A*ssembly Language*

FOR x86 PROCESSORS

Seventh Edition

*Kip Irvine*

**ASCII CONTROL CHARACTERS** The following list shows the ASCII codes generated when a control key combination is pressed. The mnemonics and descriptions refer to ASCII functions used for screen and printer formatting and data communications.

**ASCII Code\* Ctrl- Mnemonic Description**

**ALT-KEY COMBINATIONS** The following hexadecimal scan codes are produced by holding down the ALT key and pressing each character:

**ASCII Code\* Ctrl- Mnemonic Description** 00 NUL Null character 10 Ctrl-P DLE Data link escape

01 Ctrl-A SOH Start of header 11 Ctrl-Q DC1 Device control 1 02 Ctrl-B STX Start of text 12 Ctrl-R DC2 Device control 2 03 Ctrl-C ETX End of text 13 Ctrl-S DC3 Device control 3 04 Ctrl-D EOT End of transmission 14 Ctrl-T DC4 Device control 4 05 Ctrl-E ENQ Enquiry 15 Ctrl-U NAK Negative acknowledge 06 Ctrl-F ACK Acknowledge 16 Ctrl-V SYN Synchronous idle

07 Ctrl-G BEL Bell 17 Ctrl-W ETB End transmission block 08 Ctrl-H BS Backspace 18 Ctrl-X CAN Cancel 09 Ctrl-I HT Horizontal tab 19 Ctrl-Y EM End of medium 0A Ctrl-J LF Line feed 1A Ctrl-Z SUB Substitute 0B Ctrl-K VT Vertical tab 1B Ctrl-I ESC Escape 0C Ctrl-L FF Form feed 1C Ctrl-\ FS File separator

0D Ctrl-M CR Carriage return 1D Ctrl-] GS Group separator 0E Ctrl-N SO Shift out 1E Ctrl- ^ RS Record separator 0F Ctrl-O SI Shift in 1F Ctrl-† US Unit separator

\* ASCII codes are in hexadecimal. † ASCII code 1Fh is Ctrl-Hyphen (-).

**Key Scan Code Key Scan Code Key Scan Code**

1 78 A 1E N 31 2 79 B 30 O 18 3 7A C 2E P 19 4 7B D 20 Q 10 5 7C E 12 R 13

6 7D F 21 S 1F 7 7E G 22 T 14 8 7F H 23 U 16 9 80 I 17 V 2F 0 81 J 24 W 11 82 K 25 X 2D

83 L 26 Y 15 M 32 Z 2C

**KEYBOARD SCAN CODES** The following keyboard scan codes may be retrieved either by calling INT 16h or by calling INT 21h for keyboard input a second time (the first keyboard read returns 0). All codes are in hexadecimal:

**FUNCTION KEYS**

**Key Normal**

**With Shift**

**With**

**Ctrl With Alt**

F1 3B 54 5E 68

F2 3C 55 5F 69

F3 3D 56 60 6A

F4 3E 57 61 6B

F5 3F 58 62 6C

F6 40 59 63 6D

F7 41 5A 64 6E

F8 42 5B 65 6F

F9 43 5C 66 70

F10 44 5D 67 71

F11 85 87 89 8B

F12 86 88 8A 8C

**Key Alone**

**With Ctrl Key**

Home 47 77

End 4F 75

PgUp 49 84

PgDn 51 76

PrtSc 37 72

Left arrow 4B 73

Rt arrow 4D 74

Up arrow 48 8D

Dn arrow 50 91

Ins 52 92

Del 53 93

Back tab 0F 94

Gray + 4E 90

Gray − 4A 8E

Assembly Language for x86 Processors

**Seventh Edition**

KIP R. IRVINE

**Florida International University School of Computing and Information Sciences**

Boston Columbus Indianapolis New York San Francisco Upper Saddle River Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto Delhi Mexico City São Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

Vice President and Editorial Director, ECS: *Marcia Horton* Executive Editor: *Tracy Johnson* Executive Marketing Manager: *Tim Galligan* Marketing Assistant: *Jon Bryant* Program Management Team Lead: *Scott Disanno* Program Manager: *Clare Romeo* Project Manager: *Greg Dulles* Senior Operations Specialist: *Nick Sklitsis* Operations Specialist: *Linda Sager* Permissions Project Manager: *Karen Sanatar* Full-Service Project Management: *Pavithra Jayapaul, Jouve* Printer/Binder: *Courier/Westford* Typeface: *Times*

IA-32, Pentium, i486, Intel64, Celeron, and Intel 386 are trademarks of Intel Corporation. Athlon, Phenom, and Opteron are trademarks of Advanced Micro Devices. TASM and Turbo Debugger are trademarks of Borland International. Microsoft Assembler (MASM), Windows Vista, Windows 7, Windows NT, Windows Me, Windows 95, Windows 98, Windows 2000, Windows XP, MS-Windows, PowerPoint, Win32, DEBUG, WinDbg, MS-DOS, Visual Studio, Visual C++, and CodeView are registered trademarks of Microsoft Corporation. Autocad is a trademark of Autodesk. Java is a trademark of Sun Microsystems. PartitionMagic is a trademark of Symantec. All other trademarks or product names are the property of their respective owners.

