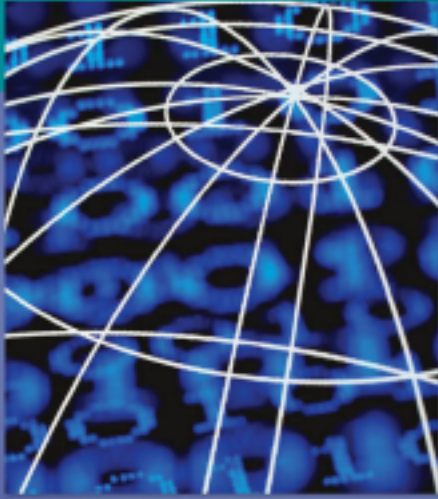


Carl Hamacher • Zvonko Vranesic • Safwat Zaky • Naraig Manjikian



# COMPUTER ORGANIZATION AND EMBEDDED SYSTEMS

Sixth Edition

*This page intentionally left blank*

*This page intentionally left blank*

# COMPUTER ORGANIZATION AND EMBEDDED SYSTEMS

*This page intentionally left blank*

# COMPUTER ORGANIZATION AND EMBEDDED SYSTEMS

SIXTH EDITION

**Carl Hamacher**

*Queen's University*

**Zvonko Vranesic**

*University of Toronto*

**Safwat Zaky**

*University of Toronto*

**Naraig Manjikian**

*Queen's University*



COMPUTER ORGANIZATION AND EMBEDDED SYSTEMS, SIXTH EDITION

Published by McGraw-Hill, a business unit of The McGraw-Hill Companies, Inc., 1221 Avenue of the Americas, New York, NY 10020. Copyright © 2012 by The McGraw-Hill Companies, Inc. All rights reserved. Previous editions 2002, 1996, and 1990. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written consent of The McGraw-Hill Companies, Inc., including, but not limited to, in any network or other electronic storage or transmission, or broadcast for distance learning.

Some ancillaries, including electronic and print components, may not be available to customers outside the United States.

This book is printed on acid-free paper.

123456789 DOC/DOC0987654321

ISBN 978-0-07-338065-0

MHID 0-07-338065-2

Vice President & Editor-in-Chief: *Marty Lange*

Vice President EDP/Central Publishing Services: *Kimberly Meriwether David*

Publisher: *Raghothaman Srinivasan*

Senior Sponsoring Editor: *Peter E. Massar*

Developmental Editor: *Darlene M. Schueller*

Senior Marketing Manager: *Curt Reynolds*

Senior Project Manager: *Lisa A. Bruflo*

Buyer: *Laura Fuller*

Design Coordinator: *Brenda A. Rolwes*

Media Project Manager: *Balaji Sundararaman*

Cover Design: *Studio Montage, St. Louis, Missouri*

Cover Image: © *Royalty-Free/CORBIS*

Compositor: *Techsetters, Inc.*

Typeface: *10/12 Times Roman*

Printer: *R. R. Donnelley & Sons Company/Crawfordsville, IN*

### Library of Congress Cataloging-in-Publication Data

Computer organization and embedded systems / Carl Hamacher ... [et al.]. – 6th ed.  
p. cm.

Includes bibliographical references.

ISBN-13: 978-0-07-338065-0 (alk. paper)

ISBN-10: 0-07-338065-2 (alk. paper)

1. Computer organization. 2. Embedded computer systems. I. Hamacher, V. Carl.

QA76.9.C643.H36 2012

004.2'2–dc22

2010050243

[www.mhhe.com](http://www.mhhe.com)

*To our families*

*This page intentionally left blank*

## About the Authors

**Carl Hamacher** received the B.A.Sc. degree in Engineering Physics from the University of Waterloo, Canada, the M.Sc. degree in Electrical Engineering from Queen's University, Canada, and the Ph.D. degree in Electrical Engineering from Syracuse University, New York. From 1968 to 1990 he was at the University of Toronto, Canada, where he was a Professor in the Department of Electrical Engineering and the Department of Computer Science. He served as director of the Computer Systems Research Institute during 1984 to 1988, and as chairman of the Division of Engineering Science during 1988 to 1990. In 1991 he joined Queen's University, where is now Professor Emeritus in the Department of Electrical and Computer Engineering. He served as Dean of the Faculty of Applied Science from 1991 to 1996. During 1978 to 1979, he was a visiting scientist at the IBM Research Laboratory in San Jose, California. In 1986, he was a research visitor at the Laboratory for Circuits and Systems associated with the University of Grenoble, France. During 1996 to 1997, he was a visiting professor in the Computer Science Department at the University of California at Riverside and in the LIP6 Laboratory of the University of Paris VI.

His research interests are in multiprocessors and multicomputers, focusing on their

interconnection networks.

**Zvonko Vranesic** received his B.A.Sc., M.A.Sc., and Ph.D. degrees, all in Electrical Engineering, from the University of Toronto. From 1963 to 1965 he worked as a design engineer with the Northern Electric Co. Ltd. in Bramalea, Ontario. In 1968 he joined the University of Toronto, where he is now a Professor Emeritus in the Department of Electrical & Computer Engineering. During the 1978–79 academic year, he was a Senior Visitor at the University of Cambridge, England, and during 1984–85 he was at the University of Paris, 6. From 1995 to 2000 he served as Chair of the Division of Engineering Science at the University of Toronto. He is also involved in research and development at the Altera Toronto Technology Center.

His current research interests include computer architecture and field-programmable VLSI technology.

He is a coauthor of four other books: *Fundamentals of Digital Logic with VHDL Design*, 3rd ed.; *Fundamentals of Digital Logic with Verilog Design*, 2nd ed.; *Microcomputer Structures*; and *Field-Programmable Gate Arrays*. In 1990, he received the Wighton Fellowship for “innovative and distinctive contributions to undergraduate laboratory instruction.” In 2004, he received the Faculty Teaching Award from the Faculty of Applied Science and Engineering at the University of Toronto.

**Safwat Zaky** received his B.Sc. degree in Electrical Engineering and B.Sc. in Mathematics, both from Cairo University, Egypt, and his M.A.Sc. and Ph.D. degrees in Electrical Engineering from the University of Toronto. From 1969 to 1972 he was with Bell Northern Research, Bramalea, Ontario, where he worked on applications of electro-optics and

vii

*This page intentionally left blank*

viii About the Authors

magnetics in mass storage and telephone switching. In 1973, he joined the University of Toronto, where he is now Professor Emeritus in the Department of Electrical and Computer Engineering. He served as Chair of the Department from 1993 to 2003 and as Vice-Provost from 2003 to 2009. During 1980 to 1981, he was a senior visitor at the Computer Laboratory, University of Cambridge, England.

He is a Fellow of the Canadian Academy of Engineering. His research interests are in the areas of computer architecture, digital-circuit design, and electromagnetic compatibility. He is a coauthor of the book *Microcomputer Structures* and is a recipient of the IEEE Third Millennium Medal and of the Vivek Goel Award for distinguished service to the University of Toronto.

**Naraig Manjikian** received his B.A.Sc. degree in Computer Engineering and M.A.Sc. degree in Electrical Engineering from the University of Waterloo, Canada, and his Ph.D. degree in Electrical Engineering from the University of Toronto. In 1997, he joined Queen’s University, Kingston, Canada, where he is now an Associate Professor in the Department of Electrical and Computer Engineering. From 2004 to 2006, he served as Undergraduate Chair for Computer Engineering. From 2006 to 2007, he served as Acting Head of the Department of Electrical and Computer Engineering, and from 2007 until 2009, he served as Associate Head for Student and Alumni Affairs. During 2003 to 2004, he was a visiting professor at McGill University, Montreal, Canada, and the University of British Columbia. During 2010 to 2011, he was a visiting professor at McGill University.

His research interests are in the areas of computer architecture, multiprocessor systems, field-programmable VLSI technology, and applications of parallel processing.

# Preface

This book is intended for use in a first-level course on computer organization and embedded systems in electrical engineering, computer engineering, and computer science curricula. The book is self-contained, assuming only that the reader has a basic knowledge of computer programming in a high-level language. Many students who study computer organization will have had an introductory course on digital logic circuits. Therefore, this subject is not covered in the main body of the book. However, we have provided an extensive appendix on logic circuits for those students who need it.

The book reflects our experience in teaching three distinct groups of students: electrical and computer engineering undergraduates, computer science undergraduates, and engineering science undergraduates. We have always approached the teaching of courses on computer organization from a practical point of view. Thus, a key consideration in shaping the contents of the book has been to carefully explain the main principles, supported by examples drawn from commercially available processors. Our main commercial examples are based on: Altera's Nios II, Freescale's ColdFire, ARM, and Intel's IA-32 architectures.

It is important to recognize that digital system design is not a straightforward process of applying optimal design algorithms. Many design decisions are based largely on heuristic judgment and experience. They involve cost/performance and hardware/software tradeoffs over a range of alternatives. It is our goal to convey these notions to the reader.

The book is aimed at a one-semester course in engineering or computer science programs. It is suitable for both hardware- and software-oriented students. Even though the emphasis is on hardware, we have addressed a number of relevant software issues.

McGraw-Hill maintains a Website with support material for the book at <http://www.mhhe.com/hamacher>.

## Scope of the Book

The first three chapters introduce the basic structure of computers, the operations that they perform at the machine-instruction level, and input/output methods as seen by a programmer. The fourth chapter provides an overview of the system software needed to translate programs written in assembly and high-level languages into machine language and to manage their execution. The remaining eight chapters deal with the organization, interconnection, and performance of hardware units in modern computers, including a coverage of embedded systems.

Five substantial appendices are provided. The first appendix covers digital logic circuits. Then, four current commercial instruction set architectures—Altera's Nios II, Freescale's ColdFire, ARM, and Intel's IA-32—are described in separate appendices.

**Chapter 1** provides an overview of computer hardware and informally introduces terms that are discussed in more depth in the remainder of the book. This chapter discusses

the basic functional units and the ways they interact to form a complete computer system. Number and character representations are discussed, along with basic arithmetic operations. An introduction to performance issues and a brief treatment of the history of

computer development are also provided.

**Chapter 2** gives a methodical treatment of machine instructions, addressing techniques, and instruction sequencing. Program examples at the machine-instruction level, expressed in a generic assembly language, are used to discuss concepts that include loops, subroutines, and stacks. The concepts are introduced using a RISC-style instruction set architecture. A comparison with CISC-style instruction sets is also included.

**Chapter 3** presents a programmer's view of basic input/output techniques. It explains how program-controlled I/O is performed using polling, as well as how interrupts are used in I/O transfers.

**Chapter 4** considers system software. The tasks performed by compilers, assemblers, linkers, and loaders are explained. Utility programs that trace and display the results of executing a program are described. Operating system routines that manage the execution of user programs and their input/output operations, including the handling of interrupts, are also described.

**Chapter 5** explores the design of a RISC-style processor. This chapter explains the sequence of processing steps needed to fetch and execute the different types of machine instructions. It then develops the hardware organization needed to implement these processing steps. The differing requirements of CISC-style processors are also considered.

**Chapter 6** provides coverage of the use of pipelining and multiple execution units in the design of high-performance processors. A pipelined version of the RISC-style processor design from Chapter 5 is used to illustrate pipelining. The role of the compiler and the relationship between pipelined execution and instruction set design are explored. Superscalar processors are discussed.

Input/output hardware is considered in **Chapter 7**. Interconnection networks, including the bus structure, are discussed. Synchronous and asynchronous operation is explained. Interconnection standards, including USB and PCI Express, are also presented.

Semiconductor memories, including SDRAM, Rambus, and Flash memory implementations, are discussed in **Chapter 8**. Caches are explained as a way for increasing the memory bandwidth. They are discussed in some detail, including performance modeling. Virtual-memory systems, memory management, and rapid address-translation techniques are also presented. Magnetic and optical disks are discussed as components in the memory hierarchy.

**Chapter 9** explores the implementation of the arithmetic unit of a computer. Logic design for fixed-point add, subtract, multiply, and divide hardware, operating on 2's complement numbers, is described. Carry-lookahead adders and high-speed multipliers are explained, including descriptions of the Booth multiplier recoding and carry-save addition techniques. Floating-point number representation and operations, in the context of the IEEE Standard, are presented.

Today, far more processors are in use in embedded systems than in general-purpose computers. **Chapters 10 and 11** are dedicated to the subject of embedded systems. First, basic aspects of system integration, component interconnections, and real-time operation are presented in Chapter 10. The use of microcontrollers is discussed. Then, Chapter 11 concentrates on system-on-a-chip (SoC) implementations, in which a single chip integrates

**Preface xi**

the processing, memory, I/O, and timer functionality needed to satisfy application-specific requirements. A substantial example shows how FPGAs and modern design tools can be used in this environment.

**Chapter 12** focuses on parallel processing and performance. Hardware multithreading and vector processing are introduced as enhancements in a single processor. Shared memory multiprocessors are then described, along with the issue of cache coherence. Interconnection networks for multiprocessors are presented.

**Appendix A** provides extensive coverage of logic circuits, intended for a reader who has not taken a course on the design of such circuits.

**Appendices B, C, D, and E** illustrate how the instruction set concepts introduced in Chapters 2 and 3 are implemented in four commercial processors: Nios II, ColdFire, ARM, and Intel IA-32. The Nios II and ARM processors illustrate the RISC design style. ColdFire has an easy-to-teach CISC design, while the IA-32 CISC architecture represents the most successful commercial design. The presentation for each processor includes assembly language examples from Chapters 2 and 3, implemented in the context of that processor. The details given in these appendices are not essential for understanding the material in the main body of the book. It is sufficient to cover only one of these appendices to gain an appreciation for commercial processor instruction sets. The choice of a processor to use as an example is likely to be influenced by the equipment in an accompanying laboratory. Instructors may wish to use more than one processor to illustrate the different design approaches.

## Changes in the Sixth Edition

Substantial changes in content and organization have been made in preparing the sixth edition of this book. They include the following:

- The basic concepts of instruction set architecture are now covered using the RISC-style approach. This is followed by a comparative examination of the CISC-style approach.
- The processor design discussion is focused on a RISC-style implementation, which leads naturally to pipelined operation.
- Two chapters on embedded systems are included: one dealing with the basic structure of such systems and the use of microcontrollers, and the other dealing with system-on-a-chip implementations.
- Appendices are used to give examples of four commercial processors. Each appendix includes the essential information about the instruction set architecture of the given processor.
- Solved problems have been included in a new section toward the end of chapters and appendices. They provide the student with solutions that can be expected for typical problems.

## Difficulty Level of Problems

The problems at the end of chapters and appendices have been classified as easy (E), medium (M), or difficult (D). These classifications should be interpreted as follows:

### xii Preface

- Easy—Solutions can be derived in a few minutes by direct application of specific information presented in one place in the relevant section of the book.
- Medium—Use of the book material in a way that does not directly follow any examples presented is usually needed. In some cases, solutions may follow the general pattern of an example, but will take longer to develop than those for easy problems.
- Difficult—Some additional insight is needed to solve these problems. If a solution requires a program to be written, its underlying algorithm or form may be quite different from that of any program example given in the book. If a hardware design is

required, it may involve an arrangement and interconnection of basic logic circuit components that is quite different from any design shown in the book. If a performance analysis is needed, it may involve the derivation of an algebraic expression.

## What Can Be Covered in a One-Semester Course

This book is suitable for use at the university or college level as a text for a one-semester course in computer organization. It is intended for the first course that students will take on computer organization.

There is more than enough material in the book for a one-semester course. The core material on computer organization and relevant software issues is given in Chapters 1 through 9. For students who have not had a course in logic circuits, the material in Appendix A should be studied near the beginning of a course and certainly prior to covering Chapter 5.

A course aimed at embedded systems should include Chapters 1, 2, 3, 4, 7, 8, 10 and 11. Use of the material on commercial processor examples in Appendices B through E can be guided by instructor and student interest, as well as by relevance to any hardware laboratory associated with a course.

## Acknowledgments

We wish to express our thanks to many people who have helped us during the preparation of this sixth edition of the book.

Our colleagues Daniel Etiemble of University of Paris South and Glenn Gulak of University of Toronto provided numerous comments and suggestions that helped significantly in shaping the material.

Blair Fort and Dan Vranesic provided valuable help with some of the programming examples.

Warren R. Carithers of Rochester Institute of Technology, Krishna M. Kavi of University of North Texas, and Nelson Luiz Passos of Midwestern State University provided reviews of material from both the fifth and sixth editions of the book.

The following people provided reviews of material from the fifth edition of the book: Goh Hock Ann of Multimedia University, Joseph E. Beaini of University of Colorado Denver, Kalyan Mohan Goli of Jawaharlal Nehru Technological University, Jaimon Jacob of Model Engineering College Ernakulam, M. Kumaresan of Anna University Coimbatore,

**Preface xiii**

Kenneth K. C. Lee of City University of Hong Kong, Manoj Kumar Mishra of Institute of Technical Education and Research, Junita Mohamad-Saleh of Universiti Sains Malaysia, Prashanta Kumar Patra of College of Engineering and Technology Bhubaneswar, Shanq Jang Ruan of National Taiwan University of Science and Technology, S. D. Samantaray of G. B. Pant University of Agriculture and Technology, Shivakumar Sastry of University of Akron, Donatella Sciuto of Politecnico of Milano, M. P. Singh of National Institute of Technology Patna, Albert Starling of University of Arkansas, Shannon Tauro of University of California Irvine, R. Thangarajan of Kongu Engineering College, Ashok Kumar Turuk of National Institute of Technology Rourkela, and Philip A. Wilsey of University of Cincinnati.

