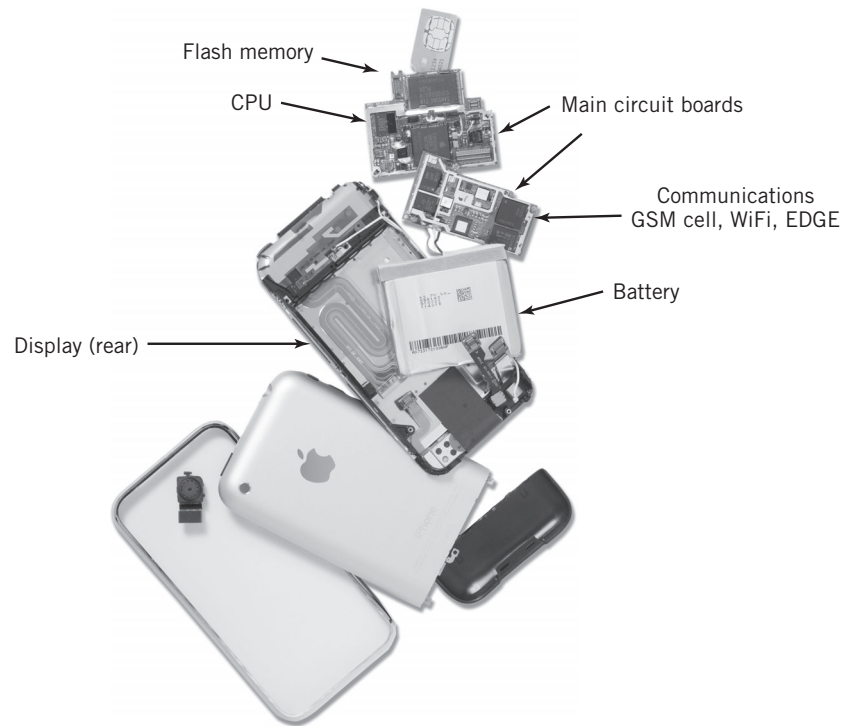
When the goal of a project is to implement a system of some type, it is sometimes convenient to view the components of a system as modules that can be implemented independently, then linked together to produce the final result. This technique can simplify analysis, design, assembly, upgrading, and even repair. It also supports collaboration during the design process, since individual components can be designed by different individuals using specifications for the interfaces.

For example, a cell phone might consist of a computer control module, a memory module, a display module, an audio input/output module, a radio transmitter/receiver module, a keypad/text input module, and a wireless networking module. Each component might have been developed by a different team. These modules, designed, constructed, and manufactured as individual assemblies, properly interfaced, wired together, and mounted into a case, constitute the design of a typical cell phone. They also represent the components that might appear in the system diagram for a cell phone. The same approach might be taken with a computer system, with a central processor module, a graphics display module, an audio module, a network module, a hard drive controller module, and so on. Figure 2.4, for example, shows the basic system hardware components that make up an iPhone.

It is also important to realize that there may be many different representations for a system, to reflect the various uses of the system model. Returning to our IT roots for an example, the representation of the business system shown in Figure 2.5(a) is a traditional hierarchical organization chart. The components are departments that perform various functions within the business. In contrast, a partial model of the same business shown in Figure 2.5(b) represents the application architecture of an IT system within this business. Take another look at Figure 1.4 for still another representation of a business. As another simple example, you could represent a house by the physical appearance of its exterior, by the function and layout of its rooms, or by the various subsystems, electrical, plumbing, heating, and so on that the house requires. Presumably, each of these representations would be useful to a different participant. In fact, we would expect an architect to provide all of these for use by the owner, the builder, and the various contractors.

FIGURE 2.4

iPhone Components



Courtesy Christopher Harting.

FIGURE 2.5(a)

Business Organization Chart

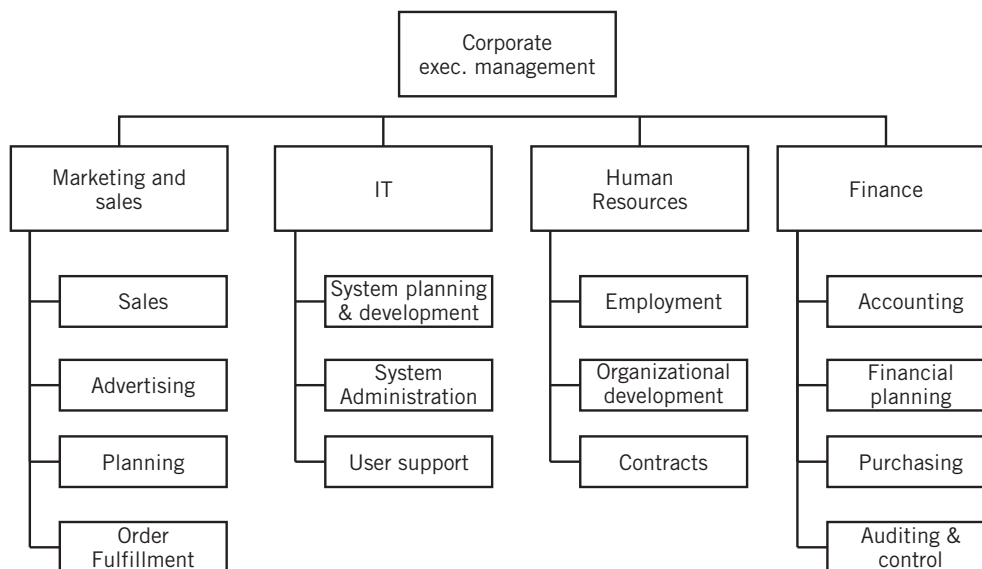
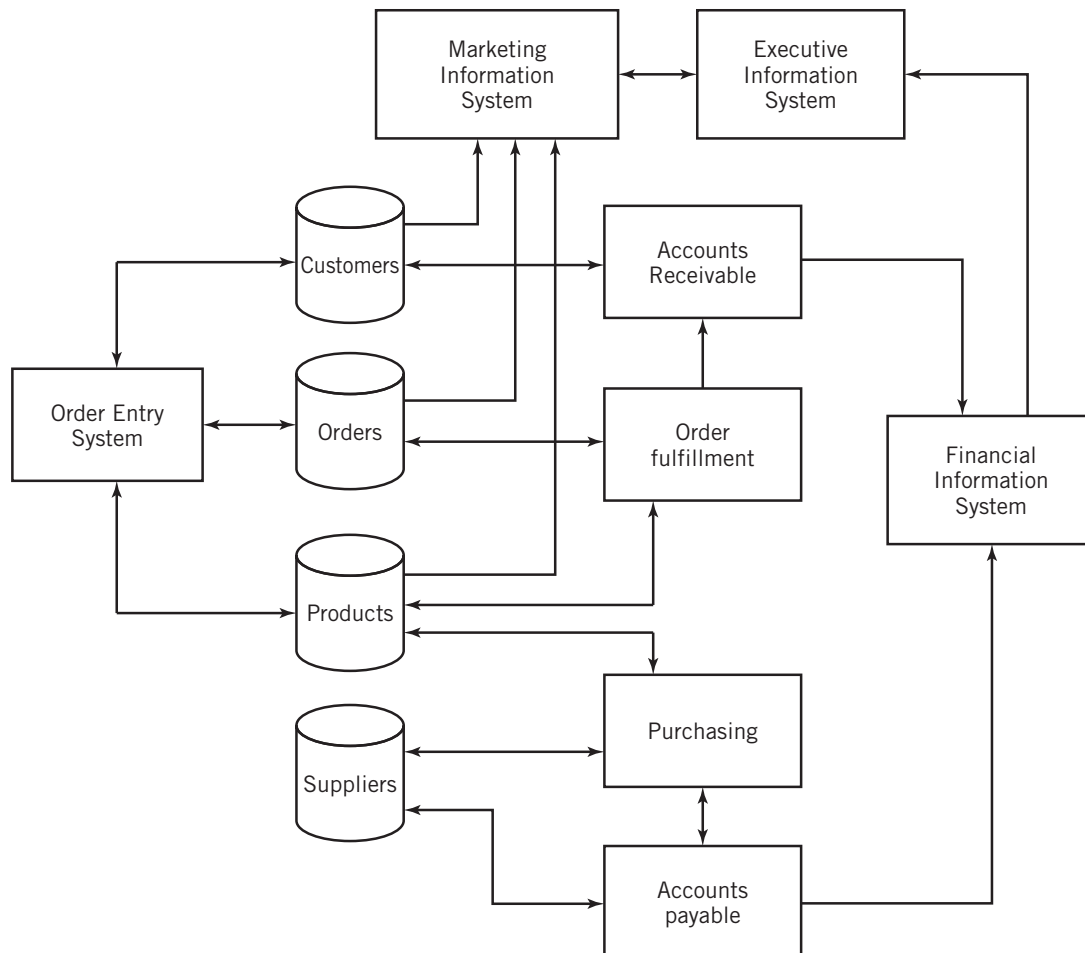


FIGURE 2.5(b)

Partial View of a Business Application Architecture



2.2 IT SYSTEM ARCHITECTURES

The use of system concepts is particularly applicable when discussing the various types of IT systems. In general, the goal of IT systems is to assist organizations to meet the strategic needs of the enterprise. Not surprisingly, IT systems are frequently complex, and the ability to separate them naturally into subsystems or components of manageable size simplifies understanding of the system as a whole. The analysis, design, and implementation of IT systems must take place at different levels of detail and frequently require collaboration among many analysts and designers. This corresponds well with the ability to decompose systems into components, hierarchically, which allows us to concentrate at the appropriate

levels of detail during each step along the way. This approach is known as a **top-down approach**. The top-down approach allows us to focus on the specific areas of interest without the distraction of details that are irrelevant for the level that we're studying. In this way, a system architect can analyze and study the IT system as a whole, encapsulating the computer systems, software systems, network architecture, and Web architecture that represent components, and focusing instead on the large picture: the purpose of each component and the requirements for the interfaces and linkages that connect and integrate them. With the IT system architecture firmly established, we can consider the individual business functions, computer systems, and networks that will link them together. For IT system analysis, this is often sufficient, at least superficially, assuming that the system architects actually understand the conditions and constraints imposed by details at the lower levels.

Although there are other, equally valid, approaches to IT system analysis and design, and many other important considerations as well, this approach suits the purposes of this book well because it allows us to establish general requirements for IT systems and then to show how the specific capabilities and characteristics of computer hardware, operating systems, networks, and data fulfill those requirements.

With these ideas in mind, let us return to the simple word processing example from Chapter 1 and reconsider it from a system architecture perspective. Recall that in this example you are sitting at your computer typing text into a word processor. We noted that the computer accepted input from your mouse and keyboard, processed it according to rules established by the application software, and produced output, which appeared on a display. From the system perspective, we can, for now, treat the whole computer, keyboard, display, printer, storage, software, and all as a single component. You're the relevant part of the environment for this discussion. Forgetting the issue of control for now, the system has an input and an output. Both of these interface with you, the environment. The data for this interface is alphanumeric text in human-readable form. Other input data to the document might include graphics drawn with the mouse, photographic images from a digital camera, bar codes, or music from an iPod or other audio source. We described this scenario earlier, in Chapter 1, as input-process-output.

A system this simple is unlikely to meet all the needs of even the smallest enterprise or, even, the least computer-literate individual. But it does serve as a starting point to recognizing the value of a system approach to the understanding of information technology.

Distributed Processing Systems

Realistically, modern IT system architectures generally rely on multiple computers connected by networks of communication channels to achieve their goals. In all but the smallest organizations, input data is collected from locations scattered throughout the organization, stored, processed, and distributed to other locations within the organization. Since modern computer hardware and networking equipment is plentiful and inexpensive, it is practical to distribute computing capability to everyone who needs it. Furthermore, the availability of the Internet and alternative structures, such as satellite communications, make global data communication practical. Web access, organization intranets, e-mail capability, analysis tools, such as Microsoft Excel, and document preparation tools are widely available and are considered essential business tools throughout most organizations. Collaboration

between different organizations, particularly in the area of automated business-to-business purchasing and sales, is commonplace.

Therefore, when envisioning effective IT systems, designers typically must create system architectures that are capable of supporting large numbers of user workstations who will have ready access to the organizational information that they need. The system must be able to reliably store and protect large amounts of organizational data. For many organizations, customers outside of the organization may also need access to the system to get information and to make purchases.

Consider a few typical simple scenarios:

- A global fast food chain collects data each day from each of its restaurants worldwide to establish sales figures and determine sales trends. This allows the company to determine which locations are most productive and which locations need assistance, which items sell best and which need to be modified or replaced, and so on.
- A large travel organization conducts much of its business online, using travel agents located all over the world. It maintains Web servers that have immediate access to large databases of client information and travel information, as well as continual and instant access to airline and hotel reservation systems to determine current airfares, seat availability, and hotel room availability. All of this information must be immediately accessible to every agent and must be current at every instant. Even brief system failures are very costly.
- A large Web-based retail sales organization sells large quantities of a wide variety of merchandise. (Think Amazon or Wal-Mart.) Orders initially come in to a central facility, where they are billed. Inventory is stored in warehouses in various countries and local regional areas to expedite delivery and reduce delivery costs. The system must be able to distribute orders to the various regional facilities efficiently; it must also maintain appropriate levels of goods at each warehouse to match sales and must be able to locate goods and arrange shipping in response to orders as they come in.

Inventory replenishment is handled by an automated purchasing IT system component that is integrated with the IT systems of the suppliers. Purchase order data is passed from the retailer to a supplier, which triggers order placement, billing and shipment components in the supplier's systems. Web technology is commonly used to satisfy the need for data and communication compatibility between the systems.

- Even conventional business order processing is inherently distributed within an organization. A purchase order, for example, might be entered into the system by a salesperson in the sales department; the order is checked by order fulfillment for inventory, then distributed to the accounting department for a credit check and billing, and sent to the warehousing area for packaging and shipping. Back orders and inventory replenishment are sent to the purchasing department. For planning and marketing purposes, data will be collected into a central location and processed into sales figures, inventory planning and purchase requirements data, and the like. In a large organization, these functions might be widely scattered over a city, country, or even the world.

The emphasis in each of these scenarios is the flow and processing of data within an organization or between organizations or between an organization and its environment. The system architecture representation of such operations is called **application architecture**. Application architecture is primarily concerned with the activities and processing of application programs and the communications between them. Since the application architecture addresses the fundamental business needs of the organization, the application architecture is typically the primary consideration in IT system design. Therefore, the system requirements and constraints set by the application architecture have major impact on the hardware architecture and network architecture requirements for the system. Within the application architecture realm the selection and layout of computer systems and communication networks is of concern primarily to the extent that it adequately supports the application software and its functionality. However, additional factors such as scalability, convenience, information availability, data security, system administration, power and space requirements, and cost may also play important roles in computer and network architectural designs.

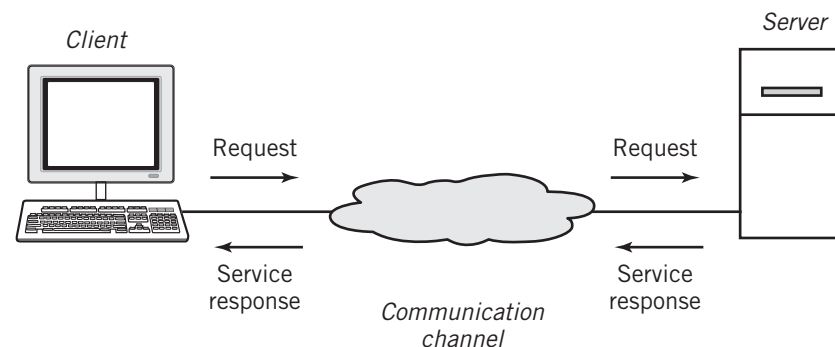
CLIENT-SERVER COMPUTING There are a variety of possible application architectures that can satisfy the requirements of modern organizations. Most, however, are based on different applications of a simple technological concept, the **client-server** model.

In a client-server configuration, a program on a client computer accepts services and resources from a complementary program on a server computer. The services and resources can include application programs, processing services, database services, Web services, file services, print services, directory services, e-mail, remote access services, even computer system initial startup service. In most cases, the client-server relationship is between complementary application programs. In certain cases, particularly for file services and printer sharing, the services are provided by programs in the operating system. Basic communication and network services are also provided by operating system programs.

Basic client-server architecture is illustrated in Figure 2.6. Notice that the link between client and server is essentially irrelevant within the application architecture view of the system. The “cloud” in the figure is intended to indicate only that there is a link of some kind between the client and the server. The link can be a network connection, an intranet

FIGURE 2.6

Basic Client-Server Architecture



or Internet connection, or some sort of direct connection. In fact, a single computer can act as both client and server, if desired. (A situation where this is the case is described in Chapter 16.)

The client-server model describes the relationship and behavior of programs in one or two computers under particular prescribed circumstances. It is important to understand that the client-server model does not require any special computer hardware. Furthermore, networking software within the operating system of each computer routinely provides basic communication capabilities. The only “special” software required is the software within the complementary application programs that provides the communications between the programs. The requests and responses take the form of data messages between the client and server that are understood by both application programs. As an example, slightly simplified, the HTTP request message sent to a Web server by a Web browser requesting a Web page consists of the word `GET` followed by a URL. If the request is successful, the message returned by the server contains the HTML text for the page.

From the description and the figure you can see that the Web browser–Web server application described as an example in Chapter 1 fits the description of a client-server application. We will return to this example momentarily.

A typical use of the client-server concept within an organization is shown in Figure 2.7. In this case, a number of clients are sharing a number of servers, showing both the **shared server** nature of client-server computing, as well as to show that there may be multiple servers offering different services on the same network. Notice, also, that the server computer labeled S2 in the figure is running two different server applications. Since computers are capable of running multiple tasks concurrently, this is a possible scenario. The only limitations to running multiple applications on a single server are the potential slowdowns that may result from the load on the server computer and the traffic on the network to that server. Overall, there is a multiple-multiple relationship between clients and servers: a server can serve multiple clients, and a client can request services from multiple servers.

The use of client-server processing as a basis for IT system architecture has a number of advantages:

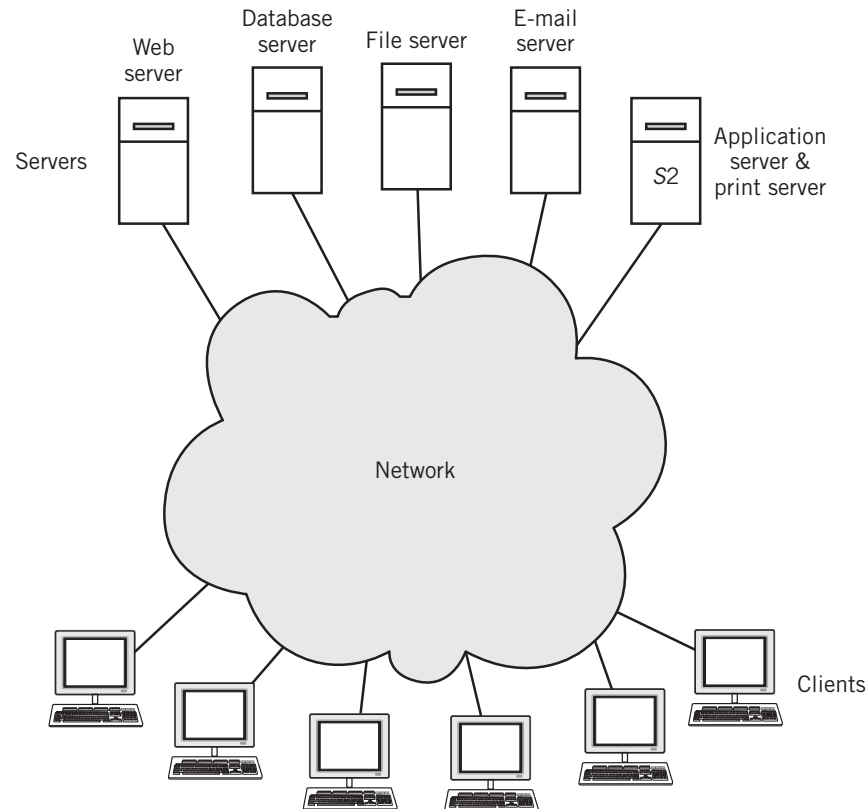
- Providing services on a single computer or on a small number of computers in a central location makes the resources and services easy to locate and available to everyone who needs them, but also allows the IT administrators to protect the resources and control and manage their use. The consistency of files and data can be managed and assured.

For example, client-server technology can ensure that every user requesting a particular program from a server will receive the same version of the program. As another example, suppose a program has a license that limits the number of simultaneous users. The program server can easily limit distribution of the program appropriately.

- The amount of data to be stored, processed, and managed may be extremely large. It is more efficient to equip a small number of computers with the power needed than to require powerful computers at every station.
- Typically, humans request information from knowledgeable sources as they need it. Thus, the client-server approach is naturally consistent with the way humans acquire and use information.

FIGURE 2.7

Clients and Servers on a Network



The most familiar example of the use of client-server technology is the Web browser–Web server model used in intranets and on the Internet. In its simplest form, this model is an example of **two-tier architecture**. Two-tier architecture simply means that there are two computers involved in the service. The key features of this architecture are a client computer running the Web browser application, a server computer running the Web server application, a communication link between them, and a set of standard protocols, in this case, HTTP, for the communication between the Web applications, HTML for the data presentation requirements, and, usually, the TCP/IP protocol suite for the networking communications.

In the simplest case, a Web browser requests a Web page that is stored as a pre-created HTML file on the server. More commonly, the user is seeking specific information, and a custom Web page must be created “on the fly”, using an application program that looks up the required data in a database, processes the data as necessary, and formats it to build the desired page dynamically.

Although it is possible to maintain the database and perform the additional database processing and page creation on the same computer as the Web server, the Web server in a

large Internet-based business may have to respond to thousands of requests simultaneously. Because response time is considered an important measure by most Web users, it is often more practical to separate the database and page processing into a third computer system. The result, shown in Figure 2.8, is called a **three-tier architecture**. Note that, in this case, the Web server machine is a client to the database application and database server on the third computer. *CGI*, the *Common Gateway Interface*, is a protocol for making communication between the Web server and the database application possible. (In the figure, we have placed the page creation application software on the database machine, but it could be located on the Web server instead if doing so would balance the loads on the two machines better.) In some situations, it is even desirable to extend this idea further. Within reason, separating different applications and processing can result in better overall control, can simplify system upgrades, and can minimize scalability issues. The most general case is known as an *n-tier architecture*.

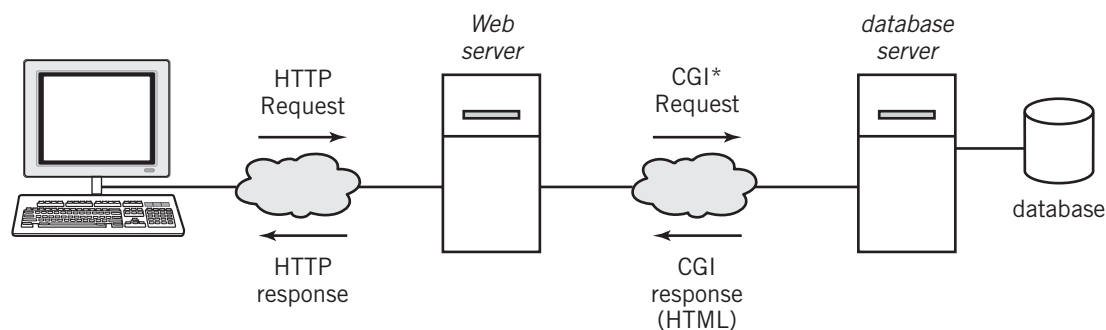
Client-server architecture is a distributed processing methodology, in which some of the processing is performed on the client system and some is performed on the server system. To see this more clearly, consider the distribution of processing between the client and server in a database application, in which the client requests specific information from a database stored on a database server.

At one extreme, the client application provides little more than a request form and a means to display the results. All of the processing is performed on the server. This might be appropriate if there is little computing power in the client. Certain so-called “thin” clients or “end-user” terminals might meet this criterion, but this situation is increasingly rare. Because this extreme case puts the entire processing load on the server, the system designer will have to specify a more powerful computer for the server; additionally, the requirements of the database server may limit the capability of the server computer system to perform other tasks or to scale for increased usage.

At the other extreme, the database server application simply accesses data from the database and passes all of the data to the client. The client application performs all of the processing. This relieves the load on the server, and it is reasonable to assume that modern client computers would be able to handle most database processing tasks relatively easily. However, the potential transfer of large amounts of raw data from the server to

FIGURE 2.8

Three-Tier Database Architecture



*CGI: Common Gateway Interface

the client for processing may put an extra burden on the network instead, requiring the system designer to specify higher speed network components at potentially higher cost and additional implementation difficulty.

A well-designed system analysis will consider the different factors, the complexity of the applications, expected network traffic, usage patterns, and the like. The optimum solution is likely to fall somewhere in the middle, with some pieces of applications on the server, others on the client.

One of the strengths of client-server architecture is its ability to enable different computer hardware and software to work together. This provides flexibility in the selection of server and client equipment tailored to the needs of both the organization and the individual users. One difficulty that sometimes arises when different computers have to work together is potential incompatibilities between the application software that resides on different equipment. This problem is commonly solved with software called **middleware**. Middleware resides logically between the servers and the clients. Typically, the middleware will reside physically on a server with other applications, but on a large system it might be installed on its own server. Either way, both clients and servers send all request and response messages to the middleware. The middleware resolves problems between incompatible message and data formats before forwarding the messages. It also manages system changes, such as the movement of a server application program from one server to another. In this case, the middleware would forward the message to the new server transparently. The middleware thus assures continued system access and stability. In general, the use of middleware can improve system performance and administration.

WEB-BASED COMPUTING The widespread success of the World Wide Web has resulted in a large base of computer users familiar with Web techniques, powerful development tools for creating Web sites and Web pages and for linking them with other applications, and protocols and standards that offer a wide and flexible variety of techniques for the collection, manipulation, and display of data and information. In addition, a powerful website is already a critical component in the system strategy of most modern organizations. Much of the data provided for the website is provided by architectural components of the organization's systems that are already in place.

Not surprisingly, these factors have led system designers to retrofit and integrate Web technology into new and existing systems, creating modern systems which take advantage of Web technology to collect, process, and present data more effectively to the users of the system.

The user of a Web-based system interacts with the system using a standard Web browser, enters data into the system by filling out Web-style forms, and accesses data using Web pages created by the system in a manner essentially identical to those used for the Internet. The organization's internal network, commonly called an **intranet**, is implemented using Web technology. To the user, integration between the intranet and the Internet is relatively seamless, limited only by the security measures designed into the system. This system architecture offers a consistent and familiar interface to users; Web-enabled applications offer access to the organization's traditional applications through the Web. Web technology can even extend the reach of these applications to employees in other parts of the world, using the Internet as the communication channel.

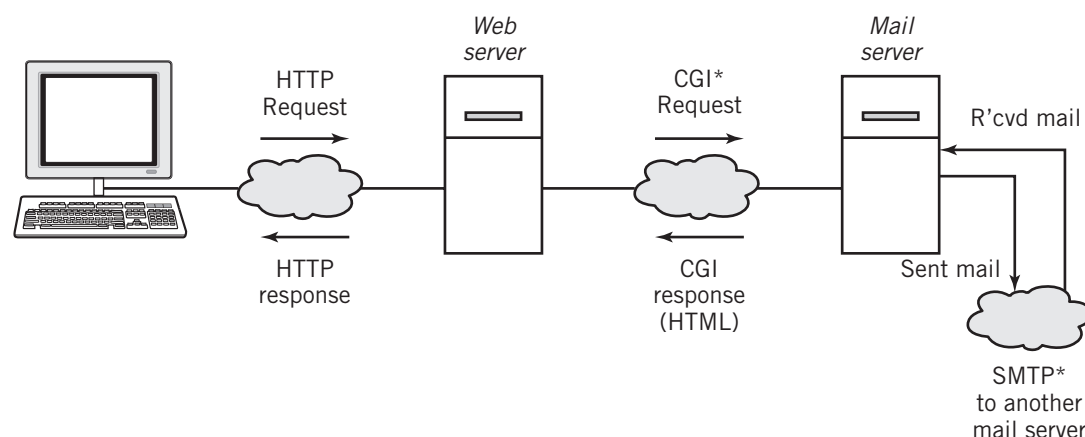
Since Web technology is based on a client-server model, it requires only a simple extension of the n -tier architecture to implement Web-based applications. As an example, Figure 2.9 shows a possible system architecture to implement Web-based e-mail. Note the similarity between this example and the three-tier database application shown in Figure 2.8.

Many organizations also now find it possible and advantageous to create system architectures that integrate parts of their systems with other organizations using Web technology and Web standards as the medium of communication. For example, an organization can integrate and automate its purchasing system with the order system of its suppliers to automate control of its inventory, leading to reduced inventory costs, as well as to rapid replacement and establishment of reliable stocks of inventory when they are needed. Internet standards such as XML allow the easy identification of relevant data within data streams between interconnected systems, making these applications possible and practical. This type of automation is a fundamental component of modern business-to-business operations.

PEER-TO-PEER COMPUTING An alternative to client-server architecture is **peer-to-peer architecture**. Peer-to-peer architecture treats the computers in a network as equals, with the ability to share files and other resources and to move them between computers. With appropriate permissions, any computer on the network can view the resources of any other computer on the network, and can share those resources. Since every computer is essentially independent, it is difficult or impossible to establish centralized control to restrict inappropriate access and to ensure data integrity. Even where the integrity of the system can be assured, it can be difficult to know where a particular file is located and no assurance that the resource holding that file is actually accessible when the file is needed. (The particular computer that holds the file may be turned off.) The system also may have several versions of the file, each stored on a different computer. Synchronization of different file versions is difficult to control and difficult to maintain. Finally, since

FIGURE 2.9

Three-Tier Web-Based E-Mail Architecture



*SMTP: Simple Mail Transfer Protocol

*CGI: Common Gateway Interface

data may pass openly through many different machines, the users of those machines may be able to steal data or inject viruses as the data passes through. All of these reasons are sufficient to eliminate peer-to-peer computing from consideration in any organizational situation where the computers in the network are controlled by more than one individual or group. In other words, nearly always.

There is one exception: peer-to-peer computing is adequate, appropriate, and useful for the movement of files between personal computers or to share a printer in a small office or home network.

Peer-to-peer technology has also proven viable as an Internet file sharing methodology outside the organizational structure, particularly for the downloading of music and video. The perceived advantage is that the heavy loads and network traffic associated with a server are eliminated. (There are legal ramifications, also, for a server that is sharing copyrighted material illegally.) This technique operates on the assumption that the computer searching for a file is able to find another computer somewhere by broadcasting a request across the Internet and establishing a connection with a nearby computer that can supply the file. Presumably, that computer already has established connections with other systems. All of these systems join together into a peer-to-peer network that can then share files. One serious downside to this approach, noted above, is the fact that the computers in an open, essentially random, peer-to-peer network can also be manipulated to spread viruses and steal identities. There are several serious documented cases of both.

An alternative, hybrid model uses client-server technology to locate systems and files that can then participate in peer-to-peer transactions. The hybrid model is used for instant messaging, for Skype and other online phone systems, and for Napster and other legal file download systems.