**Copyright © 2015, 2011, 2007, 2003 by Pearson Education, Inc., Upper Saddle River, New Jersey 07458.** All rights reserved. Manufactured in the United States of America. This publication is protected by Copyright and permissions should be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission(s) to use materials from this work, please submit a written request to Pearson Higher Education, Permissions Department, 1 Lake Street, Upper Saddle River, NJ 07458. Previously published as *Assembly Language for Intel-Based Computers.* The author and publisher of this book have used their best efforts in preparing this book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The author and pub- lisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The author and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

**Library of Congress Cataloging-in-Publication Data** Irvine, Kip R., 1951-

Assembly language for x86 processors / Kip R. Irvine, Florida International University, School of Computing and Information Sciences. — Seventh Edition.

pages cm ISBN-13: 978-0-13-376940-1 ISBN-10: 0-13-376940-2 1. IBM microcomputers–Programming. 2. X86 assembly language (Computer program language) I. Title.

QA76.8.I77 2014 005.265—dc23 2013046432

10 9 8 7 6 5 4 3 2 1

ISBN-13: 978-0-13-376940-1 ISBN-10: 0-13-376940-2

***To Jack and Candy Irvine***

*This page intentionally left blank*

Contents

**Preface xxiii**

**1 Basic Concepts 1**

**1.1 Welcome to Assembly Language 1**

1.1.1 Questions You Might Ask 3 1.1.2 Assembly Language Applications 6 1.1.3 Section Review 6

**1.2 Virtual Machine Concept 7**

1.2.1 Section Review 9

**1.3 Data Representation 9**

1.3.1 Binary Integers 10 1.3.2 Binary Addition 12 1.3.3 Integer Storage Sizes 13 1.3.4 Hexadecimal Integers 13 1.3.5 Hexadecimal Addition 15 1.3.6 Signed Binary Integers 16 1.3.7 Binary Subtraction 18 1.3.8 Character Storage 19 1.3.9 Section Review 21

**1.4 Boolean Expressions 22**

1.4.1 Truth Tables for Boolean Functions 24 1.4.2 Section Review 26

**1.5 Chapter Summary 26**

**1.6 Key Terms 27**

**1.7 Review Questions and Exercises 28**

1.7.1 Short Answer 28 1.7.2 Algorithm Workbench 30

**2 x86 Processor Architecture 32**

**2.1 General Concepts 33**

2.1.1 Basic Microcomputer Design 33 2.1.2 Instruction Execution Cycle 34

v

vi Contents

2.1.3 Reading from Memory 36 2.1.4 Loading and Executing a Program 36 2.1.5 Section Review 37

**2.2 32-Bit x86 Processors 37**

2.2.1 Modes of Operation 37 2.2.2 Basic Execution Environment 38 2.2.3 x86 Memory Management 41 2.2.4 Section Review 42

**2.3 64-Bit x86-64 Processors 42**

2.3.1 64-Bit Operation Modes 43 2.3.2 Basic 64-Bit Execution Environment 43

**2.4 Components of a Typical x86 Computer 44**

2.4.1 Motherboard 44 2.4.2 Memory 46 2.4.3 Section Review 46

**2.5 Input–Output System 47**

2.5.1 Levels of I/O Access 47 2.5.2 Section Review 49

**2.6 Chapter Summary 50**

**2.7 Key Terms 51**

**2.8 Review Questions 52**

**3 Assembly Language Fundamentals 53**

**3.1 Basic Language Elements 54**

3.1.1 First Assembly Language Program 54 3.1.2 Integer Literals 55 3.1.3 Constant Integer Expressions 56 3.1.4 Real Number Literals 57 3.1.5 Character Literals 57 3.1.6 String Literals 58 3.1.7 Reserved Words 58 3.1.8 Identifiers 58 3.1.9 Directives 59 3.1.10 Instructions 60 3.1.11 Section Review 63