Finally, we truly appreciate the support of Raghothaman Srinivasan, Peter E. Massar, Darlene M. Schueller, Lisa Bruflodt, Curt Reynolds, Brenda Rolwes, and Laura Fuller at



Carl Hamacher  
Zvonko Vranesic  
Safwat Zaky  
Naraig Manjikian

**McGraw-Hill Create™** Craft your teaching resources to match the way you teach! With McGraw-Hill Create, [www.mcgrawhillcreate.com](http://www.mcgrawhillcreate.com), you can easily rearrange chapters, combine material from other content sources, and quickly upload content you have written like your course syllabus or teaching notes. Find the content you need in Create by searching through thousands of leading McGraw-Hill textbooks. Arrange your book to fit your teaching style. Create even allows you to personalize your book's appearance by selecting the cover and adding your name, school, and course information. Order a Create book and you'll receive a complimentary print review copy in 3-5 business days or a complimentary electronic review copy (eComp) via email in minutes. Go to [www.mcgrawhillcreate.com](http://www.mcgrawhillcreate.com) today and register to experience how McGraw-Hill Create empowers you to teach your students your way.



### **McGraw-Hill Higher Education and Blackboard® have teamed up.**

Blackboard, the Web-based course management system, has partnered with McGraw-Hill to better allow students and faculty to use online materials and activities to complement face-to-face teaching. Blackboard features exciting social learning and teaching tools that foster more logical, visually impactful and active learning opportunities for students. You'll transform your closed-door classrooms into communities where students remain connected to their educational experience 24 hours a day.

This partnership allows you and your students access to McGraw-Hill's Create right from within your Blackboard course - all with one single sign-on. McGraw-Hill and Blackboard can now offer you easy access to industry leading technology and content, whether your campus hosts it, or we do. Be sure to ask your local McGraw-Hill representative for details.

# **Contents**

## **Chapter 1**

### **Basic Structure of Computers 1**

1.1 Computer Types 2   1.2 Functional Units 3   1.2.1  
Input Unit 4  
1.2.2 Memory Unit 4

2.3 Instructions and Instruction Sequencing 32  
2.3.1 Register Transfer Notation 33  
2.3.2 Assembly-Language Notation 33  
2.3.3 RISC and CISC Instruction Sets 34  
2.3.4 Introduction to RISC Instruction  
Sets 34

2.3.5 Instruction Execution and Straight-Line Sequencing 36

1.2.3 Arithmetic and Logic Unit 5

1.2.4 Output Unit 6

1.2.5 Control Unit 6

1.3 Basic Operational Concepts 7

1.4 Number Representation and Arithmetic Operations 9

1.4.1 Integers 10

1.4.2 Floating-Point Numbers 16

1.5 Character Representation 17

1.6 Performance 17

1.6.1 Technology 17

1.6.2 Parallelism 19

1.7 Historical Perspective 19

1.7.1 The First Generation 20

1.7.2 The Second Generation 20

1.7.3 The Third Generation 21

1.7.4 The Fourth Generation 21

1.8 Concluding Remarks 22

1.9 Solved Problems 22

Problems 24

References 25

Chapter 2

**Instruction Set Architecture 27**

2.1 Memory Locations and Addresses 28

2.1.1 Byte Addressability 30

2.1.2 Big-Endian and Little-Endian Assignments 30

2.1.3 Word Alignment 31

2.1.4 Accessing Numbers and Characters 32

2.2 Memory Operations 32

xv

2.3.6 Branching 37

2.3.7 Generating Memory Addresses 40

2.4 Addressing Modes 40

2.4.1 Implementation of Variables and Constants 41

2.4.2 Indirection and Pointers 42

2.4.3 Indexing and Arrays 45

2.5 Assembly Language 48

2.5.1 Assembler Directives 50

2.5.2 Assembly and Execution of Programs 53

2.5.3 Number Notation 54

2.6 Stacks 55

2.7 Subroutines 56

2.7.1 Subroutine Nesting and the Processor Stack 58

2.7.2 Parameter Passing 59

2.7.3 The Stack Frame 63

2.8 Additional Instructions 65

2.8.1 Logic Instructions 67

2.8.2 Shift and Rotate Instructions 68

2.8.3

Multiplication and Division	71
2.9 Dealing with 32-Bit Immediate Values	73
2.10 CISC Instruction Sets	74
2.10.1 Additional Addressing Modes	75
2.10.2 Condition Codes	77
2.11 RISC and CISC Styles	78
2.12 Example Programs	79
2.12.1 Vector Dot Product Program	79
2.12.2 String Search Program	81
2.13 Encoding of Machine Instructions	82
2.14 Concluding Remarks	85
2.15 Solved Problems	85
Problems	90

xvi **Contents**

## **Chapter 3**

### **Basic Input/Output 95**

3.1 Accessing I/O Devices	96
3.1.1 I/O Device Interface	97
3.1.2 Program-Controlled I/O	97
3.1.3 An Example of a RISC-Style I/O Program	101
3.1.4 An Example of a CISC-Style I/O Program	101
3.2 Interrupts	103
3.2.1 Enabling and Disabling Interrupts	106
3.2.2 Handling Multiple Devices	107
3.2.3 Controlling I/O Device Behavior	109
3.2.4 Processor Control Registers	110
3.2.5 Examples of Interrupt Programs	111
3.2.6 Exceptions	116
3.3 Concluding Remarks	119
3.4 Solved Problems	119
Problems	126

## **Chapter 4**

### **Software 129**

4.1 The Assembly Process	130
4.1.1 Two-pass Assembler	131
4.2 Loading and Executing Object Programs	131
4.3 The Linker	132
4.4 Libraries	133
4.5 The Compiler	133
4.5.1 Compiler Optimizations	134
4.5.2 Combining Programs Written in Different Languages	134
4.6 The Debugger	134
4.7 Using a High-level Language for I/O Tasks	137
4.8 Interaction between Assembly Language and C Language	139
4.9 The Operating System	143

4.9.1 The Boot-strapping Process	144
4.9.2 Managing the Execution of Application Programs	144
4.9.3 Use of Interrupts in Operating Systems	146
4.10 Concluding Remarks	149
Problems	149
References	150

## **Chapter 5**

### **Basic Processing Unit 151**

5.1 Some Fundamental Concepts	152
5.2 Instruction Execution	155
5.2.1 Load Instructions	155
5.2.2 Arithmetic and Logic Instructions	156
5.2.3 Store Instructions	157
5.3 Hardware Components	158
5.3.1 Register File	158
5.3.2 ALU	160
5.3.3 Datapath	161
5.3.4 Instruction Fetch Section	164
5.4 Instruction Fetch and Execution Steps	165
5.4.1 Branching	168
5.4.2 Waiting for Memory	171
5.5 Control Signals	172
5.6 Hardwired Control	175
5.6.1 Datapath Control Signals	177
5.6.2 Dealing with Memory Delay	177
5.7 CISC-Style Processors	178
5.7.1 An Interconnect using Buses	180
5.8 Microprogrammed Control	183
5.9 Concluding Remarks	185
5.9 Solved Problems	185
Problems	188

## **Chapter 6**

### **Pipelining 193**

6.1 Basic Concept—The Ideal Case	194
6.2 Pipeline Organization	195
6.3 Pipelining Issues	196
6.4 Data Dependencies	197
6.4.1 Operand Forwarding	198
6.4.2 Handling Data Dependencies in Software	199
6.5 Memory Delays	201
6.6 Branch Delays	202
6.6.1 Unconditional Branches	202
6.6.2 Conditional Branches	204
6.6.3 The Branch Delay Slot	204
6.6.4 Branch Prediction	205
6.7 Resource Limitations	209

6.8 Performance Evaluation	209
6.8.1 Effects of Stalls and Penalties	210
6.8.2 Number of Pipeline Stages	212
6.9 Superscalar Operation	212
6.9.1 Branches and Data Dependencies	214
6.9.2 Out-of-Order Execution	215
6.9.3 Execution Completion	216
6.9.4 Dispatch Operation	217
6.10 Pipelining in CISC Processors	218
6.10.1 Pipelining in ColdFire Processors	219
6.10.2 Pipelining in Intel Processors	219
6.11 Concluding Remarks	220
6.12 Examples of Solved Problems	220
Problems	222
References	226

## Chapter 7

# Input/Output Organization 227

7.1 Bus Structure	228
7.2 Bus Operation	229
7.2.1 Synchronous Bus	230
7.2.2 Asynchronous Bus	233
7.2.3 Electrical Considerations	236
7.3 Arbitration	237
7.4 Interface Circuits	238
7.4.1 Parallel Interface	239
7.4.2 Serial Interface	243
7.5 Interconnection Standards	247
7.5.1 Universal Serial Bus (USB)	247
7.5.2 FireWire	251
7.5.3 PCI Bus	252
7.5.4 SCSI Bus	256
7.5.5 SATA	258
7.5.6 SAS	258
7.5.7 PCI Express	258
7.6 Concluding Remarks	260
7.7 Solved Problems	260
Problems	263
References	266

## Chapter 8

# The Memory System 267

8.1 Basic Concepts	268
8.2 Semiconductor RAM Memories	270
8.2.1 Internal Organization of Memory Chips	270
8.2.2 Static Memories	271
8.2.3 Dynamic RAMs	274

8.2.4 Synchronous DRAMs	276
8.2.5 Structure of Larger Memories	279
8.3 Read-only Memories	282
8.3.1 ROM	283
8.3.2 PROM	283
8.3.3 EPROM	284
8.3.4 EEPROM	284
8.3.5 Flash Memory	284
8.4 Direct Memory Access	285
8.5 Memory Hierarchy	288
8.6 Cache Memories	289
8.6.1 Mapping Functions	291
8.6.2 Replacement Algorithms	296
8.6.3 Examples of Mapping Techniques	297
8.7 Performance Considerations	300
8.7.1 Hit Rate and Miss Penalty	301
8.7.2 Caches on the Processor Chip	302
8.7.3 Other Enhancements	303
8.8 Virtual Memory	305
8.8.1 Address Translation	306
8.9 Memory Management Requirements	310
8.10 Secondary Storage	311
8.10.1 Magnetic Hard Disks	311
8.10.2 Optical Disks	317
8.10.3 Magnetic Tape Systems	322
8.11 Concluding Remarks	323
8.12 Solved Problems	324
Problems	328
References	332

## Chapter 9

# Arithmetic 335

9.1 Addition and Subtraction of Signed Numbers	336
9.1.1 Addition/Subtraction Logic Unit	336
9.2 Design of Fast Adders	339
9.2.1 Carry-Lookahead Addition	340
9.3 Multiplication of Unsigned Numbers	344
9.3.1 Array Multiplier	344
9.3.2 Sequential Circuit Multiplier	346
9.4 Multiplication of Signed Numbers	346
9.4.1 The Booth Algorithm	348
9.5 Fast Multiplication	351
9.5.1 Bit-Pair Recoding of Multipliers	352
9.5.2 Carry-Save Addition of Summands	353

## xviii Contents

9.5.3 Summand Addition Tree using 3-2 Reducers	355
9.5.4 Summand Addition Tree using 4-2 Reducers	357

9.5.5 Summary of Fast Multiplication	359
Integer Division	360
9.7 Floating-Point Numbers and Operations	363
9.7.1 Arithmetic Operations on Floating-Point Numbers	367
9.7.2 Guard Bits and Truncation	368
9.7.3 Implementing Floating-Point Operations	369
9.8 Decimal-to-Binary Conversion	372
9.9 Concluding Remarks	372
9.10 Solved Problems	374
Problems	377
References	383

## Chapter 10

### Embedded Systems 385

10.1 Examples of Embedded Systems	386
10.1.1 Microwave Oven	386
10.1.2 Digital Camera	387
10.1.3 Home Telemetry	390
10.2 Microcontroller Chips for Embedded Applications	390
10.3 A Simple Microcontroller	392
10.3.1 Parallel I/O Interface	392
10.3.2 Serial I/O Interface	395
10.3.3 Counter/Timer	397
10.3.4 Interrupt-Control Mechanism	399
10.3.5 Programming Examples	399
10.4 Reaction Timer—A Complete Example	401
10.5 Sensors and Actuators	407
10.5.1 Sensors	407
10.5.2 Actuators	410
10.5.3 Application Examples	411
10.6 Microcontroller Families	412
10.6.1 Microcontrollers Based on the Intel 8051	413
10.6.2 Freescale Microcontrollers	413
10.6.3 ARM Microcontrollers	414
10.7 Design Issues	414
10.8 Concluding Remarks	417
Problems	418
References	420

## Chapter 11

### System-on-a-Chip—A Case Study 421

11.1 FPGA Implementation	422
11.1.1 FPGA Devices	423
11.1.2 Processor Choice	423
11.2 Computer-Aided Design Tools	424
11.2.1 Altera CAD Tools	425

11.3 Alarm Clock Example	428
11.3.1 User's View of the System	428
11.3.2 System Definition and Generation	429
11.3.3 Circuit Implementation	430
11.3.4 Application Software	431
11.4 Concluding Remarks	440
Problems	440
References	441

## Chapter 12

### Parallel Processing and Performance 443

12.1 Hardware Multithreading	444
12.2 Vector (SIMD) Processing	445
12.2.1 Graphics Processing Units (GPUs)	448
12.2.3 Shared-Memory Multiprocessors	448
12.3.1 Interconnection Networks	450
12.4 Cache Coherence	453
12.4.1 Write-Through Protocol	453
12.4.2 Write-Back protocol	454
12.4.3 Snoopy Caches	454
12.4.4 Directory-Based Cache Coherence	456
12.5 Message-Passing Multicomputers	456
Parallel Programming for Multiprocessors	456
12.7 Performance Modeling	460
12.8 Concluding Remarks	461
Problems	462
References	463

## Appendix A

### Logic Circuits 465

A.1 Basic Logic Functions	
A.1.1 Electronic Logic Gates	469
A.2 Synthesis of Logic Functions	470
A.3 Minimization of Logic Expressions	472
A.3.1 Minimization using Karnaugh Maps	475
A.3.2 Don't-Care Conditions	477
A.4 Synthesis with NAND and NOR Gates	479
A.5 Practical Implementation of Logic Gates	482
A.5.1 CMOS Circuits	484
A.5.2 Propagation Delay	489
A.5.3 Fan-In and Fan-Out Constraints	490
A.5.4 Tri-State Buffers	491
A.6 Flip-Flops	492
A.6.1 Gated Latches	493
A.6.2 Master-Slave Flip-Flop	495
A.6.3 Edge Triggering	498
A.6.4 T Flip-Flop	498
A.6.5 JK Flip-Flop	499

- A.6.6 Flip-Flops with Preset and Clear 501
- A.7 Registers and Shift Registers 502
- A.8 Counters 503
- A.9 Decoders 505
- A.10 Multiplexers 506
  - A.11 Programmable Logic Devices (PLDs) 509
    - A.11.1 Programmable Logic Array (PLA) 509
    - A.11.2 Programmable Array Logic (PAL) 511
    - A.11.3 Complex Programmable Logic Devices (CPLDs) 512
- A.12 Field-Programmable Gate Arrays 514
- A.13 Sequential Circuits 516
  - A.13.1 Design of an Up/Down Counter as a Sequential Circuit 516
  - A.13.2 Timing Diagrams 519
  - A.13.3 The Finite State Machine Model 520
- A.13.4 Synthesis of Finite State Machines 521
- A.14 Concluding Remarks 522
- Problems 522
- References 528

## Appendix B

### The Altera Nios II Processor 529

- B.1 Nios II Characteristics 530
- B.2 General-Purpose Registers 531
- B.3 Addressing Modes 532
- B.4 Instructions 533
  - B.4.1 Notation 533
  - B.4.2 Load and Store Instructions 534
  - B.4.3 Arithmetic Instructions 536
  - B.4.4 Logic Instructions 537
  - B.4.5 Move Instructions 537
  - B.4.6 Branch and Jump Instructions 538
  - B.4.7 Subroutine Linkage Instructions 541
  - B.4.8 Comparison Instructions 545
  - B.4.9 Shift Instructions 546
  - B.4.10 Rotate Instructions 547
  - B.4.11 Control Instructions 548
- B.5 Pseudoinstructions 548
- B.6 Assembler Directives 549
- B.7 Carry and Overflow Detection 551
- B.8 Example Programs 553
- B.9 Control Registers 553
- B.10 Input/Output 555
  - B.10.1 Program-Controlled I/O 556
  - B.10.2 Interrupts and Exceptions 556
- B.11 Advanced Configurations of Nios II Processor 562
  - B.11.1 External Interrupt Controller 562
  - B.11.2 Memory Management Unit 562
  - B.11.3 Floating-Point Hardware 562

## Contents xix

- B.12 Concluding Remarks 563
- B.13 Solved Problems 563
- Problems 568

## Appendix C

### The ColdFire Processor 571

- C.1 Memory Organization 572
- C.2 Registers 572
- C.3 Instructions 573
  - C.3.1 Addressing Modes 575
  - C.3.2 Move Instruction 577
  - C.3.3 Arithmetic Instructions 578
  - C.3.4 Branch and Jump Instructions 582
  - C.3.5 Logic Instructions 585
  - C.3.6 Shift Instructions 586
  - C.3.7 Subroutine Linkage Instructions 587
- C.4 Assembler Directives 593
- C.5 Example Programs 594
  - C.5.1 Vector Dot Product Program 594
  - C.5.2 String Search Program 595
- C.6 Mode of Operation and Other Control Features 596
- C.7 Input/Output 597
- C.8 Floating-Point Operations 599
  - C.8.1 FMOVE Instruction 599

## xx Contents

- C.8.2 Floating-Point Arithmetic Instructions 600
- C.8.3 Comparison and Branch Instructions 601
- C.8.4 Additional Floating-Point Instructions 601
- C.8.5 Example Floating-Point Program 602
- C.9 Concluding Remarks 603
- C.10 Solved Problems 603
  - Problems 608
  - References 609

## Appendix D

### The ARM Processor 611

- D.1 ARM Characteristics 612
  - D.1.1 Unusual Aspects of the ARM Architecture 612
- D.2 Register Structure 613
- D.3 Addressing Modes 614
  - D.3.1 Basic Indexed Addressing Mode 614
  - D.3.2 Relative Addressing Mode 615
  - D.3.3 Index Modes with Writeback 616
  - D.3.4 Offset Determination 616
  - D.3.5 Register, Immediate, and Absolute Addressing Modes 618
  - D.3.6 Addressing Mode Examples 618