Although there have been research studies to determine if there is a place for peer-to-peer technology in organizational computing, the security risks are high, the amount of control low, and the overall usefulness limited. The results to date have been disappointing.

The Role of the System Architect

In Section 2.1, we suggested that there are different ways of viewing systems. From the discussion within this section, you can see that the IT system architect must consider the system from the perspectives of application architecture, data architecture, network architecture, and computer architecture. Each of these addresses different aspects of the IT system as a whole. For example, our consideration of different general application architectures—client-server, web-based architecture, peer-to-peer architecture—ignored the networking that links the various computers together. Similarly, we attempted to minimize the effects due to the specifics of individual computer systems when exploring the various requirements of a system from the perspective of application architecture.

Ultimately, it is the responsibility of the system architect to assess the particular needs of an organization and create a system that meets those needs while attempting to achieve an optimum balance of computer power, network capability, user convenience, and budget. To do so, the architect will consider each aspect of the system: application architecture, network requirements, specification of computer systems, and data requirements, just as the architect designing a house considers flow of people through the house, overall use of

space and room layout, individual room layouts, mechanical systems, and aesthetic design as different views of the overall architecture of the house.

Although the infrastructure design as defined by the computer hardware, system software, and communication channels is subordinate to the fundamental business requirements that determine a basic IT system architecture, the system architect must understand the features and constraints that establish the feasibility and desirability of a particular infrastructure configuration.

Google: A System Architecture Example

So far, we have considered basic system concepts and simple system architectures as examples. Most IT business systems operate primarily within an organization, with limited collaboration with other, partnered organizations and carefully controlled public access. At the opposite extreme are massive systems that are widely open to the public. Google offers a primary example of such a system.

The primary mission of Google is to provide powerful, fast search capability of material on the Internet for billions of users all over the world. Income to the organization is provided from advertising that is targeted to each user based on the specific nature of the user's search. The design of Google's IT system architecture is obviously fundamental to Google's ability to achieve its mission and to meet reasonable income goals. In keeping with the focus of this book, our primary interest is in the computer and network architectures that Google uses to meet its system requirements; however we will use this example to explore the relationship between the basic system requirements, the IT system architecture created to meet those requirements, and the specific computer and network architectures that evolved from the system architecture.

Some of the basic requirements that the Google IT system must satisfy include the following:

- It must be capable of responding to millions of simultaneous requests from all over the world with pertinent, ranked search results and appropriately targeted advertising. Most desirably, the results and advertising would be matched in language, geographic suitability, and culture as much as possible to the location of the user.
- The system must be able to troll the Internet systematically and thoroughly to retrieve data and to organize the data in such a way as to make it readily available for response to user requests. There must be a processing mechanism to establish a ranking of the results to a request.
- The system must respond to requests with a reliability as near to 100 percent as is technically possible. Individual hardware and software component failures within the system must not affect system performance adversely.
- The system must be easily scalable to handle ever-increasing numbers of requests and must be cost effective.

At the application level, the requirements identify three specific processing tasks that the system must fulfill:

1. The system must accept search requests from users, identify and rank matches, create a Web page, and serve it to the user.

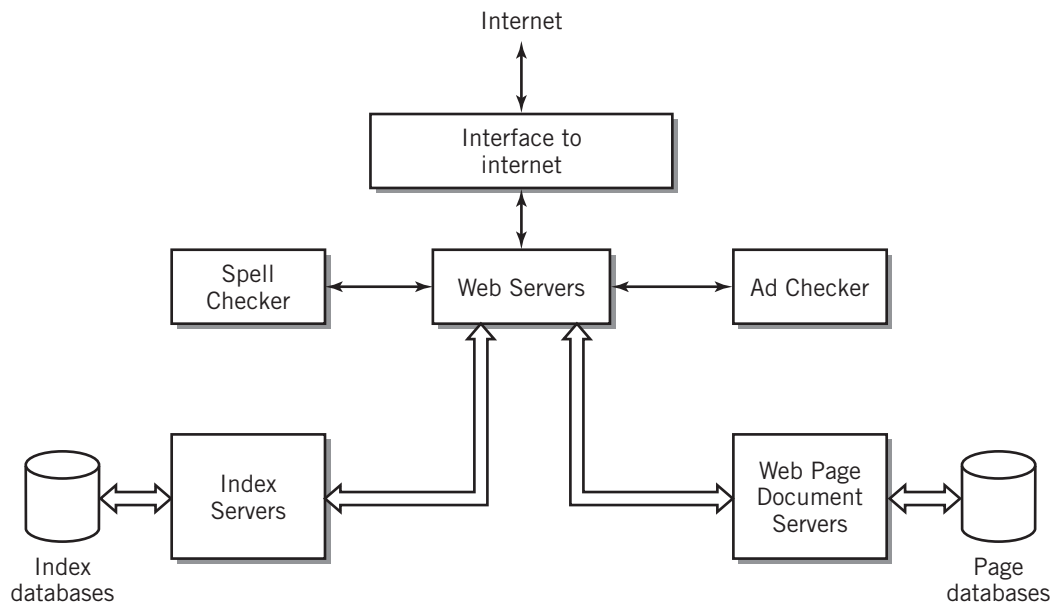
2. The system must collect data—lots of data! This task “crawls the Web”, identifies the search terms (every significant word) on every Web page it encounters, and maintains an index database connecting each term to the corresponding page. It likewise stores every Web page in a Web page database and assigns a ranking value to each entry.
3. The system must manage advertisements, identify appropriate advertisements in response to user search requests, and make the advertisements available to the Web page creation application mentioned in 1.

For this discussion we will focus on the processing of search requests. When a user types the Google URL `www.google.com` into her browser, the Web browser uses a service called *Domain Name Service (DNS)* to identify the IP address of the Web server to which the request is to be sent. Because Google must be able to handle several million requests per hour, Google provides a number of alternative IP addresses representing different sites to which the request may be redirected. Based on the approximate location from which the request was sent, the request is routed by DNS to a Google data center near that location. Google maintains more than forty separate data centers around the world to serve user requests.

A simplified system diagram of the application architecture for a Google data center is shown in Figure 2.10. All of the data centers are architecturally identical, differing only in such details as the number of processors and the hardware specifications for each processor. Each data center processes requests independently. Multiple copies of all of the

FIGURE 2.10

Google Data Center Search Application Architecture



index word data and Web page data are stored locally at every data center, and updated from master data at regular intervals.

A request enters the system from the Internet and is distributed to a Google Web server for processing. A request consists of words and phrases. There are many separate Web servers available so that many requests can be processed in parallel. The words are passed to a spell checker, to an ad server, and to a pool consisting of a large number of index servers.

The spell checker checks each word and considers possible alternatives if it believes that the user may have intended something different. When appropriate, the output of the spell checker will become part of the response sent to the user. (*“Did you mean . . . ”* is familiar to most Google users.) The ad checker searches for words in the advertising database that match the user’s request and adds the corresponding advertisement(s) to the material that will be used to create the response page.

The index servers look up each word from the request in the index database and compile a list of matching pages for each word. The list is then adjusted for multiple words and phrases and sorted in order of relevance, based on Google’s ranking algorithms. This list is then passed back to the Web server.

Next, the Web server calls upon the document servers to look up each matching page in the Web page database. The document servers return a URL, a title, and a short snippet of text for each document to the Web server. Finally, the Web server creates an HTML document from the spelling, ad, and matching page results and returns the page to the user’s Web browser.

Although the application processing just described is relatively straightforward, the implementation of this system presented a number of challenges to the system architects. The index and document databases are both massive in size. Many searches will result in a large number of “hits”; each hit must be evaluated and ranked. Each hit requires retrieval and processing of a separate page from the document database. All of this processing must occur very quickly. And the numbers of searches occurring simultaneously may also be extremely large.

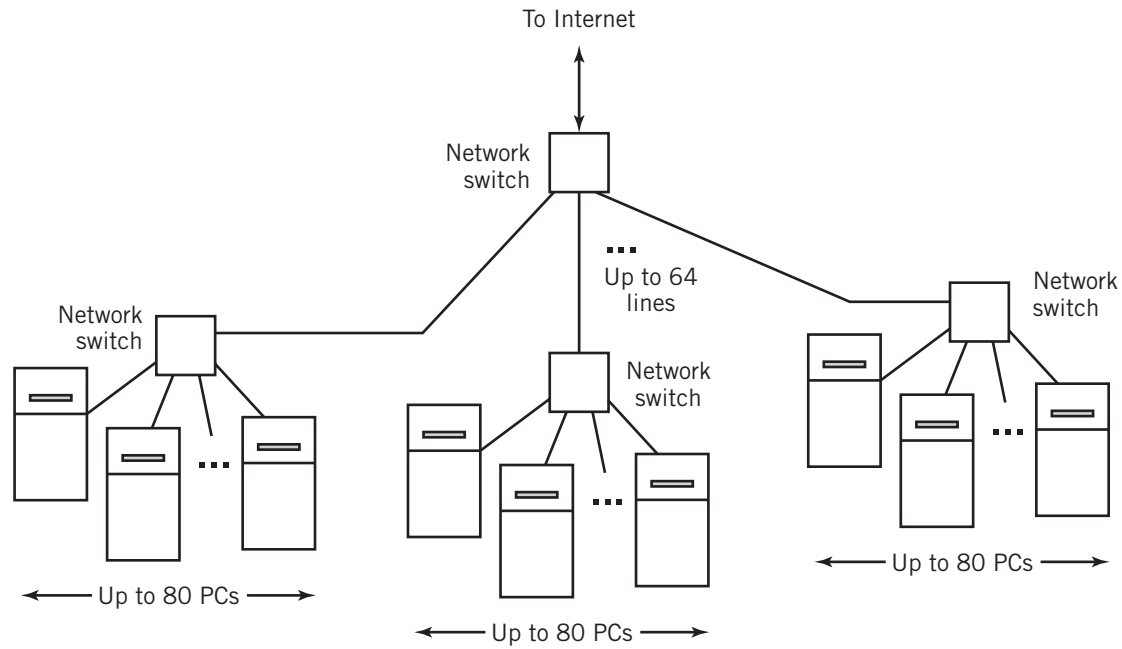
Google’s system architects responded to these challenges by recognizing that each search could be processed independently on a separate computer, except for certain bottlenecks. For example, each search request arriving from the Internet could be steered by a computer to a different Web browser. They also observed that the major bottleneck was the time required to access the databases on disks, which had to be shared among all the searches taking place. Since the data in the databases never changed as a result of a search, however, they reasoned that the databases could also be replicated and accessed in parallel.

A simplified hardware representation of their solution is shown in Figure 2.11. Groups of up to eighty computers are connected together in a network, then these networks, up to sixty-four of them, are, themselves, connected together to form a larger network, sort of like a miniature Internet of up to 5,120 computers. (There are additional switches and connections built in for reliability that are not shown in the diagram.) Each computer acts as a server, with different computers assigned to different pieces of the application architecture. Each data center is equipped similarly.

Although the computers are manufactured specifically for Google, they are essentially inexpensive commodity PCs, similar to standard, medium power, non-state-of-the-art,

FIGURE 2.11

Simplified Google System Hardware Architecture



off-the-shelf PCs. Each computer has a fairly large, but still off-the-shelf, hard disk. The index and document databases are divided up among the hard disks on many computers. (Google calls these partial databases *shards*.) This design allows different searches to access different parts of the databases simultaneously. There are multiple copies of each database, so that the failure of a PC or hard disk does not affect the overall ability of the system to conduct searches. Each computer runs standard Linux operating system software, but the application software was specially written by Google programmers.

Overall, this design allows a large number of searches to progress in parallel. The use of inexpensive PC hardware makes the solution cost-effective. The system can be scaled easily by adding more computers. Finally, the failure of a PC does not result in failure and, in fact, has minimal effect on the performance of the system overall. Thus, this solution meets the original requirements admirably. It is worth noting that a fundamental understanding of computer infrastructure was key to the system architects' solution.

This discussion provides a simple overview of the Google system. Hopefully you found even this brief look at the Google system interesting and informative. There are a number of other considerations in the Google system architecture that we have glossed over for now. However, to understand the Google architecture better, it is first necessary to continue our exploration of the hardware, software, and network components that make up the Google system, as well as every other IT system. We will return for a more in-depth discussion of the Google system architecture in Supplementary Chapter 2.

SUMMARY AND REVIEW

When working with large concepts with defined boundaries, it is often easiest to think of them in terms of systems. A system can be defined as a collection of components, linked together and organized in such a way as to be recognizable as a single unit. The components themselves may also be recognized as subsystems, to be further reduced into components, when appropriate. The area outside the boundaries of a system is its environment. The system affects and is affected by various elements of the environment. In many situations, the environment supplies inputs to the system and receives outputs from the system. The patterns of relationships, connections, constraints, and linkages among the components of a system and between a system and its environment are known collectively as the architecture of the system.

Information technology systems are systems that support the strategy and operations of organizations. The technological components of an IT system include computer hardware, application software, operating system software, networks, and data. Other components include personnel, policies, and more.

There are a number of different ways of viewing an IT system, including application architecture, network architecture, software architecture, and hardware architecture. The general architecture for an IT system includes all of these considerations.

Nearly all modern IT systems rely on distributed processing. Data comes from many sources and information is required by users distributed throughout an organization and beyond. The most common application architecture to support distributed processing is client-server architecture, in which server computer systems provide various services—Web, database, file, print, processing—to client computer systems. Client-server systems are convenient for users and offer centralized control for the organization. Client-server architecture is commonly organized in tiers, ranging from two-tier to n -tier. The alternative architecture to client-server computing, peer-to-peer computing, is used outside of organizations as a means for sharing files over the Internet, but is of limited use in organizational settings due to difficulties in establishing stable data sources, security risks, and lack of central control. It is also possible to create a hybrid architecture, with features from both client-server and peer-to-peer computing.

A specific type of client-server architecture, Web-based computing, predominates the IT scene, primarily because users are generally familiar with the use of Web browsers, the technology is standardized and already in use in most organizations, and good development tools for designing Web pages and accessing data are readily available. Both intranets and the Internet provide user access.

Protocols are the means used to communicate between computers. IT system protocols of interest to us include network protocols such as TCP/IP, I/O protocols such as USB and PCI-Express, and application protocols such as HTTP. Standards make it possible for different system components to work together. Most modern standards are global. There are standards that are defined by interested groups and de facto standards that arise from common usage.

The first step in IT system analysis and design is about finding an appropriate architecture for a particular business situation. The task can be difficult and challenging. It is easy to see why system architects need a deep understanding of the computer system

and network components that comprise the modern IT system to make the appropriate design, selections, and tradeoffs.

Hopefully this short but concentrated chapter has prepared you for the remainder of the book, which considers in detail the data, computer system hardware, operating systems, and networks that make up the technological infrastructure of an IT system.

FOR FURTHER READING

Surprisingly, there are few books that discuss system concepts and system architecture in a truly general way. Most books that claim to be about system architecture are actually specific to a particular field, usually the field of information systems. One general book about systems is by Laszlo [LASZ96]. Some IS systems design and analysis textbooks provide a brief introduction to general system concepts. (Unfortunately, many don't!) One example of a book that provides a good introduction to system concepts is Stampf [STAM05]. Chapter 1 of Stampf covers the topics in this chapter well. Wikipedia offers other references under the topic *system*.

KEY CONCEPTS AND TERMS

abstraction	interface	subsystem
application architecture	intranet	system
architecture	middleware	three-tier architecture
client-server (model)	<i>n</i> -tier architecture	top-down approach
decomposition	peer-to-peer architecture	two-tier architecture
environment	shared server	

READING REVIEW QUESTIONS

- 2.1 What are the most important ideas, keywords, and phrases that are stated in the definition of a system?
- 2.2 Explain the relationships among the following words: system, environment, boundary, interface.
- 2.3 Explain the following statement about systems: "Decomposition is inherently hierarchical."
- 2.4 Explain what is meant by the architecture of a system.
- 2.5 What does the top-down approach allow a system architect to do that might be more difficult otherwise?
- 2.6 What is the primary concern of application architecture? Give an example of application architecture, either your own, or one from the examples in the book. Explain how this example fulfills the features and requirements of the concept of application architecture.
- 2.7 Most modern computing in organizations is based on client-server models. Explain why this tends to be the case. Give an example of client-server computing that you

are familiar with and explain the characteristics of your example that fulfill the concept of client-server computing.

- 2.8 Web-based system architecture is a popular approach to many organizational systems because it offers a number of advantages to the users and to the organization over other types of systems. Discuss the primary advantages to this approach.
- 2.9 What are the principal responsibilities of a system architect?
- 2.10 Many system architects base their IT system designs on an n -tier architecture, where n is a number with value 2 or greater. Explain the difference between a single-tier architecture and an n -tier architecture. What are the main advantages claimed for an n -tier architecture?

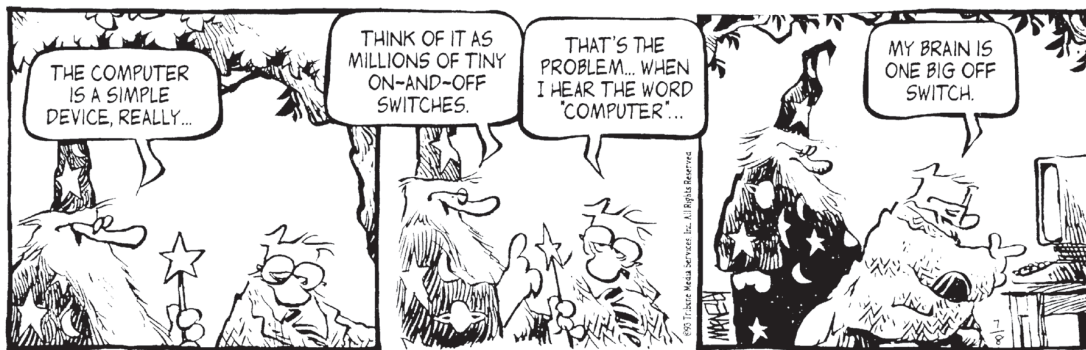
EXERCISES

- 2.1 The human body is an example of an object that can be represented as a system. Consider the various ways in which you could represent the human body as a system. Select a representation and identify the components that constitute the system. Select one component and decompose it to the next subsystem level. Now consider a completely different system representation of the human body and repeat this exercise.
- 2.2 Consider a representation of a work organization or school with which you are familiar. Identify the major components that characterize the primary operations within the organization and draw a diagram that represents the system's organization. Show and identify the links that connect the various components. Identify the major environmental factors that impact the organization.
- 2.3 Consider this textbook. Using the detailed table of contents as a reference, we can represent this textbook as a hierarchical system. As a first pass, we can define this book by the five component *parts* that make up the body of the text. Identify by general name the objects that constitute the next level of decomposition below the *parts* components. Continue to do this for at least three more levels of the hierarchy.
- 2.4 Thinking in terms of systems allows us to analyze situations that are too complicated for us to understand as a whole. What specific characteristics and features of system thinking make this possible?
- 2.5 Figure 2.8 illustrates the basic architecture for a three-tier database system. This system can be viewed as an IPO (input-processing-output) system. What is the input for this system? What environmental element generates the input? (Hint: the Web browser computer is within the system boundary.) What is the expected output from this system? What environmental element receives the output? Briefly describe the processing that takes place in this system.
- 2.6 Based on the illustration of an iPhone shown in Figure 2.4, draw a system model for an iPhone.
- 2.7 It is common to represent network connections in IT systems as a cloud. (See, for example, Figures 2.6, 2.7, 2.8, and 2.9). The cloud is obviously an *abstraction* as we defined abstraction in this chapter. What does the cloud abstraction actually represent?

- 2.8 Suppose that you have been hired to develop a website-based sales system for a large international retail sales firm. Discuss some environmental issues that are specific to the Web design of your system that you must consider if your system is to be successful at attracting and keeping purchasing customers.
- 2.9 Consider a home theatre system consisting of a television set, a receiver, a DVD player, speakers, and any other components you wish to include. Draw a system diagram for this system. Include both components and links. What are the inputs to this system? What are the outputs? (Remember that the DVD player and receiver are both components *within* the system.) Now draw a system diagram for the receiver subsystem. Include both its primary components and the links between them. What are the inputs and outputs for the receiver subsystem? Do these inputs and outputs conform to the links connected to the receiver in your original system diagram?

CHAPTER 7

THE CPU AND MEMORY



SHOE © MACNELLY. DISTRIBUTED BY KING FEATURES SYNDICATE

7.0 INTRODUCTION

The previous chapter provided a detailed introduction to the Little Man model of a computer. In that chapter we introduced a format, using a three-digit number divided into op code and address fields, for the instructions that a computer can perform. We introduced an instruction set that we indicated was representative of those found in a real computer. We also showed the steps that are performed by the Little Man in order to execute one of these instructions.

In this chapter and the next we will extend these concepts to the real computer. Our primary emphasis in this chapter is on the central processing unit (CPU), together with memory. In the real computer, memory is actually separated both physically and functionally from the CPU. Memory and the CPU are intimately related in the operation of the computer, however, and so we will treat memory together with the CPU for the convenience of our discussion. Since every instruction requires memory access,¹ it makes sense to discuss the two together.

We will use the Little Man model and its instruction set as a guideline for our discussion. The Little Man instruction set is fundamentally similar to the instruction sets of many different computers. Of course, the Little Man instruction set is based on a decimal number system, and the real CPU is binary, but this is a detail that won't concern us for most of this discussion. The CPU architectural model that we shall discuss is not based on a particular make and model, but is typical of most computers. Chapter 8 will discuss the implementation of this model in modern technology. In Supplementary Chapter 2, we shall look specifically at several popular computer models.

In this chapter you will see that the execution of instructions in the CPU together with memory is nearly identical functionally to the Little Man Computer. There is a one-to-one relationship between the various contents of the mailroom and the functional components of the CPU plus memory. The major differences occur in the facts that the CPU instruction set is created using binary numbers rather than decimal and that the instructions are performed in a simple electronic way using logic based upon Boolean algebra instead of having a Little Man running around a mailroom.

Sections 7.1 through 7.3 present a systematic introduction to the components of the CPU and memory, offering a direct comparison with the components of the Little Man Computer, and focusing on the concept of the register as a fundamental element of CPU operation. In Section 7.4, we show how simple CPU and memory register operations serve as the basic mechanism to implement the real computer's instruction set.

In Section 7.5, we turn our attention to the third major computer system component, the bus component. Buses provide the interconnection between various internal parts of the CPU, and between the CPU and memory, as well as providing

¹Recall that in the LMC every instruction must be fetched from a mailbox to be executed. The same is true in the real computer.

connections between input and output devices, the CPU, and memory. There are many different types of buses in a computer system, each optimized for a different type of task. Buses can connect two components in a point-to-point configuration or may interconnect several modules in a multipoint configuration. In general, the lines on buses carry signals that represent data, addresses, and control functions. We consider the general requirements for a bus, the features, advantages and disadvantages of different types of buses. In Chapter 11, we will focus on the specific buses that interconnect the various components of a computer system, and show you the ways in which the buses connect different parts of an entire computer system together.

In Sections 7.6, 7.7, and 7.8, we return our attention to the CPU to discuss the characteristics and features of the instruction sets provided in real computers: the different types of instructions, the formats of instruction words, and the general requirements and restraints that are required for instruction words.

You already understand from Chapter 6 how simple instructions can be combined to form the programs that you write. When you complete this chapter, you will have a good understanding of how those instructions are executed in a computer.

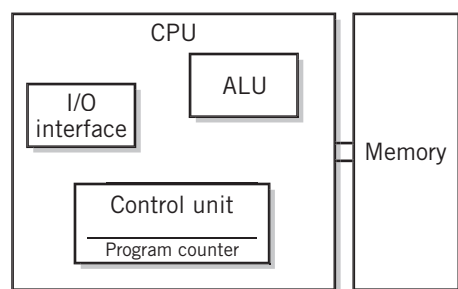
7.1 THE COMPONENTS OF THE CPU

A simplified conceptual block diagram of a CPU with memory is shown in Figure 7.1.² For comparison purposes, the block diagram for the Little Man Computer is repeated in Figure 7.2, with labels corresponding to the components in Figure 7.1.

Note the similarities between the two figures. As noted in Chapter 1, the computer unit is made up conceptually of three major components, the **arithmetic/logic unit (ALU)**, the **control unit (CU)**, and **memory**. The ALU and CU together are known as the **central processing unit (CPU)**. An input/output (I/O) interface is also included in the diagram. The I/O interface corresponds in function roughly to the input and output baskets, although its implementation and operation differ from that of the Little Man Computer in many respects.

FIGURE 7.1

System Block Diagram



The arithmetic/logic unit is the component of the CPU where data is held temporarily and where calculations take place. It corresponds directly to the calculator in the Little Man Computer.

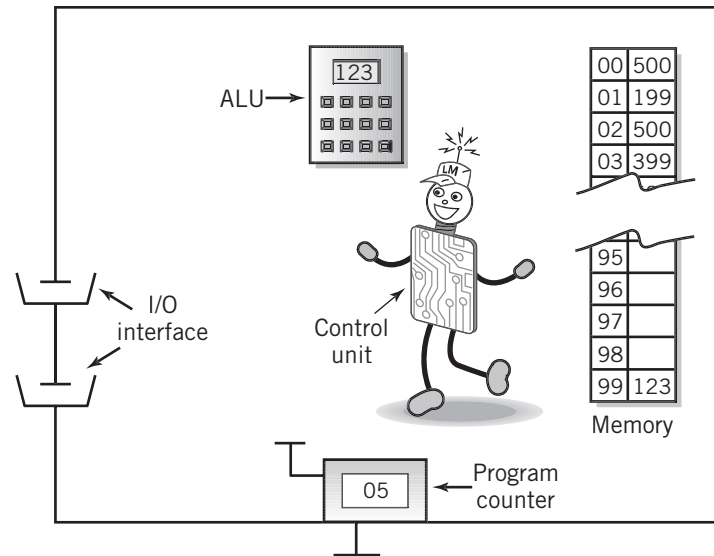
The control unit controls and interprets the execution of instructions. It does so by following a sequence of actions that correspond to the fetch-execute instruction cycle that was described in the previous chapter. Most of these actions are retrievals of instructions from memory followed by movements of data or addresses from one part of the CPU to another.

The control unit determines the particular instruction to be executed by reading the contents of a **program**

²This diagram is first attributed to John von Neumann in 1945. As discussed in Chapter 8, current technology results in a different physical layout for the components in the model; nevertheless, the basic execution of instructions is still consistent with the original model.

FIGURE 7.2

The Little Man Computer



counter (PC), sometimes called an **instruction pointer**, which is a part of the control unit. Like the Little Man's location counter, the program counter contains the address of the current instruction or the next instruction to be executed. Normally, instructions are executed sequentially. The sequence of instructions is modified by executing instructions that change the contents of the program counter. The Little Man branch instructions are examples of such instructions. A **memory management unit** within the control unit supervises the fetching of instructions and data from memory. The I/O interface is also part of the control unit. In some CPUs, these two functions are combined into a single **bus interface unit**. The program counter in the CPU obviously corresponds to the location counter in the Little Man Computer, and the control unit itself corresponds to the Little Man.

Memory, of course, corresponds directly to the mailboxes in the LMC.

7.2 THE CONCEPT OF REGISTERS

Before we discuss the way in which the CPU executes instructions, it is necessary to understand the concept of a register. A **register** is a single, permanent storage location within the CPU used for a particular, defined purpose. A register is used to hold a binary value temporarily for storage, for manipulation, and/or for simple calculations. Note that each register is wired within the CPU to perform its specific role. That is, unlike memory, where every address is just like every other address, each register serves a particular purpose. The register's size, the way it is wired, and even the operations that take place in the register reflect the specific function that the register performs in the computer.

Registers also differ from memory in that they are not addressed as a memory location would be, but instead are manipulated directly by the control unit during the execution of instructions. Registers may be as small as a single bit or as wide as several bytes, ranging usually from 1 to 128 bits.

Registers are used in many different ways in a computer. Depending on the particular use of a register, a register may hold data being processed, an instruction being executed, a memory or I/O address to be accessed, or even special binary codes used for some other purpose, such as codes that keep track of the status of the computer or the conditions of calculations that may be used for conditional branch instructions. Some registers serve many different purposes, while others are designed to perform a single, specialized task. There are even registers specifically designed to hold a number in floating point format, or a set of related values representing a list or vector, such as multiple pixels in an image.

Registers are basic working components of the CPU. You have already seen, in Chapter 6, that the computer is unable to distinguish between a value that is used as a number in a program and a value that is actually an instruction or address, except in the context of current use. When we refer to the “data” in a register, we might be talking about any of these possibilities.

You have already become acquainted with two “registers” in the Little Man Computer, namely, the calculator and the location counter.

In the CPU, the equivalent to the calculator is known as an **accumulator**. Even the short example to add two numbers in Chapter 6 showed that it is often necessary to move data to and from the accumulator to make room for other data. As a result, modern CPUs provide several accumulators; these are often known as **general-purpose registers**. Some vendors also refer to general-purpose registers as **user-visible** or **program-visible registers** to indicate that they may be accessed by the instructions in user programs. Groups of similar registers are also sometimes referred to collectively as a **register file**. General-purpose registers or accumulators are usually considered to be a part of the arithmetic/logic unit, although some computer manufacturers prefer to consider them as a separate register unit. As in the Little Man Computer, accumulator or general-purpose registers hold the data that are used for arithmetic operations as well as the results. In most computers, these registers are also used to transfer data between different memory locations, and between I/O and memory, again similar to the LMC. As you will see in Chapter 8, they can also be used for some other similar purposes.

The control unit contains several important registers.

- As already noted, the **program counter register** holds the address of the current instruction being executed.
- The **instruction register (IR)** holds the actual instruction being executed currently by the computer. In the Little Man Computer this register was not used; the Little Man himself remembered the instruction he was executing. In a sense, his brain served the function of the instruction register.
- The **memory address register (MAR)** holds the address of a memory location.
- The **memory data register (MDR)**, sometimes known as the *memory buffer register*, will hold a data value that is being stored to or retrieved from the memory location currently addressed by the memory address register.

The last two registers will be discussed in more detail in the next section, when we explain the workings of memory. Although the memory address register and memory data register are part of the CPU, operationally these two registers are more closely associated with memory itself.

The control unit will also contain several 1-bit registers, sometimes known as **flags**, that are used to allow the computer to keep track of special conditions such as arithmetic carry and overflow, power failure, and internal computer error. Usually, several flags are grouped into one or more **status registers**.

In addition, our typical CPU will contain an I/O interface that will handle input and output data as it passes between the CPU and various input and output devices, much like the LMC *in* and *out* baskets. For simplification, we will view the I/O interface as a pair of I/O registers, one to hold an I/O address that addresses a particular I/O device, the other to hold the I/O data. These registers operate similarly to the memory address and data registers. Later, in Chapter 9, we will discuss a more common way of handling I/O that uses memory as an intermediate storage location for I/O data.

Most instructions are executed by the sequenced movement of data between the different registers in the ALU and the CU. Each instruction has its own sequence.

Most registers support four primary types of operations:

1. Registers can be loaded with values from other locations, in particular from other registers or from memory locations. This operation destroys the previous value stored in the destination register, but the source register or memory location remains unchanged.
2. Data from another location can be added to or subtracted from the value previously stored in a register, leaving the sum or difference in the register.
3. Data in a register can be shifted or rotated right or left by one or more bits. This operation is important in the implementation of multiplication and division. The details of the shift operation are discussed in Section 7.6.
4. The value of data in a register can be tested for certain conditions, such as zero, positive, negative, or too large to fit in the register.