**3.2 Example: Adding and Subtracting Integers 63**

3.2.1 The *AddTwo* Program 63 3.2.2 Running and Debugging the AddTwo Program 65 3.2.3 Program Template 70 3.2.4 Section Review 70

Contents vii

**3.3 Assembling, Linking, and Running Programs 71**

3.3.1 The Assemble-Link-Execute Cycle 71 3.3.2 Listing File 71 3.3.3 Section Review 73

**3.4 Defining Data 74**

3.4.1 Intrinsic Data Types 74 3.4.2 Data Definition Statement 74 3.4.3 Adding a Variable to the AddTwo Program 75 3.4.4 Defining BYTE and SBYTE Data 76 3.4.5 Defining WORD and SWORD Data 78 3.4.6 Defining DWORD and SDWORD Data 79 3.4.7 Defining QWORD Data 79 3.4.8 Defining Packed BCD (TBYTE) Data 80 3.4.9 Defining Floating-Point Types 81 3.4.10 A Program That Adds Variables 81 3.4.11 Little-Endian Order 82 3.4.12 Declaring Uninitialized Data 83 3.4.13 Section Review 83

**3.5 Symbolic Constants 84** 3.5.1 Equal-Sign Directive 84 3.5.2 Calculating the Sizes of Arrays and Strings 85 3.5.3 EQU Directive 86 3.5.4 TEXTEQU Directive 87 3.5.5 Section Review 88

**3.6 64-Bit Programming 88**

**3.7 Chapter Summary 90**

**3.8 Key Terms 91**

3.8.1 Terms 91 3.8.2 Instructions, Operators, and Directives 92

**3.9 Review Questions and Exercises 92**

3.9.1 Short Answer 92 3.9.2 Algorithm Workbench 93

**3.10 Programming Exercises 94**

**4 Data Transfers, Addressing, and Arithmetic 95**

**4.1 Data Transfer Instructions 96**

4.1.1 Introduction 96 4.1.2 Operand Types 96 4.1.3 Direct Memory Operands 96

viii Contents

4.1.4 MOV Instruction 98 4.1.5 Zero/Sign Extension of Integers 99 4.1.6 LAHF and SAHF Instructions 101 4.1.7 XCHG Instruction 102 4.1.8 Direct-Offset Operands 102 4.1.9 Example Program (Moves) 103 4.1.10 Section Review 104

**4.2 Addition and Subtraction 105**

4.2.1 INC and DEC Instructions 105 4.2.2 ADD Instruction 105 4.2.3 SUB Instruction 106 4.2.4 NEG Instruction 106 4.2.5 Implementing Arithmetic Expressions 106 4.2.6 Flags Affected by Addition and Subtraction 107 4.2.7 Example Program (*AddSubTest*) 111 4.2.8 Section Review 112

**4.3 Data-Related Operators and Directives 112**

4.3.1 OFFSET Operator 112 4.3.2 ALIGN Directive 113 4.3.3 PTR Operator 114 4.3.4 TYPE Operator 115 4.3.5 LENGTHOF Operator 116 4.3.6 SIZEOF Operator 116 4.3.7 LABEL Directive 116 4.3.8 Section Review 117

**4.4 Indirect Addressing 117**

4.4.1 Indirect Operands 117 4.4.2 Arrays 118 4.4.3 Indexed Operands 119 4.4.4 Pointers 121 4.4.5 Section Review 122

**4.5 JMP and LOOP Instructions 123**

4.5.1 JMP Instruction 123 4.5.2 LOOP Instruction 124 4.5.3 Displaying an Array in the Visual Studio Debugger 125 4.5.4 Summing an Integer Array 126 4.5.5 Copying a String 127 4.5.6 Section Review 128

**4.6 64-Bit Programming 128**

4.6.1 MOV Instruction 128 4.6.2 64-Bit Version of SumArray 130 4.6.3 Addition and Subtraction 130 4.6.4 Section Review 131

Contents ix

**4.7 Chapter Summary 132**

**4.8 Key Terms 133**

4.8.1 Terms 133 4.8.2 Instructions, Operators, and Directives 133

**4.9 Review Questions and Exercises 134**

4.9.1 Short Answer 134 4.9.2 Algorithm Workbench 136

**4.10 Programming Exercises 137**

**5 Procedures 139**

**5.1 Stack Operations 140**

5.1.1 Runtime Stack (32-bit mode) 140 5.1.2 PUSH and POP Instructions 142 5.1.3 Section Review 145

**5.2 Defining and Using Procedures 145**

5.2.1 PROC Directive 145 5.2.2 CALL and RET Instructions 147 5.2.3 Nested Procedure Calls 148 5.2.4 Passing Register Arguments to Procedures 150 5.2.5 Example: Summing an Integer Array 150 5.2.6 Saving and Restoring Registers 152 5.2.7 Section Review 153