D.4 Instructions 621	
D.4.1 Load and Store Instructions 621	
D.4.2 Arithmetic Instructions 622	
D.4.3 Move Instructions 625	
D.4.4 Logic and Test Instructions 626	
D.4.5 Compare Instructions 627	
D.4.6 Setting Condition Code Flags 628	
D.4.7 Branch Instructions 628	
D.4.8 Subroutine Linkage Instructions 631	
D.5 Assembly Language 635	
D.5.1 Pseudoinstructions 637	
D.6 Example Programs 638	
D.6.1 Vector Dot Product 639	
D.6.2 String Search 639	
D.7 Operating Modes and Exceptions 639	
D.7.1 Banked Registers 641	
D.7.2 Exception Types 642	
D.7.3 System Mode 644	
D.7.4 Handling Exceptions 644	
D.8 Input/Output 646	
D.8.1 Program-Controlled I/O 646	
D.8.2 Interrupt-Driven I/O 648	
D.9 Conditional Execution of Instructions 648	
D.10 Coprocessors 650	
D.11 Embedded Applications and the Thumb ISA 651	
D.12 Concluding Remarks 651	
D.13 Solved Problems 652	
Problems 657	
References 660	

## Appendix E

## The Intel IA-32 Architecture 661

E.1 Memory Organization 662	
E.2 Register Structure 662	
E.3 Addressing Modes 665	
E.4 Instructions 668	
E.4.1 Machine Instruction Format 670	E.4.2
Assembly-Language Notation 670	E.4.3
Move Instruction 671	
E.4.4 Load-Effective-Address Instruction 671	
E.4.5 Arithmetic Instructions 672	
E.4.6 Jump and Loop Instructions 674	
E.4.7 Logic Instructions 677	
E.4.8 Shift and Rotate Instructions 678	E.4.9
Subroutine Linkage Instructions 679	E.4.10
Operations on Large Numbers 681	
E.5 Assembler Directives 685	
E.6 Example Programs 686	
E.6.1 Vector Dot Product Program 686	
E.6.2 String Search Program 686	
E.7 Interrupts and Exceptions 687	
E.8 Input/Output Examples 689	
E.9 Scalar Floating-Point Operations 690	
E.9.1 Load and Store Instructions 692	
E.9.2 Arithmetic Instructions 693	
E.9.3 Comparison Instructions 694	
E.9.4 Additional Instructions 694	
E.9.5 Example Floating-Point Program 694	
E.10 Multimedia Extension (MMX) Operations 695	
E.11 Vector (SIMD) Floating-Point Operations 696	
E.12 Examples of Solved Problems 697	
E.13 Concluding Remarks 702	
Problems 702	
References 703	

## chapter

# 1

# Basic Structure of Computers

## Chapter Objectives

In this chapter you will be introduced to:

- The different types of computers
  - The basic structure of a computer and its operation
  - Machine instructions and their execution
- Number and character representations
  - Addition and subtraction of binary numbers
- Basic performance issues in computer systems
- A brief history of computer development

## 2 CHAPTER 1 Basic Structure of Computers

**T**his book is about computer organization. It explains the function and design of the various units of digital computers that store and process information. It also deals with the input units of the computer which receive information from external sources and the output units which send computed results to external destinations. The input, storage, processing, and output operations are governed by a list of instructions that constitute a program.

Most of the material in the book is devoted to *computer hardware* and *computer architecture*. Computer hardware consists of electronic circuits, magnetic and optical storage devices, displays, electromechanical devices, and communication facilities. Computer architecture encompasses the specification of an instruction set and the functional behavior of the hardware units that implement the instructions.

Many aspects of programming and software components in computer systems are also discussed in the book. It is important to consider both hardware and software aspects of the design of the various computer components in order to gain a good understanding of computer systems.

### 1.1 Computer Types

Since their introduction in the 1940s, digital computers have evolved into many different types that vary widely in size, cost, computational power, and intended use. Modern computers can be divided roughly into four general categories:



- *Embedded computers* are integrated into a larger device or system in order to automatically monitor and control a physical process or environment. They are used for a specific purpose rather than for general processing tasks. Typical applications include industrial and home automation, appliances, telecommunication products, and vehicles. Users may not even be aware of the role that computers play in such systems.
- *Personal computers* have achieved widespread use in homes, educational institutions, and business and engineering office settings, primarily for dedicated individual use. They support a variety of applications such as general computation, document preparation, computer-aided design, audiovisual entertainment, interpersonal communication, and Internet browsing. A number of classifications are used for personal computers. *Desktop computers* serve general needs and fit within a typical personal workspace. *Workstation computers* offer higher computational capacity and more powerful graphical display capabilities for engineering and scientific work. Finally, *Portable* and *Notebook computers* provide the basic features of a personal computer in a smaller lightweight package. They can operate on batteries to provide mobility.
- *Servers* and *Enterprise systems* are large computers that are meant to be shared by a potentially large number of users who access them from some form of personal computer over a public or private network. Such computers may host large databases and provide information processing for a government agency or a commercial organization.
- *Supercomputers* and *Grid computers* normally offer the highest performance. They are the most expensive and physically the largest category of computers. Supercomputers are used for the highly demanding computations needed in weather forecasting, engineering design and simulation, and scientific work. They have a high cost. Grid computers provide a more cost-effective alternative. They combine a large number of personal computers and

## 1.2 Functional Units 3

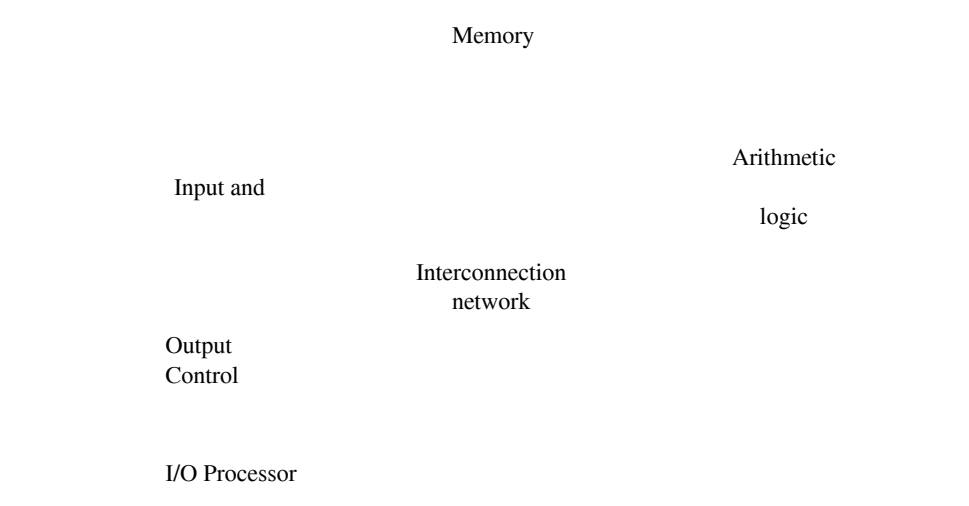
disk storage units in a physically distributed high-speed network, called a grid, which is managed as a coordinated computing resource. By evenly distributing the computational workload across the grid, it is possible to achieve high performance on large applications ranging from numerical computation to information searching.

There is an emerging trend in access to computing facilities, known as *cloud computing*. Personal computer users access widely distributed computing and storage server resources for individual, independent, computing needs. The Internet provides the necessary communication facility. Cloud hardware and software service providers operate as a utility, charging on a pay-as-you-use basis.

## 1.2 Functional Units

A computer consists of five functionally independent main parts: input, memory, arithmetic and logic, output, and control units, as shown in Figure 1.1. The input unit accepts coded information from human operators using devices such as keyboards, or from other computers over digital communication lines. The information received is stored in the computer's memory, either for later use or to be processed immediately by the arithmetic and logic unit. The processing steps are specified by a program that is also stored in the memory. Finally, the results are sent back to the outside world through the output unit. All of these actions are coordinated by the control unit. An interconnection network provides the means for the functional units to exchange information and

coordinate their actions. Later chapters will provide more details on individual units and their interconnections. We refer to the



**Figure 1.1** Basic functional units of a computer.

#### 4 CHAPTER 1 Basic Structure of Computers

arithmetic and logic circuits, in conjunction with the main control circuits, as the *processor*. Input and output equipment is often collectively referred to as the *input-output* (I/O) unit. We now take a closer look at the information handled by a computer. It is convenient to categorize this information as either instructions or data. *Instructions*, or *machine instructions*, are explicit commands that

- Govern the transfer of information within a computer as well as between the computer and its I/O devices
- Specify the arithmetic and logic operations to be performed

A *program* is a list of instructions which performs a task. Programs are stored in the memory. The processor fetches the program instructions from the memory, one after another, and performs the desired operations. The computer is controlled by the stored program, except for possible external interruption by an operator or by I/O devices connected to it. *Data* are numbers and characters that are used as operands by the instructions. Data are also stored in the memory.

The instructions and data handled by a computer must be encoded in a suitable format. Most present-day hardware employs digital circuits that have only two stable states. Each instruction, number, or character is encoded as a string of binary digits called *bits*, each having one of two possible values, 0 or 1, represented by the two stable states. Numbers are usually represented in positional binary notation, as discussed in Section 1.4. Alphanumeric characters are also expressed in terms of binary codes, as discussed in Section 1.5.

### 1.2.1 Input Unit

Computers accept coded information through input units. The most common input device is the keyboard. Whenever a key is pressed, the corresponding letter or digit is automatically translated into its corresponding binary code and transmitted to the processor.

Many other kinds of input devices for human-computer interaction are available, including the touchpad, mouse, joystick, and trackball. These are often used as graphic input devices in conjunction with displays. Microphones can be used to capture audio input which is then sampled and converted into digital codes for storage and processing. Similarly, cameras can be used to capture video input.

Digital communication facilities, such as the Internet, can also provide input to a computer from other computers and database servers.

### 1.2.2 Memory Unit

The function of the memory unit is to store programs and data. There are two classes of storage, called primary and secondary.

#### Primary Memory

*Primary memory*, also called *main memory*, is a fast memory that operates at electronic speeds. Programs must be stored in this memory while they are being executed. The

## 1.2 Functional Units 5

memory consists of a large number of semiconductor storage cells, each capable of storing one bit of information. These cells are rarely read or written individually. Instead, they are handled in groups of fixed size called *words*. The memory is organized so that one word can be stored or retrieved in one basic operation. The number of bits in each word is referred to as the *word length* of the computer, typically 16, 32, or 64 bits.

To provide easy access to any word in the memory, a distinct *address* is associated with each word location. Addresses are consecutive numbers, starting from 0, that identify successive locations. A particular word is accessed by specifying its address and issuing a control command to the memory that starts the storage or retrieval process.

Instructions and data can be written into or read from the memory under the control of the processor. It is essential to be able to access any word location in the memory as quickly as possible. A memory in which any location can be accessed in a short and fixed amount of time after specifying its address is called a *random-access memory* (RAM). The time required to access one word is called the *memory access time*. This time is independent of the location of the word being accessed. It typically ranges from a few nanoseconds (ns) to about 100 ns for current RAM units.

#### Cache Memory

As an adjunct to the main memory, a smaller, faster RAM unit, called a *cache*, is used to hold sections of a program that are currently being executed, along with any associated data. The cache is tightly coupled with the processor and is usually contained on the same integrated-circuit chip. The purpose of the cache is to facilitate high instruction execution rates.

At the start of program execution, the cache is empty. All program instructions and any required data are stored in the main memory. As execution proceeds, instructions are fetched into the processor chip, and a copy of each is placed in the cache. When the execution of an instruction requires data located in the main memory, the data are fetched and copies are also placed in the cache.

Now, suppose a number of instructions are executed repeatedly as happens in a

program loop. If these instructions are available in the cache, they can be fetched quickly during the period of repeated use. Similarly, if the same data locations are accessed repeatedly while copies of their contents are available in the cache, they can be fetched quickly.

### Secondary Storage

Although primary memory is essential, it tends to be expensive and does not retain information when power is turned off. Thus additional, less expensive, permanent *secondary storage* is used when large amounts of data and many programs have to be stored, particularly for information that is accessed infrequently. Access times for secondary storage are longer than for primary memory. A wide selection of secondary storage devices is available, including *magnetic disks*, *optical disks* (DVD and CD), and *flash memory devices*.

## 1.2.3 Arithmetic and Logic Unit

Most computer operations are executed in the *arithmetic and logic unit* (ALU) of the processor. Any arithmetic or logic operation, such as addition, subtraction, multiplication,

### 6 CHAPTER 1 Basic Structure of Computers

division, or comparison of numbers, is initiated by bringing the required operands into the processor, where the operation is performed by the ALU. For example, if two numbers located in the memory are to be added, they are brought into the processor, and the addition is carried out by the ALU. The sum may then be stored in the memory or retained in the processor for immediate use.

When operands are brought into the processor, they are stored in high-speed storage elements called *registers*. Each register can store one word of data. Access times to registers are even shorter than access times to the cache unit on the processor chip.

## 1.2.4 Output Unit

The output unit is the counterpart of the input unit. Its function is to send processed results to the outside world. A familiar example of such a device is a *printer*. Most printers employ either photocopying techniques, as in laser printers, or ink jet streams. Such printers may generate output at speeds of 20 or more pages per minute. However, printers are mechanical devices, and as such are quite slow compared to the electronic speed of a processor.

Some units, such as graphic displays, provide both an output function, showing text and graphics, and an input function, through touchscreen capability. The dual role of such units is the reason for using the single name *input/output* (I/O) unit in many cases.

## 1.2.5 Control Unit

The memory, arithmetic and logic, and I/O units store and process information and perform input and output operations. The operation of these units must be coordinated in some way. This is the responsibility of the control unit. The control unit is effectively the nerve center that sends control signals to other units and senses their states.

I/O transfers, consisting of input and output operations, are controlled by program instructions that identify the devices involved and the information to be transferred. Control circuits are responsible for generating the *timing signals* that govern the transfers and determine when a given action is to take place. Data transfers between the processor

and the memory are also managed by the control unit through timing signals. It is reasonable to think of a control unit as a well-defined, physically separate unit that interacts with other parts of the computer. In practice, however, this is seldom the case. Much of the control circuitry is physically distributed throughout the computer. A large set of control lines (wires) carries the signals used for timing and synchronization of events in all units.

The operation of a computer can be summarized as follows:

- The computer accepts information in the form of programs and data through an input unit and stores it in the memory.
- Information stored in the memory is fetched under program control into an arithmetic and logic unit, where it is processed.
- Processed information leaves the computer through an output unit.
- All activities in the computer are directed by the control unit.

### 1.3 Basic Operational Concepts 7

## 1.3 Basic Operational Concepts

In Section 1.2, we stated that the activity in a computer is governed by instructions. To perform a given task, an appropriate program consisting of a list of instructions is stored in the memory. Individual instructions are brought from the memory into the processor, which executes the specified operations. Data to be used as instruction operands are also stored in the memory.

A typical instruction might be

Load R2, LOC

This instruction reads the contents of a memory location whose address is represented symbolically by the label LOC and loads them into processor register R2. The original contents of location LOC are preserved, whereas those of register R2 are overwritten. Execution of this instruction requires several steps. First, the instruction is fetched from the memory into the processor. Next, the operation to be performed is determined by the control unit. The operand at LOC is then fetched from the memory into the processor. Finally, the operand is stored in register R2.

After operands have been loaded from memory into processor registers, arithmetic or logic operations can be performed on them. For example, the instruction

Add R4, R2, R3

adds the contents of registers R2 and R3, then places their sum into register R4. The operands in R2 and R3 are not altered, but the previous value in R4 is overwritten by the sum.

After completing the desired operations, the results are in processor registers. They can be transferred to the memory using instructions such as

Store R4, LOC

This instruction copies the operand in register R4 to memory location LOC. The original contents of location LOC are overwritten, but those of R4 are preserved.

For Load and Store instructions, transfers between the memory and the processor are initiated by sending the address of the desired memory location to the memory unit and asserting the appropriate control signals. The data are then transferred to or from the memory.

Figure 1.2 shows how the memory and the processor can be connected. It also shows some components of the processor that have not been discussed yet. The interconnections between these components are not shown explicitly since we will only discuss their functional characteristics here. Chapter 5 describes the details of the interconnections as part of processor organization.

In addition to the ALU and the control circuitry, the processor contains a number of registers used for several different purposes. The *instruction register* (IR) holds the instruction that is currently being executed. Its output is available to the control circuits, which generate the timing signals that control the various processing elements involved in executing the instruction. The *program counter* (PC) is another specialized register. It

8 CHAPTER 1 • Basic Structure of Computers Main memory

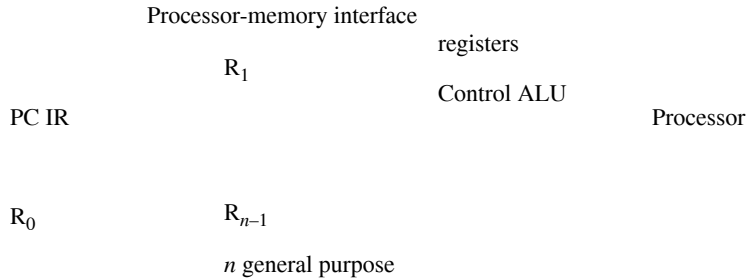


Figure 1.2 Connection between the processor and the main memory.

contains the memory address of the next instruction to be fetched and executed. During the execution of an instruction, the contents of the PC are updated to correspond to the address of the next instruction to be executed. It is customary to say that the PC *points* to the next instruction that is to be fetched from the memory. In addition to the IR and PC, Figure 1.2 shows *general-purpose registers*  $R_0$  through  $R_{n-1}$ , often called processor registers. They serve a variety of functions, including holding operands that have been loaded from the memory for processing. The roles of the general-purpose registers are explained in detail in Chapter 2.

The processor-memory interface is a circuit which manages the transfer of data between the main memory and the processor. If a word is to be read from the memory, the interface sends the address of that word to the memory along with a Read control signal. The interface waits for the word to be retrieved, then transfers it to the appropriate

processor register. If a word is to be written into memory, the interface transfers both the address and the word to the memory along with a Write control signal.