In addition, special provision is frequently made to load the value zero into a register, which is known as clearing a register, and also to invert the 0s and 1s (i.e., take the 1's complement of the value) in a register, an operation that is important when working with complementary arithmetic. It is also common to provide for the addition of the value 1 to the value in a register. This capability, which is known as incrementing the register, has many benefits, including the ability to step the program counter, to count in for loops, and to index through arrays in programs. Sometimes decrementing, or subtraction of 1, is also provided. The bit inversion and incrementing operations are combined to form the 2's complement of the value in a register. Most computers provide a specific instruction for this purpose, and also provide instructions for clearing, inverting, incrementing, and decrementing the general-purpose registers.

The control unit sets ("1") or resets ("0") status flags as a result of conditions that arise during the execution of instructions.

As an example, Figure 7.3 identifies the programmer-accessible registers in the IBM System z computers, which includes a variety of IBM mainframe models. Internal registers, such as the instruction, memory address, and memory buffer registers are not specifically

FIGURE 7.3

Programmer-Accessible Registers in IBM zSeries Computers

Register type	Number	Size of each in bits	Notes
General	16	64	For arithmetic, logical, and addressing operations; adjoining registers may be joined to form up to eight 128-bit registers
Floating point	16	64	Floating point arithmetic; registers may be joined to form 128-bit registers
PSW	1	128	Combination program counter and status-flag register, called the Program Status Word (PSW)
Control (+1 32-bit floating point control)	16	64	Various internal functions and parameters connected with the operating system; accessible only to systems programmers

identified in the table, since they are dependent on the implementation of the particular model in the series.

7.3 THE MEMORY UNIT

The Operation of Memory

To understand the details of instruction execution for the real CPU, you need first to see how instructions and data can be retrieved from memory. Real memory, like the mailboxes in the Little Man Computer, consists of cells, each of which can hold a single value, and each of which has a single address.

Two registers, the memory address register and the memory data register, act as an interface between the CPU and memory. The memory data register is called the memory buffer register by some computer manufacturers.

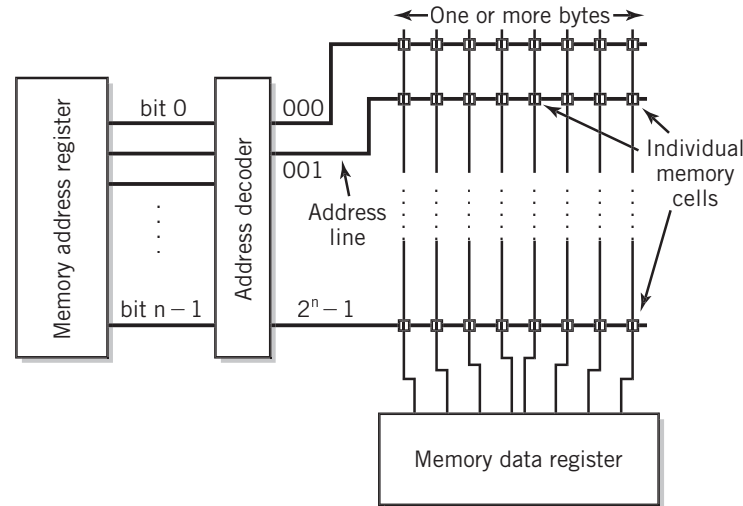
Figure 7.4 is a simplified representation of the relationship between the MAR, the MDR, and memory. Each cell in the memory unit holds 1 bit of data. The cells in Figure 7.4 are organized in rows. Each row consists of a group of one or more bytes. Each group represents the data cells for one or more consecutive memory addresses, shown in the figure as addresses 000, 001, . . . , $2^n - 1$.

In modern computers, it is common to address 8 bytes at a time to speed up memory access between the CPU and memory. The CPU can still isolate individual bytes from the group of eight for its use, however.

The memory address register holds the address in the memory that is to be “opened” for data. The MAR is connected to a decoder that interprets the address and activates a single address line into the memory. There is a separate address line for each group of cells in the memory; thus, if there are n bits of addressing, there will be 2^n address lines. (In actuality, the decoding process is somewhat more complex, involving several levels of address decoding, since there may be several millions or billions of addresses involved, but the concept described here is correct.)

FIGURE 7.4

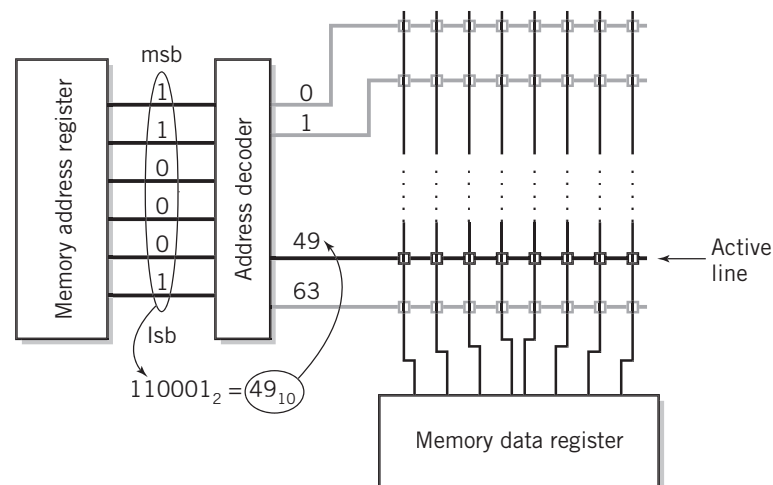
The Relationship Between the MDR, the MAR, and Memory



The memory data register is designed such that it is effectively connected to every cell in the memory unit. Each bit of the MDR is connected in a column to the corresponding bit of every location in memory. However, the addressing method assures that only a single row of cells is activated at any given time. Thus, only one memory location is addressed at any one time. A specific example of this is shown in Figure 7.5. (Note that in the drawing *msb* stands for most significant bit and *lsb* for least significant bit.)

FIGURE 7.5

MAR-MDR Example



As a simple analogy to the operation we've just described, consider the memory as being stored in a glass box, as shown in Figure 7.6. The memory data register has a window into the box. The viewer, who represents each cell in the memory data register, can see the cells in corresponding bit position for every location in memory through the window. The cells themselves are light bulbs that can be turned on (1) or off (0). The output from the memory address register is passed to an address decoder. The output from the address decoder in our analogy consists of a series of lines, each of which can light up the bulbs in a single row of cells. Only one line at a time can be activated—specifically, the one corresponding to the decoded address. The active line will light the bulbs that correspond to “1s,” leaving the “0s” dark. The viewer therefore will see only the single group of cells that is currently addressed by the memory address register. We can extend the analogy to include a “master switch” that controls all the lights, so that the data can be read only at the appropriate instant.

A more detailed picture of an individual memory cell is shown in Figure 7.7. Although this diagram is a bit complicated, it may help to clarify how data is transferred between the MDR and memory. There are three lines that control the memory cell: an address line, a read/write line, and an activation line. The address line to a particular cell is turned on only if the computer is addressing the data within that cell. The read/write line determines whether the data will be transferred from the cell to the MDR (read) or from the MDR to the cell (write). This line works by turning on one of two switches in conjunction with the address line and the activation line. The read switch, R, in the diagram turns on when the

FIGURE 7.6

A Visual Analogy for Memory

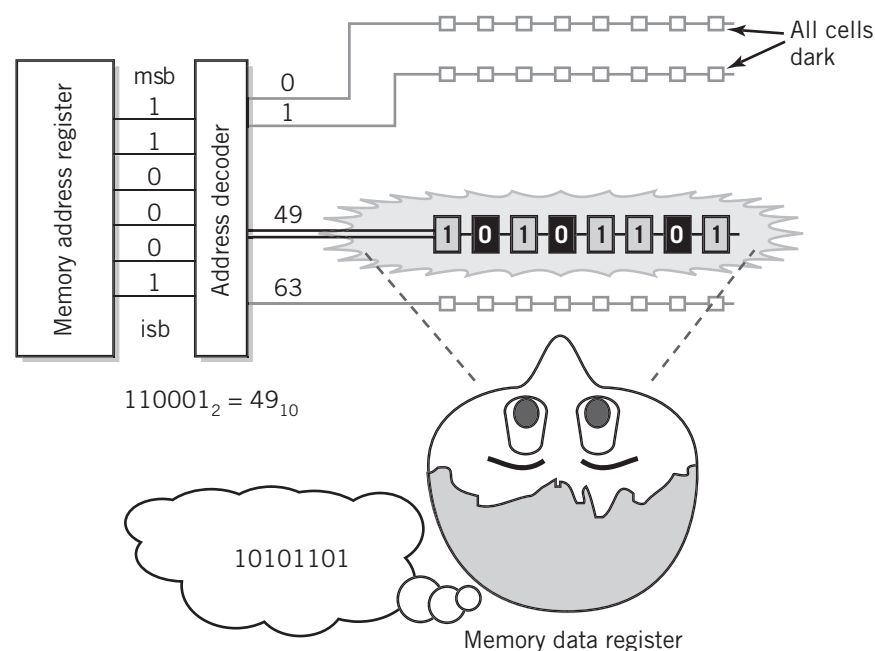
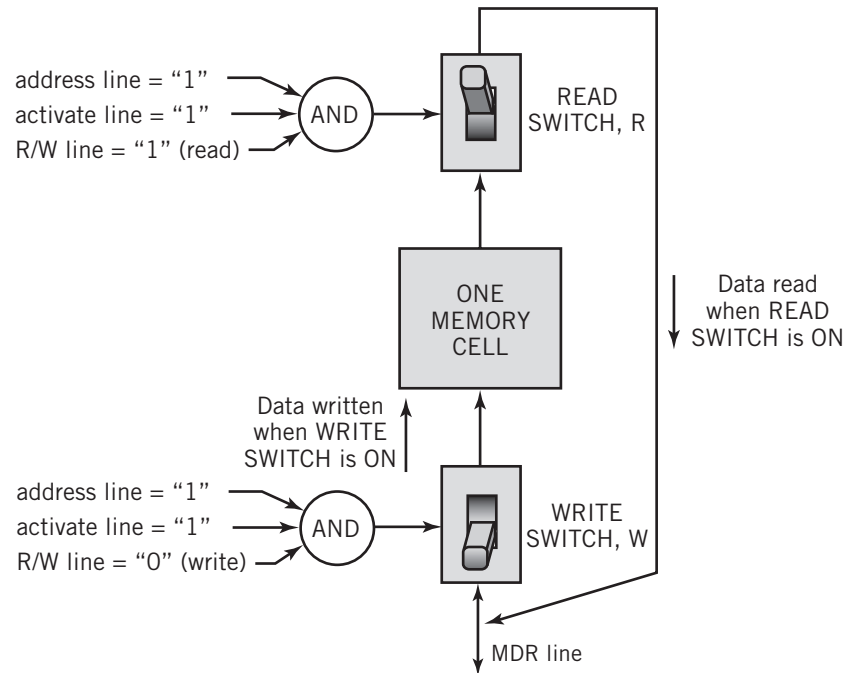


FIGURE 7.7

An Individual Memory Cell



address line and the activation line are both on (on is usually represented by 1, off by 0), and the read/write line is set to read; the switch then connects the output of the cell to the MDR line. The write switch, W, works similarly; switch W turns on when the address line and activation line are both on and the read/write switch is set to write. Switch W connects the MDR line to the input of the cell, which transfers the data bit on the MDR line to the cell for storage. Note that only one switch, at most, can be on at a given time.

The interaction between the CPU and the memory registers takes place as follows: to retrieve or store data at a particular memory location, the CPU copies an address from some register in the CPU to the memory address register. *Note that addresses are always moved to the MAR; there would never be a reason for an address transfer from the MAR to another register within the CPU*, since the CPU controls memory transfers and is obviously aware of the memory address being used. At the same time that the MAR is loaded, the CPU sends a message to the memory unit indicating whether the memory transfer is a retrieval from memory or a store to memory. This message is sent by setting the read/write line appropriately.

At the appropriate instant, the CPU momentarily turns on the switch that connects the MDR with the register by using the activation line, and the transfer takes place between memory and the MDR. The MDR is a two-way register. When the instruction being executed is to store data, the data will be transferred from another register in the CPU to the MDR, and from there it will be transferred into memory. The original data at that

location will be destroyed, replaced by the new data from the MDR. Conversely, when the instruction is to load data from memory, the data is transferred from memory to the MDR, and it will subsequently be transferred to the appropriate register in the CPU. In this case, the memory data are left intact, but the previous data value in the MDR is replaced by the new data from memory.

Memory Capacity

The number of possible memory locations in the Little Man Computer, one hundred locations, was established by the two-digit address space in each instruction. The location counter also addresses one hundred locations. There is no memory address register per se, but the Little Man is certainly aware that each memory location requires two digits. In theory, a larger location counter, say, three digits, would allow the Little Man to fetch more *instructions*, but notice that his *data* fetches and stores are still limited to the one hundred locations that the two digits of the address field in the instruction word can address.

Similarly, there are two factors that determine the capacity of memory in a real computer. The number of bits in the memory address register determines how many different address locations can be decoded, just as the two-digit addresses in the Little Man Computer resulted in a maximum of one hundred mailboxes. For a memory address register of width k bits, the number of possible memory addresses is

$$M = 2^k$$

The other factor in establishing memory capacity is of course the number of bits in the address field of the instruction set, which establishes how many memory locations can be directly addressed from the instruction.

In the Little Man Computer, we have assumed that these two size factors are the same, but in a real computer, that is not necessarily the case. Even if the size of the instruction address field is sufficient to support a larger amount of memory, the number of physical memory locations is, in fact, determined by the size of the memory address register. There are also other ways of extending the addresses specified within instructions so that we can reach more addresses than the size of the instruction address field would allow. Just to give you one common method, consider a computer that can use one of the general-purpose registers to hold an address. To find a memory location, the computer would use the value in that register as a pointer to the address. Instead of an address field, the instruction needs only to indicate which register contains the address. Using this technique, the addressing capability of the computer is determined by the size of the register. For example, a computer with 64-bit registers could address 2^{64} addresses if the MAR were wide enough. Such an extension would suggest that the MAR, and thus the actual memory capacity, is normally at least as large as the instruction address field, but it may be much larger. There is a brief discussion of simple addressing methods in Chapter 8. Additional, more sophisticated addressing methods are presented in Supplementary Chapter 3.

Ultimately, the width of the MAR determines the maximum amount of addressable memory in the computer. Today, a typical memory address register will be at least 32 bits wide, and probably much wider. Many modern CPUs support 64-bit memory addresses. A 32-bit memory address allows a memory capacity of 4 gigabytes (GB) (4×10^9 byte-size spaces), whereas 64 bits allows a memory capacity of 16×10^{18} bytes (16 exabytes or

16 billion gigabytes). In modern computers, the ultimate size of memory is more likely limited by physical space for the memory chips or by the time required to decode and access addresses in a large memory, rather than by the capability of the CPU to address such a large memory.

Of course the size of memory also affects the speed of access. The time needed for the address decoder to identify a single line out of four billion is necessarily larger than that required for a memory that is much smaller.

As an aside, it is worth noting that early models of IBM's largest mainframe computer systems had a total memory capacity of only 512 KB, (1/4000th the memory of a typical modern PC with 2 GB of memory!) and that the original IBM PC came supplied with 64 KB of memory, with a maximum capacity of 640 KB. In fact, Bill Gates, of Microsoft, was quoted at the time as saying that he could see no need for more than 640 KB of memory, ever!

The size of the data word to be retrieved or stored in a single operation is determined by the size of the memory data register and by the width of the connection between memory and the CPU. In most modern computers, data and instructions found in memory are addressed in multiples of 8-bit bytes. This establishes the minimum instruction size as 8 bits. Most instructions cannot fit practically into 8 bits. If one were to allow 3 bits for the op code (eight instruction types), only 5 bits remain for addressing. Five bits allows $2^5 = 32$ different addresses, which is clearly insufficient address space. As a result, longer instructions of 16, 24, 32, or even more bits will be stored in successive memory locations. In the interest of speed, it is generally desirable to retrieve an entire instruction with a single fetch, if possible. Additionally, data to be used in arithmetic calculations frequently requires the precision of several bytes. Therefore, most modern computer memories are designed to allow the retrieval or storage of at least 4 and, more commonly, 8 or even 16, successive bytes in a single operation. Thus, the memory data register is usually designed to retrieve the data or instruction(s) from a sequence of several successive addresses all at once, and the MDR will be several bytes wide.

Primary Memory Characteristics and Implementation

Through the history of computing there have been several different types of primary memory used, reflecting the technology and the system requirements and capabilities of the times. In the 1960s and 1970s, the dominant technology was magnetic core memory, which used a tiny core of magnetic material to hold a bit of data, and the largest machines might have had 512 KB of memory. Today, the primary memory in most computer systems is dynamic **RAM**, and most machines have 1 GB of memory, or more. RAM is an acronym that stands for *random access memory*, which is a slight misnomer, since other types of semiconductor memory can also be accessed randomly (i.e., their addresses can be accessed in any order). A more appropriate name would be *read-write memory*.

Memory today is characterized by two predominant operational factors and by a number of technical considerations. Operationally, the most important memory characteristic is whether the memory is read-write capable or read-only. Almost as important is whether the memory is **volatile** or **nonvolatile**. Nonvolatile memory retains its values when power is removed. Volatile memory loses its contents when power is removed. Magnetic core memory was nonvolatile. RAM is volatile.

Important technical considerations include the speed of memory access, the total amount of memory that can be addressed, the data width, the power consumption and heat generation, and the bit density (specified as the number of bits per square centimeter). Cost is an additional factor.

Most current computers use a mix of static and dynamic RAM for memory. The difference between static and dynamic RAM is in the technical design and is not of importance here. However, **dynamic RAM** is less expensive, requires less electrical power, generates less heat, and can be made smaller, with more bits of storage in a single integrated circuit. Dynamic RAM also requires extra electronic circuitry that “refreshes” memory periodically; otherwise the data fades away after a while and is lost. Static RAM does not require refreshing. **Static RAM** is also faster to access than dynamic RAM and is therefore useful in very-high-speed computers and for small amounts of high-speed memory, but static RAM is lower in bit density and more expensive. Both dynamic and static RAM are volatile: their contents are lost when power is turned off.

At the time of this writing, dynamic RAM is standard for most applications. The amount of data that can be stored in a single dynamic RAM chip has increased rapidly in the past few years, going from 64 Kilobits(Kb) to 512 Megabits (Mb) in fewer than fifteen years. Currently, most systems are built with chips that can hold 256 or 512 Mb of data. These chips are also designed to be packaged together in convenient plug-in packages that can supply 1–2 gigabytes of memory, or more, in a single unit. 1 gigabit (Gb) and 2 Gb RAM chips are also in production, but have not yet replaced 512 Mb chips for most system applications, with the exception of mainframe systems. Most modern systems also provide a small amount of static RAM memory that is used for high-speed access. This memory is known as *cache memory*. The use of cache memory is discussed in Chapter 8.

Although current RAM technology is fast, inexpensive, and efficient, its volatility makes some applications difficult or impossible. For example, nonvolatile RAM would make it possible to shut off a computer without losing the programs and data in memory. This would make it possible to restart the computer into its previous state without rebooting, would eliminate the undesirable effects of power failures and laptop battery discharge, and would simplify the use of computers in situations where power conservation is critical, such as in long distance space missions. The desire for nonvolatile RAM has led to considerable research on alternative technologies for creating and producing nonvolatile RAM.

There are a small number of memory technologies in current use that are capable of nonvolatile random access, but none in current large-scale production is capable of replacing standard SRAM and DRAM for use in primary memory. Foremost among these technologies is **flash memory**. Flash memory uses a concept called *hot carrier injection* to store bits of data. Flash memory allows rewriting of cells by erasing groups of memory cells selectively, and then writing the new pattern into the cells. Flash memory serves as an inexpensive form of nonvolatile storage for portable computer storage, digital cameras, MP3 players, and other electronic devices; however it is unsuitable for primary memory because the rewrite time is extremely slow compared to standard RAM and the number of rewrites over the lifetime of the ROM is somewhat limited. Flash memory is viewed primarily as a potential replacement for slow long-term storage devices such as magnetic disks and CD or DVD devices, although the significantly higher cost of flash memory is still a factor at this point in time.

A number of nonvolatile memory technologies that might be capable of replacing traditional RAM appear to be nearing production. These include magnetorestrictive RAM (MRAM), ferroelectric RAM (FeRAM), phase-change RAM (PRAM), and carbon nano[tube] RAM (NRAM). You will probably be reading about one or more of these in the future.

ROM, or *read-only memory*, is used for situations where the software is built semi-permanently into the computer, is required as part of the computer's software, and is not expected to change over the life of the computer, except perhaps very infrequently. Bootstrap programs and basic I/O system drivers fall into this category. Early ROM memory was made up of integrated circuits with fuses in them that could be blown. These fuses were similar to, but much smaller than, the fuses that you might have in your home. A blown fuse might represent a "0," an intact fuse a "1." Modern ROM memories use a different technology, such as **EEPROM** or flash memory. EEPROM (Erasable Electrically Programmable ROM) uses a concept called Fowler-Nordheim tunneling to achieve rewritability. Because of its cost, need for special circuitry, and speed, EEPROM has mostly been replaced by flash memory. Regardless of technology, ROM is nonvolatile. Thus, although electrical power is required to access the data, the data remains consistent with or without power.

7.4 THE FETCH-EXECUTE INSTRUCTION CYCLE

The fetch-execution instruction cycle is *the* basis for every capability of the computer. This seems like a strong statement, but think about it: the purpose of the computer is to execute instructions similar to those that we have already introduced. And, as you've already seen from the Little Man Computer, the operation of every instruction is defined by its fetch-execute instruction cycle. Ultimately, the operation of a computer as a whole is defined by the primary operations that can be performed with registers, as explained in Section 7.2: to move data between registers, to add or subtract data to a register, to shift data within a register, and to test the value in a register for certain conditions, such as negative, positive, or zero.

With the importance of the instruction cycle in mind, we can consider how these few operations can be combined to implement each of the instructions in a computer. The registers that will be of the most importance to us for this discussion will be the general-purpose registers or accumulators used to hold data values between instructions (A or GR), the program counter (PC), which holds the address of the current instruction, the instruction register (IR), which will hold the current instruction while it is being executed, and the memory address and data registers (MAR and MDR), used for accessing memory.

To begin, review carefully the steps that the Little Man took to execute an instruction. (You may want to read Section 6.6 again to refresh your memory.) You will recall that there were two phases in the process. First, the Little Man fetched the instruction from memory and read it. This phase was identical for every instruction. Then, he interpreted the instruction and performed the actions required for that particular instruction.

He repeated this cycle endlessly, until he was given the instruction to stop.

The **fetch-execute instruction cycle** in a CPU works similarly. As noted, much of the procedure consists of copying data from one register to another. You should always be

aware that data copying does not affect the “from” register, but it obviously replaces the previous data in the “to” register with the new data being copied.

Remember that every instruction must be fetched from memory before it can be executed. Therefore, the first step in the instruction cycle always requires that the instruction must be fetched from memory. (Otherwise, how would the computer know what instruction to perform?) Since the address of the current instruction to be executed is identified by the value in the program counter register, the first step will be to transfer that value into the memory address register, so that the computer can retrieve the instruction located at that address.

We will use the following notation to indicate the transfer of a data value from one register to another:

$$\text{REG}_a \rightarrow \text{REG}_b$$

Then, in this notation, the first step in the execution of every instruction will be

$$(\text{step 1}) \text{ PC} \rightarrow \text{MAR}$$

As explained in the description of memory, this will result in the instruction being transferred from the specified memory location to the memory data register. The next step is to transfer that instruction to the instruction register:

$$(\text{step 2}) \text{ MDR} \rightarrow \text{IR}$$

The instruction register will hold the instruction through the rest of the instruction cycle. It is the particular instruction in the IR that will control the particular steps that make up the remainder of the cycle. These two steps comprise the fetch phase of the instruction cycle.

The remaining steps are, of course, instruction dependent. Let us consider the steps required to complete a `LOAD` instruction.

The next thing that the Little Man did was to read the address part of the `LOAD` instruction. He then walked over to the mailbox specified by that address, read the data, and copied it into the calculator. The real CPU will operate similarly, substituting register transfers for the Little Man, of course. Thus,

$$(\text{step 3}) \text{ IR}[\text{address}] \rightarrow \text{MAR}$$

The notation `IR [address]` is used to indicate that only the address part of the contents of the instruction register is to be transferred. This step prepares the memory module to read the actual data that will be copied into the “calculator,” which in this case will be the accumulator:

$$(\text{step 4}) \text{ MDR} \rightarrow \text{A}$$

The CPU increments the program counter, and the cycle is complete and ready to begin the next instruction (actually this step can be performed any time after the previous instruction is retrieved, and is usually performed early in the cycle in parallel with other steps).

$$(\text{step 5}) \text{ PC} + 1 \rightarrow \text{PC}$$

Notice the elegant simplicity of this process! The `LOAD` instruction requires only five steps. Four of the steps simply involve the movement of data from one register to another.

The fifth step is nearly as simple. It requires the addition of the value 1 to the contents of a register, and the new value is returned to the same register. This type of addition is common in computers. In most cases, the result of an addition or subtraction is returned to one of the original registers.

The remaining instructions operate similarly. Compare, for example, the steps required to perform the `STORE` and the `ADD` instructions with those of the `LOAD` instruction, discussed earlier.

The `STORE` instruction

```
PC → MAR
MDR → IR
IR [address] → MAR
A → MDR
PC + 1 → PC
```

The `ADD` instruction

```
PC → MAR
MDR → IR
IR [address] → MAR
A + MDR → A
PC + 1 → PC
```

Study these examples carefully. For practice, relate them to the steps the Little Man performs to execute the corresponding instruction. Notice that the only step that changes in these three instructions is the fourth step.

The fetch-execute cycles for the remaining instructions are left as an exercise (see Exercise 7.5 at the end of this chapter).

The following example, with comments, recaps the above discussion in the context of a three-instruction program segment that loads a number from memory, adds a second number to it, and stores the result back to the first memory location. Note that each instruction is made up of its corresponding fetch-execute cycle. The program segment is executed by processing each step of each fetch-execute cycle in sequence.

Assume that the following values are present just prior to execution of this segment:

```
Program Counter: 65
Value in Mem Location 65: 590 (LOAD 90)
Value in Mem Location 66: 192 (ADD 92)
Value in Mem Location 67: 390 (STORE 90)
Value in Mem Location 90: 111
Value in Mem Location 92: 222
```

EXAMPLE

1st instruction LOAD 90:	PC → MAR	MAR now has 65
	MDR → IR	IR contains the instruction: 590
	----- ←	end of fetch
	IR [address] → MAR	MAR now has 90, the location of the data
	MDR → A	Move 111 from MDR to A
	PC + 1 → PC	PC now points to 66.
----- end of execution, end of first instruction		

2nd instruction ADD 92:	PC → MAR	MAR now contains 66
	MDR → IR	IR contains the instructions: 192
	----- ← end of fetch	
	IR [address] → MAR	MAR now has 92
	A + MDR → A	111+222=333 in A
	PC + 1 → PC	PC now points to 67
	end of execution, end of second instruction	
3rd instruction STORE 90:	PC → MAR	MAR now contains 67
	MDR → IR	IR contains 390
	----- ← end of fetch	
	IR [address] → MAR	MAR now holds 90
	A → MDR	The value in A, 333, moves to mem
		location 90
	PC + 1 → PC	PC now points to 68
	end of execution, end of third instruction	
←	ready for next instruction	

7.5 BUSES

Bus Characteristics

You have already seen that instructions are executed within the CPU by moving “data” in many different forms from register to register and between registers and memory. The different forms that the “data” can take include instructions and addresses, in addition to actual numerical data. “Data” moves between the various I/O modules, memory, and the CPU in similar fashion. The physical connection that makes it possible to transfer data from one location in the computer system to another is called a **bus**. From our previous discussion of the way that the CPU and memory work together, it is probably already obvious to you that there must be a bus of some kind linking the CPU and memory; similarly, buses internal to the CPU can be used to link registers together at the proper times to implement the fetch-execute cycles introduced in Section 7.4.

Specifically, a bus may be defined as a group of electrical, or, less commonly, optical, conductors suitable for carrying computer signals from one location to another. The electrical conductors may be wires, or they may be conductors on a printed circuit. Optical conductors work similarly, using light that is directed from point to point in special thin clear glass fibers. Optical conductors can carry data much faster than electrical conductors, but their cost is high, which has limited their use to date. Nonetheless, there is considerable lab research into ways to integrate more optical circuits into computers.

Buses are used most commonly for transferring data between computer peripherals and the CPU, for transferring data between the CPU and memory, and for transferring data between different points within the CPU. A bus might be a tiny fraction of an inch long, carrying data between various parts of the CPU within an integrated circuit chip; it might be a few inches long, carrying data between the CPU chip and memory; it might even be

hundreds of feet long, carrying data between different computers connected together in a network.

The characteristics of buses are dependent on their particular use within the computer environment. A bus can be characterized by the number of separate wires or optical conductors in the bus; by its *throughput*, that is, the data transfer rate measured in bits per second; by the data width (in bits) of the data being carried; by the number and type of attachments that the bus can support; by the distance between the two end points; by the type of control required; by the defined purpose of the bus; by the addressing capacity; by whether the lines on the bus are uniquely defined for a single type of signal or shared; and by the various features and capabilities that the bus provides. The bus must also be specified electrically and mechanically; by the voltages used; by the timing and control signals that the bus provides, by the protocol used to operate and control the bus, by the number of pins on the connectors, if any; even by the size of the cards that plug into the connector. A bus would not be very useful if the cards that it was to interconnect did not fit into the space allotted! Unfortunately for the concept of standardization, there are dozens of different buses in use, although a few are far more common than others.

The need to characterize buses comes from the necessity of interfacing the bus to other components that are part of the computer system. Buses that are internal to the CPU are usually not characterized formally at all, since they serve special purposes and do not interface to the outside world. Buses that are used in this way are sometimes known as *dedicated* buses. Buses that are intended for more general use must have a well-defined standard; standard buses generally have a name. PCI Express, USB, IDE, and SATA are all examples of named buses.

Each conductor in the bus is commonly known as a **line**. Lines on a bus are often assigned names, to make individual lines easier to identify. In the simplest case, each line carries a single electrical signal. The signal might represent one bit of a memory address, or a sequence of data bits, or a timing control that turns a device on and off at the proper time. Sometimes, a conductor in a bus might also be used to carry power to a module. In other cases, a single line might represent some combination of functions.

The lines on a bus can be grouped into as many as four general categories: data, addressing, control, and power. Data lines carry the “data” that is being moved from one location to another. Address lines specify the recipient of data on the bus. Control lines provide control and timing signals for the proper synchronization and operation of the bus and of the modules and other components that are connected to the bus. A bus connecting only two specific 32-bit registers within a CPU, for example, may require just thirty-two data lines plus one control line to turn the bus on at the correct time. A backplane that interconnects a 64-bit data width CPU, a large memory, and many different types of peripherals might require many more than a hundred lines to perform its function.