**5.3 Linking to an External Library 153**

5.3.1 Background Information 154 5.3.2 Section Review 155

**5.4 The Irvine32 Library 155**

5.4.1 Motivation for Creating the Library 155 5.4.2 Overview 157 5.4.3 Individual Procedure Descriptions 158 5.4.4 Library Test Programs 170 5.4.5 Section Review 178

**5.5 64-Bit Assembly Programming 178**

5.5.1 The *Irvine64* Library 178 5.5.2 Calling 64-Bit Subroutines 179 5.5.3 The x64 Calling Convention 179 5.5.4 Sample Program that Calls a Procedure 180

**5.6 Chapter Summary 182**

**5.7 Key Terms 183**

5.7.1 Terms 183 5.7.2 Instructions, Operators, and Directives 183

x Contents

**5.8 Review Questions and Exercises 183**

5.8.1 Short Answer 183 5.8.2 Algorithm Workbench 186

**5.9 Programming Exercises 187**

**6 Conditional Processing 189**

**6.1 Conditional Branching 190**

**6.2 Boolean and Comparison Instructions 190**

6.2.1 The CPU Status Flags 191 6.2.2 AND Instruction 191 6.2.3 OR Instruction 192 6.2.4 Bit-Mapped Sets 194 6.2.5 XOR Instruction 195 6.2.6 NOT Instruction 196 6.2.7 TEST Instruction 196 6.2.8 CMP Instruction 197 6.2.9 Setting and Clearing Individual CPU Flags 198 6.2.10 Boolean Instructions in 64-Bit Mode 199 6.2.11 Section Review 199

**6.3 Conditional Jumps 199**

6.3.1 Conditional Structures 199 6.3.2 J*cond* Instruction 200 6.3.3 Types of Conditional Jump Instructions 201 6.3.4 Conditional Jump Applications 204 6.3.5 Section Review 208

**6.4 Conditional Loop Instructions 209**

6.4.1 LOOPZ and LOOPE Instructions 209 6.4.2 LOOPNZ and LOOPNE Instructions 209 6.4.3 Section Review 210

**6.5 Conditional Structures 210**

6.5.1 Block-Structured IF Statements 210 6.5.2 Compound Expressions 213 6.5.3 WHILE Loops 214 6.5.4 Table-Driven Selection 216 6.5.5 Section Review 219

**6.6 Application: Finite-State Machines 219**

6.6.1 Validating an Input String 219 6.6.2 Validating a Signed Integer 220 6.6.3 Section Review 224

**6.7 Conditional Control Flow Directives 225**

6.7.1 Creating IF Statements 226 6.7.2 Signed and Unsigned Comparisons 227 6.7.3 Compound Expressions 228 6.7.4 Creating Loops with .REPEAT and .WHILE 231

Contents xi

**6.8 Chapter Summary 232**

**6.9 Key Terms 233**

6.9.1 Terms 233 6.9.2 Instructions, Operators, and Directives 234

**6.10 Review Questions and Exercises 234**

6.10.1 Short Answer 234 6.10.2 Algorithm Workbench 236

**6.11 Programming Exercises 237**

6.11.1 Suggestions for Testing Your Code 237 6.11.2 Exercise Descriptions 238

**7 Integer Arithmetic 242**

**7.1 Shift and Rotate Instructions 243**

7.1.1 Logical Shifts and Arithmetic Shifts 243 7.1.2 SHL Instruction 244 7.1.3 SHR Instruction 245 7.1.4 SAL and SAR Instructions 246 7.1.5 ROL Instruction 247 7.1.6 ROR Instruction 247 7.1.7 RCL and RCR Instructions 248 7.1.8 Signed Overflow 249 7.1.9 SHLD/SHRD Instructions 249 7.1.10 Section Review 251

**7.2 Shift and Rotate Applications 251**

7.2.1 Shifting Multiple Doublewords 252 7.2.2 Binary Multiplication 253 7.2.3 Displaying Binary Bits 254 7.2.4 Extracting File Date Fields 254 7.2.5 Section Review 255

**7.3 Multiplication and Division Instructions 255**

7.3.1 MUL Instruction 255 7.3.2 IMUL Instruction 257 7.3.3 Measuring Program Execution Times 260 7.3.4 DIV Instruction 262 7.3.5 Signed Integer Division 264 7.3.6 Implementing Arithmetic Expressions 267 7.3.7 Section Review 269