Let us now consider some typical operating steps. A program must be in the main memory in order for it to be executed. It is often transferred there from secondary storage through the input unit. Execution of the program begins when the PC is set to point to the

#### 1.4 Number Representation and Arithmetic Operations 9

first instruction of the program. The contents of the PC are transferred to the memory along with a Read control signal. When the addressed word (in this case, the first instruction of the program) has been fetched from the memory it is loaded into register IR. At this point, the instruction is ready to be interpreted and executed.

Instructions such as Load, Store, and Add perform data transfer and arithmetic operations. If an operand that resides in the memory is required for an instruction, it is fetched by sending its address to the memory and initiating a Read operation. When the operand has been fetched from the memory, it is transferred to a processor register. After operands have been fetched in this way, the ALU can perform a desired arithmetic operation, such as Add, on the values in processor registers. The result is sent to a processor register. If the result is to be written into the memory with a Store instruction, it is transferred from the processor register to the memory, along with the address of the location where the result is to be stored, then a Write operation is initiated.

At some point during the execution of each instruction, the contents of the PC are incremented so that the PC points to the next instruction to be executed. Thus, as soon as the execution of the current instruction is completed, the processor is ready to fetch a new instruction.

In addition to transferring data between the memory and the processor, the computer accepts data from input devices and sends data to output devices. Thus, some machine instructions are provided for the purpose of handling I/O transfers.

Normal execution of a program may be preempted if some device requires urgent service. For example, a monitoring device in a computer-controlled industrial process may detect a dangerous condition. In order to respond immediately, execution of the current program must be suspended. To cause this, the device raises an *interrupt* signal, which is a request for service by the processor. The processor provides the requested service by executing a program called an *interrupt-service routine*. Because such diversions may alter the internal state of the processor, its state must be saved in the memory before servicing the interrupt request. Normally, the information that is saved includes the contents of the PC, the contents of the general-purpose registers, and some control information. When the interrupt-service routine is completed, the state of the processor is restored from the memory so that the interrupted program may continue.

This section has provided an overview of the operation of a computer. Detailed discussion of these concepts is given in subsequent chapters, first from the point of view of the programmer in Chapters 2, 3, and 4, and then from the point of view of the hardware designer in later chapters.

## 1.4 Number Representation and Arithmetic Operations

The most natural way to represent a number in a computer system is by a string of bits, called a binary number. We will first describe binary number representations for integers as well as arithmetic operations on them. Then we will provide a brief introduction to the representation of floating-point numbers.

### 1.4.1 Integers

Consider an  $n$ -bit vector

$$B = b_{n-1} \dots b_1 b_0$$

where  $b_i = 0$  or  $1$  for  $0 \leq i \leq n - 1$ . This vector can represent an unsigned integer value  $V(B)$  in the range  $0$  to  $2^n - 1$ , where

$$V(B) = b_{n-1} \times 2^{n-1} + \dots + b_1 \times 2^1 + b_0 \times 2^0$$

We need to represent both positive and negative numbers. Three systems are used for representing such numbers:

- Sign-and-magnitude
- 1's-complement
- 2's-complement

In all three systems, the leftmost bit is  $0$  for positive numbers and  $1$  for negative numbers. Figure 1.3 illustrates all three representations using 4-bit numbers. Positive values have identical representations in all systems, but negative values have different representations. In the *sign-and-magnitude* system, negative values are represented by changing the most

$B$ Values represented							
$b_3 b_2 b_1 b_0$ Sign and				magnitude 1's complement 2's complement			
	1	1	1	0 1	4 - 5 - 6 - 0		
0 0 0 0	1	1	0 0 1	0 1	- 7	+ 7 + 6 +	
0 0 0 1	1	1	1	0 1	+ 7 + 6 + 5 + 4 + 3		
1	1	1	1		+ 7 + 6 + 5 + 4 + 3 + 2 + 1 +		
1	1	1	0 1	0 1	5 + 4 + 3 + 2 + 1 + 0 - 8 - 7		
1	1	0 0 1	0 1		+ 2 + 1 + 0 - 7 - 6 - 6 - 5 -		
1	0 0 0 0	1	0 1	0 - 0 - 1 - 5 - 4 - 4 - 3 - 2			
1	0 0 0 1	0 0 0 1	0 0 1	- 2 - 3 - 3 - 2 - 1 - 1			

**Figure 1.3** Binary, signed-integer representations.

significant bit ( $b_3$  in Figure 1.3) from  $0$  to  $1$  in the  $B$  vector of the corresponding positive value. For example,  $+5$  is represented by  $0101$ , and  $-5$  is represented by  $1101$ .

In *1's-complement* representation, negative values are obtained by complementing each bit of the corresponding positive number. Thus, the representation for  $-3$  is obtained by complementing each bit in the vector  $0011$  to yield  $1100$ . The same operation, bit complementing, is done to convert a negative number to the corresponding positive value. Converting either way is referred to as forming the *1's-complement* of a given number. For  $n$ -bit numbers, this operation is equivalent to subtracting the number from  $2^n - 1$ . In the case of the 4-bit numbers in Figure 1.3, we subtract from  $2^4 - 1 = 15$ , or  $1111$  in binary.

Finally, in the *2's-complement* system, forming the *2's-complement* of an  $n$ -bit



number is done by subtracting the number from  $2^n$ . Hence, the 2's-complement of a number is obtained by adding 1 to the 1's-complement of that number.

Note that there are distinct representations for +0 and -0 in both the sign-and-magnitude and 1's-complement systems, but the 2's-complement system has only one representation for 0. For 4-bit numbers, as shown in Figure 1.3, the value -8 is representable in the 2's-complement system but not in the other systems. The sign-and-magnitude system seems the most natural, because we deal with sign-and-magnitude decimal values in manual computations. The 1's-complement system is easily related to this system, but the 2's-complement system may appear somewhat unnatural. However, we will show that the 2's-complement system leads to the most efficient way to carry out addition and subtraction operations. It is the one most often used in modern computers.

### Addition of Unsigned Integers

Addition of 1-bit numbers is illustrated in Figure 1.4. The sum of 1 and 1 is the 2-bit vector 10, which represents the value 2. We say that the *sum* is 0 and the *carry-out* is 1. In order to add multiple-bit numbers, we use a method analogous to that used for manual computation with decimal numbers. We add bit pairs starting from the low-order (right) end of the bit vectors, propagating carries toward the high-order (left) end. The carry-out from a bit pair becomes the *carry-in* to the next bit pair to the left. The carry-in must be added to a bit pair in generating the sum and carry-out at that position. For example, if both bits of a pair are 1 and the carry-in is 1, then the sum is 1 and the carry-out is 1, which represents the value 3.

$$\begin{array}{r}
 \begin{array}{cc}
 0 & 0 \\
 + & 1 \\
 \hline
 1 & 1 \\
 \hline
 1 & 0
 \end{array} \\
 \begin{array}{cc}
 0 & 0 \\
 + & 1 \\
 \hline
 1 & 1 \\
 \hline
 1 & 0
 \end{array} \\
 \begin{array}{cc}
 0 & 0 \\
 + & 1 \\
 \hline
 1 & 1 \\
 \hline
 1 & 0
 \end{array}
 \end{array}$$

Carry-out

**Figure 1.4** Addition of 1-bit numbers.

### Addition and Subtraction of Signed Integers

We introduced three systems for representing positive and negative numbers, or, simply, *signed numbers*. These systems differ only in the way they represent negative values. Their relative merits from the standpoint of ease of performing arithmetic operations can be summarized as follows. The sign-and-magnitude system is the simplest representation, but it is also the most awkward for addition and subtraction operations. The 1's-complement method is somewhat better. The 2's-complement system is the most efficient method for performing addition and subtraction operations.

To understand 2's-complement arithmetic, consider addition modulo  $N$  (abbreviated as mod  $N$ ). A helpful graphical device for the description of addition of unsigned integers mod  $N$  is a circle with the values 0 through  $N - 1$  marked along its perimeter, as shown in Figure 1.5a. Consider the case  $N = 16$ , shown in part (b) of the figure. The decimal values 0 through 15 are represented by their 4-bit binary values 0000 through 1111 around the outside of the circle. In terms of decimal values, the operation  $(7 + 5) \bmod 16$  yields the value 12. To perform this operation graphically, locate 7 (0111) on the outside of the circle and then move 5 units in the clockwise direction to arrive at the answer 12 (1100). Similarly,  $(9 + 14) \bmod 16 = 7$ ; this is modeled on the circle by locating 9 (1001) and moving 14 units in the clockwise direction past the zero position to arrive at the answer 7

(0111). This graphical technique works for the computation of  $(a + b) \bmod 16$  for any unsigned integers  $a$  and  $b$ ; that is, to perform addition, locate  $a$  and move  $b$  units in the clockwise direction to arrive at  $(a + b) \bmod 16$ .

Now consider a different interpretation of the mod 16 circle. We will reinterpret the binary vectors outside the circle to represent the signed integers from  $-8$  through  $+7$  in the 2's-complement representation as shown inside the circle.

Let us apply the mod 16 addition technique to the example of adding  $+7$  to  $-3$ . The 2's-complement representation for these numbers is 0111 and 1101, respectively. To add these numbers, locate 0111 on the circle in Figure 1.5*b*. Then move 1101 (13) steps in the clockwise direction to arrive at 0100, which yields the correct answer of  $+4$ . Note that the 2's-complement representation of  $-3$  is interpreted as an unsigned value for the number of steps to move.

If we perform this addition by adding bit pairs from right to left, we obtain

$$\begin{array}{r} 0111 \\ + 1101 \\ \hline 1\ 0100 \end{array}$$

↑  
Carry-out

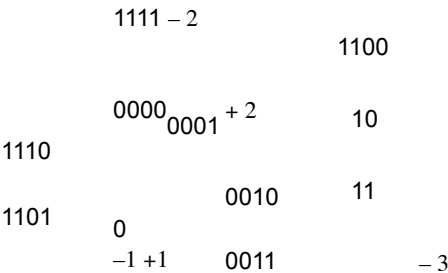
If we ignore the carry-out from the fourth bit position in this addition, we obtain the correct answer. In fact, this is always the case. Ignoring this carry-out is a natural result of using mod  $N$  arithmetic. As we move around the circle in Figure 1.5*b*, the value next to 1111 would normally be 10000. Instead, we go back to the value 0000.

The rules governing addition and subtraction of  $n$ -bit signed numbers using the 2's complement representation system may be stated as follows:

**1.4 Number Representation and Arithmetic Operations**  $\overset{0}{N-1} \ 1$

$$\begin{array}{l} N-2 \\ 2 \end{array}$$

(a) Circle representation of integers mod N



	- 4	+ 4	0	1000
	- 5	+ 5	0101	+ 7
		+ 6		0110
			1010	0111
			010	1001
			- 7	- 8
- 6				
	+ 3			

(b) Mod 16 system for 2's-complement numbers

**Figure 1.5** Modular number systems and the 2's-complement system.

- To *add* two numbers, add their  $n$ -bit representations, ignoring the carry-out bit from the most significant bit (MSB) position. The sum will be the algebraically correct value in 2's-complement representation if the actual result is in the range  $-2^{n-1}$  through  $+2^{n-1} - 1$ .
- To *subtract* two numbers  $X$  and  $Y$ , that is, to perform  $X - Y$ , form the 2's-complement of  $Y$ , then add it to  $X$  using the *add* rule. Again, the result will be the algebraically correct value in 2's-complement representation if the actual result is in the range  $-2^{n-1}$  through  $+2^{n-1} - 1$ .

#### 14 CHAPTER 1 Basic Structure of Computers

Figure 1.6 shows some examples of addition and subtraction in the 2's-complement system. In all of these 4-bit examples, the answers fall within the representable range of  $-8$  through  $+7$ . When answers do not fall within the representable range, we say that *arithmetic overflow* has occurred. A later subsection discusses such situations. The four addition operations (a) through (d) in Figure 1.6 follow the add rule, and the six subtraction operations (e) through (j) follow the subtract rule. The subtraction operation requires forming the 2's-complement of the subtrahend (the bottom value). This operation

	11011	001		01001101	
	001			010	0011
(a) (c) (e)		00101	() +2 ()	1110	10010
	00100	101	() +2 ()	01111	101
	100	+3	() +6 ()	101	1110
(f) (g) (h)		() +5	+3	0100	10011
	00100	01100	() -5 ()	11010	111
	011	011	-2	+++++111	1000
(i) (j)	0101	() -7	() -7 ()	0100	00100
	10111	10011	() -3 ()	00101	011
	110	011	-7	+++	100
	1001	+1	() -7 ()	1110	() +4 ()
				01101	-6
					() -2



## 1.4.2 Floating-Point Numbers

Until now we have only considered integers, which have an implied binary point at the right end of the number, just after bit  $b_0$ . If we use a full word in a 32-bit word length computer to represent a signed integer in 2's-complement representation, the range of values that can be represented is  $-2^{31}$  to  $+2^{31} - 1$ . In decimal terms, this range is somewhat smaller than  $-10^{10}$  to  $+10^{10}$ .

The same 32-bit patterns can also be interpreted as fractions in the range  $-1$  to  $+1 - 2^{-31}$  if we assume that the implied binary point is just to the right of the sign bit; that is, between bit  $b_{31}$  and bit  $b_{30}$  at the left end of the 32-bit representation. In this case, the magnitude of the smallest fraction representable is approximately  $10^{-10}$ .

Neither of these two *fixed-point* number representations has a range that is sufficient for many scientific and engineering calculations. For convenience, we would like to have a binary number representation that can easily accommodate both very large integers and very small fractions. To do this, a computer must be able to represent numbers and operate on them in such a way that the position of the binary point is variable and is automatically adjusted as computation proceeds. In this case, the binary point is said to *float*, and the numbers are called *floating-point numbers*.

Since the position of the binary point in a floating-point number varies, it must be indicated explicitly in the representation. For example, in the familiar decimal scientific notation, numbers may be written as  $6.0247 \times 10^{23}$ ,  $3.7291 \times 10^{-27}$ ,  $-1.0341 \times 10^2$ ,  $-7.3000 \times 10^{-14}$ , and so on. We say that these numbers have been given to 5 *significant digits* of precision. The *scale factors*  $10^{23}$ ,  $10^{-27}$ ,  $10^2$ , and  $10^{-14}$  indicate the actual position of the decimal point with respect to the significant digits. The same approach can be used to represent binary floating-point numbers in a computer, except that it is more appropriate to use 2 as the base of the scale factor. Because the base is fixed, it does not need to be given in the representation. The exponent may be positive or negative.

We conclude that a binary floating-point number can be represented by:

- a sign for the number
- some significant bits
- a signed scale factor exponent for an implied base of 2

An established international IEEE (Institute of Electrical and Electronics Engineers) standard for 32-bit floating-point number representation uses a sign bit, 23 significant bits, and 8 bits for a signed exponent of the scale factor, which has an implied base of 2. In decimal terms, the range of numbers represented is roughly  $\pm 10^{-38}$  to  $\pm 10^{38}$ , which is adequate for most scientific and engineering calculations. The same IEEE standard also defines a 64-bit representation to accommodate more significant bits and more bits for the signed exponent, resulting in much higher precision and a much larger range of values.

Floating-point number representation and arithmetic operations on floating-point numbers are considered in detail in Chapter 9. Some of the commercial processors described in Appendices B to E include operations on floating-point numbers in their instruction sets and have processor registers dedicated to holding floating-point numbers.

1.6 Performance 17

## 1.5 Character Representation

The most common encoding scheme for characters is ASCII (American Standard Code

for Information Interchange). Alphanumeric characters, operators, punctuation symbols, and control characters are represented by 7-bit codes as shown in Table 1.1. It is convenient to use an 8-bit *byte* to represent and store a character. The code occupies the low-order seven bits. The high-order bit is usually set to 0. Note that the codes for the alphabetic and numeric characters are in increasing sequential order when interpreted as unsigned binary numbers. This facilitates sorting operations on alphabetic and numeric data.

The low-order four bits of the ASCII codes for the decimal digits 0 to 9 are the first ten values of the binary number system. This 4-bit encoding is referred to as the *binary-coded decimal* (BCD) code.

## 1.6 Performance

The most important measure of the performance of a computer is how quickly it can execute programs. The speed with which a computer executes programs is affected by the design of its instruction set, its hardware and its software, including the operating system, and the technology in which the hardware is implemented. Because programs are usually written in a high-level language, performance is also affected by the compiler that translates programs into machine language. We do not describe the details of compilers or operating systems in this book. However, Chapter 4 provides an overview of software, including a discussion of the role of compilers and operating systems. This book concentrates on the design of instruction sets, along with memory, processor, and I/O hardware, and the organization of both small and large computers. Section 1.2.2 describes how caches can improve memory performance. Some performance aspects of instruction sets are discussed in Chapter 2. In this section, we give an overview of how performance is affected by technology, as well as processor and system organization.

### 1.6.1 Technology

The technology of Very Large Scale Integration (VLSI) that is used to fabricate the electronic circuits for a processor on a single chip is a critical factor in the speed of execution of machine instructions. The speed of switching between the 0 and 1 states in logic circuits is largely determined by the size of the transistors that implement the circuits. Smaller transistors switch faster. Advances in fabrication technology over several decades have reduced transistor sizes dramatically. This has two advantages: instructions can be executed faster, and more transistors can be placed on a chip, leading to more logic functionality and more memory storage capacity.