The bus that connects the CPU and memory, for example, needs address lines to pass the address stored in the MAR to the address decoder in memory and data lines to transfer data between the CPU and the memory MDR. The control lines provide timing signals for the data transfer, define the transfer as a read or write, specify the number of bytes to transfer, and perform many other functions.

In reality, all of the lines except for the power lines in a bus can be used in different ways. Each line in a bus may serve a single, dedicated purpose, such as a bus line that carries the twelfth bit of an address, for example. Alternatively, a line may be configured to serve

different purposes at different times. A single line might be used to carry each of the bits of an address in sequence, followed by the bits of data, for example. At their two extremes, buses are characterized as **parallel** or **serial**. By definition, a parallel bus is simply a bus in which there is an individual line for each bit of data, address, and control being used. This means that all the bits being transferred on the bus can be transferred simultaneously. A serial bus is a bus in which data is transferred sequentially, one bit at a time, using a single data line pair. (A data return line is required to complete the circuit, just as there are two wires in a standard 110-volt power circuit. Multiple data lines can share the same data return line, commonly known as a *ground* line, but in some cases it is possible to reduce noise and other interference by using a separate return line for each data line.)

A bus line may pass data in one direction only, or may be used to pass data in both directions. A unidirectional line is called a **simplex line**. A bidirectional line may carry data one direction at a time, in which case it is called a **half-duplex line**, or in both directions simultaneously, known as a **full-duplex line**. The same nomenclature is also used to describe data communication channels, because, ultimately, the basic concepts of bus lines and communication channels are essentially similar.

Buses are also characterized by the way that they interconnect the various components to which they are attached. A bus that carries signals from a single specific source to a single specific destination is identified as a **point-to-point bus**. Point-to-point buses that connect an external device to a connector are often referred to as **cables**, as in a printer cable or a network cable. Thus, the cable that connects the USB port in a personal computer from the computer to a printer is an example of a point-to-point bus. The internal connectors into which external cables can be plugged are often called **ports**. Typical ports on a personal computer might include parallel printer ports, network ports, USB ports, and firewire ports.

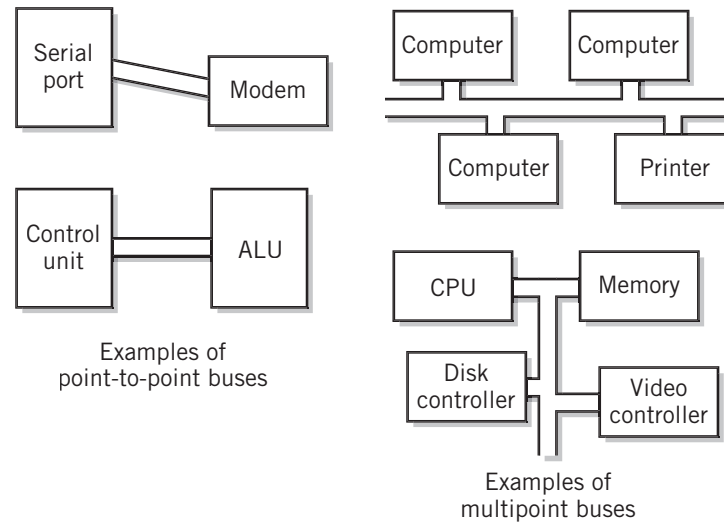
Alternatively, a bus may be used to connect several points together. Such a bus is known as a **multipoint bus**, or sometimes as a *multidrop bus*. It is also referred to as a **broadcast bus**, because the signals produced by a source on the bus are “broadcast” to every other point on the bus in the same way as a radio station broadcasts to anyone who tunes in. The bus in a traditional Ethernet network is an example of a broadcast bus: the signal being sent by a particular computer on the network is received by every other computer connected to the network. (The operation of Ethernet is discussed in Chapter 13.) In most cases, a multipoint bus requires addressing signals on the bus to identify the desired destination that is being addressed by the source at a particular time. Addressing is not required with a point-to-point bus, since the destination is already known, but an address may be required if the message is being passed *through* the destination point to another location. Addressing is also not required for a multipoint bus where the signal is actually intended to reach all the other locations at once; this is sometimes the case for buses that are internal to the CPU. Addressing may be integral to the lines of the bus itself, or may be part of the protocol that defines the meaning of the data signals being transported by the bus.

Typical point-to-point and multipoint bus configurations are illustrated in Figure 7.8

A parallel bus that carries, say, 64 bits of data and 32 bits of address on separate data and address lines would require a bus width of 96 lines, even before control lines are considered. The parallel bus is characterized by high throughput capability because all the bits of a data word are transferred at once. Virtually every bus internal to the CPU is a parallel bus, since the high speed is essential to CPU operation. Also most internal

FIGURE 7.8

Point-to-Point and Multipoint Buses



operations and registers are inherently parallel, and the use of serial buses would require additional circuitry to convert the parallel data to serial and back again. Until recently, the buses that connected the CPU with memory and various high speed I/O modules such as disk and display controllers were also parallel, for similar reasons.

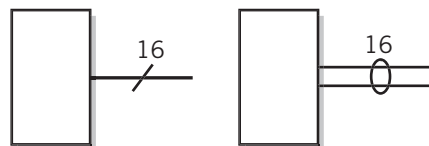
The parallel bus does have a number of disadvantages, though. Parallel buses are expensive and consume a considerable amount of space. Connectors used with parallel buses are also expensive because of the large number of pins involved. External parallel buses, such as printer cables are also expensive because of the large number of lines required. More seriously, parallel buses are subject to radio-generated electrical interference between the different lines at high data transfer rates. The higher the data rate, the worse the interference, which ultimately limits the speed at which the parallel bus can operate. Additionally, there is a slight difference in time delay on different lines, known as *skew*, as signals traverse the bus. The transfer rate, and thus the clock speed of the bus, is also limited by the requirement that the data must not change faster than the maximum skew time. Both of these problems can cause data corruption. Finally, the cost of fiber optic technology makes a parallel optical cable impractical.

Data on a serial bus is transferred sequentially, one bit at a time. Although you might think that the throughput of a serial bus would be lower than that of a parallel bus theoretically capable of the same per line transfer rate, the limitations noted above make serial bus transmission attractive in many circumstances. Indeed, with advances in serial bus technology, serial buses are now preferred for many, if not most, applications requiring high data transfer rates.

Generally, a serial bus has a single data line pair and perhaps a few control lines. (For simultaneous two-way communication, a second data line pair can be added.) There are

FIGURE 7.9

Alternative Bus Notations



no separate address lines in a serial bus. Serial buses are often set up for point-to-point connection; no addressing is required in this case. If addressing is required in a serial bus application, the address may be **multiplexed** with the data. What this means is that the same line is used for both address and data at different times; if an address is required, for example, the address might be sent first, one bit at a time, followed by the data. At its simplest, the serial bus can be reduced to a single data line pair, used for data, control, and addressing. Using modern materials such as fiber optics, very high transfer rates may be achieved. In general, control is handled using a bus protocol that establishes agreement as to the meaning and timing of each signal on the line among the components connected to the line.

It is also possible to design a parallel bus that multiplexes addresses and data on the same lines, as the PCI bus does, or multiplexes 32-bit data on sixteen data lines, for example. For example, the Pentium 4 multiplexes 128-bit data words to fit a 64-bit data path on the Pentium system bus.

To use a bus, the circuits that are connected to the bus must agree on a **bus protocol**. Recall from Chapter 1 that a protocol is an agreement between two or more entities that establishes a clear, common path of communication and understanding between them. A bus protocol is simply a specification that spells out the meaning of each line and each signal on each line for this purpose. Thus, a particular control line on a bus might be defined as a line that determines if the bus is to be used for memory read or memory write. Both the CPU and memory would have to agree, for example, that a “0” on that particular line means “memory read” and a “1” on the line means “memory write”. The line might have a name like $\overline{\text{MREAD/MWRITE}}$, where the bar over MWRITE means that a “0” is the active state. The bar itself stands for “NOT”.³

Buses are frequently notated on diagrams using widened lines to indicate buses. Sometimes a number is also present on the diagram. The number indicates the number of separate lines in the bus. Two alternative ways of notating buses in diagrams are shown in Figure 7.9.

7.6 CLASSIFICATION OF INSTRUCTIONS

Nearly every instruction in a computer performs some sort of operation on one or more source data values, which results in one or more destination data values. The operation may be a move or load, it may be an addition or subtraction, it may be an input or output, or it may be one of many other operations that we have already discussed.

³A pound sign (#) following the name is sometimes used to stand for “NOT” instead.

Actually, if you think about the classes of instructions that we have discussed, you will realize that there are only a very few instructions that do *not* operate on data. Some of these are concerned with the flow of the program itself, such as unconditional `JUMP` instructions. There are also instructions that control the administration of the computer itself; the only example in the Little Man Computer instruction set is the `COFFEE BREAK` or `HALT` that causes the computer to cease executing instructions. Another example on many computers is the `NO OPERATION` instruction that does nothing but waste time (which can be useful when a programmer wants to create a time delay for some reason).

Most modern computers also provide instructions that aid the operating system software in its work, by providing security, controlling memory access, and performing other functions. Because the operating system will frequently be controlling many tasks and users, these instructions must not be available to the users' application programs. Only the operating system can execute these instructions. These instructions are known as **privileged instructions**. The `HALT` instruction is usually a privileged instruction, because you would not want an individual user to stop the computer while other users are still in the middle of their tasks.

Computer manufacturers usually group the instruction set into various categories of instructions, such as data movement instructions, arithmetic instructions, shift and rotate instructions, input/output instructions, conditional branch instructions, jump instructions, and special-purpose instructions.

Within each category, the instructions usually have a similar instruction word format, support similar addressing modes, and execute in a similar way. A typical instruction set, divided into eight categories, appears in Figure 7.10. This figure represents nearly all the user-accessible instructions in the Motorola 68000 series of microprocessors used in early Apple Macintosh computers.⁴ The privileged instructions are not listed in the diagram, nor are exception-handling instructions that are used primarily by system programmers. These constitute an additional two categories for the 68000 series CPUs. Incidentally, notice that this CPU does not have any I/O instructions. That is because the CPU is designed in such a way that the move instructions can also be used for I/O. Notice particularly that, except for the lack of I/O instructions, the categories conform fairly well to the Little Man Computer instruction set. The additional instructions in this CPU are mostly variations on instructions that are familiar to you plus special control instructions. The 68000 series CPUs also support a math coprocessor, which adds a category of floating point arithmetic instructions. The floating point math instructions are built directly into 68000 series processors starting with the 68040 CPU.

Data Movement Instructions (LOAD, STORE, and Other Moves)

Because the move instructions are the most frequently used, and therefore the most basic to the computer, computer designers try to provide a lot of flexibility in these instructions. The `MOVE` category commonly includes instructions to move data from memory to general registers, from general registers to memory, between different general registers, and, in

⁴Although the 68000 CPU series is old, it is still used in embedded computer systems. It was selected for this illustration because of its clean design, with few extraneous bells and whistles.

FIGURE 7.10

68000 Instruction Set

Mnemonic	Operation	Mnemonic	Operation
Data Movement Instructions		Shift and Rotate Instructions	
CAS*	Compare and swap with operand	ASL	Arithmetic shift register left
CAS2*	Compare upper/lower and swap	ASR	Arithmetic shift right
EXG	Exchange registers	LSL	Logical shift left
LEA	Load effective address	LSR	Logical shift right
LINK	Link and allocate stack	ROL	Rotate left
MOVE	Move src to dst	ROR	Rotate right
MOVE16	Move src to dst (68030-68060 only)	ROXL	Rotate left with extend bit
MOVEA	Move src to address register	ROXR	Rotate right with extend bit
MOVEM	Move multiple registers at once	SWAP	Swap words of a long word
MOVEP	Move to peripheral	Bit Manipulation Instructions	
MOVEQ	Move short data to dst	BCHG	Change bit
PEA	Push effective address to stack	BCLR	Clear bit
UNLK	Unlink stack	BTEST	Set bit
Integer Arithmetic Instructions		BTST	Test bit
ADD	Add src to dst	Bit Field Instructions	
ADDA	Add src to address register	BFCHG*	Change bit field
ADDI	Add immediate data to dst	BFCLR*	Clear bit field
ADDQ	Add short data to dst	BFEXTS*	Extract and sign extend bit field
ADDX	Add with extend bit to dst	BFEXTU*	Extract and zero extend bit field
SUB, SUBA, SUBI, SUBQ, SUBX	Subtracts act similarly to adds	BFFFO*	Find first set bit in bit field
MULS	Signed multiply	BFINS*	Insert bit field
MULU	Unsigned multiply	BFSET*	Set bit field
DIVS	Signed divide	BFTST*	Test bit field
DIVU	Unsigned divide	Binary Coded Decimal Instructions	
DIVSL*	Long signed divide	ABCD	Add src to dst
DIVUL*	Unsigned long divide	NBCD	Negate destination
CLR	Clear value in register	PACK*	Pack src to dst
CMP	Compare src to dst	SBCD*	Subtract src from dst
CMPA	Compare src to address register	UNPK*	Unpack src to dst
CMPI	Compare immediate data to dst	Program Flow Instructions	
CMPM	Compare memory	Bcc	Branch on condition code cc
CMP2*	Compare register to upper/lower bounds	BRA	Branch unconditionally
EXT	Sign extend	BSR	Branch to subroutine
EXTB	Sign extend byte	CALLM*	Call module
NEG	Negate register	DBcc	Test, decrement, and branch on condition
NEGX	Negate with extend	JMP	Jump to address
Boolean Logic Instructions		JSR	Jump to subroutine
AND	AND src to dst	NOP	No operation
ANDI	AND immediate data to dst	RTD*	Return and deallocate stack (also 68010)
EOR	Exclusive OR src to dst	RTE	Return from exception (privileged)
EORI	Exclusive OR immediate data to dst	RTM*	Return from module
NOT	NOT destination	RTR	Return and restore condition codes
OR	OR src to dst	RTS	Return from subroutine
ORI	OR immediate data to dst	TRAP	Trap to system
Scc	Test condition codes and set operand	*(68020–68060 only)	
TAS	Test and set operand	(src = source; dst = destination; cc = condition code indicator, e.g. BGT branch of greater than)	
TST	Test operand and set condition codes		
TRAPcc*	Trap on condition		

some computers, directly between different memory locations without affecting any general register. There may be many different addressing modes available within a single computer.

Additionally, variations on these instructions are frequently used to handle different data sizes. Thus, there may be a `LOAD BYTE` instruction, a `LOAD HALF-WORD` (2 bytes), a `LOAD WORD` (4 bytes), and a `LOAD DOUBLE WORD` (8 bytes) within the same instruction set. (Incidentally, the concept of a “word” is not consistent between manufacturers. To some manufacturers the size of a word is 16 bits; to others, it is 32 or even 64 bits).

The Little Man `LOAD` and `STORE` instructions are simple, though adequate, examples of `MOVE` instructions. Other than expanding the addressing mode capabilities and adding multiple word size capabilities, which we have already discussed, the major limitation of the Little Man `LOAD` and `STORE` instructions is the fact that they are designed to operate with a single accumulator.

When we expand the number of accumulators or general-purpose registers, we must expand the instruction to determine which register we wish to use. Thus, the instruction must provide a field for the particular register. Fortunately, it takes very few bits to describe a register. Even sixteen registers require only 4 bits. On the other hand, if the computer uses the registers to hold pointers to the actual memory addresses as its standard addressing mode, the required instruction size may actually decrease, since fewer bits are required for the address field in this case.

Additionally, it is desirable to have the capability to move data directly between registers, since such moves do not require memory access and are therefore faster to execute. In fact, some modern CPUs, including the Sun SPARC and IBM PowerPC architectures, provide only one pair of `LOAD/STORE` or `MOVE` instructions for moving data between the CPU and memory. All other instructions in these CPUs move and manipulate data only between registers. This allows the instruction set to be executed much more rapidly. There is a detailed examination of the Power PC computer and its variants in Supplementary Chapter 2.

Arithmetic Instructions

Every CPU instruction set includes integer addition and subtraction. Except for a few special-purpose CPUs, every CPU today also provides instructions for integer multiplication and division. Many instruction sets provide integer arithmetic for several different word sizes. As with the `MOVE` instructions, there may be several different integer arithmetic instruction formats providing various combinations of register and memory access in different addressing modes.

In addition, most current CPUs also provide floating point arithmetic capabilities. On older PCs with 80386 or earlier processors, a floating point math coprocessor unit had to be purchased separately and installed in a socket provided for that purpose on the motherboard of the computer. Because of the expense, most users would not exercise this option. Extensive floating point calculations are required for many graphics applications, such as CAD/CAM programs, animation, and computer games; the presence of floating point instructions reduces the processing time significantly. Floating point instructions usually operate on a separate set of floating point data registers with 64-, 80-, or 128-bit word sizes. The modern instruction set usually also contains instructions that convert data between integer and floating point formats.

As noted in Chapter 5, most modern CPUs also provide at least a minimal set of arithmetic instructions for BCD or packed decimal format, which simplifies the programming of business data processing applications.

Of course, it is not absolutely necessary to provide all these different instruction options. Multiplication and division can be performed with repeated addition and subtraction, respectively. In computers there is an even easier technique. In elementary school you probably learned the “long” multiplication and division methods which multiply or divide numbers one digit at a time and shift the results until the entire operation is complete. Because of the simplicity of binary multiplication ($1 \times 1 = 1$, all other results are 0), the computer can implement the same method using only adds or subtracts together with shift instructions. Internally, the multiplication and division instructions simply implement in hardware this same method. Since the fetch-execute cycle requires a single-bit shift and register add step for each bit in the multiplier, multiply and divide instructions execute slowly compared to other instructions.

Even the subtract instruction is theoretically not necessary, since we showed in Chapter 4 that integer subtraction is performed internally by the process of complementing and adding.

As we already noted, the same is true of BCD and floating point instructions. On the now rare computers that do not provide floating point instructions, there is usually a library of software procedures that are used to simulate floating point instructions.

Boolean Logic Instructions

Most modern instruction sets provide instructions for performing Boolean algebra. Commonly included are a NOT instruction, which inverts the bits on a single operand, as well as AND, (inclusive) OR, and EXCLUSIVE-OR instructions, which require two source arguments and a destination.

Single Operand Manipulation Instructions

In addition to the NOT instruction described in the previous paragraph, most computers provide other convenient single operand instructions. Most of these instructions operate on the value in a register, but some instruction sets provide similar operations on memory values as well. Most commonly, the instruction set will contain instructions for NEGATING a value, for INCREMENTING a value, for DECREMENTING a value, and for setting a register to zero. There are sometimes others. On some computers, the increment or decrement instruction causes a branch to occur automatically when zero is reached; this simplifies the design of loops by allowing the programmer to combine the test and branch into a single instruction.

Bit Manipulation Instructions

Most instruction sets provide instructions for setting and resetting individual bits in a data word. Some instruction sets also provide instructions for operating on multiple bits at once. Bits can also be tested, and used to control program flow. These instructions allow programmers to design their own “flags” in addition to commonly provided negative/positive, zero/nonzero, carry/borrow, and overflow arithmetic flags.

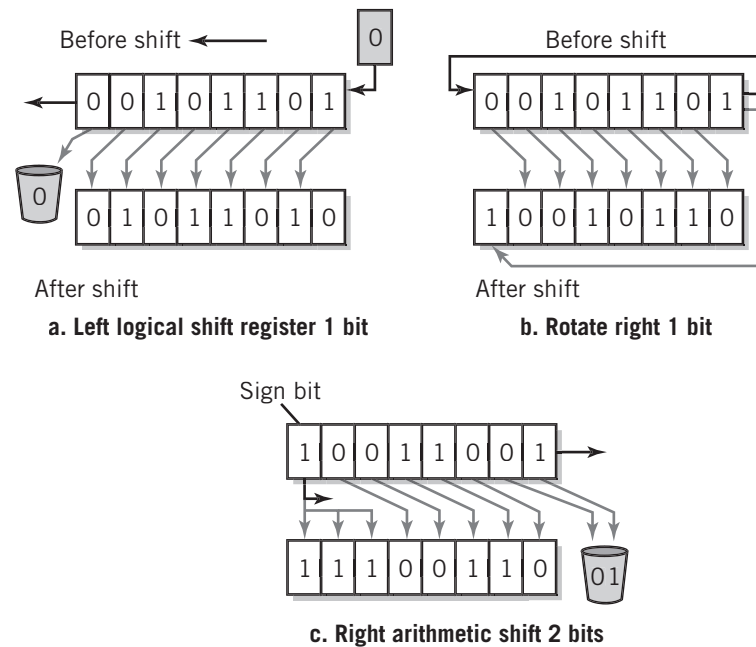
Shift and Rotate Instructions

Shift and **rotate operations** have been mentioned previously as a means to implement multiplication and division. Shifts and rotate operations have other programming applications, and CPU instruction sets commonly provide a variety of different shift and rotate instructions for the programmer to use. As shown in Figure 7.11, shift instructions move the data bits left or right one or more bits. Rotate instructions also shift the data bits left or right, but the bit that is shifted out of the end is placed into the vacated space at the other end. Depending on the design of the particular instruction set, bits shifted out the end of the word may be shifted into a different register or into the carry or overflow flag bit, or they may simply “fall off the end” and be lost.

Two different kinds of shifts are usually provided. The data word being shifted might be logical or it might be numeric. **Logical shift** instructions simply shift the data as you would expect, and zeros are shifted in to replace the bit spaces that have been vacated. **Arithmetic shift** instructions are commonly used to multiply or divide the original value by a power of 2. Therefore, the instruction does not shift the leftmost bit, since that bit usually represents the algebraic sign of the numeric value—obviously the sign of a number must be maintained. Left arithmetic shifts do not shift the left bit, but zeros replace the bits from the right as bits are moved to the left. This will effectively double the numeric value for each shift of one bit. On the other hand, right arithmetic shifts fill the space of moved bits with the sign bit rather than with zero. This has the effect of halving the value

FIGURE 7.11

Typical Register Shifts and Rotates



for each bit shifted, while maintaining the sign of the value. It may not seem obvious to you that this works correctly, but it becomes more apparent if you recall that negative numbers in complementary arithmetic count backward starting from the value -1 , which is represented in 2's complement by all ones.

Rotate instructions take the bits as they exit and rotate them back into the other end of the register. Some instructions sets include the carry or overflow bit as part of the rotation. Some CPUs also allow the rotation to take place between two registers. Rotate instructions can be used to exchange the 2 bytes of data in a 16-bit word, for example, by rotating the word by 8 bits.

Program Control Instructions

Program control instructions control the flow of a program. Program control instructions include jumps and branches, both unconditional and conditional, and also **subroutine CALL** and **RETURN** instructions. Various conditional tests are provided, including those with which you are already familiar: branch on zero, branch on nonzero, branch on positive, branch on negative, branch on carry, and so on.

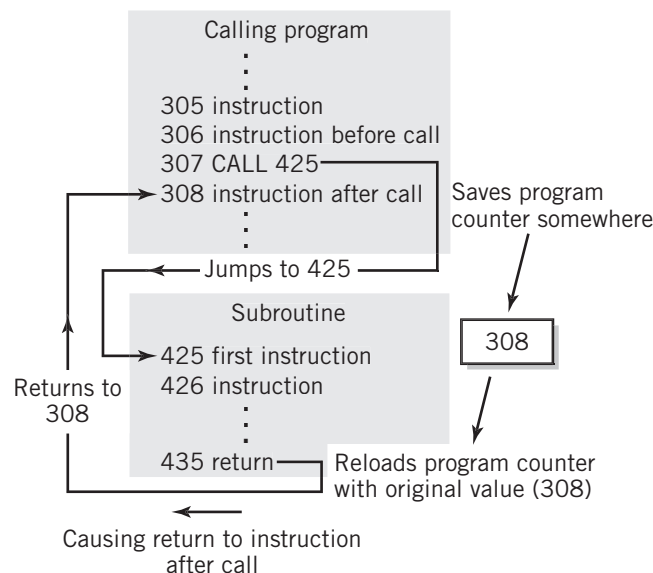
CALL instructions, sometimes known as JUMP SUBROUTINE instructions, are used to implement subroutine, procedure and function calls. Thus, CALL instructions are important as a means to enable program modularization.

From your programming experience, recall what happens when your program calls a subroutine or procedure. The program jumps to the starting location of the subroutine

and executes the code in the subroutine. When the subroutine is completed, program execution returns to the calling program and continues with the instruction following the call. The machine language CALL instruction works the same way. A jump to the starting location of the subroutine occurs, and execution continues from that point. The only difference between a CALL instruction and a normal JUMP instruction is that the CALL instruction must also save somewhere the program counter address from which the jump occurred, so that the program may return to the instruction in the calling program following the call after the subroutine is completed. The RETURN instruction restores the original value to the program counter, and the calling program proceeds from where it left off. Operation of the CALL and RETURN instructions are illustrated in Figure 7.12.

FIGURE 7.12

Operation of CALL and RETURN Instructions



Different computers use different methods to save the return address. One common method is to store the return address on a memory stack; the `RETURN` instruction operates by removing the address from the stack and moving it to the program counter. The use of stacks is discussed briefly in the next section. Another method for performing `CALLS` and `RETURNS` is explored in Exercise S3.14.

Stack Instructions

One of the most important data storage structures in programming is the **stack**. A stack is used to store data when the most recently used data will also be the first needed. For that reason, stacks are also known as *LIFO*, for *last-in, first-out*, structures. As an analogy, stacks are frequently described by the way plates are stored and used in a cafeteria. New plates are added to the top of the stack, or *pushed*, and plates already on the stack move down to make room for them. Plates are removed from the top of the stack, or *popped*, so that the last plates placed on the stack are the first removed. Similarly, the last number entered onto a computer memory stack will be the first number available when the stack is next accessed. Any data that must be retrieved in reverse order from the way it was entered is a candidate for the use of stacks. Figure 7.13 shows the process of adding to and removing numbers from the stack.

Stacks are an efficient way of storing intermediate data values during complex calculations. In fact, storage in Hewlett-Packard calculators is organized around a stack of memory. As we already noted, stacks are also an excellent method for storing the return addresses and arguments from subroutine calls. Program routines that are recursive must “call themselves.” Suppose the return address were stored in a fixed location, as shown in Figure 7.14a. If the routine is called a second time, from within itself, Figure 7.14b, the original returning address (56) is lost and replaced by the new return address (76). The program is stuck in an infinite loop between 76 and 85. In Figure 7.15, the return address is stored on a stack. This time when the routine is again called, the original address is simply

FIGURE 7.13

Using a Stack

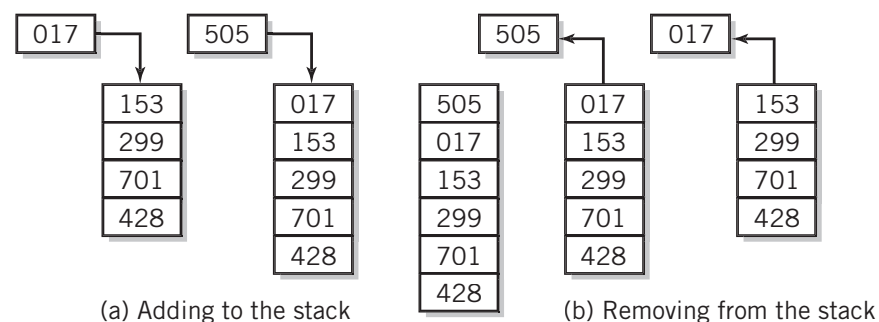
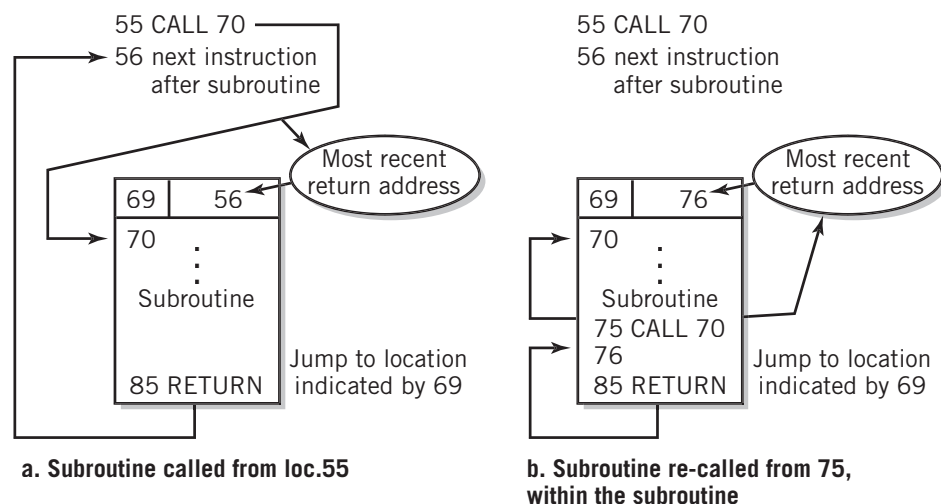


FIGURE 7.14

Fixed Location Subroutine Return Address Storage



pushed down the stack, below the most recent address. Notice that the program “winds its way back out” in the reverse order from which the routines were entered. This is exactly what we want: we always return from the last called subroutine to the one just previous. J. Linderman of Bentley University notes that the same technique would be used to back out of a maze for which the explorer has written down each turn that she made after entering.

There are many other interesting applications for stacks in computers, but further discussion is beyond the scope of this book. The curious reader is referred to *For Further Reading* for references.

Computers do not generally provide special memory for stack use, although many machines provide special `STACK` instructions to simplify the bookkeeping task. Instead, the programmer sets aside one or more blocks of regular memory for this purpose. The “bottom” of the stack is a fixed memory location, and a **stack pointer** points to the “top” of the stack, that is, the most recent entry. This is shown in Figure 7.16. A new entry is added to the stack, or pushed, by incrementing the stack pointer, and then storing the data at that location. An entry is removed from the stack, or popped, by copying the value pointed to and then decrementing the stack pointer. If a register is provided for the stack pointer, register-deferred addressing can be used for this purpose. (You should note that memory is drawn upside-down in Figure 7.16 so that incrementing the stack pointer moves it *upward*.)

Many instruction sets provide `PUSH` and `POP` instructions as direct support for stacks, but stacks can be implemented easily without special instructions. (Exercise S3.15 illustrates one solution.) Some computers also specify the use of a particular general-purpose register as a stack pointer register.