**7.4 Extended Addition and Subtraction 269**

7.4.1 ADC Instruction 269 7.4.2 Extended Addition Example 270 7.4.3 SBB Instruction 272 7.4.4 Section Review 272

xii Contents

**7.5 ASCII and Unpacked Decimal Arithmetic 273**

7.5.1 AAA Instruction 274 7.5.2 AAS Instruction 276 7.5.3 AAM Instruction 276 7.5.4 AAD Instruction 276 7.5.5 Section Review 277

**7.6 Packed Decimal Arithmetic 277**

7.6.1 DAA Instruction 277 7.6.2 DAS Instruction 279 7.6.3 Section Review 279

**7.7 Chapter Summary 279**

**7.8 Key Terms 280**

7.8.1 Terms 280 7.8.2 Instructions, Operators, and Directives 280

**7.9 Review Questions and Exercises 281**

7.9.1 Short Answer 281 7.9.2 Algorithm Workbench 282

**7.10 Programming Exercises 284**

**8 Advanced Procedures 286**

**8.1 Introduction 287**

**8.2 Stack Frames 287**

8.2.1 Stack Parameters 288 8.2.2 Disadvantages of Register Parameters 288 8.2.3 Accessing Stack Parameters 290 8.2.4 32-Bit Calling Conventions 293 8.2.5 Local Variables 295 8.2.6 Reference Parameters 297 8.2.7 LEA Instruction 298 8.2.8 ENTER and LEAVE Instructions 298 8.2.9 LOCAL Directive 300 8.2.10 The Microsoft x64 Calling Convention 301 8.2.11 Section Review 302

**8.3 Recursion 302**

8.3.1 Recursively Calculating a Sum 303 8.3.2 Calculating a Factorial 304 8.3.3 Section Review 311

**8.4 INVOKE, ADDR, PROC, and PROTO 311**

8.4.1 INVOKE Directive 311 8.4.2 ADDR Operator 312 8.4.3 PROC Directive 313 8.4.4 PROTO Directive 316

Contents xiii

8.4.5 Parameter Classifications 319 8.4.6 Example: Exchanging Two Integers 320 8.4.7 Debugging Tips 321 8.4.8 WriteStackFrame Procedure 322 8.4.9 Section Review 323

**8.5 Creating Multimodule Programs 323**

8.5.1 Hiding and Exporting Procedure Names 323 8.5.2 Calling External Procedures 324 8.5.3 Using Variables and Symbols across Module Boundaries 325 8.5.4 Example: ArraySum Program 326 8.5.5 Creating the Modules Using Extern 326 8.5.6 Creating the Modules Using INVOKE and PROTO 330 8.5.7 Section Review 333

**8.6 Advanced Use of Parameters (Optional Topic) 333**

8.6.1 Stack Affected by the USES Operator 333 8.6.2 Passing 8-Bit and 16-Bit Arguments on the Stack 335 8.6.3 Passing 64-Bit Arguments 336 8.6.4 Non-Doubleword Local Variables 337

**8.7 Java Bytecodes (Optional Topic) 339**

8.7.1 Java Virtual Machine 339 8.7.2 Instruction Set 340 8.7.3 Java Disassembly Examples 341 8.7.4 Example: Conditional Branch 344

**8.8 Chapter Summary 346**

**8.9 Key Terms 347**

8.9.1 Terms 347 8.9.2 Instructions, Operators, and Directives 348

**8.10 Review Questions and Exercises 348**

8.10.1 Short Answer 348 8.10.2 Algorithm Workbench 348

**8.11 Programming Exercises 349**

**9 Strings and Arrays 352**

**9.1 Introduction 352**

**9.2 String Primitive Instructions 353**

9.2.1 MOVSB, MOVSW, and MOVSD 354 9.2.2 CMPSB, CMPSW, and CMPSD 355 9.2.3 SCASB, SCASW, and SCASD 356 9.2.4 STOSB, STOSW, and STOSD 356 9.2.5 LODSB, LODSW, and LODSD 356 9.2.6 Section Review 357

xiv Contents

**9.3 Selected String Procedures 357** 9.3.1 Str\_compare Procedure 358 9.3.2 Str\_length Procedure 359 9.3.3 Str\_copy Procedure 359 9.3.4 Str\_trim Procedure 360 9.3.5 Str\_ucase Procedure 363 9.3.6 *String Library Demo* Program 364 9.3.7 String Procedures in the Irvine64 Library 365 9.3.8 Section Review 368 **9.4 Two-Dimensional Arrays 368**