**Table 1.1** The 7-bit ASCII code.

Bit positions	Bit positions	654
3210	000 001 010 011 100 101 110 111 0000	NUL DLE SPACE 0 @ P ´ p 0001 SOH
DC1 ! 1 A Q a q	0010 STX DC2 ¨ 2 B R b r	0011 ETX DC3 # 3 C S c s
0100 EOT DC4 \$	4 D T d t	0101 ENQ NAK % 5 E U e u
0110 ACK SYN & 6 F V f v	0111 BEL ETB ´ 7 G	W g w
1000 BS CAN ( 8 H X h x	1001 HT EM ) 9 I Y i y	1010 LF SUB *: J Z j z
1011		

VT ESC + ; K [ k { 1100 FF FS , < L / 1 | 1101 CR GS - = M ] m } 1110 SO RS . > N ^ n ~  
 1111 SI US / ? O — ° DEL NUL Null/Idle SI Shift in

SOH Start of header DLE Data link escape  
 STX Start of text DC1-DC4 Device control

ETX End of text NAK Negative acknowledgment EOT End of transmission SYN  
 Synchronous idle

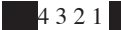
ENQ Enquiry ETB End of transmitted block ACK Acknowledgment CAN Cancel (error  
 in data) BEL Audible signal EM End of medium

BS Back space SUB Special sequence

HT Horizontal tab ESC Escape

LF Line feed FS File separator  
 VT Vertical tab GS Group separator  
 FF Form feed RS Record separator  
 CR Carriage return US Unit separator

SO Shift out DEL Delete/Idle

Bit positions of code format = 

## 1.7 Historical Perspective 19

### 1.6.2 Parallelism

Performance can be increased by performing a number of operations in parallel. Parallelism can be implemented on many different levels.

#### Instruction-level Parallelism

The simplest way to execute a sequence of instructions in a processor is to complete all steps of the current instruction before starting the steps of the next instruction. If we overlap the execution of the steps of successive instructions, total execution time will be reduced. For example, the next instruction could be fetched from memory at the same time that an arithmetic operation is being performed on the register operands of the current instruction. This form of parallelism is called *pipelining*. It is discussed in detail in Chapter 6.

#### Multicore Processors

Multiple processing units can be fabricated on a single chip. In technical literature, the term *core* is used for each of these processors. The term processor is then used for the complete chip. Hence, we have the terminology *dual-core*, *quad-core*, and *octo-core* processors for chips that have two, four, and eight cores, respectively.

#### Multiprocessors

Computer systems may contain many processors, each possibly containing multiple cores. Such systems are called *multiprocessors*. These systems either execute a number of different application tasks in parallel, or they execute subtasks of a single large task in parallel. All processors usually have access to all of the memory in such systems, and the term *shared-memory multiprocessor* is often used to make this clear. The high performance of these systems comes with much higher complexity and cost, arising from the use of multiple processors and memory units, along with more complex interconnection networks.

In contrast to multiprocessor systems, it is also possible to use an interconnected group of complete computers to achieve high total computational power. The computers normally have access only to their own memory units. When the tasks they are executing need to share data, they do so by exchanging *messages* over a communication network. This property distinguishes them from shared-memory multiprocessors, leading to the

## 1.7 Historical Perspective

Electronic digital computers as we know them today have been developed since the 1940s. A long, slow evolution of mechanical calculating devices preceded the development of electronic computers. Here, we briefly sketch the history of computer development. A more extensive coverage can be found in Hayes [1].

### 20 CHAPTER 1 Basic Structure of Computers

In the 300 years before the mid-1900s, a series of increasingly complex mechanical devices, constructed from gear wheels, levers, and pulleys, were used to perform the basic operations of addition, subtraction, multiplication, and division. Holes on punched cards were mechanically sensed and used to control the automatic sequencing of a list of calculations, which essentially provided a programming capability. These devices enabled the computation of complete mathematical tables of logarithms and trigonometric functions as approximated by polynomials. Output results were punched on cards or printed on paper. Electromechanical relay devices, such as those used in early telephone switching systems, provided the means for performing logic functions in computers built in the late 1930s and early 1940s.

During World War II, the first electronic computer was designed and built at the University of Pennsylvania, using the vacuum tube technology developed for radios and military radar equipment. Vacuum tube circuits were used to perform logic operations and to store data. This technology initiated the modern era of electronic digital computers.

Development of the technologies used to fabricate processors, memories, and I/O units of computers has been divided into four generations: the first generation, 1945 to 1955; the second generation, 1955 to 1965; the third generation, 1965 to 1975; and the fourth generation, 1975 to the present.

### 1.7.1 The First Generation

The key concept of a stored program was introduced at the same time as the development of the first electronic digital computer. Programs and their data were located in the same memory, as they are today. This facilitates changing existing programs and data or preparing and loading new programs and data. Assembly language was used to prepare programs and was translated into machine language for execution.

Basic arithmetic operations were performed in a few milliseconds, using vacuum tube technology to implement logic functions. This provided a 100- to 1000-fold increase in speed relative to earlier mechanical and electromechanical technology. Mercury delay-line memory was used at first. I/O functions were performed by devices similar to typewriters. Magnetic core memories and magnetic tape storage devices were also developed.

### 1.7.2 The Second Generation

The transistor was invented at AT&T Bell Laboratories in the late 1940s and quickly re



placed the vacuum tube in implementing logic functions. This fundamental technology shift marked the start of the second generation. Magnetic core memories and magnetic drum storage devices were widely used in the second generation. Magnetic disk storage devices were developed in this generation. The earliest high-level languages, such as Fortran, were developed, making the preparation of application programs much easier. Compilers were developed to translate these high-level language programs into assembly language, which was then translated into executable machine-language form. IBM became a major computer manufacturer during this time.

## 1.7 Historical Perspective 21

### 1.7.3 The Third Generation

Texas Instruments and Fairchild Semiconductor developed the ability to fabricate many transistors on a single silicon chip, called integrated-circuit technology. This enabled faster and less costly processors and memory elements to be built. Integrated-circuit memories began to replace magnetic core memories. This technological development marked the beginning of the third generation. Other developments included the introduction of microprogramming, parallelism, and pipelining. Operating system software allowed efficient sharing of a computer system by several user programs. Cache and virtual memories were developed. Cache memory makes the main memory appear faster than it really is, and virtual memory makes it appear larger. System 360 mainframe computers from IBM and the line of PDP minicomputers from Digital Equipment Corporation were dominant commercial products of the third generation.

### 1.7.4 The Fourth Generation

By the early 1970s, integrated-circuit fabrication techniques had evolved to the point where complete processors and large sections of the main memory of small computers could be implemented on single chips. This marked the start of the fourth generation. Tens of thousands of transistors could be placed on a single chip, and the name Very Large Scale Integration (VLSI) was coined to describe this technology. A complete processor fabricated on a single chip became known as a microprocessor. Companies such as Intel, National Semiconductor, Motorola, Texas Instruments, and Advanced Micro Devices have been the driving forces of this technology. Current VLSI technology enables the integration of multiple processors (cores) and cache memories on a single chip.

A particular form of VLSI technology, called Field Programmable Gate Arrays (FPGAs), has allowed system developers to design and implement processor, memory, and I/O circuits on a single chip to meet the requirements of specific applications, especially in embedded computer systems. Sophisticated computer-aided-design tools make it possible to develop FPGA-based products quickly. Companies such as Altera and Xilinx provide this technology, along with the required software development systems.

Embedded computer systems, portable notebook computers, and versatile mobile telephone handsets are now in widespread use. Desktop personal computers and workstations interconnected by wired or wireless local area networks and the Internet, with access to database servers and search engines, provide a variety of powerful computing platforms.

Organizational concepts such as parallelism and hierarchical memories have evolved to produce the high-performance computing systems of today as the fourth generation has matured. Supercomputers and Grid computers, at the upper end of high-performance computing, are used for weather forecasting, scientific and engineering computations, and simulations.

## 1.8 Concluding Remarks

This chapter has introduced basic concepts about the structure of computers and their operation. Machine instructions and programs have been described briefly. The addition and subtraction of binary numbers has been explained. Much of the terminology needed to deal with these subjects has been defined. Subsequent chapters provide detailed explanations of these terms and concepts, with an emphasis on architecture and hardware.

## 1.9 Solved Problems

This section presents some examples of the types of problems that a student may be asked to solve, and shows how such problems can be solved.

**Example 1.1 Problem:** List the steps needed to execute the machine instruction

Load R2, LOC

in terms of transfers between the components shown in Figure 1.2 and some simple control commands. An overview of the steps needed is given in Section 1.3. Assume that the address of the memory location containing this instruction is initially in register PC.

**Solution:** The required steps are:

- Send the address of the instruction word from register PC to the memory and issue a Read control command.
- Wait until the requested word has been retrieved from the memory, then load it into register IR, where it is interpreted (decoded) by the control circuitry to determine the operation to be performed.
- Increment the contents of register PC to point to the next instruction in memory.
- Send the address value LOC from the instruction in register IR to the memory and issue a Read control command.
- Wait until the requested word has been retrieved from the memory, then load it into register R2.

**Example 1.2 Problem:** Quantify the effect on performance that results from the use of a cache in the case of a program that has a total of 500 instructions, including a 100-instruction loop that is executed 25 times. Determine the ratio of execution time without the cache to execution time with the cache. This ratio is called the *speedup*.

**1.9 Solved Problems 23**

Assume that main memory accesses require 10 units of time and cache accesses require 1 unit of time. We also make the following further assumptions so that we can simplify calculations in order to easily illustrate the advantage of using a cache:

- Program execution time is proportional to the total amount of time needed to fetch instructions from either the main memory or the cache, with operand data accesses being ignored.
- Initially, all instructions are stored in the main memory, and the cache is empty.

The cache is large enough to contain all of the loop instructions.

**Solution:** Execution time without the cache is

$$T = 400 \times 10 + 100 \times 10 \times 25 = 29,000$$

Execution time with the cache is

$$T_{cache} = 500 \times 10 + 100 \times 1 \times 24 = 7,400$$

Therefore, the speedup is

$$T/T_{cache} = 3.92$$

**Problem:** Convert the following pairs of decimal numbers to 5-bit 2's-complement numbers, then perform addition and subtraction on each pair. Indicate whether or not overflow occurs for each case.

(a) 7 and 13

(b) -12 and 9

**Solution:** The conversion and operations are:

(a)  $7_{10} = 00111_2$  and  $13_{10} = 01101_2$

Adding these two positive numbers, we obtain 10100, which is a negative number. Therefore, overflow has occurred.

To subtract them, we first form the 2's-complement of 01101, which is 10011.

Then we perform addition with 00111 to obtain 11010, which is  $-6_{10}$ , the correct answer.

(b)  $-12_{10} = 10100_2$  and  $9_{10} = 01001_2$

Adding these two numbers, we obtain 11101 =  $-3_{10}$ , the correct answer.

To subtract them, we first form the 2's-complement of 01001, which is 10111.

Then we perform addition of the two negative numbers 10100 and 10111 to obtain 01011, which is a positive number. Therefore, overflow has occurred.

## Problems

**1.1 [E]** Repeat Example 1.1 for the machine instruction

Add R4, R2, R3

which is discussed in Section 1.3.

**1.2 [E]** Repeat Example 1.1 for the machine instruction

Store R4, LOC

which is discussed in Section 1.3.

- 1.3 [M]** (a) Give a short sequence of machine instructions for the task “Add the contents of memory location A to those of location B, and place the answer in location C”.

Instructions

Load  $R_i$ , LOC

and

Store  $R_i$ , LOC

are the only instructions available to transfer data between the memory and the general purpose registers. Add instructions are described in Section 1.3. Do not change the contents of either location A or B.

- (b) Suppose that Move and Add instructions are available with the formats

Move Location1, Location2

and

Add Location1, Location2

These instructions move or add a copy of the operand at the second location to the first location, overwriting the original operand at the first location. Either or both of the operands can be in the memory or the general-purpose registers. Is it possible to use fewer instructions of these types to accomplish the task in part (a)? If yes, give the sequence.

- 1.4 [M]** (a) A program consisting of a total of 300 instructions contains a 50-instruction loop that is executed 15 times. The processor contains a cache, as described in Section 1.2.2. Fetching and executing an instruction that is in the main memory requires 20 time units. If the instruction is found in the cache, fetching and executing it requires only 2 time units. Ignoring operand data accesses, calculate the ratio of program execution time without the cache to execution time with the cache. This ratio is called the *speedup* due to the use of the cache. Assume that the cache is initially empty, that it is large enough to hold the loop, and that the program starts with all instructions in the main memory.
- (b) Generalize part (a) by replacing the constants 300, 50, 15, 20, and 2 with the variables  $w$ ,  $x$ ,  $y$ ,  $m$ , and  $c$ . Develop an expression for speedup.
- (c) For the values  $w = 300$ ,  $x = 50$ ,  $m = 20$ , and  $c = 2$  what value of  $y$  results in a speedup of 5?

## References 25

- (d) Consider the form of the expression for speedup developed in part (b). What is the upper limit on speedup as the number of loop iterations,  $y$ , becomes larger and larger?

- 1.5 [M]** (a) A processor cache is discussed in Section 1.2.2. Suppose that execution time for a program is proportional to instruction fetch time. Assume that fetching an instruction from the cache takes 1 time unit, but fetching it from the main memory takes 10 time units. Also, assume that a requested instruction is found in the cache with probability 0.96. Finally, assume that if an instruction is not found in the cache it must first be fetched from the main memory into the cache and then fetched from the cache to be executed. Compute the ratio of program execution time without the cache to program execution time with the cache. This ratio is called the *speedup* resulting from the presence of the cache.

- (b) If the size of the cache is doubled, assume that the probability of not finding a requested instruction there is cut in half. Repeat part (a) for a doubled cache size.

- 1.6 [E]** Extend Figure 1.4 to incorporate both possibilities for a carry-in (0 or 1) to each of the four cases shown in the figure. Specify both the sum and carry-out bits for each of the eight new cases.
- 1.7 [M]** Convert the following pairs of decimal numbers to 5-bit 2's-complement numbers, then add them. State whether or not overflow occurs in each case.
- (a) 4 and 11
  - (b) 6 and 14
  - (c) -13 and 12
  - (d) -4 and 8
  - (e) -2 and -9
  - (f) -9 and -14
- 1.8 [M]** Repeat Problem 1.7 for the subtract operation, where the second number of each pair is to be subtracted from the first number. State whether or not overflow occurs in each case.
- 1.9 [E]** A memory byte location contains the pattern 01010011. What decimal value does this pattern represent when interpreted as a binary number? What does it represent as an ASCII code?
- 1.10 [E]** A way to detect overflow when adding two 2's-complement numbers is given at the end of Section 1.4.1. State how to detect overflow when subtracting two such numbers.

## References

1. J. P. Hayes, *Computer Architecture and Organization*, 3rd Ed., McGraw-Hill, New York, 1998.

*This page intentionally left blank*  
**chapter**

# 2

## Instruction Set Architecture

### Chapter Objectives

In this chapter you will learn about:

- Machine instructions and program execution
- Addressing methods for accessing register and memory operands

- Assembly language for representing machine instructions, data, and programs
- Stacks and subroutines

## 28 CHAPTER 2 Instruction Set Architecture

This chapter considers the way programs are executed in a computer from the machine instruction set view point. Chapter 1 introduced the general concept that both program instructions and data operands are stored in the memory. In this chapter, we discuss how instructions are composed and study the ways in which sequences of instructions are brought from the memory into the processor and executed to perform a given task. The addressing methods that are commonly used for accessing operands in memory locations and processor registers are also presented.

The emphasis here is on basic concepts. We use a generic style to describe machine instructions and operand addressing methods that are typical of those found in commercial processors. A sufficient number of instructions and addressing methods are introduced to enable us to present complete, realistic programs for simple tasks. These generic programs are specified at the assembly-language level, where machine instructions and operand addressing information are represented by symbolic names. A complete instruction set, including operand addressing methods, is often referred to as the *instruction set architecture* (ISA) of a processor. For the discussion of basic concepts in this chapter, it is not necessary to define a complete instruction set, and we will not attempt to do so. Instead, we will present enough examples to illustrate the capabilities of a typical instruction set.

The concepts introduced in this chapter and in Chapter 3, which deals with input/output techniques, are essential for understanding the functionality of computers. Our choice of the generic style of presentation makes the material easy to read and understand. Also, this style allows a general discussion that is not constrained by the characteristics of a particular processor.

Since it is interesting and important to see how the concepts discussed are implemented in a real computer, we supplement our presentation in Chapters 2 and 3 with four examples of popular commercial processors. These processors are presented in Appendices B to E. Appendix B deals with the Nios II processor from Altera Corporation. Appendix C presents the ColdFire processor from Freescale Semiconductor, Inc. Appendix D discusses the ARM processor from ARM Ltd. Appendix E presents the basic architecture of processors made by Intel Corporation. The generic programs in Chapters 2 and 3 are presented in terms of the specific instruction sets in each of the appendices.

The reader can choose only one processor and study the material in the corresponding appendix to get an appreciation for commercial ISA design. However, knowledge of the material in these appendices is not

essential for understanding the material in the main body of the book.

The vast majority of programs are written in high-level languages such as C, C++, or Java. To execute a high-level language program on a processor, the program must be translated into the machine language for that processor, which is done by a compiler program. Assembly language is a readable symbolic representation of machine language. In this book we make extensive use of assembly language, because this is the best way to describe how computers work.

We will begin the discussion in this chapter by considering how instructions and data are stored in the memory and how they are accessed for processing.

## 2.1 Memory Locations and Addresses

We will first consider how the memory of a computer is organized. The memory consists of many millions of storage *cells*, each of which can store a *bit* of information having the value 0 or 1. Because a single bit represents a very small amount of information, bits are seldom handled individually. The usual approach is to deal with them in groups of fixed size. For

### 2.1 Memory Locations and Addresses 29

this purpose, the memory is organized so that a group of  $n$  bits can be stored or retrieved in a single, basic operation. Each group of  $n$  bits is referred to as a *word* of information, and  $n$  is called the *word length*. The memory of a computer can be schematically represented as a collection of words, as shown in Figure 2.1.