FIGURE 7.15

Stack Subroutine Return Address Storage

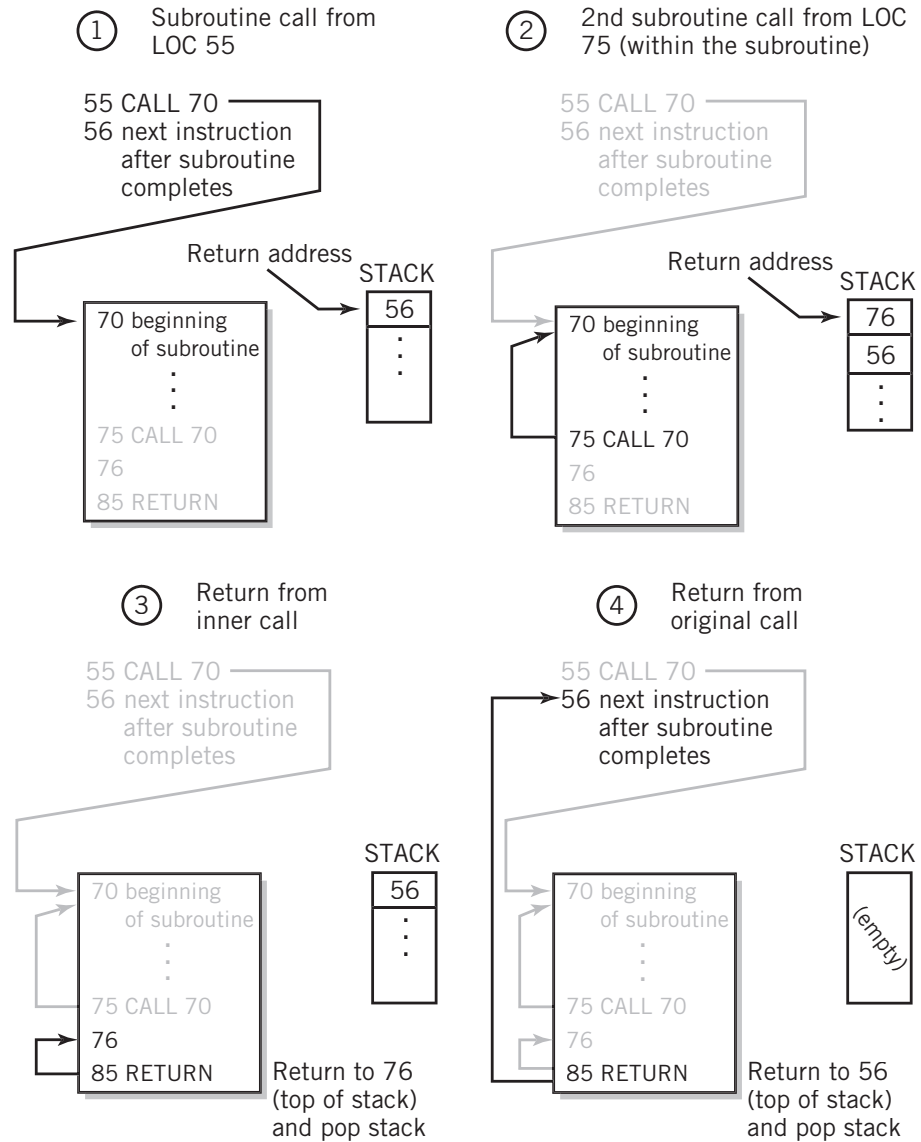
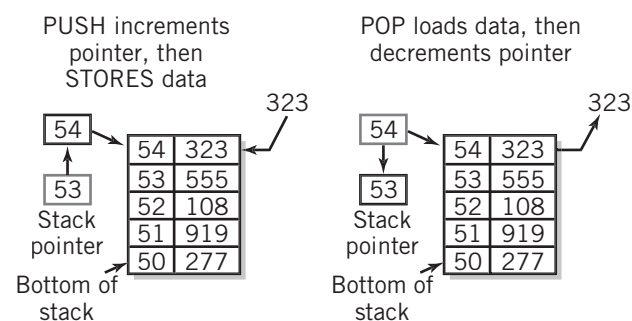


FIGURE 7.16

Using a Block of Memory as a Stack



Multiple Data Instructions

Multimedia applications rank high in computational demands on the CPU in modern PCs and workstations. In response to the demand, CPU designers have created specialized instructions that speed up and simplify multimedia processing operations.

Multimedia operations are commonly characterized by a number of simple operations applied identically to every piece of data in the set. As a simple example, the brightness of an image might be modified by multiplying the value of every pixel in the image by a common scale factor. Or, a

measure of similarity between two images could be established by subtracting all the pixel values in one image from the corresponding pixel values in a second image and averaging the results.

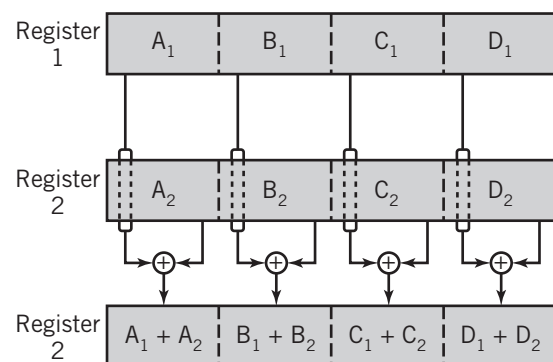
Multiple data instructions perform a single operation on multiple pieces of data simultaneously. For this reason they are also known as **SIMD** instructions. SIMD stands for **Single Instruction, Multiple Data**. The SIMD instructions provided on Intel Pentium processors are typical. The processor provides eight 128-bit registers specifically for SIMD instruction use and also allows the use of the standard 64-bit floating point registers for this purpose. The Pentium CPU SIMD instructions can process from two 64-bit integers up to sixteen 8-bit integer arithmetic operations or up to two 64-bit floating point number operations simultaneously as well as providing instructions for packing and unpacking the values and moving them between registers and memory, and a

variety of other related instructions. Other vendors, including AMD, IBM, Sun, Transmeta, and VIA provide compatible or similar SIMD instructions. The IBM Cell processor, which serves as the CPU in the Sony Playstation 3, provides a particularly powerful SIMD capability that accounts for much of the Playstation's graphics strength. Figure 7.17 shows the operation of a SIMD ADD instruction.

Although multimedia operations are a primary application for these instructions, these instructions can be applied to any vector or array processing application, and are useful for a number of purposes in addition to multimedia processing, including voice-to-text processing, the solutions to large-scale economics problems, weather prediction, and data encryption and decryption.

FIGURE 7.17

Operation of a 4-Wide SIMD ADD Instruction



Other Instructions

The remainder of the instructions includes input/output instructions and machine control instructions. In most systems both groups are privileged instructions. Input/output instructions are generally privileged instructions because we do not want input and output requests from different users and programs interfering with each other. Consider, for example, two users requesting printer output on a shared printer at the same time, so that each page of output is divided back and forth between the two users. Obviously, such output would not be acceptable. Instead, these requests would be made to the operating system that controls the printer, which would set priorities, maintain queues, and service the requests. We will deal with the subject of input/output in Chapters 9 and 10, and with operating systems in Chapters 15 through 18.

7.7 INSTRUCTION WORD FORMATS

Instructions in the Little Man Computer were made up entirely of three-digit decimal numbers, with a single-digit op code, and a two-digit **address field**. The address field was used in various ways: for most instructions, the address field contained the two-digit address where data for the instruction could be found (e.g., LOAD) or was to be placed (STORE). In a few instructions, the address field was unused (e.g., HALT). For the branch instructions, the address field space was used instead to hold the address of the next instruction to be executed. For the I/O instructions, the address field became a sort of extension of the op code. In reality, the I/O address field contained the “address” of an I/O device, in our case 01 for the in basket and 02 for the out basket.

The instruction set in a typical real CPU is similar. Again, the instruction word can be divided into an op code and zero or more address fields. A simple 32-bit instruction format with one address field might look like that shown in Figure 7.18. In this example, the 32 bits are divided into an 8-bit op code and 24 bits of address field.

In the Little Man Computer, reference to an *address* specifically referred to a *memory* address. However, we have already noted that the computer might have several general-purpose registers and that it would be necessary for the programmer to select a particular register to use as a part of the instruction. To be more general, we will use the word “address” to refer to any data location, whether it is a user-accessible register or a memory location. We will use the more specific expression *memory address* when we want to specify that the address is actually a memory location.

In general, computer instructions that manipulate data require the specification of at least two locations for the data: one or more *source* locations and one *destination* location. These locations may be expressed *explicitly*, as address fields in the instruction word, or *implicitly*, as part of the definition of the instruction itself. The instruction format of the Little Man LOAD instruction, for example, takes the data from the single address field as the

FIGURE 7.18

A Simple 32-bit Instruction Format

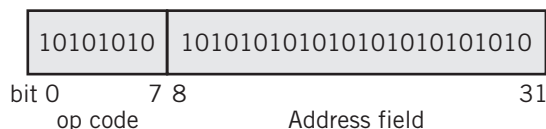


FIGURE 7.19

Typical Two Operation Register Move Format

op code	source register	destination register
MOVE	5	10

explicit source address. Explicit addresses in the Little Man Computer are always memory addresses. The destination address in this case is **implicit**: this instruction always uses the accumulator register as a destination. The Little Man ADD and SUBTRACT instructions require *two* sources and a destination. Source data addressed by the instruction's single explicit address field is added to the value in the implicitly stated accumulator, with the result placed implicitly in the accumulator.

For a particular instruction, the source(s) and destination may be the same or may be different. For example, an instruction that complements a value to change its sign would usually be done “in place”; that is, the source and destination register or memory location is usually the same. The Little Man ADD instruction uses the accumulator register both as a source for one of the numbers to be added and as the destination for the result. On the other hand, when we move data, using a LOAD or STORE or some other type of MOVE operation, two operands are required. The source and destination are obviously different, or the move would not be useful! A register-to-register MOVE, for example, might use an instruction format such as that shown in Figure 7.19. In the figure, the instruction word consists of an opcode and two fields that point to registers. As shown, this instruction would move data from register 5 to register 10. Unless the operation is done in place, the sources are normally left unchanged by the instruction, whereas the destination is almost always changed.

The source and destination addresses may be registers or memory locations. Since most modern computers have multiple registers available to the user it is usually necessary to provide at least two explicit address fields, even for an address-register move, since the number of the particular register must be specified in the instruction.

The sources and destinations of data for an instruction, whether implicit or explicit, are also known as **operands**. Thus, instructions that move data from one place to another have two operands: one source operand and one destination operand. Arithmetic operations such as ADD and SUBTRACT require three operands. Explicit address fields are also known as *operand fields*.

Most commonly, instructions that manipulate data will have one address field for operations that happen in place, and two or three address fields for move and arithmetic operations. On some computers one or more of the addresses may be implicit, and no address field is required for the implicit address. However, in modern computers most address references are explicit, even for register addresses, because this increases the generality and flexibility of the instruction. Thus, most computer instructions will consist of an op code and one, two, or three explicit address fields. Some textbooks refer to instructions with one, two, or three explicit address fields as *unary*, *binary*, or *ternary* instructions, respectively.

7.8 INSTRUCTION WORD REQUIREMENTS AND CONSTRAINTS

The size of the instruction word, in bits, is dependent on the particular CPU architecture, particularly by the design of its instruction set. The size of the instruction word may be

fixed at, say, 32 bits, or it may vary depending on the usage of the address fields. The Sun Sparc CPU, for example, takes the former approach: every instruction word is exactly 32 bits wide. Conversely, some of the basic instruction words for the x86 microprocessor line used in the common PC, for example, are as small as 1 or 2 bytes long, but there are some instructions in the Pentium microprocessor that are as many as 15 bytes long. The IBM Series z architecture is an evolutionary extension of upward compatible CPU architectures dating back to the 1960s. The legacy instructions in the IBM Series z CPU are mostly 4 bytes, or 32 bits long, with a few 2-byte or 6-byte long instructions. To expand the architecture to 64-bit addressing and data, IBM added a number of new instructions. These are all 6 bytes in length.

The challenge in establishing an instruction word size is the need to provide both enough op code bits to support a reasonable set of different instructions as well as enough address field bits to meet the ever growing demand for increasing amounts of addressable memory. Consider again, for example, the extremely straightforward instruction format shown in Figure 7.18. This format assumes a single address field with a 32-bit fixed length instruction. With the division shown, we have access to $2^8 = 256$ different instructions and $2^{24} =$ approximately 16 million memory addresses.

Even if the designer creates a smaller instruction set, with fewer op codes, the amount of memory that may be specified in a 32-bit instruction word is severely limited by modern standards. Most of today's computers support an address size of at least 32 bits. Many newer machines support 64-bit addresses.

Further, with additional registers, the simple instruction format shown in Figure 7.18 must be expanded to handle explicit addressing of multiple registers, including moves between registers, as well as identifying the proper register in operations between registers and memory. In short, the simple instruction format used in the Little Man Computer is inadequate for the instruction sets in modern computers.

The use of instructions of different lengths is one of several techniques developed by instruction set designers to allow more flexibility in the design of the instruction set. Simple instructions can be expressed in a small word, perhaps even a single byte, whereas more complicated instructions will require instruction words many bytes long. Longer instructions are stored in successive bytes of memory. Thus, a Little Man HALT, IN, or OUT instruction would be stored in a single location. A LOAD might require two successive locations to store memory addresses of five digits or three locations for an eight-digit address. The use of variable length instructions is efficient in memory usage, since each instruction is only as long as it needs to be.

There are a number of important disadvantages to variable length instructions, however. Most modern computers increase CPU processing speed by "pipelining" instructions, that is, by fetching a new instruction while the previous one is still completing execution, similar to the processing on an automobile assembly line. Variable length instructions complicate pipelining, because the starting point of the new instruction is not known until the length of the previous instruction has been determined. If you extend this idea to multiple instructions, you can see the difficulty of maintaining a smooth assembly line. This issue is discussed in more detail in Chapter 8. Because pipelining has become so important to processing speed in modern computers, the use of variable length instructions has fallen out of favor for new CPU designs. Nearly all new CPU designs use fixed length instructions exclusively.

As we mentioned previously in our discussion of memory size, an effective alternative to large instructions or variable instruction words is to store the address that would otherwise be located in an instruction word address field at some special location that can hold a large address, such as a general purpose register, and use a small address field within the instruction to point to the register location. There are a number of variations on this theme. This technique is used, even on systems that provide variable length instructions. A single CPU might provide a number of different variations to increase the flexibility of the instruction set. This flexibility also includes the ability to code programs that process lists of data more efficiently. The various ways of addressing registers and memory are known as addressing modes. The Little Man Computer provides only a single mode, known as direct addressing. The alternative just described is called register deferred addressing. An example of a deferred `LOAD` instruction is shown in Figure 7.20. This instruction would load the data value stored at memory address 3BD421 into general purpose register 7. There are a number of addressing modes discussed in detail in Supplementary Chapter 3. The use of different addressing modes is the most important method for minimizing the size of instruction words and for writing efficient programs.

Examples of instruction formats from two different CPUs are shown in Figure 7.21. There may be several different formats within a single CPU. We have shown only a partial set for each machine, although the SPARC set is complete except for small variations. (There are twenty-three different IBM formats in all.) It is not necessary that you understand every detail in Figure 7.21, but it is useful to note the basic similarities between the instruction set formats in different computers.

FIGURE 7.20

Deferred Register Addressing

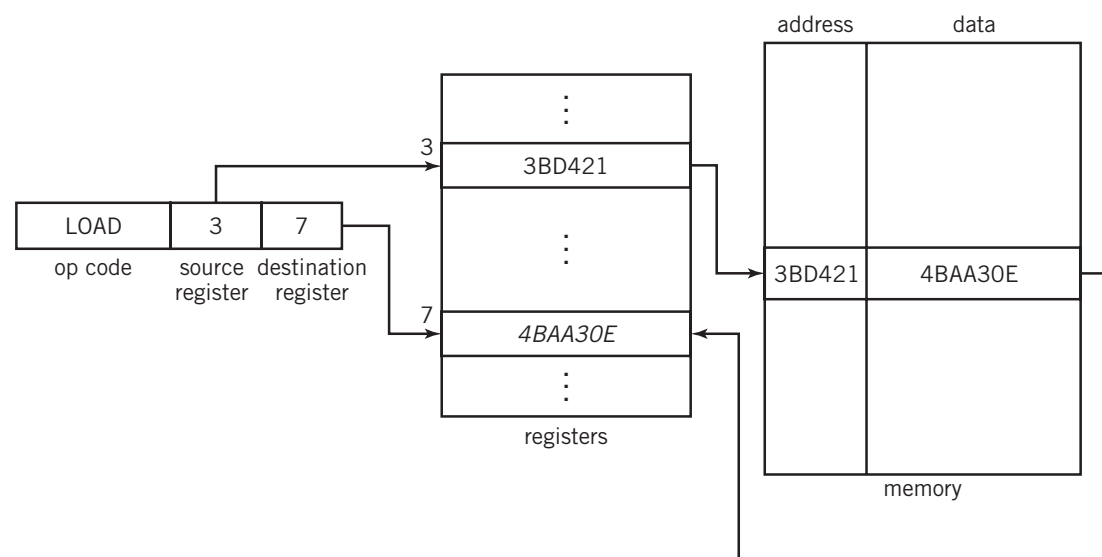
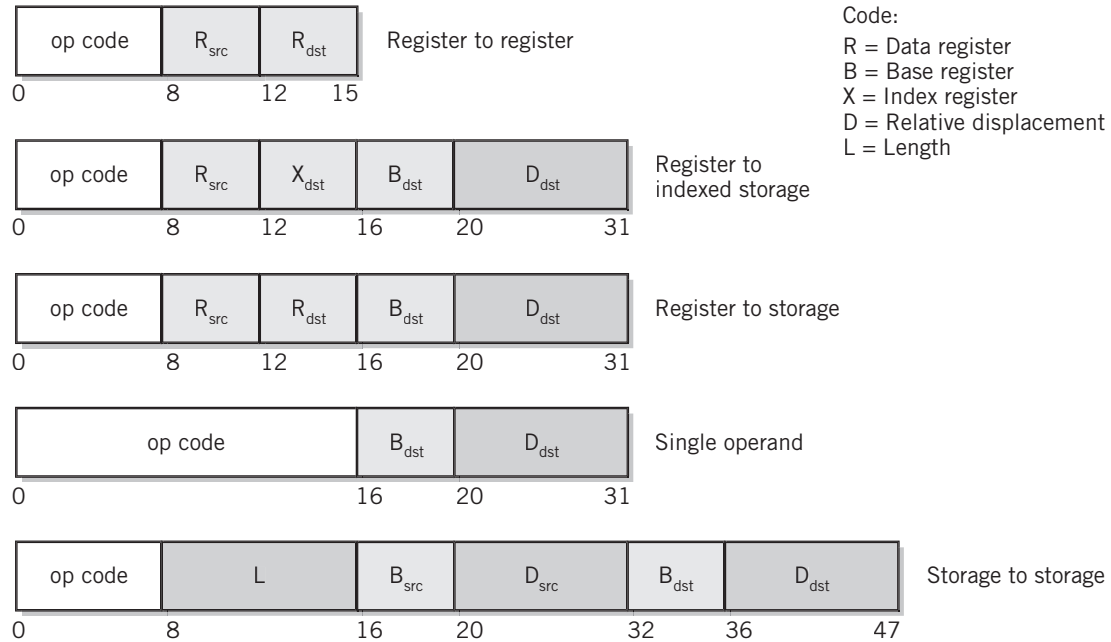


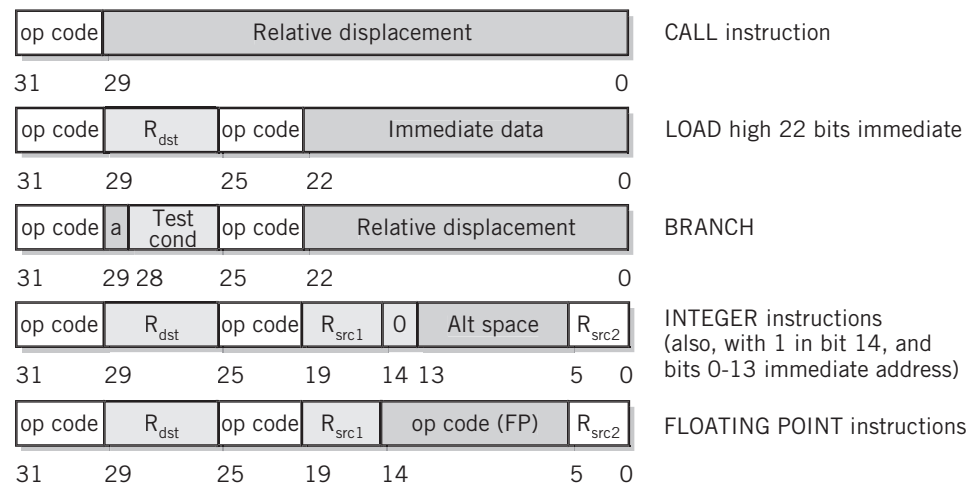
FIGURE 7.2I

Examples of Instruction Formats



Code:
R = Data register
B = Base register
X = Index register
D = Relative displacement
L = Length

IBM mainframe formats



SPARC formats

SUMMARY AND REVIEW

Functionally, the operation of the CPU, together with memory, is essentially identical to that of the Little Man Computer. For each component of the Little Man Computer, there is a corresponding component in the computer unit.

Within the CPU, the most important components are registers. Data may be moved between registers, may be added or subtracted from the current contents of a register, and can be shifted or rotated within a register or between registers. Each instruction in the instruction set is executed by performing these simple operations, using the appropriate choice of registers and operations in the correct sequence for the particular instruction. The sequence of operations for a particular instruction is known as its fetch-execute cycle. A fetch-execute cycle exists for every instruction in the instruction set. Fetch-execute instruction cycles constitute the basis for all program execution in the computer. The sequence for each instruction corresponds closely to the actions taken by the Little Man in performing a similar instruction.

The operation of memory is intimately related to two registers in particular, the memory address register and the memory data register. Addresses placed into the MAR are decoded in memory, resulting in the activation of a single memory address line. At the proper instant, data can then be transferred in either direction between that memory location and the MDR. The direction is specified by a read/write control line. The number of available memory locations is established by the size of the MAR; the data word size is established by the size of the MDR.

Interconnections between various parts of a computer are provided by buses. There are many different types of buses. Buses connect different modules within the CPU. They also connect the CPU to memory and to the I/O peripherals. Buses can connect two components in a point-to-point configuration or may interconnect several modules in a multipoint configuration. Buses may be parallel or serial. In general, the lines on buses carry signals that represent data, address, and control functions.

Instructions fall naturally into a small number of categories: moves, integer arithmetic, floating point arithmetic, data flow control, and so forth. There are also privileged instructions, which control functions internal to the CPU and are accessible only to the operating system.

Instructions in a real CPU are made up of an op code and up to three address field operands. The size of the instruction word is CPU dependent. Some computers use variable length instruction words. Other computers use a fixed length instruction, most commonly, 32 bits in length.

FOR FURTHER READING

There are many excellent textbooks that describe the implementation and operation of the components of the computer system. A brief, but very clear, explanation of the fetch-execute cycle can be found in Davis and Rajkumar [DAV02]. Three classic engineering textbooks that discuss the topics of this chapter in great detail are those authored by Stallings [STAL05], Patterson and Hennessy [PATT07], and Tanenbaum [TAN05]. Wikipedia offers a brief, but clear, introduction to the principal concepts of von Neumann architecture. There are many books and papers describing various components and techniques associated

with the implementation and operation of the CPU and memory. Also see the For Further Reading section in Chapter 8 for more suggestions.

KEY CONCEPTS AND TERMS

accumulator	full-duplex line	program counter (PC)
address field	general-purpose register	program counter register
arithmetic/logic unit (ALU)	half-duplex line	Program Status Word (PSW)
arithmetic shift	implicit source address	privileged instruction
broadcast bus	instruction pointer	RAM
bus	instruction register (IR)	register
bus interface bridge	line (bus)	register file
bus protocol	logical shift	ROM
cable	memory	rotate operation
central processing unit (CPU)	memory address register (MAR)	serial bus
control unit (CU)	memory data register (MDR)	shift operation
dynamic RAM	memory management unit	SIMD
EEPROM (electronically erasable programmable ROM)	multiplex	simplex line
explicit source address	multipoint bus	stack
fetch-execute instruction cycle	nonvolatile memory	stack pointer
flag	operands	static RAM
flash memory	parallel bus	status register
	point-to-point bus	subroutine call and return
	port	user-visible register
		volatile memory

READING REVIEW QUESTIONS

- 7.1 What does *ALU* stand for? What is its corresponding component in the Little Man Computer? What does *CU* stand for? What is its corresponding LMC component?
- 7.2 What is a *register*? Be precise. Name at least two components in the LMC that meet the qualifications for a register. Name several different kinds of values that a register might hold.
- 7.3 What is the purpose of the *instruction register*? What takes the place of the instruction register in the LMC?
- 7.4 When a value is copied from one register to another, what happens to the value in the source register? What happens to the value in the destination register?
- 7.5 There are four primary operations that are normally performed on a register. Describe each operation.
- 7.6 Explain the relationship between the memory address register, the memory data register, and memory itself.
- 7.7 If the memory register for a particular computer is 32 bits wide, how much memory can this computer support?

- 7.8 What is the difference between volatile and nonvolatile memory? Is RAM volatile or nonvolatile? Is ROM volatile or nonvolatile?
- 7.9 Registers perform a very important role in the fetch-execute cycle. What is the function of registers in the fetch-execute instruction cycle?
- 7.10 Explain each of the fetch part of the fetch-execute cycle. At the end of the fetch operation, what is the status of the instruction? Specifically, what has the fetch operation achieved that prepares the instruction for execution? Explain the similarity between this operation and the corresponding operation performed steps performed by the Little Man.
- 7.11 Once the fetch operation is complete, what is the first step of the execution phase for any instruction that accesses a memory address for data (e.g., LOAD, STORE)?
- 7.12 Using the ADD instruction as a model, show the fetch-execute cycle for a SUBTRACT instruction.
- 7.13 Assume the following values in various registers and memory locations at a given point in time:
PC: 20 A: 150 Memory location 20: 160 [ADD 60] Memory location 60: 30.
Show the values that are stored in each of the following registers at the completion of the instruction: PC, MAR, MDR, IR, and A.
- 7.14 Define a *bus*. What are buses used for?
- 7.15 What three types of “data” might a bus carry?
- 7.16 Explain how data travels on a bus when the bus is *simplex*. *Half-duplex*. *Full-duplex*.
- 7.17 What is the difference between a *multipoint* bus and a *point-to-point* bus? Draw diagrams that illustrate the difference.
- 7.18 Briefly describe each of the major disadvantages of parallel buses.
- 7.19 Which Little Man Computer instructions would be classified as *data movement* instructions?
- 7.20 What operations would you expect the arithmetic class of instructions to perform?
- 7.21 Explain the difference between SHIFT and ROTATE instructions.
- 7.22 What do *program control instructions* do? Which LMC instructions would be classified as program control instructions?
- 7.23 What is a stack? Explain how a stack works. Create a diagram that shows how PUSH and POP instructions are used to implement a stack.
- 7.24 What is a *privileged instruction*? Which LMC instructions would normally be privileged?
- 7.25 Show a 32-bit instruction format that allows 32 different op codes. How many bits are available for addressing in your format?
- 7.26 Show an instruction format that could be used to move data or perform arithmetic between two registers. Assume that the instruction is 32 bits wide and that the computer has sixteen general-purpose data registers. If the op code uses 8 bits, how many bits are spares, available for other purposes, such as special addressing techniques?

EXERCISES

- 7.1 Draw side-by-side flow diagrams that show how the Little Man executes a store instruction and the corresponding CPU fetch-execute cycle.
- 7.2 Suppose that the following instructions are found at the given locations in memory:
- ```

20 LDA 50
21 ADD 51
50 724
51 006

```
- Show the contents of the IR, the PC, the MAR, the MDR, and A at the conclusion of instruction 20.
  - Show the contents of each register as each step of the fetch-execute cycle is performed for instruction 21.
- 7.3 One large modern computer has a 48-bit memory address register. How much memory can this computer address?
- 7.4 Why are there two different registers (MAR and MDR) associated with memory? What are the equivalents in the Little Man Computer?
- 7.5 Show the steps of the CPU fetch-execute cycle for the remaining instructions in the Little Man instruction set.
- 7.6 Most of the registers in the machine have two-way copy capability; that is, you can copy to them from another register, and you can copy from them to another register. The MAR, on the other hand, is always used as a destination register; you only copy to the MAR. Explain clearly why this is so.
- 7.7
- What is the effect of shifting an unsigned number in a register two bits to the left? One bit to the right? Assume that 0s are inserted to replace bit locations at the end of the register that have become empty due to the shift.
  - Suppose the number is signed, that is, stored using 2's complement. Now what is the effect of shifting the number?
  - Suppose that the shift excludes the sign bit, so that the sign bit always remains the same. Furthermore, suppose that during a right shift, the sign bit is always used as the insertion bit at the left end of the number (instead of 0). Now what is the effect of these shifts?
- 7.8 If you were building a computer to be used in outer space, would you be likely to use some form of flash memory or RAM as main memory? Why?
- 7.9 Using the register operations indicated in this chapter, show the fetch-execute cycle for an instruction that produces the 2's complement of the number in A. Show the fetch-execute cycle for an instruction that clears A (i.e., sets A to 0).
- 7.10 Many older computers used an alternative to the `BRANCH ON CONDITION` instruction called `SKIP ON CONDITION` that worked as follows: if the condition were true, the computer would skip the following instruction and go on to the one after; otherwise, the next instruction in line would be executed. Programmers usually place a jump instruction in the "in-between" location to branch on a FALSE

condition. Normally, the skip instruction was designed to skip one memory location. If the instruction set uses variable length instructions, however, the task is more difficult, since the skip must still skip around the entire instruction. Assume a Little Man mutant that uses a variable length instruction. The op code is in the first word, and there may be as many as three words following. To make life easy, assume that the third digit of the op code word is a number from 1 to 4, representing the number of words in the instruction. Create a fetch-execute cycle for this machine.

- 7.11 Suppose that the instruction format for a modified Little Man Computer requires two consecutive locations for each instruction. The high-order digits of the instruction are located in the first mail slot, followed by the low-order digits. The IR is large enough to hold the entire instruction and can be addressed as IR [high] and IR [low] to load it. You may assume that the op code part of the instruction uses IR [high] and that the address is found in IR [low]. Write the fetch-execute cycle for an ADD instruction on this machine.
- 7.12 The Little Prince Computer (LPC) is a mutant variation on the LMC. (The LPC is so named because the differences are a royal pain.) The LPC has one additional instruction. The extra instruction requires two consecutive words:

0XX  
0YY

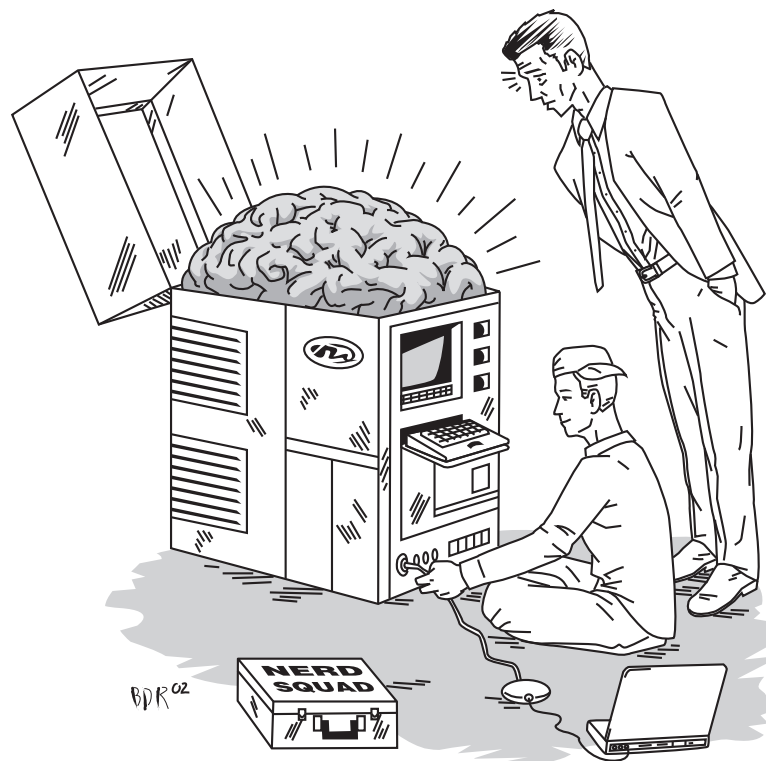
This instruction, known as move, moves data directly from location XX to location YY without affecting the value in the accumulator. To execute this instruction, the Little Prince would need to store the XX data temporarily. He can do this by writing the value on a piece of paper and holding it until he retrieves the second address. The equivalent in a real CPU might be called the intermediate address register, or IAR. Write the fetch-execute cycle for the LPC MOVE instruction.