9.4.1 Ordering of Rows and Columns 368 9.4.2 Base-Index Operands 369 9.4.3 Base-Index-Displacement Operands 371 9.4.4 Base-Index Operands in 64-Bit Mode 372 9.4.5 Section Review 373 **9.5 Searching and Sorting Integer Arrays 373**

9.5.1 Bubble Sort 373 9.5.2 Binary Search 375 9.5.3 Section Review 382 **9.6 Java Bytecodes: String Processing (Optional Topic) 382**

**9.7 Chapter Summary 383**

**9.8 Key Terms and Instructions 384**

**9.9 Review Questions and Exercises 384**

9.9.1 Short Answer 384 9.9.2 Algorithm Workbench 385 **9.10 Programming Exercises 386**

**10 Structures and Macros 390**

**10.1 Structures 390**

10.1.1 Defining Structures 391 10.1.2 Declaring Structure Variables 393 10.1.3 Referencing Structure Variables 394 10.1.4 Example: Displaying the System Time 397 10.1.5 Structures Containing Structures 399 10.1.6 Example: Drunkard’s Walk 399 10.1.7 Declaring and Using Unions 403 10.1.8 Section Review 405 **10.2 Macros 405**

10.2.1 Overview 405 10.2.2 Defining Macros 406 10.2.3 Invoking Macros 407

Contents xv

10.2.4 Additional Macro Features 408 10.2.5 Using the Book’s Macro Library (32-bit mode only) 412 10.2.6 Example Program: Wrappers 419 10.2.7 Section Review 420 **10.3 Conditional-Assembly Directives 420**

10.3.1 Checking for Missing Arguments 421 10.3.2 Default Argument Initializers 422 10.3.3 Boolean Expressions 423 10.3.4 IF, ELSE, and ENDIF Directives 423 10.3.5 The IFIDN and IFIDNI Directives 424 10.3.6 Example: Summing a Matrix Row 425 10.3.7 Special Operators 428 10.3.8 Macro Functions 431 10.3.9 Section Review 433 **10.4 Defining Repeat Blocks 433** 10.4.1 WHILE Directive 433 10.4.2 REPEAT Directive 434 10.4.3 FOR Directive 434 10.4.4 FORC Directive 435 10.4.5 Example: Linked List 436 10.4.6 Section Review 437 **10.5 Chapter Summary 438**

**10.6 Key Terms 439**

10.6.1 Terms 439 10.6.2 Operators and Directives 439 **10.7 Review Questions and Exercises 440**

10.7.1 Short Answer 440 10.7.2 Algorithm Workbench 440 **10.8 Programming Exercises 442**

**11 MS-Windows Programming 445 11.1 Win32 Console Programming 445** 11.1.1 Background Information 446 11.1.2 Win32 Console Functions 450 11.1.3 Displaying a Message Box 452 11.1.4 Console Input 455 11.1.5 Console Output 461 11.1.6 Reading and Writing Files 463 11.1.7 File I/O in the Irvine32 Library 468 11.1.8 Testing the File I/O Procedures 470 11.1.9 Console Window Manipulation 473 11.1.10 Controlling the Cursor 476

xvi Contents

11.1.11 Controlling the Text Color 477 11.1.12 Time and Date Functions 479 11.1.13 Using the 64-Bit Windows API 482 11.1.14 Section Review 484

**11.2 Writing a Graphical Windows Application 484**

11.2.1 Necessary Structures 484 11.2.2 The MessageBox Function 486 11.2.3 The WinMain Procedure 486 11.2.4 The WinProc Procedure 487 11.2.5 The ErrorHandler Procedure 488 11.2.6 Program Listing 488 11.2.7 Section Review 492

**11.3 Dynamic Memory Allocation 492**

11.3.1 HeapTest Programs 496 11.3.2 Section Review 499

**11.4 x86 Memory Management 499**

11.4.1 Linear Addresses 500 11.4.2 Page Translation 503 11.4.3 Section Review 505

**11.5 Chapter Summary 505**

**11.6 Key Terms 507**

**11.7 Review Questions and Exercises 507**

11.7.1 Short Answer 507 11.7.2 Algorithm Workbench 508

**11.8 Programming Exercises 509**

**12 Floating-Point Processing and Instruction Encoding 511**

**12.1 Floating-Point Binary Representation 511**

12.1.1 IEEE Binary Floating-Point Representation 512 12.1.2 The Exponent 514 12.1.3 Normalized Binary Floating-Point Numbers 514 12.1.4 Creating the IEEE Representation 514 12.1.5 Converting Decimal Fractions to Binary Reals 516 12.1.6 Section Review 518