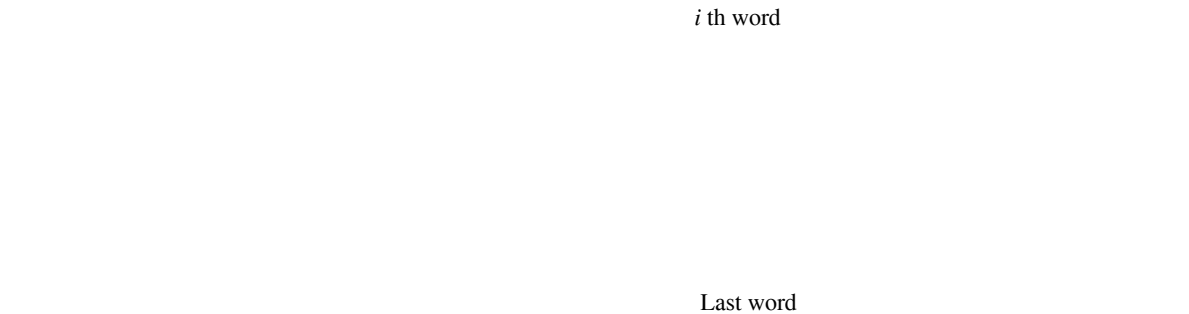
Modern computers have word lengths that typically range from 16 to 64 bits. If the word length of a computer is 32 bits, a single word can store a 32-bit signed number or four ASCII-encoded characters, each occupying 8 bits, as shown in Figure 2.2. A unit of 8 bits is called a *byte*. Machine instructions may require one or more words for their representation. We will discuss how machine instructions are encoded into memory words in a later section, after we have described instructions at the assembly-language level.

Accessing the memory to store or retrieve a single item of information, either a word or a byte, requires distinct names or *addresses* for each location. It is customary to use numbers from 0 to  $2^k - 1$ , for some suitable value of  $k$ , as the addresses of successive locations in the memory. Thus, the memory can have up to  $2^k$  addressable locations. The  $2^k$  addresses constitute the *address space* of the computer. For example, a 24-bit address generates an address space of  $2^{24}$  (16,777,216) locations. This number is usually written as 16M (16 mega), where 1M is the number  $2^{20}$  (1,048,576). A 32-bit address creates an address space of  $2^{32}$  or 4G (4 giga) locations, where 1G is  $2^{30}$ . Other notational conventions

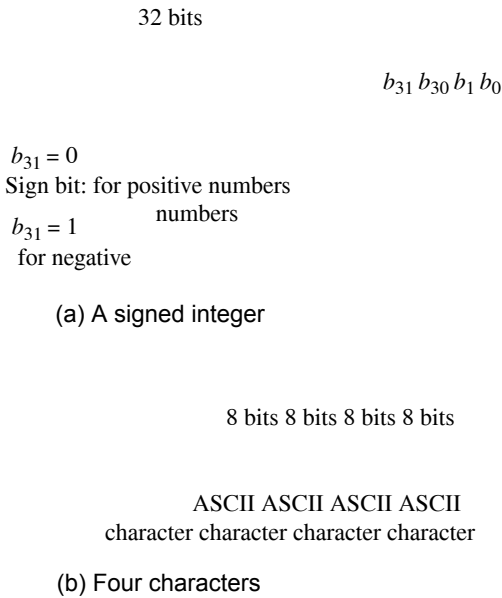
$n$  bits

First word

Second word



**Figure 2.1** Memory words.



**Figure 2.2** Examples of encoded information in a 32-bit word.

that are commonly used are K (kilo) for the number  $2^{10}$  (1,024), and T (tera) for the number  $2^{40}$ .

### 2.1.1 Byte Addressability

We now have three basic information quantities to deal with: bit, byte, and word. A byte is always 8 bits, but the word length typically ranges from 16 to 64 bits. It is impractical

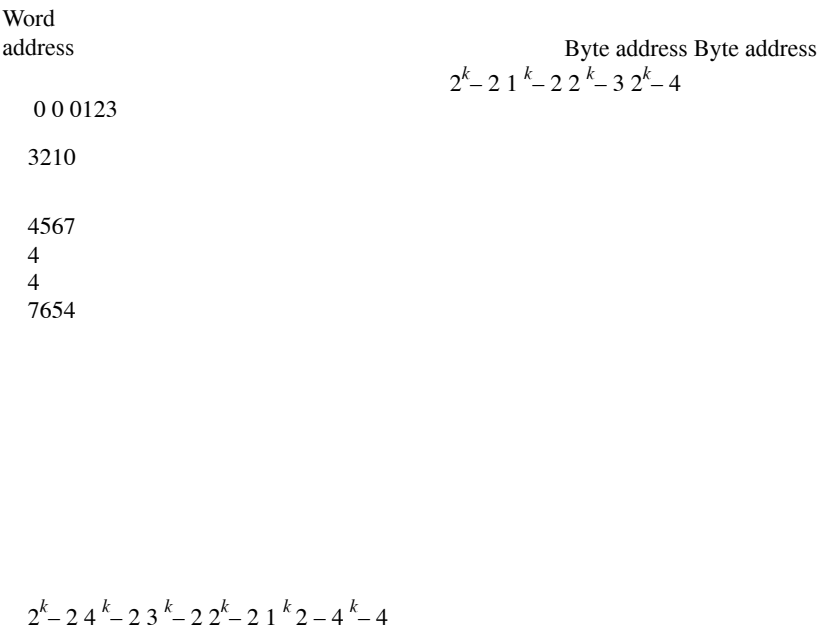


to assign distinct addresses to individual bit locations in the memory. The most practical assignment is to have successive addresses refer to successive byte locations in the memory. This is the assignment used in most modern computers. The term *byte-addressable memory* is used for this assignment. Byte locations have addresses 0, 1, 2,... Thus, if the word length of the machine is 32 bits, successive words are located at addresses 0, 4, 8,..., with each word consisting of four bytes.

### 2.1.2 Big-Endian and Little-Endian Assignments

There are two ways that byte addresses can be assigned across words, as shown in Figure 2.3. The name *big-endian* is used when lower byte addresses are used for the more significant bytes (the leftmost bytes) of the word. The name *little-endian* is used for the opposite ordering, where the lower byte addresses are used for the less significant bytes (the rightmost bytes) of the word. The words “more significant” and “less significant” are used in relation to the weights (powers of 2) assigned to bits when the word represents a number. Both little-endian and big-endian assignments are used in commercial machines. In both cases, byte addresses 0, 4, 8,..., are taken as the addresses of successive words in the memory

2.1 Memory Locations and Addresses 31



(a) Big-endian assignment (b) Little-endian assignment **Figure 2.3** Byte

and word addressing.

of a computer with a 32-bit word length. These are the addresses used when accessing the memory to store or retrieve a word.

In addition to specifying the address ordering of bytes within a word, it is also necessary to specify the labeling of bits within a byte or a word. The most common convention, and the one we will use in this book, is shown in Figure 2.2a. It is the most

natural ordering for the encoding of numerical data. The same ordering is also used for labeling bits within a byte, that is,  $b_7, b_6, \dots, b_0$ , from left to right.

### 2.1.3 Word Alignment

In the case of a 32-bit word length, natural word boundaries occur at addresses 0, 4, 8, ..., as shown in Figure 2.3. We say that the word locations have *aligned* addresses if they begin at a byte address that is a multiple of the number of bytes in a word. For practical reasons associated with manipulating binary-coded addresses, the number of bytes in a word is a power of 2. Hence, if the word length is 16 (2 bytes), aligned words begin at byte addresses 0, 2, 4, ..., and for a word length of 64 ( $2^3$  bytes), aligned words begin at byte addresses 0, 8, 16, ...

There is no fundamental reason why words cannot begin at an arbitrary byte address. In that case, words are said to have *unaligned* addresses. But, the most common case is to use aligned addresses, which makes accessing of memory operands more efficient, as we will see in Chapter 8.

## 32 CHAPTER 2 \* Instruction Set Architecture

### 2.1.4 Accessing Numbers and Characters

A number usually occupies one word, and can be accessed in the memory by specifying its word address. Similarly, individual characters can be accessed by their byte address. For programming convenience it is useful to have different ways of specifying addresses in program instructions. We will deal with this issue in Section 2.4.

## 2.2 Memory Operations

Both program instructions and data operands are stored in the memory. To execute an instruction, the processor control circuits must cause the word (or words) containing the instruction to be transferred from the memory to the processor. Operands and results must also be moved between the memory and the processor. Thus, two basic operations involving the memory are needed, namely, *Read* and *Write*.

The Read operation transfers a copy of the contents of a specific memory location to the processor. The memory contents remain unchanged. To start a Read operation, the processor sends the address of the desired location to the memory and requests that its contents be read. The memory reads the data stored at that address and sends them to the processor.

The Write operation transfers an item of information from the processor to a specific memory location, overwriting the former contents of that location. To initiate a Write operation, the processor sends the address of the desired location to the memory, together with the data to be written into that location. The memory then uses the address and data to perform the write.

The details of the hardware implementation of these operations are treated in Chapters 5 and 6. In this chapter, we consider all operations from the viewpoint of the ISA, so we concentrate on the logical handling of instructions and operands.

## 2.3 Instructions and Instruction Sequencing

The tasks carried out by a computer program consist of a sequence of small steps, such as adding two numbers, testing for a particular condition, reading a character from the keyboard, or sending a character to be displayed on a display screen. A computer must have instructions capable of performing four types of operations:

- Data transfers between the memory and the processor registers
- Arithmetic and logic operations on data
- Program sequencing and control
- I/O transfers

We begin by discussing instructions for the first two types of operations. To facilitate the discussion, we first need some notation.

## 2.3 Instructions and Instruction Sequencing 33

### 2.3.1 Register Transfer Notation

We need to describe the transfer of information from one location in a computer to another. Possible locations that may be involved in such transfers are memory locations, processor registers, or registers in the I/O subsystem. Most of the time, we identify such locations symbolically with convenient names. For example, names that represent the addresses of memory locations may be LOC, PLACE, A, or VAR2. Predefined names for the processor registers may be R0 or R5. Registers in the I/O subsystem may be identified by names such as DATAIN or OUTSTATUS. To describe the transfer of information, the contents of any location are denoted by placing square brackets around its name. Thus, the expression

$$R2 \leftarrow [LOC]$$

means that the contents of memory location LOC are transferred into processor register R2. As another example, consider the operation that adds the contents of registers R2 and R3, and places their sum into register R4. This action is indicated as

$$R4 \leftarrow [R2] + [R3]$$

This type of notation is known as *Register Transfer Notation* (RTN). Note that the right hand side of an RTN expression always denotes a value, and the left-hand side is the name of a location where the value is to be placed, overwriting the old contents of that location.

In computer jargon, the words “transfer” and “move” are commonly used to mean “copy.” Transferring data from a *source* location A to a *destination* location B means that the contents of location A are read and then written into location B. In this operation, only the contents of the destination will change. The contents of the source will stay the same.

### 2.3.2 Assembly-Language Notation

We need another type of notation to represent machine instructions and programs. For this, we use *assembly language*. For example, a generic instruction that causes the transfer described above, from memory location LOC to processor register R2, is specified by the statement

## Load R2, LOC

The contents of LOC are unchanged by the execution of this instruction, but the old contents of register R2 are overwritten. The name Load is appropriate for this instruction, because the contents read from a memory location are *loaded* into a processor register.

The second example of adding two numbers contained in processor registers R2 and R3 and placing their sum in R4 can be specified by the assembly-language statement

Add R4, R2, R3

In this case, registers R2 and R3 hold the source operands, while R4 is the destination.

## 34 CHAPTER 2 • Instruction Set Architecture

An *instruction* specifies an operation to be performed and the operands involved. In the above examples, we used the English words Load and Add to denote the required operations. In the assembly-language instructions of actual (commercial) processors, such operations are defined by using *mnemonics*, which are typically abbreviations of the words describing the operations. For example, the operation Load may be written as LD, while the operation Store, which transfers a word from a processor register to the memory, may be written as STR or ST. Assembly languages for different processors often use different mnemonics for a given operation. To avoid the need for details of a particular assembly language at this early stage, we will continue the presentation in this chapter by using English words rather than processor-specific mnemonics.

### 2.3.3 RISC and CISC Instruction Sets

One of the most important characteristics that distinguish different computers is the nature of their instructions. There are two fundamentally different approaches in the design of instruction sets for modern computers. One popular approach is based on the premise that higher performance can be achieved if each instruction occupies exactly one word in memory, and all operands needed to execute a given arithmetic or logic operation specified by an instruction are already in processor registers. This approach is conducive to an implementation of the processing unit in which the various operations needed to process a sequence of instructions are performed in “pipelined” fashion to overlap activity and reduce total execution time of a program, as we will discuss in Chapter 6. The restriction that each instruction must fit into a single word reduces the complexity and the number of different types of instructions that may be included in the instruction set of a computer. Such computers are called *Reduced Instruction Set Computers* (RISC).

An alternative to the RISC approach is to make use of more complex instructions which may span more than one word of memory, and which may specify more complicated operations. This approach was prevalent prior to the introduction of the RISC approach in the 1970s. Although the use of complex instructions was not originally identified by any particular label, computers based on this idea have been subsequently called *Complex Instruction Set Computers* (CISC).

We will start our presentation by concentrating on RISC-style instruction sets because they are simpler and therefore easier to understand. Later we will deal with CISC-style instruction sets and explain the key differences between the two approaches.

### 2.3.4 Introduction to RISC Instruction Sets

Two key characteristics of RISC instruction sets are:

- Each instruction fits in a single word.
- A *load/store architecture* is used, in which
  - Memory operands are accessed only using Load and Store instructions.
  - All operands involved in an arithmetic or logic operation must either be in processor registers, or one of the operands may be given explicitly within the instruction word.

## 2.3 Instructions and Instruction Sequencing 35

At the start of execution of a program, all instructions and data used in the program are stored in the memory of a computer. Processor registers do not contain valid operands at that time. If operands are expected to be in processor registers before they can be used by an instruction, then it is necessary to first bring these operands into the registers. This task is done by Load instructions which copy the contents of a memory location into a processor register. Load instructions are of the form

Load destination, source

or more specifically

Load processor\_register, memory\_location

The memory location can be specified in several ways. The term *addressing modes* is used to refer to the different ways in which this may be accomplished, as we will discuss in Section 2.4.

Let us now consider a typical arithmetic operation. The operation of adding two numbers is a fundamental capability in any computer. The statement

$$C = A + B$$

in a high-level language program instructs the computer to add the current values of the two variables called A and B, and to assign the sum to a third variable, C. When the program containing this statement is compiled, the three variables, A, B, and C, are assigned to distinct locations in the memory. For simplicity, we will refer to the addresses of these locations as A, B, and C, respectively. The contents of these locations represent the values of the three variables. Hence, the above high-level language statement requires the action

$$C \leftarrow [A] + [B]$$

to take place in the computer. To carry out this action, the contents of memory locations A and B are fetched from the memory and transferred into the processor where their sum is computed. This result is then sent back to the memory and stored in location C.

The required action can be accomplished by a sequence of simple machine instructions. We choose to use registers R2, R3, and R4 to perform the task with four instructions:

Load R2, A  
Load R3, B  
Add R4, R2, R3  
Store R4, C

We say that Add is a *three-operand*, or a *three-address*, instruction of the form

Add destination, source1, source2

The Store instruction is of the form

Store source, destination

where the source is a processor register and the destination is a memory location. Observe that in the Store instruction the source and destination are specified in the reverse order from the Load instruction; this is a commonly used convention.

## 36 CHAPTER 2 Instruction Set Architecture

Note that we can accomplish the desired addition by using only two registers, R2 and R3, if one of the source registers is also used as the destination for the result. In this case the addition would be performed as

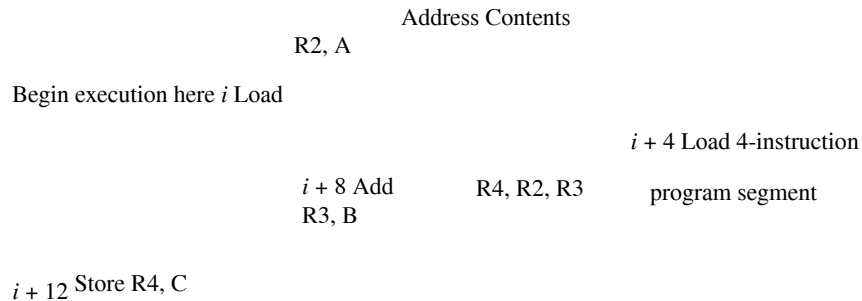
Add R3, R2, R3

and the last instruction would become

Store R3, C

### 2.3.5 Instruction Execution and Straight-Line Sequencing

In the preceding subsection, we used the task  $C = A + B$ , implemented as  $C \leftarrow [A] + [B]$ , as an example. Figure 2.4 shows a possible program segment for this task as it appears in the memory of a computer. We assume that the word length is 32 bits and the memory is byte-addressable. The four instructions of the program are in successive word locations, starting at location  $i$ . Since each instruction is 4 bytes long, the second, third, and fourth instructions are at addresses  $i + 4$ ,  $i + 8$ , and  $i + 12$ . For simplicity, we assume that a desired



**Figure 2.4** A program for  $C \leftarrow [A] + [B]$ .

A

B

C

memory address can be directly specified in Load and Store instructions, although this is not possible if a full 32-bit address is involved. We will resolve this issue later in Section 2.4. Let us consider how this program is executed. The processor contains a register called the *program counter* (PC), which holds the address of the next instruction to be executed. To begin executing a program, the address of its first instruction ( $i$  in our example) must be placed into the PC. Then, the processor control circuits use the information in the PC to fetch and execute instructions, one at a time, in the order of increasing addresses. This is called *straight-line sequencing*. During the execution of each instruction, the PC is incremented by 4 to point to the next instruction. Thus, after the Store instruction at location  $i + 12$  is executed, the PC contains the value  $i + 16$ , which is the address of the first instruction of the next program segment.

Executing a given instruction is a two-phase procedure. In the first phase, called *instruction fetch*, the instruction is fetched from the memory location whose address is in the PC. This instruction is placed in the *instruction register* (IR) in the processor. At the start of the second phase, called *instruction execute*, the instruction in IR is examined to determine which operation is to be performed. The specified operation is then performed by the processor. This involves a small number of steps such as fetching operands from the memory or from processor registers, performing an arithmetic or logic operation, and storing the result in the destination location. At some point during this two-phase procedure, the contents of the PC are advanced to point to the next instruction. When the execute phase of an instruction is completed, the PC contains the address of the next instruction, and a new instruction fetch phase can begin.