- 7.13 Generally, the distance that a programmer wants to move from the current instruction location on a `BRANCH ON CONDITION` is fairly small. This suggests that it might be appropriate to design the `BRANCH` instruction in such a way that the new location is calculated relative to the current instruction location. For example, we could design a different LMC instruction 8CX. The C digit would specify the condition on which to branch, and X would be a single-digit relative address. Using 10's complement, this would allow a branch of  $-5$  to  $+4$  locations from the current address. If we were currently executing this instruction at location 24, 803 would cause a branch on negative to location 27. Write a fetch-execute cycle for this `BRANCH ON NEGATIVE RELATIVE` instruction. You may ignore the condition code for this exercise, and you may also assume that the complementary addition is handled correctly. The single-digit address, X, is still found in IR [address].
- 7.14 As computer words get larger and larger, there is a law of diminishing returns: the speed of execution of real application programs does not increase and may, in fact, decrease. Why do you suppose that this is so?
- 7.15 Most modern computers provide a large number of general-purpose registers and very few memory access instructions. Most instructions use these registers to hold data instead of memory. What are the advantages to such an architecture?

- 7.16 Create the fetch-execute cycle for an instruction that moves a value from general purpose register-1 to general purpose register-2. Compare this cycle to the cycle for a `LOAD` instruction. What is the major advantage of the `MOVE` over the `LOAD`?
- 7.17 What are the trade-offs in using a serial bus versus a parallel bus to move data from one place to another?
- 7.18 Until recently, most personal computers used a parallel PCI bus as a backplane to interconnect the various components within the computer, but the PCI bus was rarely, if ever, used to connect external devices to the computer. Modern computers often use a serial adaptation of the PCI bus called PCI Express, which is sometimes made available as a port to connect external devices. Identify at least three shortcomings of the original PCI bus that made external use of the bus impractical. Explain how the PCI Express bus overcomes each of these limitations.
- 7.19 Explain why skew is not a factor in a serial bus.
- 7.20 Point-to-point buses generally omit lines for addressing. Why is this possible? Suppose a point-to-point bus is used to connect two components together where one of the components actually represents multiple addresses. How could a bus with no address lines be used to satisfy the requirement for different addresses in this case?

## CHAPTER 8

# CPU AND MEMORY: DESIGN, ENHANCEMENT, AND IMPLEMENTATION



Adapted by Benjamin Reece

## 8.0 INTRODUCTION

The Little Man Computer design, implemented in binary form, may be sufficient to implement any program, but it is not necessarily a convenient way to do so. It is like traveling overseas by freight steamer instead of by fast plane: it might be fun, but it sure ain't the easiest way to get the job done! Computers today are more sophisticated and flexible, providing a greater variety of instructions, improved methods of addressing memory and manipulating data, and implementation techniques that allow instructions to be executed quickly and efficiently.

In Chapter 7, we discussed the principal features of a CPU: the basic architecture of the CPU, register concept, instruction set, instruction formats, means of addressing memory, and the fetch-execute cycle. In this chapter we will investigate some of the additional design features and implementation techniques that help to give the modern CPU its power.

It probably won't surprise you to know that there are a large number of different ways of performing these tasks. At the same time, it is important to recognize, right from the outset, that additional features and a particular choice of organization do not change the fundamental operation of the computer as we have already described it. Rather, they represent variations on the ideas and techniques that we have already described. These variations can simplify the programmer's task and possibly speed up program execution by creating shortcuts for common operations. However, nothing introduced in this chapter changes the most important idea: that the computer is nothing more than a machine capable of performing simple operations at very high speeds.

The first section investigates different CPU architectures, with particular focus on the modern manifestation and organization of traditional architectures. The section also briefly considers two interesting recent architectures, the Transmeta VLIW and Intel EPIC architectures.

In the second section we consider various CPU features and enhancements, with an emphasis on alternatives to the traditional control unit/ALU CPU organization. We explain how these alternative organizations address major bottlenecks that limit CPU execution speed, with a number of innovative techniques for improving CPU performance.

Section 8.3 looks at memory enhancements. The most significant improvement in memory access speed is cache memory. Cache memory is discussed in considerable depth.

In Section 8.4, we present a general model that includes the features, enhancements, and techniques described in Section 8.2. This model represents the organization of most current CPUs.

Section 8.5 considers the concept of multiprocessing: a computer organization consisting of multiple CPUs directly connected together, sharing memory, major buses,

and I/O. This organization adds both performance enhancement and additional design challenges. We also briefly introduce a complementary feature, simultaneous multithreading. Two types of multiprocessors are presented: the symmetrical multiprocessor is more common. It is well-suited for general purpose computing. An alternative, the master-slave multiprocessor is useful for computer applications characterized by computationally intense, repetitive operations, such as graphics processing.

Finally, in Section 8.6, we present a brief commentary on the implementation of the CPU organization that we have discussed in previous sections.

It is not our intention to overwhelm you in this chapter with myriad details to memorize, nor to help you create a new career as an assembly language programmer or computer hardware engineer, but this chapter will at least introduce you to the major concepts, methods, and terminology used in modern computers. When reading this chapter, remember to keep your focus on the larger picture: the details are just variations on a theme.

## 8.1 CPU ARCHITECTURES

### Overview

A CPU architecture is defined by the basic characteristics and major features of the CPU. (CPU architecture is sometimes called **instruction set architecture (ISA)**.) These characteristics include such things as the number and types of registers, methods of addressing memory, and basic design and layout of the instruction set. It does *not* include consideration of the implementation, instruction execution speed, details of the interface between the CPU and associated computer circuitry, and various optional features. These details are usually referred to as the computer's **organization**. The architecture may or may not include the absence or presence of particular instructions, the amount of addressable memory, or the data widths that are routinely processed by the CPU. Some architectures are more tightly defined than others.

These ideas about computer architecture should not surprise you. Consider house architecture. A split-level ranch house, for example, is easily recognized by its general characteristics, even though there may be wide differences in features, internal organization, and design from one split-level ranch to the next. Conversely, an A-frame house or a Georgian house is recognized by specific, well-defined features that must be present in the design to be recognized as A-frame or Georgian.

There have been many CPU architectures over the years, but only a few with longevity. In most cases, that longevity has resulted from evolution and expansion of the architecture to include new features, always with protection of the integrity of the original architecture, as well as with improved design, technology, and implementation of the architecture.

At present, important CPU architectural families include the IBM mainframe series, the Intel x86 family, the IBM POWER/PowerPC architecture, and the Sun SPARC family. Each of these is characterized by a lifetime exceeding twenty years. The original IBM mainframe architecture is more than forty-five years old. Architectural longevity protects the investment of users by allowing continued use of program applications through system upgrades and replacements.

Most CPU architectures today are variations on the traditional design described in Chapter 7.<sup>1</sup> There have also been a few interesting attempts to create other types, including a stack-based CPU with no general-purpose registers, and two recent architectures called **very long instruction word (VLIW)** from Transmeta and **explicitly parallel instruction computers (EPIC)** from Intel. VLIW and EPIC architectures are too new to assess their long-term value.

It should be noted that each of these architectures is consistent with the broad characteristics that define a von Neumann computer.

## Traditional Modern Architectures

Early CPU architectures were characterized by comparatively few general-purpose registers, a wide variety of memory addressing techniques, a large number of specialized instructions, and instruction words of varying sizes. Researchers in the late 1970s and early 1980s concluded that these characteristics inhibited the efficient organization of the CPU. In particular, their studies revealed that

- Specialized instructions were used rarely, but added hardware complexity to the instruction decoder that slowed down execution of the other instructions that are used frequently.
- The number of data memory accesses and total MOVE instructions could be reduced by increasing the number of general-purpose registers and using those registers to manipulate data and perform calculations. The time to locate and access data in memory is much longer than that required to process data in a register and requires more steps in the fetch-execute cycle of instructions that access memory than those that don't.
- Permitting the use of general purpose registers to hold memory addresses, also, would allow the addressing of large amounts of memory while reducing instruction word size, addressing complexity, and instruction execution time, as well as simplifying the design of programs that require indexing. Reducing the number of available addressing methods simplifies CPU design significantly.
- The use of fixed-length, fixed-format instruction words with the op code and address fields in the same position for every instruction would allow instructions to be fetched and decoded independently and in parallel. With variable-length instructions it is necessary to wait until the previous instruction is decoded in order to establish its length and instruction format.

The Intel x86 is characteristic of older architectures; it has comparatively few general purpose registers, numerous addressing methods, dozens of specialized instructions, and instruction word formats that vary from 1 to 15 bytes in length. In contrast, every instruction

---

<sup>1</sup>Historically, traditional architecture was loosely categorized into one of two types, CISC (complex instruction set computers) or RISC (reduced instruction set computers). In modern times, the dividing line between CISC and RISC architectures has become increasingly blurred as many of the features of each have migrated across the dividing line. Because modern architectures contain the major features of both, it is no longer useful to distinguish one from the other.

in the newer Sun SPARC architecture is the same 32-bit length; there are only five primary instruction word formats, shown earlier in Figure 7.21; and only a single, register-based, LOAD/STORE memory addressing mode.

## VLIW and EPIC Architectures

VLIW (very long instruction word) and EPIC (explicitly parallel instruction computer) architectures represent recent approaches to architectural design. VLIW architecture is represented by the Transmeta Crusoe and Efficeon families of CPU processors. The Intel Itanium IA-64 series is based on EPIC architecture. The basic goal of each of these architectures is to increase execution speed by processing instruction operations in parallel. The primary difficulty in doing so results from the inherently sequential order of the instructions in a program. In particular, the data used in an instruction may depend on the result from a previous instruction. This situation is known as a **data dependency**. Also, branches and loops may alter the sequence, resulting in **control dependency**. Data and control dependencies are discussed in more depth in Section 8.2.

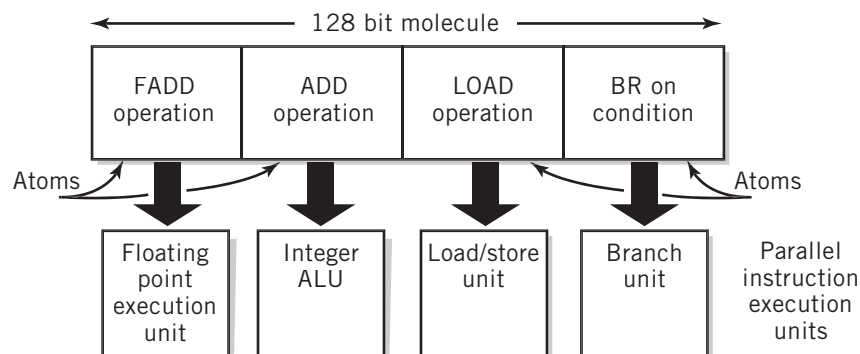
The Transmeta Crusoe architecture is based on a 128-bit instruction word called a molecule. The molecule is divided into four 32-bit *atoms*. Each atom represents an operation similar to those of a normal 32-bit instruction word, however, the atoms are designed in such a way that all four operations may be executed simultaneously in separate execution units. Figure 8.1 shows an example of a typical molecule.

The Crusoe CPU provides 64 general-purpose registers to assure adequate register space for rapid register-to-register processing.

Although a programmer could write programs directly for a Crusoe CPU with the 128-bit word instruction set, that is not the primary goal of the Crusoe architecture. Indeed, the fine details of the instruction set have not been publicly released to date. Instead, the Crusoe CPU is intended for use with a specific software program that translates instruction sets on the fly from other types of CPUs to the Crusoe instruction set for execution on

**FIGURE 8.1**

VLIW Format



the Crusoe CPU. This translator is called a **code-morphing layer**.<sup>2</sup> It is a fundamental component within the Crusoe architecture. It is permanently resident in memory and processes every instruction prior to execution. In addition to instruction translation, the code-morphing layer also reorders the instructions as necessary to eliminate data dependencies and other bottlenecks. Although this sounds like an inefficient way to process instructions, Transmeta has demonstrated that the simplicity of its VLIW design and the sophistication of its code-morphing software allow execution of the Pentium instruction set at speeds comparable to the native execution speeds of a Pentium processor. Transmeta claims, with apparent justification, that this simplicity allows a much simpler CPU design, with fewer transistors and a much lower power consumption, resulting from the elimination of complicated hardware implementation features commonly used to achieve high execution speeds in a conventional CPU design. The Efficeon CPU extends the Crusoe architecture to 256 bits, representing eight 32-bit atoms to be executed simultaneously.

At present, Transmeta has provided a code-morphing layer only for the Pentium CPU family. However, if there were reason to do so, Transmeta could easily create code-morphing layers for other CPUs.

The EPIC architecture, designed by Intel for its IA-64 processor family, attempts to achieve similar goals by slightly different means. The basic instruction set architecture is new, although Intel has built x86 capability into the CPU to support compatibility with its earlier architecture. The IA-64 offers 128 64-bit general-purpose registers and 128 82-bit floating point registers. All instructions are 41 bits wide.

Like the VLIW architecture, the EPIC architecture also organizes instructions into bundles prior to CPU execution, however the methodology and goal are somewhat different. In this case, the instructions *do* represent the native instruction set of the architecture. Instructions are presented to the CPU for execution in 128-bit bundles that include a group of three instructions plus 5 bits that identify the type of each instruction in the bundle.

An assembly language programmer is expected to follow a set of published guidelines that identify dependencies and allow the parallel execution of each bundle. Additionally, bits within each instruction word provide information to the execution unit that identify potential dependencies and other bottlenecks and help the programmer to optimize the code for fast execution. High-level language EPIC compilers must also create code that satisfies the guidelines.

A fundamental difference between the Transmeta VLIW and the Intel EPIC architectures is the placement of responsibility for correct instruction sequencing. The VLIW architecture allows any sequence of instructions to enter the CPU for processing. The code-morphing software, integral to the architecture, handles proper sequencing. The EPIC architecture places the burden on the assembly language programmer or on the program compiler software.

This does not suggest that one architecture is superior to the other. It simply indicates a different approach to the solution of dependencies. Note that the Transmeta VLIW does

---

<sup>2</sup>On a lesser scale, code morphing can also be used to translate complex variable-width instruction words to simpler fixed-width equivalents for faster execution. This technique allows the retention of legacy architectures while permitting the use of modern processing methods. Modern x86 implementations use this approach.

not allow direct assembly language access to the CPU. *All* program code must be processed through code-morphing software. Each architecture offers an interesting new approach to program execution with potential benefits.

## 8.2 CPU FEATURES AND ENHANCEMENTS

### Introduction

We have already introduced you to the fundamental model of a traditional CPU, represented by an instruction set, registers, and a fetch-execute instruction cycle. Additionally, we have presented some of the bells and whistles that have enhanced CPU capability and performance. Some of the enhancements that were introduced in Chapter 7 include direct support for floating point arithmetic, BCD arithmetic, and multimedia processing, as well as the inclusion of additional addressing modes, which simplify data access, increase potential memory size capability while maintaining reasonable instruction word sizes, and improve list and array processing. In this chapter we have already presented a number of architectural enhancements that can improve performance including features that allow parallel execution of instructions to improve processing speed, register-oriented instructions, the use of fixed-width instructions, and integral code-morphing software.

Since the purpose of a computer is to execute programs, the ability of the CPU to execute instructions quickly is an important contributor to performance. Once a particular architecture is established, there remain a number of different ways to increase the instruction execution performance of a computer. One method is to provide a number of CPUs in the computer rather than just one. Since a single CPU can process only one instruction at a time, each additional CPU would, in theory, multiply the performance of the computer by the number of CPUs included. We will return to a discussion of this technique later, in Section 8.5.

Of more interest at the moment are approaches that can be used to improve the performance of an individual CPU. In our introduction to CPU architectures, we suggested a number of possibilities. Some of these require new design, such as the large number of registers and register-to-register instructions that are characteristic of newer architectures. As we already noted, even with older instruction sets, it is often possible to use code morphing to create an intermediate instruction set that is used within the CPU as a substitute for the more complex, original instruction set.

Another difficulty to be overcome when attempting system optimization is that some computer instructions inherently require a large number of fetch-execute steps. Integer division and floating point arithmetic instructions are in this category. Obviously, CPU architects cannot create modern instruction sets that omit these instructions.

In this section, we consider a number of different, but interrelated, approaches to CPU optimization that are applicable to nearly any CPU design. Interestingly enough, you will see that similar approaches can be found in such diverse operations as automobile assembly plants and restaurants.

In Chapter 7, you learned that the fetch-execute cycle is the basic operation by which instructions get executed. You also observed that the steps in a fetch-execute cycle generally must be performed in a particular sequence: an instruction must be fetched and identified before it can be executed, for example. Otherwise the machine would have no way of

knowing what to execute. And so on, step by step, through the entire instruction cycle. *(The first step in cooking spaghetti is to add water to the pot.)* CPU performance can be improved by any method that can perform the fetch-execute cycle steps more quickly or more efficiently.

Then, a program is executed by performing the fetch-execute cycle in a specified sequence, where the sequence is sometimes determined by the program itself during execution. To be provably correct during program execution, the sequence must be maintained and data dependencies resolved in proper order. *(The “cook spaghetti,” “drain spaghetti,” and “prepare sauce” instructions must be completed before the sauce is mixed into the spaghetti.)*

Observe that the limitation to performance results from the serial nature of CPU processing: each instruction requires a sequence of fetch-execute cycle steps, and the program requires the execution of a sequence of these instructions. Thus, the keys to increased performance must rely on methods that either reduce the number of steps in the fetch-execute cycle or reduce the time required for each step in the cycle and, ultimately, reduce the time for each instruction in the program.

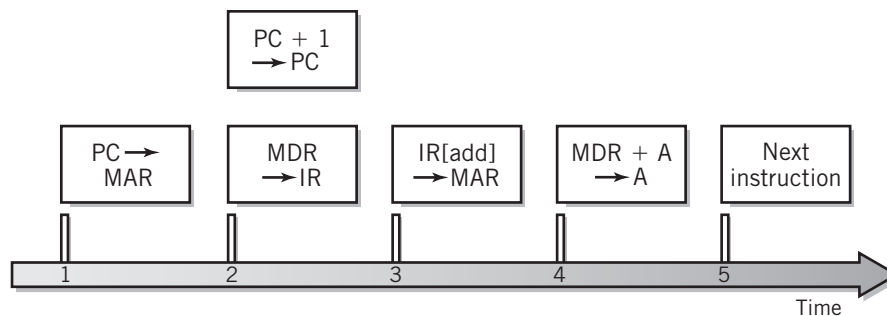
## Fetch-Execute Cycle Timing Issues

As a first step, consider the problem of controlling the timing of each step in the fetch-execute cycle to guarantee perfect CPU operation, to assure that each step follows the previous step, in perfect order, as quickly as possible. There must be enough time between steps to assure that each operation is complete and that data is where it is supposed to be before the next step takes place. As you saw in Chapter 7, most steps in the fetch-execute cycle work by copying, combining, or moving data between various registers. When data is copied, combined, or moved between registers, it takes a short, but finite, amount of time for the data to “settle down” in the new register, that is, for the results of the operation to be correct. This occurs in part because the electronic switches that connect the registers operate at slightly different speeds. (We’re actually talking billionths of a second here!) Also, design allowances must be made for the fact that some operations take longer than others; for example, addition takes more time than a simple data movement. Even more significant is the amount of time that it takes for the address stored in the MAR to activate the correct address in memory. The latter time factor is due to the complex electronic circuitry that is required to identify one group of memory cells out of several million or billion possibilities. This means that reducing the number of memory access steps by using registers for most data operations will inherently improve performance. (We discuss methods to reduce the memory access time, itself, in Section 8.3.) To assure adequate time for each step, the times at which different events take place are synchronized to the pulses of an electronic clock. The **clock** provides a master control as to when each step in the instruction cycle takes place. The pulses of the clock are separated sufficiently to assure that each step has time to complete, with the data settled down, before the results of that step are required by the next step. Thus, use of a faster clock alone does not work if the circuitry cannot keep up.

A timing cycle for a Little Man **ADD** instruction is shown in Figure 8.2. Each block in the diagram represents one step of the fetch-execute cycle. Certain steps that do not have to access memory and which are not dependent on previous steps can actually be performed at the same time. This can reduce the overall number of cycles required for the

**FIGURE 8.2**

Fetch-Execute Timing Diagram

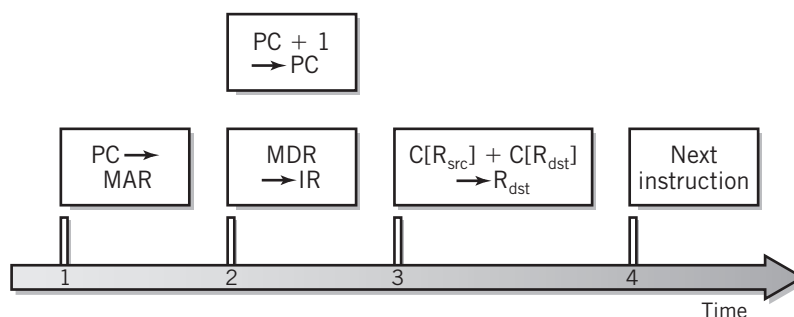


instruction, which speeds up the computer. In this diagram, the data from the program counter has been copied into the memory address register in the first step and is no longer needed. Therefore, the program counter can be incremented at any time after the first step. In Figure 8.2 the PC is incremented in parallel with the  $MDR \rightarrow IR$  step. As shown in the figure, the **ADD** instruction is completed in four clock cycles.

Figure 8.3 shows the improvement possible by using multiple data registers to implement an **ADD** instruction. Since the register-to-register add can be done directly, the number of steps in the cycle is reduced from four to three, with only a single execute step, and the extra time required for the memory access is eliminated.

The built-in clock runs continuously whenever the power to the computer is on. The frequency of its pulses is controlled by a quartz crystal, similar to that which might control your wristwatch. The frequency of the clock and the number of steps required by each instruction determine the speed with which the computer performs useful work.

The pulses of the clock are combined with the data in the instruction register to control electronic switches that open and close in the right sequence to move data from

**FIGURE 8.3**Fetch-Execute Cycle for Register-to-Register **ADD** Instruction

Note:  $C[R_{dst}]$  = contents of destination register

register to register in accordance with the instruction cycle for the particular instruction. The memory activation line described in Section 7.3 is an example of a timing line. The activation line is set up so that it will not turn on until the correct address decode line in the MAR has had time to settle down. If this were not the case, several address lines might be partially turned on, and the data transferred between the memory and MDR might be incorrect. Such errors can obviously not be tolerated, so it is important to control timing accurately.

Conceptually, each pulse of the clock is used to control one step in the sequence, although it is sometimes possible to perform multiple operations within a single step. The clock in the original IBM PC, for example, ran at 4.77 MHz (MHz is pronounced megahertz), which meant that the machine could perform 4.77 million steps every second. If a typical instruction in the IBM PC requires about ten steps, then the original IBM PC could execute about  $(4.77/10)$  or about 0.5 million PC instructions per second. A PC running at 8 MHz, with everything else equal, would perform approximately twice as fast.

There are several factors that determine the number of instructions that a computer can perform in a second. Obviously the clock speed is one major factor. Some current PC computers run their clocks at 3 GHz (pronounced gigahertz) or even more to achieve higher instruction cycle rates.

## A Model for Improved CPU Performance

The current organizational model of a CPU uses three primary, interrelated techniques to address the limitations of the conventional CU/ALU model and to improve performance.

- Implementation of the fetch-execute cycle is divided into two separate units: a fetch unit to retrieve and decode instructions and an execution unit to perform the actual instruction operation. This allows independent, concurrent operation of the two parts of the fetch-execute cycle.
- The model uses an assembly line technique called *pipelining* to allow overlapping between the fetch-execute cycles of sequences of instructions. This reduces the average time needed to complete an instruction.
- The model provides separate execution units for different types of instructions. This makes it possible to separate instructions with different numbers of execution steps for more efficient processing. It also allows the parallel execution of unrelated instructions by directing each instruction to its own execution unit. You have already seen this method applied to the Transmeta and Itanium architectures in Section 8.1.

We next consider each of these techniques in turn.

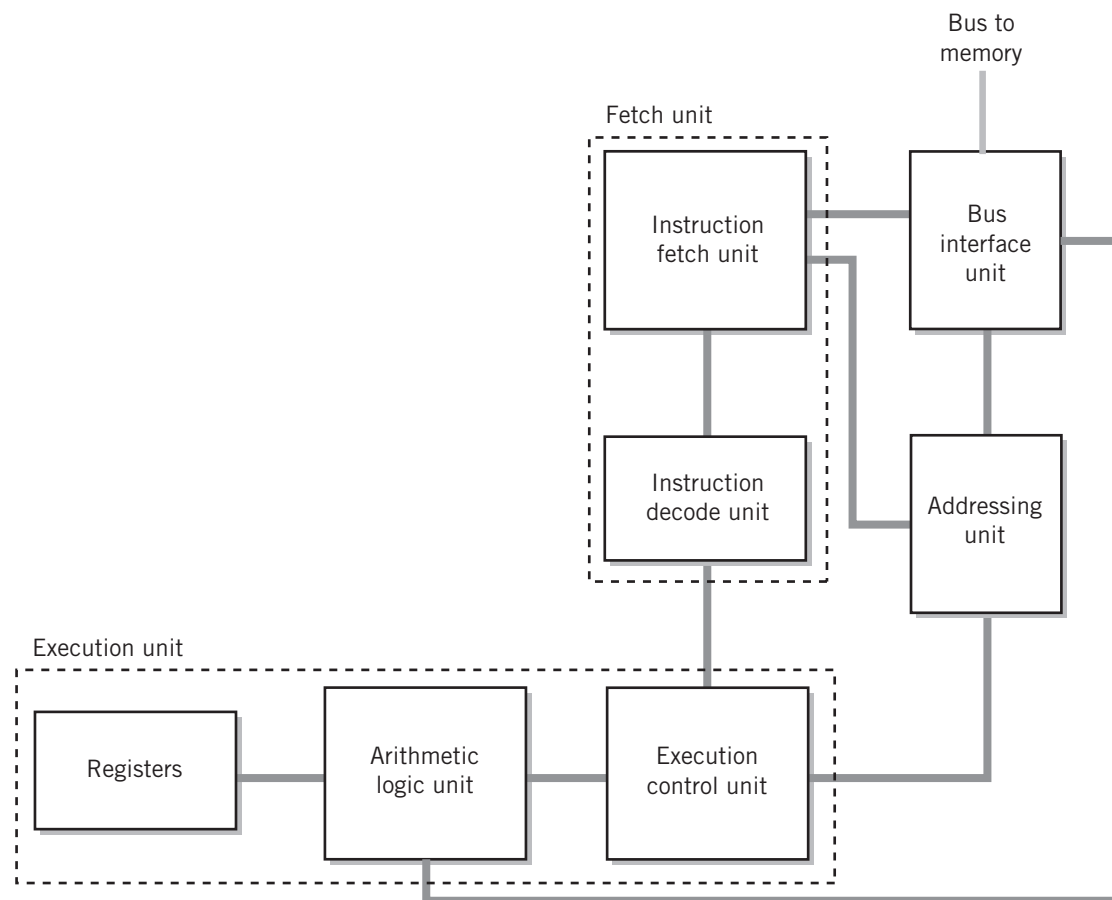
**SEPARATE FETCH UNIT/EXECUTE UNIT** Picture a modified Little Man Computer in which the Little Man has been given an assistant. The assistant will fetch and decode the instructions from the mailboxes at a pace that allows the Little Man to spend his time executing instructions, one after another. Note that a similar division of labor is used in a restaurant: waiters and waitresses gather the food orders from the customers and pass them to the cooks for processing.

The current preferred CPU implementation model divides the CPU similarly into two units, which correspond roughly to the fetch and execute parts of the instruction cycle. To achieve maximum performance, these two parts operate as independently from each other as possible, recognizing, of course, that an instruction must be fetched before it can be decoded and executed. Figure 8.4 illustrates this alternative CPU organization.

The **fetch unit** portion of the CPU consists of an instruction fetch unit and an instruction decode unit. Instructions are fetched from memory by the fetch unit, based on the current address stored in an instruction pointer (IP) register. The fetch unit is designed to fetch several instructions at a time in parallel. The IP register effectively acts as a program counter, but is given a different name to emphasize that there are a number of instructions in the pipeline simultaneously. There is a bus interface unit that provides the logic and memory registers necessary to address memory over the bus. Once an instruction is fetched, it is held in a buffer until it can be decoded and executed. The number of instructions held will depend upon the size of each instruction, the width of the memory bus and memory

**FIGURE 8.4**

Alternative CPU Organization



data register<sup>3</sup>, and the size of the buffer. As instructions are executed, the fetch unit takes advantage of time when the bus is not otherwise being used and attempts to keep the buffer filled with instructions. In general, modern memory buses are wide enough and fast enough that they do not limit instruction retrieval.

Recall that in Figure 8.3 we showed that register-to-register operations could be implemented with only a single memory access, in the fetch portion of the fetch-execute cycle. Fetching the instructions in advance allows the execution of these instructions to take place quickly, without the delay required to access memory.

Instructions in the fetch unit buffer are sent to the instruction decoder unit. The decoder unit identifies the op code. From the op code it determines the type of the instruction. If the instruction set is made up of variable length instructions, it also determines the length of the particular instruction. The decoder then assembles the complete instruction with its operands, ready for execution.

The **execution unit** contains the arithmetic/logic unit and the portion of the control unit that identifies and controls the steps that comprise the execution part for each different instruction. The remainder of what we previously called the control unit is distributed throughout the model, controlling the fetching and decoding of instructions at the correct times, and in the correct order, address generation for instructions and operands, and so forth. The ALU provides the usual computational abilities for the general registers and condition flags.

When the execution unit is ready for an instruction, the instruction decoder unit passes the next instruction to the control unit for execution. Instruction operands requiring memory references are sent to the addressing unit. The addressing unit determines the memory address required, and the appropriate data read or write request is then processed by the bus interface unit.

The bus interface and addressing units operate independently of the instruction pipeline and provide services to the fetch, decode, and execution units as requested by each unit.

**PIPELINING** Look at Figure 8.2 again. In the figure, there are two stages to the execution phase of the instruction cycle. If each stage is implemented separately, so that the instruction simply passes from one stage to the next as it is executed, only one stage is in use at any given time. If there are more steps in the cycle, the same is still true. Thus, to speed up processing even more, modern computers overlap instructions, so that more than one instruction is being worked on at a time. This method is known as **pipelining**. The pipelining concept is one of the major advances in modern computing design. It has been responsible for large increases in program execution speed.