**12.2 Floating-Point Unit 518**

12.2.1 FPU Register Stack 519 12.2.2 Rounding 521 12.2.3 Floating-Point Exceptions 523 12.2.4 Floating-Point Instruction Set 523

Contents xvii

12.2.5 Arithmetic Instructions 526 12.2.6 Comparing Floating-Point Values 530 12.2.7 Reading and Writing Floating-Point Values 533 12.2.8 Exception Synchronization 534 12.2.9 Code Examples 535 12.2.10 Mixed-Mode Arithmetic 537 12.2.11 Masking and Unmasking Exceptions 538 12.2.12 Section Review 539

**12.3 x86 Instruction Encoding 539** 12.3.1 Instruction Format 540 12.3.2 Single-Byte Instructions 541 12.3.3 Move Immediate to Register 541 12.3.4 Register-Mode Instructions 542 12.3.5 Processor Operand-Size Prefix 543 12.3.6 Memory-Mode Instructions 544 12.3.7 Section Review 547

**12.4 Chapter Summary 547**

**12.5 Key Terms 549**

**12.6 Review Questions and Exercises 549**

12.6.1 Short Answer 549 12.6.2 Algorithm Workbench 550

**12.7 Programming Exercises 551**

**13 High-Level Language Interface 555**

**13.1 Introduction 555**

13.1.1 General Conventions 556 13.1.2 .MODEL Directive 557 13.1.3 Examining Compiler-Generated Code 559 13.1.4 Section Review 564

**13.2 Inline Assembly Code 564**

13.2.1 \_\_asm Directive in Visual C++ 564 13.2.2 File Encryption Example 566 13.2.3 Section Review 569

**13.3 Linking 32-Bit Assembly Language Code to C/C++ 570**

13.3.1 IndexOf Example 570 13.3.2 Calling C and C++ Functions 574 13.3.3 Multiplication Table Example 576 13.3.4 Calling C Library Functions 579 13.3.5 Directory Listing Program 582 13.3.6 Section Review 583

xviii Contents

**13.4 Chapter Summary 583**

**13.5 Key Terms 584**

**13.6 Review Questions 584**

**13.7 Programming Exercises 585**

***Chapters are available from the Companion Web site***

**14 16-Bit MS-DOS Programming 14.1**

**14.1 MS-DOS and the IBM-PC 14.1**

14.1.1 Memory Organization 14.2 14.1.2 Redirecting Input-Output 14.3 14.1.3 Software Interrupts 14.4 14.1.4 INT Instruction 14.5 14.1.5 Coding for 16-Bit Programs 14.6 14.1.6 Section Review 14.7

**14.2 MS-DOS Function Calls (INT 21h) 14.7**

14.2.1 Selected Output Functions 14.9 14.2.2 Hello World Program Example 14.11 14.2.3 Selected Input Functions 14.12 14.2.4 Date/Time Functions 14.16 14.2.5 Section Review 14.20

**14.3 Standard MS-DOS File I/O Services 14.20**

14.3.1 Create or Open File (716Ch) 14.22 14.3.2 Close File Handle (3Eh) 14.23 14.3.3 Move File Pointer (42h) 14.23 14.3.4 Get File Creation Date and Time 14.24 14.3.5 Selected Library Procedures 14.24 14.3.6 Example: Read and Copy a Text File 14.25 14.3.7 Reading the MS-DOS Command Tail 14.27 14.3.8 Example: Creating a Binary File 14.30 14.3.9 Section Review 14.33

**14.4 Chapter Summary 14.33**

**14.5 Programming Exercises 14.35**

**15 Disk Fundamentals 15.1**

**15.1 Disk Storage Systems 15.1**

15.1.1 Tracks, Cylinders, and Sectors 15.2 15.1.2 Disk Partitions (Volumes) 15.4 15.1.3 Section Review 15.4

Contents xix

**15.2 File Systems 15.5**

15.2.1 FAT12 15.6 15.2.2 FAT16 15.6 15.2.3 FAT32 15.6 15.2.4 NTFS 15.7 15.2.5 Primary Disk Areas 15.7 15.2.6 Section Review 15.8

**15.3 Disk Directory 15.9**

15.3.1 MS-DOS Directory Structure 15.10 15.3.2 Long Filenames in MS-Windows 15.12 15.3.3 File Allocation Table (FAT) 15.14 15.3.4 Section Review 15.14