### 2.3.6 Branching

Consider the task of adding a list of  $n$  numbers. The program outlined in Figure 2.5 is a generalization of the program in Figure 2.4. The addresses of the memory locations containing the  $n$  numbers are symbolically given as NUM1, NUM2,..., NUM $n$ , and separate Load and Add instructions are used to add each number to the contents of register R2. After all the numbers have been added, the result is placed in memory location SUM.

Instead of using a long list of Load and Add instructions, as in Figure 2.5, it is possible to implement a program loop in which the instructions read the next number in the list and add it to the current sum. To add all numbers, the loop has to be executed as many times as there are numbers in the list. Figure 2.6 shows the structure of the desired program. The body of the loop is a straight-line sequence of instructions executed repeatedly. It starts at location LOOP and ends at the instruction Branch\_if\_[R2]>0. During each pass through this loop, the address of the next list entry is determined, and that entry is loaded into R5 and added to R3. The address of an operand can be specified in various ways, as will be described in Section 2.4. For now, we concentrate on how to create and control a program loop.

Assume that the number of entries in the list,  $n$ , is stored in memory location N, as shown. Register R2 is used as a counter to determine the number of times the loop is executed. Hence, the contents of location N are loaded into register R2 at the beginning of the program. Then, within the body of the loop, the instruction

Subtract R2, R2, #1

	Load R3, NUM2
$i + 8$	Add Load R3, NUM3
$i + 12$	Add R2, R2, R3
$i + 16$	
	Load Add R3, NUM $n$
$i + 8n - 12$	$i$ R2, R2, R3 R2, R2, R3
$+ 8n - 8$	
$i + 8n - 4$	NUM2
SUM	NUM $n$
NUM1	Store R2, SUM

**Figure 2.5** A program for adding  $n$  numbers.

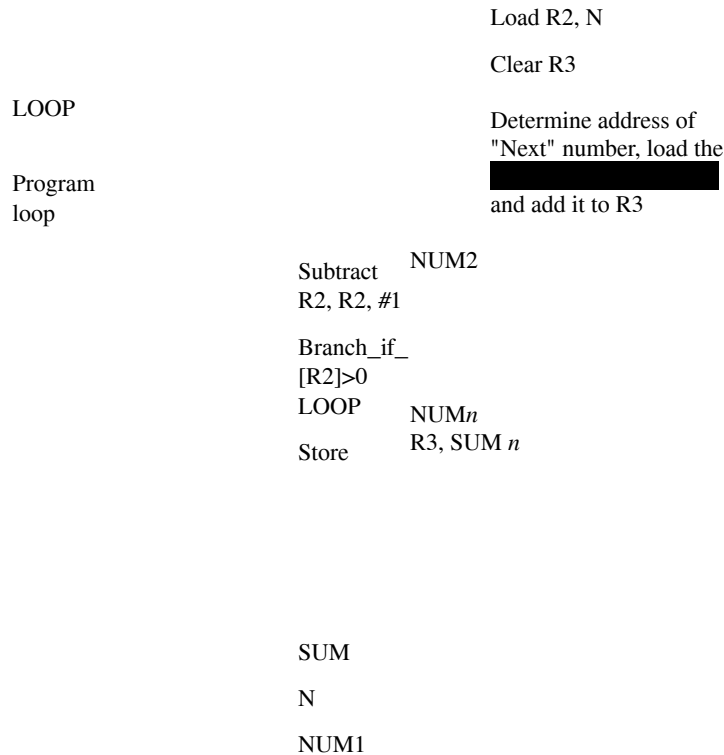
reduces the contents of R2 by 1 each time through the loop. (We will explain the significance of the number sign ‘#’ in Section 2.4.1.) Execution of the loop is repeated as long as the contents of R2 are greater than zero.

We now introduce *branch* instructions. This type of instruction loads a new address into the program counter. As a result, the processor fetches and executes the instruction at this new address, called the *branch target*, instead of the instruction at the location that follows the branch instruction in sequential address order. A *conditional branch* instruction causes a branch only if a specified condition is satisfied. If the condition is not satisfied, the PC is incremented in the normal way, and the next instruction in sequential address order is fetched and executed.

In the program in Figure 2.6, the instruction

Branch\_if\_[R2]>0 LOOP





**Figure 2.6** Using a loop to add  $n$  numbers.

is a conditional branch instruction that causes a branch to location LOOP if the contents of register R2 are greater than zero. This means that the loop is repeated as long as there are entries in the list that are yet to be added to R3. At the end of the  $n$ th pass through the loop, the Subtract instruction produces a value of zero in R2, and, hence, branching does not occur. Instead, the Store instruction is fetched and executed. It moves the final result from R3 into memory location SUM.

The capability to test conditions and subsequently choose one of a set of alternative ways to continue computation has many more applications than just loop control. Such a capability is found in the instruction sets of all computers and is fundamental to the programming of most nontrivial tasks.

One way of implementing conditional branch instructions is to compare the contents of two registers and then branch to the target instruction if the comparison meets the specified

requirement. For example, the instruction that implements the action

Branch\_if\_[R4]>[R5] LOOP

may be written in generic assembly language as

```
Branch_greater_than R4, R5, LOOP
```

or using an actual mnemonic as

```
BGT R4, R5, LOOP
```

It compares the contents of registers R4 and R5, without changing the contents of either register. Then, it causes a branch to LOOP if the contents of R4 are greater than the contents of R5.

A different way of implementing branch instructions uses the concept of condition codes, which we will discuss in Section 2.10.2.

2.3.7 Generating Memory Addresses

Let us return to Figure 2.6. The purpose of the instruction block starting at LOOP is to add successive numbers from the list during each pass through the loop. Hence, the Load instruction in that block must refer to a different address during each pass. How are the addresses specified? The memory operand address cannot be given directly in a single Load instruction in the loop. Otherwise, it would need to be modified on each pass through the loop. As one possibility, suppose that a processor register, *R<sub>i</sub>*, is used to hold the memory address of an operand. If it is initially loaded with the address NUM1 before the loop is entered and is then incremented by 4 on each pass through the loop, it can provide the needed capability.

This situation, and many others like it, give rise to the need for flexible ways to specify the address of an operand. The instruction set of a computer typically provides a number of such methods, called *addressing modes*. While the details differ from one computer to another, the underlying concepts are the same. We will discuss these in the next section.

2.4 Addressing Modes

We have now seen some simple examples of assembly-language programs. In general, a program operates on data that reside in the computer’s memory. These data can be organized in a variety of ways that reflect the nature of the information and how it is used. Programmers use *data structures* such as lists and arrays for organizing the data used in computations.

Programs are normally written in a high-level language, which enables the programmer to conveniently describe the operations to be performed on various data structures. When translating a high-level language program into assembly language, the compiler generates appropriate sequences of low-level instructions that implement the desired operations. The

Table 2.1 RISC-type addressing modes.

Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	<i>R<sub>i</sub></i>	EA = <i>R<sub>i</sub></i>

Absolute LOC EA = LOC

Register indirect (Ri) EA = [Ri]

Index X(Ri) EA = [Ri] + X

Base with index (Ri,Rj) EA = [Ri] + [Rj]

EA = effective address

Value = a signed number

X = index value

different ways for specifying the locations of instruction operands are known as *addressing modes*. In this section we present the basic addressing modes found in RISC-style processors. A summary is provided in Table 2.1, which also includes the assembler syntax we will use for each mode. The assembler syntax defines the way in which instructions and the addressing modes of their operands are specified; it is discussed in Section 2.5.

### 2.4.1 Implementation of Variables and Constants

Variables are found in almost every computer program. In assembly language, a variable is represented by allocating a register or a memory location to hold its value. This value can be changed as needed using appropriate instructions.

The program in Figure 2.5 uses only two addressing modes to access variables. We access an operand by specifying the name of the register or the address of the memory location where the operand is located. The precise definitions of these two modes are:

*Register mode*—The operand is the contents of a processor register; the name of the register is given in the instruction.

*Absolute mode*—The operand is in a memory location; the address of this location is given explicitly in the instruction.

Since in a RISC-style processor an instruction must fit in a single word, the number of bits that can be used to give an absolute address is limited, typically to 16 bits if the word length is 32 bits. To generate a 32-bit address, the 16-bit value is usually extended to 32 bits by replicating bit  $b_{15}$  into bit positions  $b_{31-16}$  (as in sign extension). This means that an absolute address can be specified in this manner for only a limited range of the full address space. We will deal with the issue of specifying full 32-bit addresses in Section 2.9. To keep our examples simple, we will assume for now that all addresses of memory locations involved in a program can be specified in 16 bits.

## 42 CHAPTER 2 • Instruction Set Architecture

The instruction

Add R4, R2, R3

uses the Register mode for all three operands. Registers R2 and R3 hold the two source operands, while R4 is the destination.

The Absolute mode can represent global variables in a program. A declaration such

as Integer NUM1, NUM2, SUM;

in a high-level language program will cause the compiler to allocate a memory location to each of the variables NUM1, NUM2, and SUM. Whenever they are referenced later in the

program, the compiler can generate assembly-language instructions that use the Absolute mode to access these variables.

The Absolute mode is used in the instruction

Load R2, NUM1

which loads the value in the memory location NUM1 into register R2.

Constants representing data or addresses are also found in almost every computer program. Such constants can be represented in assembly language using the Immediate addressing mode.

*Immediate mode*—The operand is given explicitly in the instruction.

For example, the instruction

Add R4, R6, 200<sub>immediate</sub>

adds the value 200 to the contents of register R6, and places the result into register R4. Using a subscript to denote the Immediate mode is not appropriate in assembly languages. A common convention is to use the number sign (#) in front of the value to indicate that this value is to be used as an immediate operand. Hence, we write the instruction above in the form

Add R4, R6, #200

In the addressing modes that follow, the instruction does not give the operand or its address explicitly. Instead, it provides information from which an *effective address* (EA) can be derived by the processor when the instruction is executed. The effective address is then used to access the operand.

### 2.4.2 Indirection and Pointers

The program in Figure 2.6 requires a capability for modifying the address of the memory operand during each pass through the loop. A good way to provide this capability is to use a processor register to hold the address of the operand. The contents of the register are then changed (incremented) during each pass to provide the address of the next number in the list that has to be accessed. The register acts as a *pointer* to the list, and we say that an item

Main memory

Load R2, (R5)

B

B Operand

**Figure 2.7** Register indirect addressing.  
**2.4 Addressing Modes 43** R5

in the list is accessed *indirectly* by using the address in the register. The desired capability is provided by the indirect addressing mode.

*Indirect mode*—The effective address of the operand is the contents of a register that is specified in the instruction.

We denote indirection by placing the name of the register given in the instruction in parentheses as illustrated in Figure 2.7 and Table 2.1.

To execute the Load instruction in Figure 2.7, the processor uses the value B, which is in register R5, as the effective address of the operand. It requests a Read operation to fetch the contents of location B in the memory. The value from the memory is the desired operand, which the processor loads into register R2. Indirect addressing through a memory location is also possible, but it is found only in CISC-style processors.

Indirection and the use of pointers are important and powerful concepts in programming. They permit the same code to be used to operate on different data. For example, register R5 in Figure 2.7 serves as a pointer for the Load instruction to load an operand from the memory into register R2. At one time, R5 may point to location B in memory. Later, the program may change the contents of R5 to point to a different location, in which case the same Load instruction will load the value from that location into R2. Thus, a program segment that includes this Load instruction is conveniently reused with only a change in the pointer value.

Let us now return to the program in Figure 2.6 for adding a list of numbers. Indirect addressing can be used to access successive numbers in the list, resulting in the program shown in Figure 2.8. Register R4 is used as a pointer to the numbers in the list, and the operands are accessed indirectly through R4. The initialization section of the program loads the counter value *n* from memory location N into R2. Then, it uses the Clear instruction to clear R3 to 0. The next instruction uses the Immediate addressing mode to place the address value NUM1, which is the address of the first number in the list, into R4. Observe that we cannot use the Load instruction to load the desired immediate value, because the Load instruction can operate only on memory source operands. Instead, we use the Move instruction

Move R4, #NUM1

#### 44 CHAPTER 2 Instruction Set Architecture

```
Load R2, N Load the size of the list.
Clear R3 Initialize sum to 0.
Move R4, #NUM1 Get address of the first number.
LOOP: Load R5, (R4) Get the next number.
Add R3, R3, R5 Add this number to sum.
Add R4, R4, #4 Increment the pointer to the list.
Subtract R2, R2, #1 Decrement the counter.
Branch_if_[R2]>0 LOOP Branch back if not finished.
Store R3, SUM Store the final sum.
```

**Figure 2.8** Use of indirect addressing in the program of Figure 2.6.

In many RISC-type processors, one general-purpose register is dedicated to holding a

constant value zero. Usually, this is register R0. Its contents cannot be changed by a program instruction. We will assume that R0 is used in this manner in our discussion of RISC-style processors. Then, the above Move instruction can be implemented as

Add R4, R0, #NUM1

It is often the case that Move is provided as a *pseudoinstruction* for the convenience of programmers, but it is actually implemented using the Add instruction.

The first three instructions in the loop in Figure 2.8 implement the unspecified instruction block starting at LOOP in Figure 2.6. The first time through the loop, the instruction

Load R5, (R4)

fetches the operand at location NUM1 and loads it into R5. The first Add instruction adds this number to the sum in register R3. The second Add instruction adds 4 to the contents of the pointer R4, so that it will contain the address value NUM2 when the Load instruction is executed in the second pass through the loop.

As another example of pointers, consider the C-language statement

A = \*B;

where B is a pointer variable and the '\*' symbol is the operator for indirect accesses. This statement causes the contents of the memory location pointed to by B to be loaded into memory location A. The statement may be compiled into

Load R2, B  
Load R3, (R2)  
Store R3, A

Indirect addressing through registers is used extensively. The program in Figure 2.8 shows the flexibility it provides.

## 2.4 Addressing Modes 45

### 2.4.3 Indexing and Arrays

The next addressing mode we discuss provides a different kind of flexibility for accessing operands. It is useful in dealing with lists and arrays.

*Index mode*—The effective address of the operand is generated by adding a constant value to the contents of a register.

For convenience, we will refer to the register used in this mode as the *index register*. Typically, this is just a general-purpose register. We indicate the Index mode symbolically as

$X(Ri)$

where X denotes a constant signed integer value contained in the instruction and Ri is the name of the register involved. The effective address of the operand is given by

$$EA = X + [Ri]$$

The contents of the register are not changed in the process of generating the effective address.

In an assembly-language program, whenever a constant such as the value X is needed, it may be given either as an explicit number or as a symbolic name representing a numerical value. The way in which a symbolic name is associated with a specific

numerical value will be discussed in Section 2.5. When the instruction is translated into machine code, the constant X is given as a part of the instruction and is restricted to fewer bits than the word length of the computer. Since X is a signed integer, it must be sign-extended (see Section 1.4) to the register length before being added to the contents of the register.

Figure 2.9 illustrates two ways of using the Index mode. In Figure 2.9a, the index register, R5, contains the address of a memory location, and the value X defines an *offset* (also called a *displacement*) from this address to the location where the operand is found. An alternative use is illustrated in Figure 2.9b. Here, the constant X corresponds to a memory address, and the contents of the index register define the offset to the operand. In either case, the effective address is the sum of two values; one is given explicitly in the instruction, and the other is held in a register.

To see the usefulness of indexed addressing, consider a simple example involving a list of test scores for students taking a given course. Assume that the list of scores, beginning at location LIST, is structured as shown in Figure 2.10. A four-word memory block comprises a record that stores the relevant information for each student. Each record consists of the student's identification number (ID), followed by the scores the student earned on three tests. There are  $n$  students in the class, and the value  $n$  is stored in location N immediately in front of the list. The addresses given in the figure for the student IDs and test scores assume that the memory is byte addressable and that the word length is 32 bits.

We should note that the list in Figure 2.10 represents a two-dimensional array having  $n$  rows and four columns. Each row contains the entries for one student, and the columns give the IDs and test scores.

Suppose that we wish to compute the sum of all scores obtained on each of the tests and store these three sums in memory locations SUM1, SUM2, and SUM3. A possible

## 46 CHAPTER 2 • Instruction Set Architecture

Load R2, 20(R5)

R5  
1000 1000

20 = offset

1020 Operand

(a) Offset is given as a constant

Load R2, 1000(R5)

1000 20  
R5

20 = offset

1020 Operand

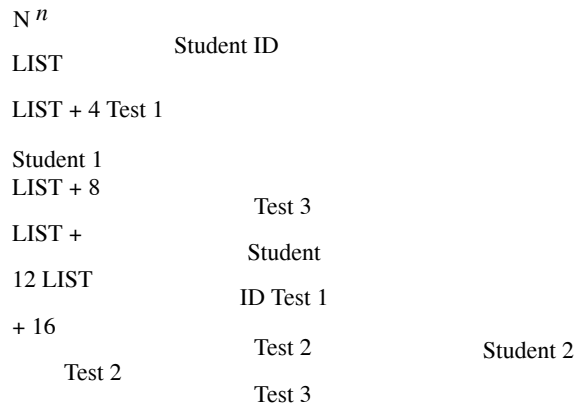
(b) Offset is in the index register

**Figure 2.9** Indexed addressing.

program for this task is given in Figure 2.11. In the body of the loop, the program uses the Index addressing mode in the manner depicted in Figure 2.9a to access each of the three scores in a student's record. Register R2 is used as the index register. Before the loop is entered, R2 is set to point to the ID location of the first student record which is the address LIST.