In its simplest form, the idea of pipelining is that as each instruction completes a step, the following instruction moves into the stage just vacated. Thus, when the first instruction is completed, the next one is already one stage short of completion. If there are many steps in the fetch-execute cycle, we can have several instructions at various points in the cycle. The method is similar to an automobile assembly line, where several cars are in different degrees of production at the same time. It still takes the same amount of time to complete

---

<sup>3</sup>Recall that in Chapter 7 we noted that it is common modern practice to retrieve several bytes from memory with each memory access.

one instruction cycle (or one car), but the pipelining technique results in a large overall increase in the average number of instructions performed in a given time.

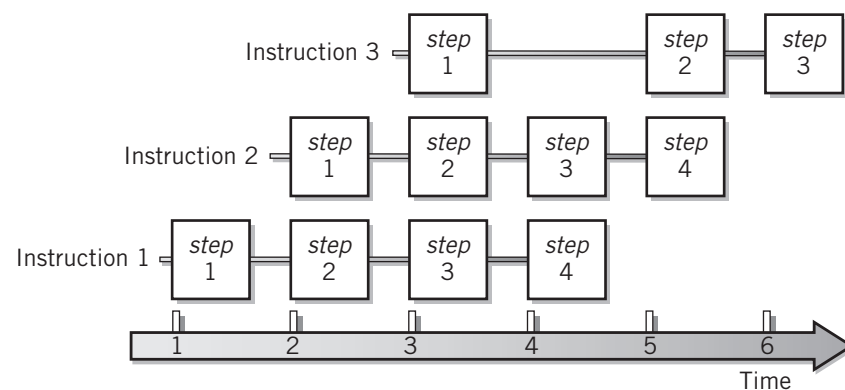
Of course, a branch instruction may invalidate all the instructions in the pipeline at that instant if the branch is taken, and the computer still must have the data from the previous instruction if the next instruction requires it in order to proceed. Modern computers use a variety of techniques to compensate for the branching problem. One common approach is to maintain two or more separate pipelines so that instructions from both possible outcomes can be processed until the direction of the branch is clear. Another approach attempts to predict the probable branch path based on the history of previous execution of the same instruction. The problem of waiting for data results from previous instructions can be alleviated by separating the instructions so that they are not executed one right after the other. Many modern computer designs contain logic that can reorder instructions as they are executed to keep the pipelines full and to minimize situations where a delay is necessary. **Instruction reordering** also makes it possible to provide parallel pipelines, with duplicate CPU logic, so that multiple instructions can actually be executed simultaneously. This technique is equivalent to providing multiple car assembly lines. It is known as **superscalar processing**. We will look at superscalar processing again in the next section.

Pipelining and instruction reordering complicate the electronic circuitry required for the computer and also require careful design to eliminate the possibility of errors occurring under unusual sequences of instructions. (Remember that the programmer must always be able to assume that instructions are executed in the specified order.) Despite the added complexity, these methods are now generally accepted as a means for meeting the demand for more and more computer power. The additional task of analyzing, managing, and steering instructions to the proper execution unit at the proper time is usually combined with instruction fetching and decoding to form a single **instruction unit** that handles all preparation of instructions for execution.

A diagram illustrating pipelining is shown in Figure 8.5. For simplicity, instruction reordering has not been included. The figure shows three instructions, one for each row in the diagram. The “steps” in the diagram represent the sequence of steps in the fetch-execute

**FIGURE 8.5**

Pipelining



cycle for each instruction. Timing marks are indicated along the horizontal axis. The F-E cycle for instruction 3 shows a delay between step 1 and step 2; such a delay might result because the second step of the instruction needs a result from step 3 of the previous instruction, for example, the data in a particular register.

**MULTIPLE, PARALLEL EXECUTION UNITS** It is not useful to pipe different types of instructions through a single pipeline. Different instructions have different numbers of steps in their cycles and, also, there are differences in each step. Instead, the instruction decode unit steers instructions into specific execution units. Each execution unit provides a pipeline that is optimized for one general type of instruction. Typically, a modern CPU will have a LOAD/STORE unit, an integer arithmetic unit, a floating point arithmetic unit, and a branch unit. More powerful CPUs may have multiple execution units for the more commonly used instruction types and, perhaps, may provide other types of execution units as well. Again, an analogy may aid in understanding the concept of multiple, parallel execution units. A simple automobile plant analogy would note that most automobile plants have separate assembly lines for different car models. The most popular models might have multiple assembly lines operating in parallel.

The use of multiple execution units operating in parallel makes it possible to perform the actual execution of several instructions simultaneously.

## Scalar and Superscalar Processor Organization

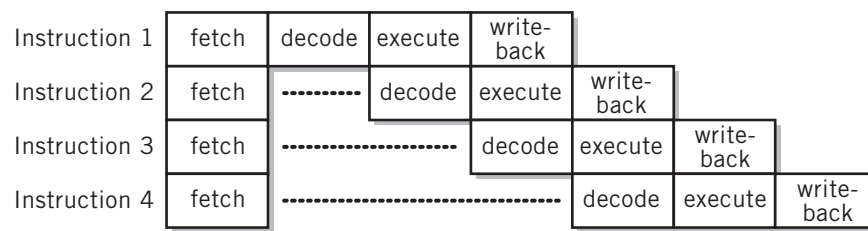
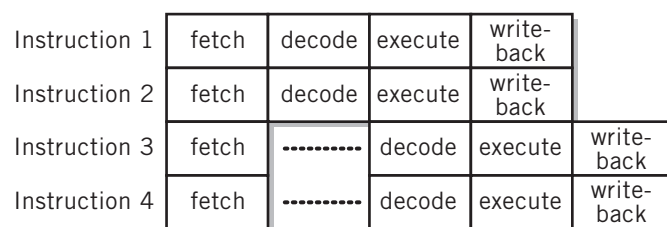
The previous discussion has shown you that modern CPUs achieve high performance by separating the two major phases of the fetch-execute cycle into separate components, then further separating the execution phase into a number of independent execution units, each with pipeline capability. Once a pipeline is filled, an execution unit can complete an instruction with each clock tick. With a single execution unit pipeline, ignoring holes in the pipeline resulting from different instruction types and branch conditions, the CPU can average instruction execution approximately equal to the clock speed of the machine. A processor fulfilling this condition is called a **scalar processor**. With multiple execution units it is possible to process instructions in parallel, with an average rate of more than one instruction per clock cycle. The ability to process more than one instruction per clock cycle is known as **superscalar processing**. Superscalar processing is a standard feature in modern CPUs. Superscalar processing can increase the throughput by double or more. Commonly, current CPU designs produce speed increases of between two and five times.

It is important to remember that pipelining and superscalar processing techniques do not affect the cycle time of any individual instruction. An instruction fetch-execute cycle that requires six clock cycles from start to finish will require six clock cycles whether instructions are performed one at a time or pipelined in parallel with a dozen other instructions. It is the average instruction cycle time that is improved by performing some form of parallel execution. If an individual instruction must be completed for any reason before another can be executed, the CPU must stall for the full cycle time of the first instruction.

Figure 8.6 illustrates the difference between scalar and superscalar processing with pipelining in the execution unit. In the illustration the execution phase of the fetch-execute

**FIGURE 8.6**

## Scalar versus Superscalar Processing

**a. Scalar****b. Superscalar**

cycle is divided into three parts that can be executed separately. Thus, the diagram is divided into steps that fetch, decode, execute, and write back the results of the execute operation. Presumably, each step is carried out by a separate component within the execution unit. To simplify the illustration, we have also assumed that in each case the pipeline is full. Generally, a single fetch unit pipeline is sufficient to fetch multiple instructions, even when multiple execution units are present.

In the scalar processor, Figure 8.6a, each step is assumed to take one clock cycle. If the instructions are all of the same length, they will finish consecutively, as shown in the diagram. More complexity in the instruction set will create bubbles in the pipeline, but does not alter the basic idea that we are illustrating. Panel b of the figure assumes the presence of two execution units. It also assumes that the instructions executing in parallel are independent of each other; that is, the execution of one does not depend upon results from the other. Therefore, two instructions can be executed at a time in parallel, resulting in a substantial improvement in overall instruction completion performance.

Superscalar processing complicates the design of a CPU considerably. There are a number of difficult technical issues that must be resolved to make it possible to execute multiple instructions simultaneously. The most important of these are

- Problems that arise from instructions completing in the wrong order
- Changes in program flow due to branch instructions
- Conflicts for internal CPU resources, particularly general-purpose registers

**OUT-OF-ORDER PROCESSING** Out-of-order instruction execution can cause problems because a later instruction may depend on the results from an earlier instruction. This situation is known as a **hazard** or a *dependency*. If the later instruction completes ahead of the earlier one, the effect of the earlier instruction upon the later cannot be satisfied. The most common type of a dependency is a **data dependency**. This is a situation in which the later instruction is supposed to use the results from the earlier instruction in its calculation. There are other types of dependencies also.

With multiple execution units, it is possible for instructions to complete in the wrong order. There are a number of ways in which this can occur. In the simplest case, an instruction with many steps in its cycle may finish after an instruction with just a few steps, even if it started earlier. As a simple example, a `MULTIPLY` instruction takes longer to execute than a `MOVE` or `ADD` instruction. If a `MULTIPLY` instruction is followed in the program by an `ADD` instruction that adds a constant to the results of the multiplication, the result will be incorrect if the `ADD` instruction is allowed to complete ahead of the `MULTIPLY` instruction. This is an example of data dependency. Data dependency can take several different forms.

Many data dependencies are sufficiently obvious that they can be detected by the CPU. In this case, execution of the dependent instruction is suspended until the results of the earlier instruction are available. This suspension may, itself, cause out-of-order execution, since it may allow another, still later, instruction to complete ahead of the suspended instruction. Some CPUs provide reservation stations within each execution unit or a general instruction pool to hold suspended instructions so that the execution unit may continue processing other instructions.

Finally, some systems intentionally allow out-of-order instruction execution. These CPUs can actually search ahead for instructions without apparent dependencies, to keep the execution units busy. Current Intel x86 CPUs, for example, can search twenty to thirty instructions ahead, if necessary, to find instructions available for execution.

**BRANCH INSTRUCTION PROCESSING** Branch instructions must always be processed ahead of subsequent instructions, since the addresses of the proper subsequent instructions to fetch are determined from the branch instruction. For unconditional branch instructions, this is simple. Branch instructions are identified immediately as they enter the instruction fetch pipeline. The address in the instruction is decoded and used to fill the instruction fetch pipeline with instructions from the new location. Normally, no delay is incurred.

Unfortunately, conditional branch instructions are more difficult, because the condition decision may depend on the results from instructions that have not yet been executed. These situations are known as *flow* or *branch dependencies*. If the wrong branch is in the pipeline, the pipeline must be flushed and refilled, wasting time. Worse yet, an instruction from the wrong branch, that is, one that should not have been executed, can alter a previous result that is still needed.

The solution to the conditional branching problem may be broken into two parts: methods to optimize correct branch selection and methods to prevent errors as a result of conditional branch instructions. Selection of the wrong branch is time wasting, but not fatal. By contrast, incorrect results *must* be prevented.

Errors are prevented by setting the following guideline: although instructions may be executed out of order, they must be completed in the correct order. Since branches and subtle data dependencies can occur, the execution of an instruction out of order may or

may not be valid, so the instruction is executed *speculatively*, that is, on the assumption that its execution will be useful. For this purpose, a separate bank of registers is used to hold results from these instructions until previous instructions are complete. The results are then transferred to their actual register and memory locations, in correct program instruction order. This technique of processing is known as **speculative execution**. On occasion, the results from some speculatively executed instructions must be thrown away, but on the whole, speculative execution results in a performance boost sufficient to justify the extra complexity required.

A few systems place the burden for error prevention on the assembly language programmer or program language compiler by requiring that a certain number of instructions following a conditional branch instruction be independent of the branch. In these systems, one or more instructions sequentially following the branch are *always* executed, regardless of the outcome of the branch.

There are various creative methods that are used in CPUs to optimize conditional branch processing. One possible solution to this problem is to maintain two separate instruction fetch pipelines, one for each possible branch outcome. Instructions may be executed speculatively from *both* branches until the correct pipeline is known. Another solution is to have the CPU attempt to predict the correct path based on program usage or past performance. A loop, for example, may be expected to execute many times before exiting. Therefore, the CPU might assume that a branch to a previous point in the program is usually taken. Some systems provide a **branch history table**, a small amount of dedicated memory built into the CPU that maintains a record of previous choices for each of several branch instructions that have been used in the program being executed to aid in prediction. A few systems even include a “hint” bit in the branch instruction word that can be set by the programmer to tell the CPU the more probable outcome of the branch. Of course, when a branch prediction is incorrect, there is a time delay to purge and refill the fetch pipeline and speculative instructions, but, overall, branch prediction is effective.

**CONFLICT OF RESOURCES** Conflicts between instructions that use the same registers can be prevented by using the same bank of registers that is used to hold the results of speculative instructions until instruction completion. This register bank is given different names by different vendors. They are called variously **rename registers** or **logical registers** or **register alias tables**. The registers in the bank can be renamed to correspond logically to any physical register and assigned to any execution unit. This would allow two instructions using the “same” register to execute simultaneously without holding up each other’s work. At completion of an instruction, the CPU then selects the corresponding physical register and copies the result into it. This must occur in the specified program instruction order.

## 8.3 MEMORY ENHANCEMENTS

Within the instruction fetch-execute cycle, the slowest steps are those that require memory access. Therefore, any improvement in memory access can have a major impact on program processing speed.

The memory in modern computers is usually made up of dynamic random access memory circuit chips. DRAM is inexpensive. Each DRAM chip is capable of storing millions

of bits of data. Dynamic RAM has one major drawback, however. With today's fast CPUs, the access time of DRAM is too slow to keep up with the CPU, and delays must be inserted into the LOAD/STORE execution pipeline to allow memory to keep up. Thus, the use of DRAM is a potential bottleneck in processing. Instructions must be fetched from memory and data must be moved from memory into registers for processing.

The fetch-execute CPU implementation introduced in Section 8.2 reduces instruction fetch delay to a minimum with modern instruction prefetch and branch control technologies, and the increased adoption of register-to-register instructions also reduces delays. Nonetheless, memory accesses are always required ultimately to move the data from memory to register and back, and improvements in memory access still have an impact on processing speed.

As mentioned in Chapter 7, static RAM, or SRAM, is an alternative type of random access memory that is two to three times as fast as DRAM. The inherent memory capacity of SRAM is severely limited, however. SRAM design requires a lot of chip real estate compared to DRAM, due to the fact that SRAM circuitry is more complex and generates a lot of heat that must be dissipated. One or two MB of SRAM requires more space than 64 MB of DRAM, and will cost considerably more.

With today's memory requirements, SRAM is not a practical solution for large amounts of memory except in very expensive computers; therefore, designers have created alternative approaches to fulfill the need for faster memory access. Three different approaches are commonly used to enhance the performance of memory:

- Wide path memory access
- Memory interleaving
- Cache memory

These three methods are complementary. Each has slightly different applicability, and they may be used together in any combination to achieve a particular goal. Of these techniques, the use of cache memory has the most profound effect on system performance.

## Wide Path Memory Access

As mentioned in Chapter 7, Section 7.3, the simplest means to increase memory access is to widen the data path so as to read or write several bytes or words between the CPU and memory with each access; this technique is known as **wide path memory access**. Instead of reading 1 byte at a time, for example, the system can retrieve 2, 4, 8, or even 16 bytes, simultaneously. Most instructions are several bytes long, in any case, and most data is at least 2 bytes, and frequently more. This solution can be implemented easily by widening the bus data path and using a larger memory data register. The system bus on most modern CPUs, for example, has a 64-bit data path and is commonly used to read or write 8 bytes of data with a single memory access.

Within the CPU, these bytes can be separated as required and processed in the usual way. With modern CPU implementation, instruction groups can be passed directly to the instruction unit for parallel execution. As the number of bytes simultaneously accessed is increased, there is a diminishing rate of return, since the circuitry required to separate and direct the bytes to their correct locations increases in complexity, fast memory access becomes more difficult, and yet it becomes less likely that the extra bytes will actually be

used. Even a 64-bit data path is adequate to assure that a pipeline will remain filled and bursts of consecutive 64-bit reads or writes can handle situations that require high-speed access to large blocks of data. Very few systems read and write more than 8 bytes at a time. Most systems read and write a fixed number of bytes at a time, but there are a few systems that can actually read and write a variable number of bytes.

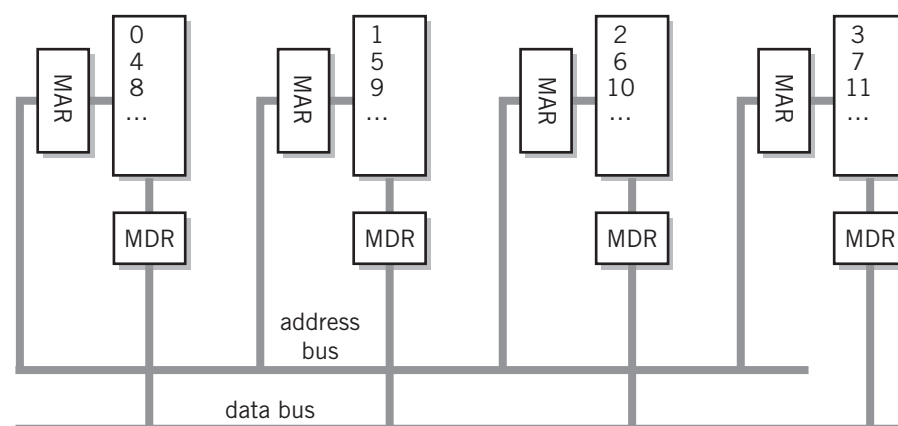
Modern computers are commonly built with standard, off-the-shelf memory circuits and chips that include wide path memory access as a standard feature.

## Memory Interleaving

Another method for increasing the effective rate of memory access is to divide memory into parts, called **memory interleaving**, so that it is possible to access more than one location at a time. Then, each part would have its own address register and data register, and each part is independently accessible. Memory can then accept one read/write request from each part simultaneously. Although it might seem to you that the obvious way to divide up memory would be in blocks, for example, by separating the high addresses into one block and the low addresses into the other, it turns out that as a practical matter it is usually more useful to divide the memory so that successive access points, say, groups of 8 bytes (see above), are in different blocks. Breaking memory up this way is known as ***n*-way interleaving**, where a value of 2 or 4 or some other value is substituted for *n*, depending on the number of separate blocks. For example, two-way interleaving would be designed so that it would be possible to access an odd memory address and an even memory address concurrently. If 8-byte wide access is provided, this would allow the concurrent access to 16 successive bytes at a time. A memory with eight-way interleaving would allow access to eight different locations simultaneously, but the system could not access locations 0, 8, 16, or 24 at the same time, for instance, nor 1, 9, 17, or 25. It *could* access locations 16 and 25 or 30 and 31 concurrently, however. Since memory accesses tend to be successive, memory interleaving can be effective. A diagram of four-way interleaving is shown in Figure 8.7.

**FIGURE 8.7**

Four-Way Memory Interleaving



This method is particularly applicable when multiple devices require access to the same memory. The IBM mainframe architecture, for example, is designed to allow multiple CPUs to access a common memory area; the I/O channel subsystem also has access to the storage area. Thus, several different components may make memory requests at the same time. The IBM S/3033 computer, for example, partitioned memory into eight **logical storage elements**. Each element can independently accept a memory request. Thus, eight memory requests can be processed concurrently.

The personal computer memory that holds images while they are being displayed, known as video RAM, is another example. Changes to part of the video RAM can be made at the same time that another part of the video RAM is being used to produce the actual display on the monitor.

## Cache Memory

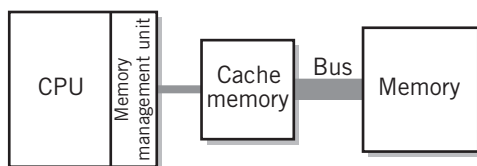
A different strategy is to position a small amount of high-speed memory, for example, SRAM, between the CPU and main storage. This high-speed memory is invisible to the programmer and cannot be directly addressed in the usual way by the CPU. Because it represents a “secret” storage area, it is called **cache memory**. This concept is illustrated in Figure 8.8.

Cache memory is organized differently than regular memory. Cache memory is organized into blocks. Each block provides a small amount of storage, perhaps between 8 and 64 bytes, also known as a **cache line**. The block will be used to hold an exact reproduction of a corresponding amount of storage from somewhere in main memory. Each block also holds a **tag**. The tag identifies the location in main memory that corresponds to the data being held in that block. In other words, taken together, the tags act as a directory that can be used to determine exactly which storage locations from main memory are also available in the cache memory. A typical 64 KB cache memory might consist of 8000 (actually 8192) 8-byte blocks, each with tag.

A simplified, step-by-step illustration of the use of cache memory is shown in Figure 8.9. Every CPU request to main memory, whether data or instruction, is seen first by cache memory. A hardware **cache controller** checks the tags to determine if the memory location of the request is presently stored within the cache. If it is, the cache memory is used as if it were main memory. If the request is a read, the corresponding word from cache memory is simply passed to the CPU. Similarly, if the request is a write, the data from the CPU is stored in the appropriate cache memory location. Satisfying a request in this way is known as a **hit**.

**FIGURE 8.8**

Cache Memory

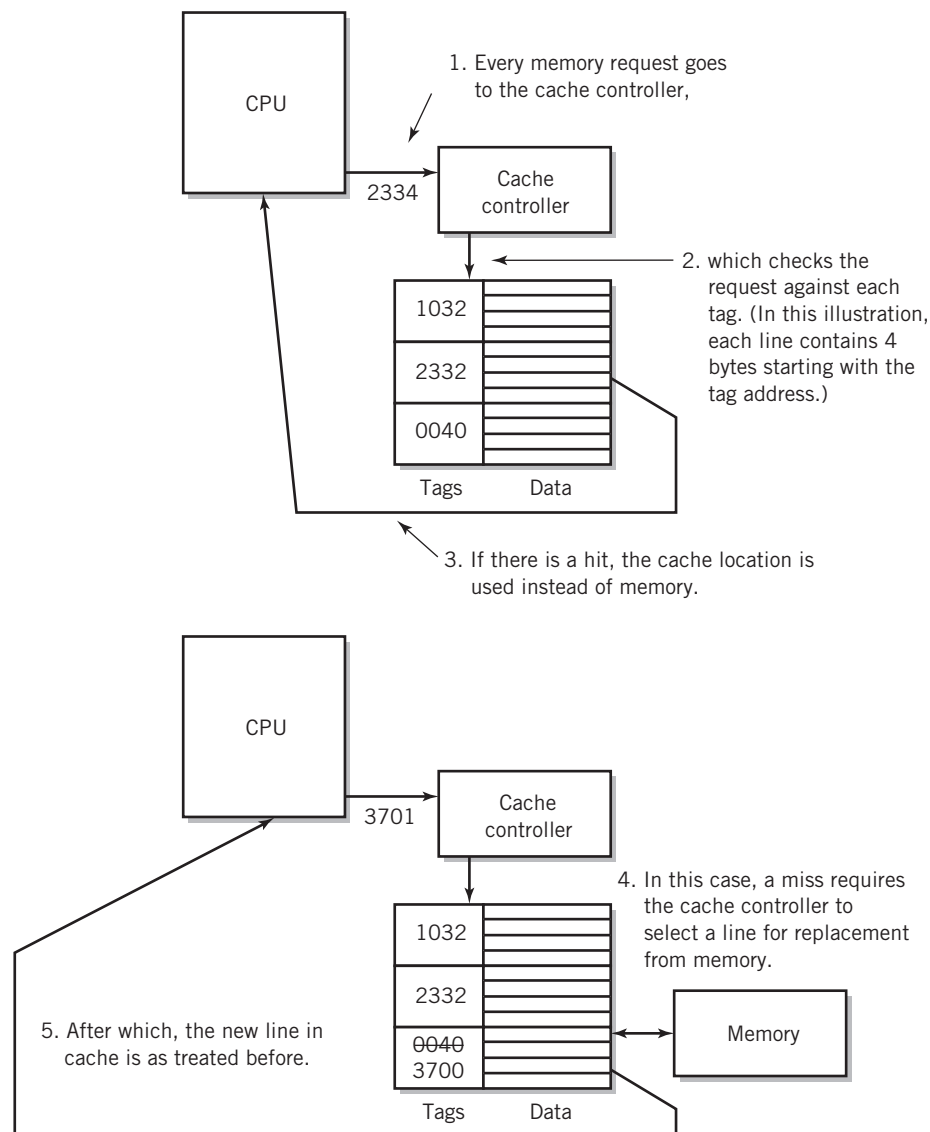


If the required memory data is not already present in cache memory, an extra step is required. In this case, a cache line that includes the required location is copied from memory to the cache. Once this is done, the transfer is made to or from cache memory, as before. The situation in which the request is not already present in cache memory is known as a **miss**. The ratio of hits to the total number of requests is known as the **hit ratio**.

When cache memory is full, some block in cache memory must be selected for replacement. Various algorithms

**FIGURE 8.9**

## Step-by-Step Use of Cache



have been implemented by different computer designers to make this selection, but most commonly, some variation on a *least recently used*, or *LRU*, algorithm is used. An LRU algorithm, as the name implies, keeps track of the usage of each block and replaces the block that was last used the longest time ago.

Cache blocks that have been read, but not altered, can simply be read over during replacement. Memory write requests impose an additional burden on cache memory

operations, since written data must also be written to the main memory to protect the integrity of the program and its data. Two different methods of handling the process of returning changed data from cache to main storage are in common use. The first method, **write through**, writes data back to the main memory immediately upon change in the cache. This method has the advantage that the two copies, cache and main memory, are always kept identical. Some designers use an alternative technique known variously as *store in*, *write back*, or *copy back*. With this technique, the changed data is simply held in cache until the cache line is to be replaced. The **write back** method is faster, since writes to memory are made only when a cache line is actually replaced, but more care is required in the design to ensure that there are no circumstances under which data loss could occur. If two different programs were using the same data in separate cache blocks, for example, and one program changed the data, the design must assure that the other program has access to the updated data.

The entire cache operation is managed by the cache controller. This includes tag searching and matching, write through or write back, and implementation of the algorithm that is used for cache block replacement. The CPU and software are unaware of the presence of cache memory and the activities of the cache controller. We note in passing that to be effective, these operations must be controlled completely by hardware. It is possible to envision using a program to implement the cache block replacement algorithm, for example, but this is not feasible. Since memory accesses would be required to execute the program, this would defeat the entire purpose of cache memory, which is to provide access quickly to a single memory location.

Cache memory works due to a principle known as **locality of reference**. The locality of reference principle states that at any given time, most memory references will be confined to one or a few small regions of memory. If you consider the way that you were taught to write programs, this principle makes sense. Instructions are normally executed sequentially; therefore, adjoining words are likely to be accessed. In a well-written program, most of the instructions being executed at a particular time are part of a small loop or a small procedure or function. Likewise, the data for the program is likely taken from an array. Variables for the program are all stored together. Studies have verified the validity of the locality principle. Cache memory hit ratios of 90 percent and above are common with just a small amount of cache. Since requests that can be fulfilled by the cache memory are fulfilled much faster, the cache memory technique can have a significant impact on the overall performance of the system. Program execution speed improvements of 50 percent and more are common.

The hit ratio is an important measure of system performance. Cache hits can access memory data at or near the speed that instructions are executed, even with sophisticated instruction steering and multiple execution units. When a miss occurs, however, there is a time delay while new data is moved to the cache. The time to move data to the cache is called **stall time**. The stall time is typically long compared to instruction execution time. This can result in a condition in which there are no instructions available to feed to the execution units; the pipelines empty and instruction execution is stalled until the needed cache line is available, reducing performance.

Some modern architectures even provide program instructions to request cache preloading for data or instructions that will be needed soon. This improves execution speed even more. Also, some system designers interleave the cache or implement separate caches

for instructions and data. This allows even more rapid access, since the instruction and its operands can be accessed simultaneously much of the time. Furthermore, design of a separate instruction cache can be simplified, since there is no need to write the instruction cache back to main memory if the architecture imposes a pure coding requirement on the programmer. The trade-off is that accommodating separate instruction and data caches requires additional circuit complexity, and many system designers opt instead for a combined, or *unified*, cache that holds both data and instructions.

It is also possible to provide more than one level of cache memory. Consider the two level cache memory shown in Figure 8.10. This memory will work as follows. The operation begins when the CPU requests an instruction (or piece of data) be read (or written) from memory. If the cache controller for the level closest to the CPU, which we'll call level 1 (normally abbreviated as L1), determines that the requested memory location is presently in the level 1 cache, the instruction is immediately read into the CPU.

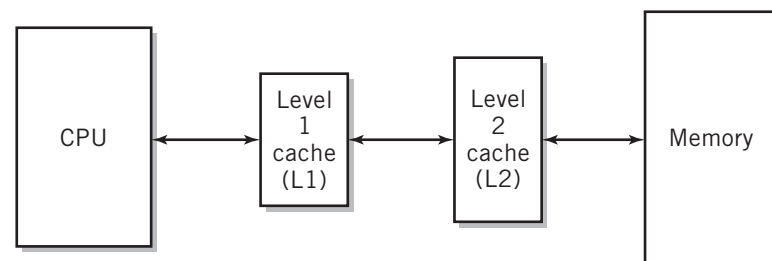
Suppose, however, that the instruction is *not* presently in level 1 cache. In this case, the request is passed on to the controller for level 2 cache. Level 2 cache works in exactly the same way as level 1 cache. If the instruction is presently in the level 2 cache, a cache line containing the instruction is moved to the level 1 cache and then to the CPU. If not, then the level 2 cache controller requests a level 2 cache line from memory, the level 1 cache receives a cache line from the level 2 cache, and the instruction is transferred to the CPU. This technique can be extended to more levels, but there is usually little advantage in expanding beyond level 3.

What does the second level buy us? Most system designers believe that more cache would improve performance enough to be worthwhile. In this case, the system designers provide a second level of cache, external to the chip. A personal computer secondary cache commonly provides an additional 512 KB–2 MB of cache. A typical AMD Athlon 64 processor provides 64 KB of L1 data cache, 64 KB of L1 instruction cache, and 512 KB or 1 MB of level 2 cache within the same package as the CPU. The use of a dedicated on-chip bus between level 1 cache and level 2 cache provides faster response than connecting the level 1 cache to memory or to a level 2 cache on the regular memory bus.

To be useful, the second level of cache must have significantly more memory than the first level; otherwise, the two cache levels would contain the same data, and the secondary cache would serve no purpose. It is also normal to provide a larger cache line in the secondary cache. This increases the likelihood that requests to the secondary cache can be met without going out to main memory every time.

**FIGURE 8.10**

Two-Level Cache



Before leaving the subject of memory caching, a side note: the concept of caching also shows up in other, unrelated but useful, areas of computer system design. For example, caching is used to reduce the time necessary to access data from a disk. In this case, part of main memory can be allocated for use as a **disk cache**. When a disk read or write request is made, the system checks the disk cache first. If the required data is present, no disk access is necessary; otherwise, a disk cache line made up of several adjoining disk blocks is moved from the disk into the disk cache area of memory. Most disk manufacturers now provide separate buffer memory for this purpose. This feature is implemented within the hardware of a disk controller. Another example is the cache of previous Web pages provided by Web browser application software.