On the first pass through the loop, test scores of the first student are added to the running sums held in registers R3, R4, and R5, which are initially cleared to 0. These scores are accessed using the Index addressing modes 4(R2), 8(R2), and 12(R2). The index register R2 is then incremented by 16 to point to the ID location of the second student. Register R6, initialized to contain the value  $n$ , is decremented by 1 at the end of each pass through the loop. When the contents of R6 reach 0, all student records have been accessed, and

## 2.4 Addressing Modes 47



**Figure 2.10** A list of students' marks.

Move R2, #LIST Get the address LIST.



```

Clear R3
Clear R4
Clear R5
Load R6, N Load the value n.
LOOP: Load R7, 4(R2) Add the mark for next student's Add R3, R3, R7 Test
1 to the partial sum. Load R7, 8(R2) Add the mark for that student's Add R4,
R4, R7 Test 2 to the partial sum. Load R7, 12(R2) Add the mark for that
student's Add R5, R5, R7 Test 3 to the partial sum. Add R2, R2, #16
Increment the pointer.
Subtract R6, R6, #1 Decrement the counter.
Branch_if_[R6]>0 LOOP Branch back if not finished. Store
R3, SUM1 Store the total for Test 1.
Store R4, SUM2 Store the total for Test 2.
Store R5, SUM3 Store the total for Test 3.

```

**Figure 2.11** Indexed addressing used in accessing test scores in the list in Figure 2.10.

## 48 CHAPTER 2 Instruction Set Architecture

the loop terminates. Until then, the conditional branch instruction transfers control back to the start of the loop to process the next record. The last three instructions transfer the accumulated sums from registers R3, R4, and R5, into memory locations SUM1, SUM2, and SUM3, respectively.

It should be emphasized that the contents of the index register, R2, are not changed when it is used in the Index addressing mode to access the scores. The contents of R2 are changed only by the last Add instruction in the loop, to move from one student record to the next.

In general, the Index mode facilitates access to an operand whose location is defined relative to a reference point within the data structure in which the operand appears. In the example just given, the ID locations of successive student records are the reference points, and the test scores are the operands accessed by the Index addressing mode.

We have introduced the most basic form of indexed addressing that uses a register  $R_i$  and a constant offset  $X$ . Several variations of this basic form provide for efficient access to memory operands in practical programming situations (although they may not be included in some processors). For example, a second register  $R_j$  may be used to contain the offset  $X$ , in which case we can write the Index mode as

$$(R_i, R_j)$$

The effective address is the sum of the contents of registers  $R_i$  and  $R_j$ . The second register is usually called the *base* register. This form of indexed addressing provides more flexibility in accessing operands, because both components of the effective address can be changed.

Yet another version of the Index mode uses two registers plus a constant, which can be denoted as

$$X(R_i, R_j)$$

In this case, the effective address is the sum of the constant  $X$  and the contents of registers  $R_i$  and  $R_j$ . This added flexibility is useful in accessing multiple components inside each item in a record, where the beginning of an item is specified by the  $(R_i, R_j)$  part of the addressing mode.

Finally, we should note that in the basic Index mode

$$X(R_i)$$

if the contents of the register are equal to zero, then the effective address is just equal to the sign-extended value of X. This has the same effect as the Absolute mode. If register R0 always contains the value zero, then the Absolute mode is implemented simply as

X(R0)

## 2.5 Assembly Language

Machine instructions are represented by patterns of 0s and 1s. Such patterns are awkward to deal with when discussing or preparing programs. Therefore, we use symbolic names to represent the patterns. So far, we have used normal words, such as Load, Store, Add, and

### 2.5 Assembly Language 49

Branch, for the instruction operations to represent the corresponding binary code patterns. When writing programs for a specific computer, such words are normally replaced by acronyms called *mnemonics*, such as LD, ST, ADD, and BR. A shorthand notation is also useful when identifying registers, such as R3 for register 3. Finally, symbols such as LOC may be defined as needed to represent particular memory locations. A complete set of such symbolic names and rules for their use constitutes a programming language, generally referred to as an *assembly language*. The set of rules for using the mnemonics and for specification of complete instructions and programs is called the *syntax* of the language.

Programs written in an assembly language can be automatically translated into a sequence of machine instructions by a program called an *assembler*. The assembler program is one of a collection of utility programs that are a part of the system software of a computer. The assembler, like any other program, is stored as a sequence of machine instructions in the memory of the computer. A user program is usually entered into the computer through a keyboard and stored either in the memory or on a magnetic disk. At this point, the user program is simply a set of lines of alphanumeric characters. When the assembler program is executed, it reads the user program, analyzes it, and then generates the desired machine language program. The latter contains patterns of 0s and 1s specifying instructions that will be executed by the computer. The user program in its original alphanumeric text form is called a *source program*, and the assembled machine-language program is called an *object program*. We will discuss how the assembler program works in Section 2.5.2 and in Chapter 4. First, we present a few aspects of assembly language itself.

The assembly language for a given computer may or may not be case sensitive, that is, it may or may not distinguish between capital and lower-case letters. In this section, we use capital letters to denote all names and labels in our examples to improve the readability of the text. For example, we write a Store instruction as

ST R2, SUM

The mnemonic ST represents the binary pattern, or *operation (OP) code*, for the operation performed by the instruction. The assembler translates this mnemonic into the binary OP code that the computer recognizes.

The OP-code mnemonic is followed by at least one blank space or tab character. Then the information that specifies the operands is given. In the Store instruction above, the source operand is in register R2. This information is followed by the specification of the destination operand, separated from the source operand by a comma. The destination operand is in the memory location that has its binary address represented by the name

SUM.

Since there are several possible addressing modes for specifying operand locations, an assembly-language instruction must indicate which mode is being used. For example, a numerical value or a name used by itself, such as SUM in the preceding instruction, may be used to denote the Absolute mode. The number sign usually denotes an immediate operand. Thus, the instruction

ADD R2, R3, #5

adds the number 5 to the contents of register R3 and puts the result into register R2. The number sign is not the only way to denote the Immediate addressing mode. In some assembly languages, the Immediate addressing mode is indicated in the OP-code mnemonic.

## 50 CHAPTER 2 \* Instruction Set Architecture

For example, the previous Add instruction may be written as

ADDI R2, R3, 5

The suffix I in the mnemonic ADDI states that the second source operand is given in the Immediate addressing mode.

Indirect addressing is usually specified by putting parentheses around the name or symbol denoting the pointer to the operand. For example, if register R2 contains the address of a number in the memory, then this number can be loaded into register R3 using the instruction

LD R3, (R2)

### 2.5.1 Assembler Directives

In addition to providing a mechanism for representing instructions in a program, assembly language allows the programmer to specify other information needed to translate the source program into the object program. We have already mentioned that we need to assign numerical values to any names used in a program. Suppose that the name TWENTY is used to represent the value 20. This fact may be conveyed to the assembler program through an *equate* statement such as

TWENTY EQU 20

This statement does not denote an instruction that will be executed when the object program is run; in fact, it will not even appear in the object program. It simply informs the assembler that the name TWENTY should be replaced by the value 20 wherever it appears in the program. Such statements, called *assembler directives* (or *commands*), are used by the assembler while it translates a source program into an object program.

To illustrate the use of assembly language further, let us reconsider the program in Figure 2.8. In order to run this program on a computer, it is necessary to write its source code in the required assembly language, specifying all of the information needed to generate the corresponding object program. Suppose that each instruction and each data item occupies one word of memory. Also assume that the memory is byte-addressable and that the word length is 32 bits. Suppose also that the object program is to be loaded in the main memory as shown in Figure 2.12. The figure shows the memory addresses where the machine instructions and the required data items are to be found after the program is loaded for execution. If the assembler is to produce an object program according to this arrangement, it has to know

- How to interpret the names
- Where to place the instructions in the memory
- Where to place the data operands in the memory

To provide this information, the source program may be written as shown in Figure 2.13. The program begins with the assembler directive, **ORIGIN**, which tells the assembler program where in the memory to place the instructions that follow. It specifies that the instructions

Load R2, N  
**2.5 Assembly Language 51**

```

100
Clear
R3

104
R5, (R4)    R4, #4 R2, R2,
LOOP        116 120 124  #1
108 112    Add      128
Move R4,    Add      Branch_if_[R2
              ]>0
#NUM1 Load Subtract LOOP
              R3, R3, R5 R4,
              R3, SUM

200 204
SUM N      208 212
NUM1
NUM2

NUMn
132        804
Store

```

**Figure 2.12** Memory arrangement for the program in Figure 2.8.

of the object program are to be loaded in the memory starting at address 100. It is followed by the source program instructions written with the appropriate mnemonics and syntax. Note that we use the statement

BGT R2, R0, LOOP

to represent an instruction that performs the operation

Branch\_if\_[R2]>0 LOOP

The second ORIGIN directive tells the assembler program where in the memory to place the data block that follows. In this case, the location specified has the address 200. This is intended to be the location in which the final sum will be stored. A 4-byte space for the sum is reserved by means of the assembler directive RESERVE. The next word, at address 204, has to contain the value 150 which is the number of entries in the list.

## 52 CHAPTER 2 Instruction Set Architecture

Memory Addressing  
address or data  
label Operation information

Assembler directive ORIGIN 100

Statements that LD R2, N  
generate CLR R3  
machine MOV R4, #NUM1  
instructions LOOP: LD R5, (R4)  
ADD R3, R3, R5  
ADD R4, R4, #4  
SUB R2, R2, #1  
BGT R2, R0, LOOP  
ST R3, SUM  
next instruction

Assembler directives ORIGIN 200  
SUM: RESERVE 4  
N: DATAWORD 150  
NUM1: RESERVE 600  
END

**Figure 2.13** Assembly language representation for the program in Figure 2.12.

The DATAWORD directive is used to inform the assembler of this requirement. The next RESERVE directive declares that a memory block of 600 bytes is to be reserved for data. This directive does not cause any data to be loaded in these locations. Data may be loaded in the memory using an input procedure, as we will explain in Chapter 3. The last statement in the source program is the assembler directive END, which tells the assembler that this is the end of the source program text.

We previously described how the EQU directive can be used to associate a specific value, which may be an address, with a particular name. A different way of associating addresses with names or labels is illustrated in Figure 2.13. Any statement that results in instructions or data being placed in a memory location may be given a memory address

label. The assembler automatically assigns the address of that location to the label. For example, in the data block that follows the second ORIGIN directive, we used the labels SUM, N, and NUM1. Because the first RESERVE statement after the ORIGIN directive is given the label SUM, the name SUM is assigned the value 200. Whenever SUM is encountered in the program, it will be replaced with this value. Using SUM as a label in

**2.5 Assembly Language 53**

this manner is equivalent to using the assembler directive

SUM EQU 200

Similarly, the labels N and NUM1 are assigned the values 204 and 208, respectively, because they represent the addresses of the two word locations immediately following the word location with address 200.

Most assembly languages require statements in a source program to be written in the form

Label: Operation Operand(s) Comment

These four *fields* are separated by an appropriate delimiter, perhaps one or more blank or tab characters. The Label is an optional name associated with the memory address where the machine-language instruction produced from the statement will be loaded. Labels may also be associated with addresses of data items. In Figure 2.13 there are four labels: LOOP, SUM, N, and NUM1.

The Operation field contains an assembler directive or the OP-code mnemonic of the desired instruction. The Operand field contains addressing information for accessing the operands. The Comment field is ignored by the assembler program. It is used for documentation purposes to make the program easier to understand.

We have introduced only the very basic characteristics of assembly languages. These languages differ in detail and complexity from one computer to another.

## 2.5.2 Assembly and Execution of Programs

A source program written in an assembly language must be assembled into a machine language object program before it can be executed. This is done by the assembler program, which replaces all symbols denoting operations and addressing modes with the binary codes used in machine instructions, and replaces all names and labels with their actual values.

The assembler assigns addresses to instructions and data blocks, starting at the addresses given in the ORIGIN assembler directives. It also inserts constants that may be given in DATAWORD commands, and it reserves memory space as requested by RESERVE commands.

A key part of the assembly process is determining the values that replace the names. In some cases, where the value of a name is specified by an EQU directive, this is a straightforward task. In other cases, where a name is defined in the Label field of a given instruction, the value represented by the name is determined by the location of this instruction in the assembled object program. Hence, the assembler must keep track of addresses as it generates the machine code for successive instructions. For example, the names LOOP and SUM in the program of Figure 2.13 will be assigned the values 112 and 200, respectively.

In some cases, the assembler does not directly replace a name representing an address with the actual value of this address. For example, in a branch instruction, the name that specifies the location to which a branch is to be made (the branch target) is not replaced by the actual address. A branch instruction is usually implemented in machine

code by specifying the branch target as the distance (in bytes) from the present address in the Program Counter

## 54 CHAPTER 2 Instruction Set Architecture

to the target instruction. The assembler computes this *branch offset*, which can be positive or negative, and puts it into the machine instruction. We will show how branch instructions may be implemented in Section 2.13.

The assembler stores the object program on the secondary storage device available in the computer, usually a magnetic disk. The object program must be loaded into the main memory before it is executed. For this to happen, another utility program called a *loader* must already be in the memory. Executing the loader performs a sequence of input operations needed to transfer the machine-language program from the disk into a specified place in the memory. The loader must know the length of the program and the address in the memory where it will be stored. The assembler usually places this information in a header preceding the object code. Having loaded the object code, the loader starts execution of the object program by branching to the first instruction to be executed, which may be identified by an address label such as `START`. The assembler places that address in the header of the object code for the loader to use at execution time.

When the object program begins executing, it proceeds to completion unless there are logical errors in the program. The user must be able to find errors easily. The assembler can only detect and report syntax errors. To help the user find other programming errors, the system software usually includes a *debugger* program. This program enables the user to stop execution of the object program at some points of interest and to examine the contents of various processor registers and memory locations.

In this section, we introduced some important issues in assembly and execution of programs. Chapter 4 provides a more detailed discussion of these issues.

### 2.5.3 Number Notation

When dealing with numerical values, it is often convenient to use the familiar decimal notation. Of course, these values are stored in the computer as binary numbers. In some situations, it is more convenient to specify the binary patterns directly. Most assemblers allow numerical values to be specified in different ways, using conventions that are defined by the assembly-language syntax. Consider, for example, the number 93, which is represented by the 8-bit binary number 01011101. If this value is to be used as an immediate operand, it can be given as a decimal number, as in the instruction

```
ADDI R2, R3, 93
```

or as a binary number identified by an assembler-specific prefix symbol such as a percent sign, as in

```
ADDI R2, R3, %01011101
```

Binary numbers can be written more compactly as *hexadecimal*, or *hex*, numbers, in which four bits are represented by a single hex digit. The first ten patterns 0000, 0001,..., 1001, referred to as *binary-coded decimal* (BCD), are represented by the digits 0, 1,..., 9. The remaining six 4-bit patterns, 1010, 1011,..., 1111, are represented by the letters A, B,..., F. In hexadecimal representation, the decimal value 93 becomes 5D. In assembly language, a hex representation is often identified by the prefix 0x (as in the C language) or

by a dollar sign prefix. Thus, we would write

ADDI R2, R3, 0x5D

## 2.6 Stacks

Data operated on by a program can be organized in a variety of ways. We have already encountered data structured as lists. Now, we consider an important data structure known as a *stack*. A *stack* is a list of data elements, usually words, with the accessing restriction that elements can be added or removed at one end of the list only. This end is called the top of the stack, and the other end is called the bottom. The structure is sometimes referred to as a *pushdown* stack. Imagine a pile of trays in a cafeteria; customers pick up new trays from the top of the pile, and clean trays are added to the pile by placing them onto the top of the pile. Another descriptive phrase, *last-in-first-out* (LIFO) stack, is also used to describe this type of storage mechanism; the last data item placed on the stack is the first one removed when retrieval begins. The terms *push* and *pop* are used to describe placing a new item on the stack and removing the top item from the stack, respectively.

In modern computers, a stack is implemented by using a portion of the main memory for this purpose. One processor register, called the *stack pointer* (SP), is used to point to a particular stack structure called the *processor stack*, whose use will be explained shortly.

Data can be stored in a stack with successive elements occupying successive memory locations. Assume that the first element is placed in location BOTTOM, and when new elements are pushed onto the stack, they are placed in successively lower address locations. We use a stack that grows in the direction of decreasing memory addresses in our discussion, because this is a common practice.

Figure 2.14 shows an example of a stack of word data items. The stack contains numerical values, with 43 at the bottom and -28 at the top. The stack pointer, SP, is used to keep track of the address of the element of the stack that is at the top at any given time. If we assume a byte-addressable memory with a 32-bit word length, the push operation can be implemented as

Subtract SP, SP, #4  
Store R<sub>j</sub>, (SP)

where the Subtract instruction subtracts 4 from the contents of SP and places the result in SP. Assuming that the new item to be pushed on the stack is in processor register R<sub>j</sub>, the Store instruction will place this value on the stack. These two instructions copy the word from R<sub>j</sub> onto the top of the stack, decrementing the stack pointer by 4 before the store (push) operation. The pop operation can be implemented as

Load R<sub>j</sub>, (SP)  
Add SP, SP, #4

These two instructions load (pop) the top value from the stack into register R<sub>j</sub> and then increment the stack pointer by 4 so that it points to the new top element. Figure 2.15 shows the effect of each of these operations on the stack in Figure 2.14.



Stack

BOTTOM  $2^k - 1$

top element

Bottom  
element

-28 17

**Figure 2.14** A stack of words in the memory.

## 2.7 Subroutines

In a given program, it is often necessary to perform a particular task many times on different data values. It is prudent to implement this task as a block of instructions that is executed each time the task has to be performed. Such a block of instructions is usually called a *subroutine*. For example, a subroutine may evaluate a mathematical function, or it may sort a list of values into increasing or decreasing order.

It is possible to reproduce the block of instructions that constitute a subroutine at every place where it is needed in the program. However, to save space, only one copy of this block is placed in the memory, and any program that requires the use of the subroutine simply branches to its starting location. When a program branches to a subroutine we say that it is *calling* the subroutine. The instruction that performs this branch operation is named a Call instruction.

After a subroutine has been executed, the calling program must resume execution, continuing immediately after the instruction that called the subroutine. The subroutine is said to *return* to the program that called it, and it does so by executing a Return instruction. Since the subroutine may be called from different places in a calling program, provision must be made for returning to the appropriate location. The location where the calling