All of these examples of caching share the common attribute that they increase performance by providing faster access to data, anticipating its potential need in advance, then storing that data temporarily where it is rapidly available.

## 8.4 THE COMPLEAT MODERN SUPERSCALAR CPU

Figure 8.11 is a model of a CPU block diagram that includes all the ideas just discussed. The design shown in this diagram is very similar to the one used in Sun SPARC and IBM Power and PowerPC processors and, with minor variations, to that used in various generations of the Intel Pentium and Itanium families, as well as various IBM mainframe processors. As you would expect, the CPU is organized into modules that reflect the superscalar, pipelined nature of the architecture. Although it is difficult to identify the familiar components that we introduced in Chapter 7, the control unit, arithmetic/logic unit, program counter, and the like, they are indeed embedded into the design, as you saw in Figure 8.4. The control unit operation is distributed through much of the diagram, controlling each step of the usual fetch-execute cycle as instructions flow through different blocks in the CPU. The functions of the arithmetic/logic unit are found within the integer unit. The program counter is part of the instruction unit.

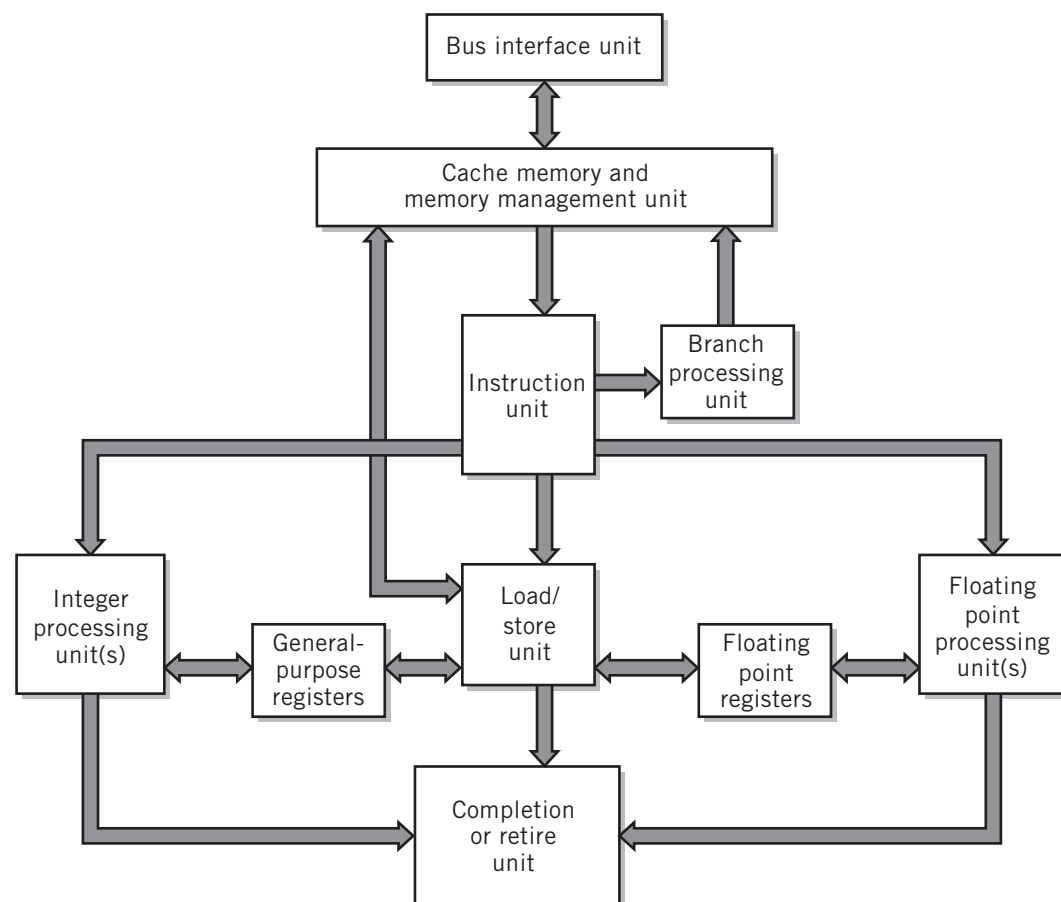
In operation, instructions are fetched from memory by the memory management unit as they are needed for execution, and placed into a pipeline within the instruction unit. The instructions are also partially decoded within the instruction unit, to determine the type of instruction that is being executed. This allows branch instructions to be passed quickly to the branch processing unit for analysis of future instruction flow.

Instructions are actually executed in one of several types of execution units. Each execution unit has a pipeline designed to optimize the steps of the execute cycle for a particular type of instruction.

As you can see from the block diagram, there are separate execution units for branch instructions, for integer instructions, for floating point instructions, and for load and store instructions. Some processors provide multiple integer execution units to increase the processing capacity of the CPU still further. Some models also have a separate system register unit for executing system-level instructions. Some CPUs combine the load/store instructions into the integer unit. The PowerPC provides reservation stations in each execution unit. The Intel Pentium processors provide a general instruction pool where decoded instructions from the instruction unit are held as they await operand data from memory and from unresolved data dependencies. The Pentium instruction pool also holds

**FIGURE 8.11**

Modern CPU Block Diagram



completed instructions after execution until they can be retired in order. The Pentium also separates the LOAD and STORE execution units.

The instruction unit is responsible for maintaining the fetch pipeline and for dispatching instructions. Because branch instructions affect the addresses of the following instructions in the pipeline, they are processed immediately. Other instructions are processed as space becomes available in the appropriate execution unit(s). Branch prediction is usually built into the branch unit. When conditional branches occur, execution of instructions continues speculatively along the predicted branch until the condition is resolved. Also, the use of multiple execution units makes it possible that instructions will execute in the wrong order, since some instructions may have to wait for operands resulting from other instructions and since the pipelines in each execution unit are of different lengths. As we noted earlier, some current superscalar processors can look ahead several instructions to find instructions that can be processed independently of the program order to prevent

delays or errors arising from data dependency. The ability to process instructions out of order is an important factor in the effectiveness of these processors. The completion or “retire” unit accepts or rejects speculative instructions, stores results in the appropriate physical registers and cache memory locations, and retires instructions in the correct program order, assuring correct program flow. The Crusoe and Itanium architectures prevent out-of-order retirement by reordering the instructions prior to execution. This simplifies the CPU design.

From this discussion, you can see that the modern CPU includes many sophisticated features designed to streamline the basically simple fetch-execute cycle for high-performance processing. The modern CPU features different types of execution units, tailored to the needs of different types of instructions, and a complex steering system that can steer instructions through the instruction unit to available execution units, manage operands, and retire instructions in correct program order. The goal of each of these techniques is to increase the parallelism of instruction execution, while maintaining the basic sequential characteristic of a von Neumann computer architecture.

As a brief diversion, consider the similarities between the operation of the modern CPU and the operation of a moderately large restaurant. Each of the waiters and waitresses taking orders represent the fetch unit fetching instructions. Customers’ orders are fed to the kitchen, where they are sorted into categories: soup orders to the soup chef, salads to the salad chef, entrées to the entrée chef, and so on. Typically, the entrée chef will have the most complex orders to fill, equivalent to the longest pipeline in the CPU. If the kitchen is large, the entrée area will be further subdivided into multiple execution areas: frying, baking, and so on, and there may be multiple cooks working in the busiest areas. As with the programs being executed in a computer, there are dependencies between the various cooks. For example, green beans must be blanched before they may be placed in the salad. Finally, we observe that, like computer program instructions, the restaurant must provide food from the kitchen to the customers in the proper sequence, and with appropriate timing, to satisfy the customers’ requirements.

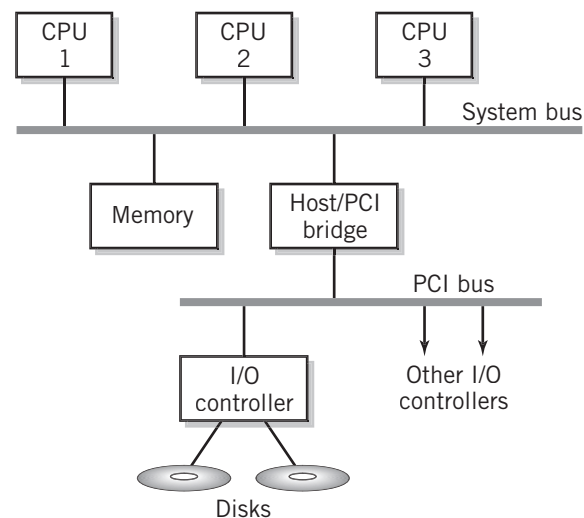
In this section we have introduced the basic ideas of superscalar processing, briefly indicated the difficulties, and explained the reasoning for its use. There are many excellent references listed in the For Further Reading if you are interested in more of the details of superscalar processing and modern CPU design.

## 8.5 MULTIPROCESSING

One obvious way to increase performance in a computer system is to increase the number of CPUs. Computers that have multiple CPUs within a single computer, sharing some or all of the system’s memory and I/O facilities, are called **multiprocessor systems**, or sometimes **tightly coupled systems**. When multiple CPU processors are supplied within a single integrated circuit, they are more commonly called **multicore processors**. Figure 8.12 shows a typical multiprocessor configuration. All the processors in a multiprocessor configuration have access to the same programs and data in shared memory and to the same I/O devices, so it is possible to divide program execution between different CPUs. Furthermore, programs or pieces of programs may be run in any CPU that is available, so that each additional processor extends the power available for multitasking in a multiprocessing system, at least within the capability of the shared components, the memory, buses, and I/O controllers.

**FIGURE 8.12**

Typical Multiprocessing System Configuration



Under ideal conditions, each CPU processes its own assigned sequence of program instructions independently. Thus, a dual-core processor effectively doubles the number of instructions executed in a given time, a quad-core processor would quadruple the rate, and so forth. Of course this assumes that there are multiple independent tasks available to be executed simultaneously. Since modern computer systems are normally executing many programs and segments of programs concurrently this is nearly always the case.

In practice, increasing the number of CPUs is, in fact, usually effective, although, as the number of CPUs increases, the value of the additional CPUs diminishes because of the overhead required to distribute the instructions in a useful way among the different CPUs and the conflicts among the CPUs for shared resources, such as memory, I/O, and access to the shared buses. With the exception of certain, specialized systems, there are rarely more than sixteen CPUs

sharing the workload in a multiprocessing computer; more commonly today, a multiprocessor might consist of two, four, or eight core CPUs within a single chip. Still, each core in the chip is a full-blown superscalar CPU, of the type discussed in the previous sections of this chapter.

Although increased computing power is a significant motivation for multiprocessing, there are other considerations that make multiprocessing attractive:

- Since the execution speed of a CPU is directly related to the clock speed of the CPU, equivalent processing power can be achieved at much lower clock speeds, reducing power consumption, heat, and stress within the various computer components.
- Programs can be divided into independent pieces, and the different parts executed simultaneously on multiple CPUs.
- With multiprocessing, increasing computational power may be achieved by adding more CPUs, which is relatively inexpensive.
- Data dependencies and cache memory misses can stall the pipelines in a single CPU. Multiprocessing allows the computer to continue instruction execution in the other CPUs, increasing overall throughput.

Assignment of work to the various processors is the responsibility of the operating system. Work is assigned from among the programs available to be executed, or, more commonly, from independent segments of those programs called **threads**. Since each of the CPUs has access to the same memory and I/O, any CPU can theoretically execute any thread or program currently in memory, including the operating system. This raises the question of control of the system. There are two basic ways of configuring a multiprocessing system:

- **Master-slave multiprocessing**, in which one CPU, the *master*, manages the system, and controls all resources and scheduling. Only the master may execute the operating system. Other CPUs are *slaves*, performing work assigned to them by the master.
- **Symmetrical multiprocessing (SMP)**, in which each CPU has identical access to the operating system, and to all system resources, including memory. Each CPU schedules its own work, within parameters, constraints, and priorities set by the operating system. In a normal SMP configuration, each of the CPUs is identical.

A number of CPUs also implement a simplified, limited form of multiprocessing using parallel execution units within a single CPU to process two or more threads simultaneously. This technique is called **simultaneous thread multiprocessing (STM)**. STM is also known as *hyperthreading*. STM is particularly useful in dealing with cache stalls, because the CPU can be kept busy working on the alternative thread or threads. The operating system manages STM in a manner similar to SMP. Since STM operates within a single CPU and SMP operates between CPUs, STM and SMP can be used together.

For general purpose computing, the symmetrical configuration has many advantages. Because every CPU is equal, every CPU has equal access to the operating system. Any CPU can execute any task and can process any interrupt.<sup>4</sup> Processors are all kept equally busy, since each processor can dispatch its own work as it needs it. Thus, the workload is well balanced. It is easy to implement fault-tolerant computing with a symmetrical configuration—critical operations are simply dispatched to all CPUs simultaneously. Furthermore, a failure in a single CPU may reduce overall system performance, but it will not cause system failure. As an interesting aside, note that a program may execute on a different CPU each time it is dispatched, although most SMP systems provide a means to *lock* a program onto a particular CPU, if desired. Thus, the symmetrical configuration offers important capabilities for multiprocessing: maximum utilization of each CPU, flexibility, high reliability, and optional support for fault-tolerant computing. Most modern general purpose multiprocessing systems are SMP systems.

Because of the somewhat limited flexibility in distributing the workload, the master-slave configuration is usually considered less suitable for general purpose computing. In a master-slave configuration the master is likely to be the busiest CPU in the system. If a slave requires a work assignment while the master is busy, the slave will have to wait until the master is free. Furthermore, since the master handles all I/O requests and interrupts, a heavily loaded system will cause a backload in the master. If slaves are dependent on the results of these requests, the system is effectively stalled.

Conversely, there are a number of specialized computing applications for which the master-slave configuration is particularly well suited. These applications are characterized by a need for a master control program, supported by repetitive or continuous, computation- and data-intensive, time-critical tasks. For example, the processor in a game controller

---

<sup>4</sup>Interrupts are a special feature of the CPU in which outside events such as mouse movements and power interruptions can affect the sequence of instructions processed by the CPU. Interrupts are discussed in detail in Chapter 9.

must execute the code that plays the game. At the same time, it requires support that can rapidly calculate and display new images based on movements of the objects in the image, compute shadings and light reflections resulting from the movements; often, the processor must create new pixel values for every pixel in the image. It must also be able to create the appropriate reactions and display in response to events that occur, such as explosions or fires or objects bouncing off a wall, and more.

Many important applications in economics, biology, physics, and finance, particularly those based on simulation and modeling, have similar requirements.

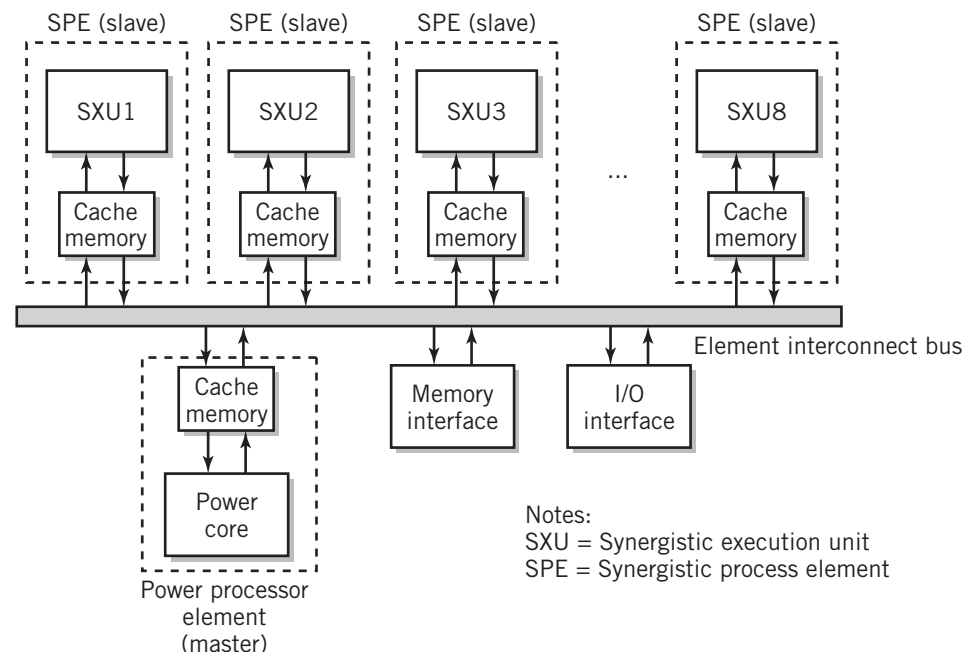
### EXAMPLE

The recently developed Cell Broadband Engine processor is organized in a master-slave configuration. It was developed jointly by IBM, Sony, and Toshiba as the first of a new generation of processors intended for use in high-performance intensive computing applications. It is the main processor used in the Sony PlayStation 3.

A block diagram of the Cell processor is shown in Figure 8.13. The master processor is similar to a 64-bit PowerPC CPU. There are eight slave processors. A high-speed bus interconnects the master processor and each of the slave processors. For those interested, a more detailed description of the PowerPC Cell processor can be found in Gschwind, et. al. [GSCH06].

**FIGURE 8.13**

Cell Processor Block Diagram



## 8.6 A FEW COMMENTS ON IMPLEMENTATION

It is not the intention of this book to discuss the electronic implementation of the computer in any detail. A brief introduction is provided in Supplementary Chapter 1, but the details of such a discussion are better left to an engineering textbook. There are several good computer engineering textbooks listed in the *For Further Reading* for the supplementary chapter if you are interested in learning more about how the computer works.

Although the increased capacity of current integrated circuit technology has allowed computer designers the option of creating very complex circuits, much of that capacity is currently used to provide increased capability in the form of multiple execution units, increased amounts of cache memory, and multicore processing; the basic design and implementation of a processor is simpler than you might imagine.

If you look again at the instruction classes that constitute the operations of a CPU together with the fetch-execute cycles that make up each of the instructions, you can see that the great majority of operations within the CPU consist of moving data from one register to another. The steps

PC  $\rightarrow$  MAR and  
MDR  $\rightarrow$  IR

are examples of this type of operation.

In addition, we must include the capability to add data to a register with data from another register or from a constant (usually the constant 1 or  $-1$ ), the capability to perform simple Boolean functions (AND, OR, NOT) on data in registers, and the capability to shift the data in a register to the left or right. Finally, the CPU must include the capability to make simple decisions based on the values stored in flags and registers (conditional branches). All of these operations are under the timed control of a clock. Control unit logic opens and closes switches at the right times to control the individual operations and the movement of data from one component within the CPU to another.

And for all practical purposes, that's about it. The small number of different operations used in a CPU suggest that the CPU can be directly implemented in electronic hardware, and indeed that is the case. In Supplementary Chapter 1, we demonstrate for the curious reader, in somewhat simplified fashion, that all of the preceding functions can be implemented using logic gates that perform Boolean algebra. The registers, flags, and counters that control timing are made up of electronic devices called flip-flops, which are, themselves, made up of logic gates.

So, as you can see, the basic hardware implementation of the CPU is relatively straightforward and simple. Although the addition of pipelining, superscaling, and other features complicates the design, it is possible, with careful design, to implement and produce an extremely fast and efficient CPU at low cost and in large quantities.

## SUMMARY AND REVIEW

---

In this chapter we presented a number of different techniques that are used to enhance the power and flexibility of a CPU. We began with a discussion of three different approaches to CPU architecture, with particular emphasis on traditional computer architecture. We presented the advantages, disadvantages, and trade-offs for each architecture.

Next, we looked at the various aspects of instruction execution in a CPU, with the purpose of improving performance. This discussion culminated in the presentation of a model for an alternative organization that preserves the basic rules of execution, but allows much faster instruction execution. The important features of this model include separating the fetch-execute cycle into two separate fetch and execute units that can operate in parallel, pipelining to allow instructions to execute in an assembly line, and multiple execution units to allow parallel execution of unrelated instructions. A variety of innovative techniques, including rename registers, speculative execution, out-of-order execution, and branch prediction help to reduce bottlenecks and contribute to the performance. We noted that the resulting model is capable of superscalar processing, with instructions processed at average rates far exceeding the cycle rate of the clock.

We then turned our attention to memory enhancements, particularly the techniques and benefits of cache memory, a fast, intermediate memory between the CPU and regular memory. Following this, we put together a model of the complete superscalar CPU that contained all of the features that we had presented up to this point.

To increase performance even further, it is possible to combine CPUs into multiple units that share memory, buses, I/O, and other resources, a concept called multiprocessing. We presented two different configurations of multiprocessors.

We concluded the chapter with a brief introduction to the technology used to implement the modern processor.

With all of the advances and variations in technology, architecture, and organization that we have introduced in this chapter, and despite all of the different types of computers, applications, and uses for computers that are available today, it is important to remember that, regardless of the specifics, every current CPU conforms to the basic model created by Von Neumann more than a half century ago. There is little evidence that the basic concepts that govern CPU operation are likely to change in the near future.

## FOR FURTHER READING

---

The many references used to write this chapter are all listed in the bibliography section at the end of the book. The following books and articles are particularly useful for their clear descriptions and explanations of these topics. Stallings [STAL05] and Tanenbaum [TANE05] describe the different types of architectures, focusing on the differences between CISC and RISC architectures in great depth. Information about the VLIW and EPIC architectures may be found at the Transmeta and Intel websites, respectively. The IBM website contains a wealth of information about zSeries, POWER, and Cell architectures, including the *Redbooks*, which are free, downloadable, book-length explanations of various computer topics. These range in difficulty from beginner to highly technical. Intel.com (for the x86 series) and sun.com (for the SPARC architecture) are other useful brand-specific websites.

Instruction sets, instruction formats, and addressing are discussed at length in every computer architecture textbook. The book by Patterson and Hennessy [PATT07] covers the topics of Chapters 7 and 8 thoroughly, and has the additional benefit of being highly readable. A more advanced treatment, by the same authors, is found in [HENN06]. Good discussions of multiprocessing are also found in Patterson and Hennessy and in Tanenbaum. Two readable websites introducing the Cell processor are Gschwind, et. al. [GSCH06] and Moore [MOOR06].

A different approach to this material is to compare the architectures of various machines. The book by Tabak [TAB95] looks at several different CPUs in detail. Most of these CPUs are obsolete, but the comparison between different architectures is useful. There are textbooks and trade books devoted to the architecture of every major CPU. The most thorough discussion of the x86 architecture is found in Mueller [MUE08]. I also recommend Brey [BREY08], Messmer [MESS01], and Sargent and Shoemaker [SARG95] for the Intel x86 series. The PC System Architecture series is a collection of short books describing the architectures of various parts of computers. Volume 5 [SHAN04] describes the evolution and architecture of the Pentium 4. The case studies provided in Supplementary Chapter 2 are additional information resources.

Stallings also includes an entire chapter on superscalar processors. Clear, detailed discussions of all aspects of CPU and memory design can be found in the two books by Patterson and Hennessy [PATT07, HENN06]. There are many additional references in each of these books. Specific discussions of the superscalar processing techniques for particular CPUs can be found in Liptay [LIPT92] for the IBM ES/9000 mainframe, in Becker and colleagues [BECK93], Thompson and Ryan [THOM94], Burgess and colleagues [BURG94], and Ryan [RYAN93] for the PowerPC, and “Tour of the P6” [THOR95] for the P6.

An alternative approach to the topics in this chapter can be found in any assembly language textbook. There are many good books on these topics, with new ones appearing every day. A website such as Amazon is a good resource for identifying the best currently available.

## KEY CONCEPTS AND TERMS

|                             |                                    |                                   |
|-----------------------------|------------------------------------|-----------------------------------|
| branch history table        | instruction set architecture (ISA) | scalar processing                 |
| cache controller            | instruction unit                   | simultaneous thread               |
| cache line                  | locality of reference              | multiprocessing (STM)             |
| clock                       | logical register                   | speculative execution             |
| code morphing layer         | logical storage elements           | stall time                        |
| control dependency          | master-slave                       | superscalar processing            |
| data dependency             | multiprocessing                    | symmetrical                       |
| disk cache                  | memory interleaving                | multiprocessing (SMP)             |
| execution unit              | miss                               | tag                               |
| explicitly parallel         | multiprocessor systems             | threads                           |
| instruction computer (EPIC) | multicore processor                | tightly coupled system            |
| fetch unit                  | n-way interleaving                 | very long instruction word (VLIW) |
| hazard                      | organisation                       | wide path memory access           |
| hit                         | pipelining                         | write-back                        |
| hit ratio                   | register alias table               | write through                     |
| instruction reordering      | rename register                    |                                   |

## READING REVIEW QUESTIONS

- 8.1 The x86 series is an example of a CPU *architecture*. As you are probably aware, there are a number of different chips, including some from different manufacturers even,

that qualify as x86 CPUs. What, *exactly*, defines the x86 architecture? What word defines the difference between the various CPUs that share the same architecture? Name at least one different CPU architecture.

- 8.2 What is the major performance advantage that results from the use of multiple general-purpose data registers?
- 8.3 Explain the advantage in implementing separate fetch and execute units in a CPU. What additional task is implemented in the fetch unit as a performance enhancement measure?
- 8.4 Explain how pipelining serves to reduce the average number of steps in the execution part of the fetch-execute cycle.
- 8.5 Which class of instructions can reduce performance by potentially invalidating the instructions in a pipeline? Identify two methods that can be used to partially overcome this problem.
- 8.6 Most CPUs today are *superscalar*. What does that mean?
- 8.7 The use of multiple execution units can improve performance but also cause problems called *hazards* or *dependencies*. Explain how a hazard can occur. How can hazards be managed?
- 8.8 What is a *rename register*? What is it used for?
- 8.9 What performance improvement is offered by *memory interleaving*?
- 8.10 What specific performance improvement is offered by the use of cache memory?
- 8.11 Describe how cache memory is organized. What is a *cache line*? How is it used?
- 8.12 Explain the *hit ratio* in cache memory.
- 8.13 Explain the difference between cache *write-through* and cache *write-back*. Which method is safer? Which method is faster?
- 8.14 Explain what takes place when cache memory is full.
- 8.15 Explain the *locality of reference* principle and its relationship to cache memory performance and the hit ratio.
- 8.16 When a system has multiple levels of cache memory, L2 always has more memory than L1. Why is this necessary?
- 8.17 Modern computers are usually described as multicore. What does this mean? Under ideal conditions, what performance gain would be achieved using a four-core processor over a single-core processor?
- 8.18 Identify and briefly explain two different ways of configuring a multiprocessing system. Which configuration is more effective for general purpose computing? Which configuration is more effective for handling specialized processing tasks, such as those used in game applications?

## EXERCISES

- 8.1 Find a good reference that describes the x86 chip. Discuss the features of the architecture that make superscalar processing possible in this chip. What limitations does the Pentium architecture impose on its superscalar processing?
- 8.2 Consider a CPU that implements a single instruction fetch-decode-execute-write-back pipeline for scalar processing. The execution unit of this pipeline assumes

that the execution stage requires one step. Describe, and show in diagram form, what happens when an instruction that requires one execution step follows one that requires four execution steps.

- 8.3
  - a. Consider a CPU with two parallel integer execution units. An addition instruction requires 2 clock pulses to complete execution, and a multiplication requires 15 clock pulses. Now assume the following situation: the program is to multiply two numbers, located in registers R2 and R4, and store the results in R5. The following instruction adds the number in R5 to the number in R2 and stores the result in R5. The CPU does not stall for data dependencies, and both instructions have access to an execution unit simultaneously. The initial values of R2, R4, and R5 are 3, 8, and 0, respectively. What is the result? Now assume that the CPU does handle data dependencies correctly. What is the result? If we define wasted time as time in which an execution unit is not busy, how much time is wasted in this example?
  - b. Now assume that a later instruction in the fetch pipeline has no data dependencies. It adds the value in R1, initially 4, to the value in R4 and stores the result in R5. Data dependencies are handled correctly. There are no rename registers, and the CPU retires instructions in order. What happens? If the CPU provides rename registers, what happens? What effect does out-of-order execution have upon the time required to execute this program?
- 8.4 Suppose that a CPU always executes the two instructions following a branch instruction, regardless of whether the branch is taken or not. Explain how this can eliminate most of the delay resulting from branch dependency in a pipelined CPU. What penalties or restrictions does this impose on the programs that are executed on this machine?
- 8.5 Some systems use a branch prediction method known as static branch prediction, so called because the prediction is made on the basis of the instruction, without regard to history. One possible scenario would have the system predict that all conditional backward branches are taken and all forward conditional branches are not taken. Recall your experience with programming in the Little Man Computer language. Would this algorithm be effective? Why or why not? What aspects of normal programming, in any programming language, support your conclusion?
- 8.6 How would you modify the Little Man Computer to implement the pipelined instruction fetch-execution unit model that was described in this chapter? What would it take to supply multiple execution units? Describe your modified LMC in detail and show how an instruction flows through it.
- 8.7
  - a. Suppose we are trying to determine the speed of a computer that executes the Little Man instruction set. The LOAD and STORE instructions each make up about 25% of the instructions in a typical program; ADD, SUBTRACT, IN, and OUT take 10% each. The various branches each take about 5%. The HALT instruction is almost never used (a maximum of once each program, of course!). Determine the average number of instructions executed each second if the clock ticks at 100 MHz.
  - b. Now suppose that the CPU is pipelined, so that each instruction is fetched while another instruction is executing. (You may also neglect the time required

to refill the pipeline during branches and at the start of program execution.)  
 What is the average number of instructions that can be executed each second with the same clock in this case?

- 8.8 The goal of scalar processing is to produce, on average, the execution of one instruction per clock tick. If the clock ticks at a rate of 2 GHz, how many instructions per second can this computer execute? How many instructions would a 2 GHz superscalar processor that processes three instructions per clock cycle execute?
- 8.9 Consider a cache memory that provides three hundred 16-byte blocks. Now consider that you are processing all the data in a two-dimensional array of, say, four hundred rows by four hundred columns, using a pair of nested loops. Assume that the program stores the array column by column. You can write your program to nest the loops in either direction, that is, process row by row or column by column. Explain which way you would choose to process the data. What is the advantage? Conversely, what is the disadvantage of processing the data the other way? What effect does choosing the incorrect way have on system performance?
- 8.10 Carefully discuss what happens when a cache miss occurs. Does this result in a major slowdown in execution of the instruction? If so, why?
- 8.11 What is the purpose of the tag in a cache memory system?
- 8.12 Describe the trade-offs between the memory cache write-through and write-back techniques.
- 8.13 Carefully describe the advantages and disadvantages of master-slave multiprocessing and symmetrical multiprocessing. Which would you select for fault-tolerant computing? Why?
- 8.14 Locate information about the Cell Processor. Describe the tasks performed by the various slave processors. What is the primary role of the master processor? Explain the advantages of master-slave multiprocessing over other forms of processing for this application. Can you think of some other types of computer problems that would benefit from this approach?
- 8.15 As you know, a single CPU processes one instruction at a time. Adding a second CPU (or *core*, in current terminology) allows the system to process two instructions at a time, simultaneously, effectively doubling the processing power of the system. A third core will offer triple the processing power of a single CPU, and so on. However, studies have shown that, in general, the expected increase in computing power starts to decline when the number of cores grows large, beyond eight or so. Why would you expect this to be the case? For what types of computing problems might this *not* be true?