**15.4 Reading and Writing Disk Sectors 15.15**

15.4.1 Sector Display Program 15.16 15.4.2 Section Review 15.19

**15.5 System-Level File Functions 15.20**

15.5.1 Get Disk Free Space (7303h) 15.20 15.5.2 Create Subdirectory (39h) 15.23 15.5.3 Remove Subdirectory (3Ah) 15.23 15.5.4 Set Current Directory (3Bh) 15.23 15.5.5 Get Current Directory (47h) 15.24 15.5.6 Get and Set File Attributes (7143h) 15.24 15.5.7 Section Review 15.25

**15.6 Chapter Summary 15.25**

**15.7 Programming Exercises 15.26**

**16 BIOS-Level Programming 16.1**

**16.1 Introduction 16.1**

16.1.1 BIOS Data Area 16.2

**16.2 Keyboard Input with INT 16h 16.3**

16.2.1 How the Keyboard Works 16.3 16.2.2 INT 16h Functions 16.4 16.2.3 Section Review 16.8

**16.3 VIDEO Programming with INT 10h 16.8**

16.3.1 Basic Background 16.8 16.3.2 Controlling the Color 16.10 16.3.3 INT 10h Video Functions 16.12 16.3.4 Library Procedure Examples 16.22 16.3.5 Section Review 16.23

xx Contents

**16.4 Drawing Graphics Using INT 10h 16.23**

16.4.1 INT 10h Pixel-Related Functions 16.24 16.4.2 DrawLine Program 16.25 16.4.3 Cartesian Coordinates Program 16.27 16.4.4 Converting Cartesian Coordinates to Screen Coordinates 16.29 16.4.5 Section Review 16.30

**16.5 Memory-Mapped Graphics 16.30**

16.5.1 Mode 13h: 320 X 200, 256 Colors 16.30 16.5.2 Memory-Mapped Graphics Program 16.32 16.5.3 Section Review 16.34

**16.6 Mouse Programming 16.35**

16.6.1 Mouse INT 33h Functions 16.35 16.6.2 Mouse Tracking Program 16.40 16.6.3 Section Review 16.44

**16.7 Chapter Summary 16.45**

**16.8 Programming Exercises 16.46**

**17 Expert MS-DOS Programming 17.1**

**17.1 Introduction 17.1**

**17.2 Defining Segments 17.2**

17.2.1 Simplified Segment Directives 17.2 17.2.2 Explicit Segment Definitions 17.4 17.2.3 Segment Overrides 17.7 17.2.4 Combining Segments 17.7 17.2.5 Section Review 17.9

**17.3 Runtime Program Structure 17.9**

17.3.1 Program Segment Prefix 17.10 17.3.2 COM Programs 17.10 17.3.3 EXE Programs 17.11 17.3.4 Section Review 17.13

**17.4 Interrupt Handling 17.13**

17.4.1 Hardware Interrupts 17.14 17.4.2 Interrupt Control Instructions 17.16 17.4.3 Writing a Custom Interrupt Handler 17.16 17.4.4 Terminate and Stay Resident Programs 17.19 17.4.5 Application: The No\_Reset Program 17.19 17.4.6 Section Review 17.23

**17.5 Hardware Control Using I/O Ports 17.23**

17.5.1 Input–Output Ports 17.24 17.5.2 PC Sound Program 17.24

**17.6 Chapter Summary 17.26**

Contents xxi

**Appendix A MASM Reference 587 Appendix B The x86 Instruction Set 609 Appendix C Answers to Section Review**

**Questions 644**

***Appendices are available from the Companion Web site***

**Appendix D BIOS and MS-DOS Interrupts D.1 Appendix E Answers to Review Questions**

**(Chapters 14–17) E.1**

**Index 664**

*This page intentionally left blank*

Preface

*Assembly Language for x86 Processors, Seventh Edition*, teaches assembly language program- ming and architecture for x86 and Intel64 processors. It is an appropriate text for the following types of college courses:

**•** Assembly Language Programming

**•** Fundamentals of Computer Systems

**•** Fundamentals of Computer Architecture Students use Intel or AMD processors and program with **Microsoft Macro Assembler (MASM)**, running on recent versions of Microsoft Windows. Although this book was originally designed as a programming textbook for college students, it serves as an effective supplement to computer architecture courses. As a testament to its popularity, previous editions have been translated into numerous languages.

*Emphasis of Topics* This edition includes topics that lead naturally into subsequent courses in computer architecture, operating systems, and compiler writing:

**•** Virtual machine concept

**•** Instruction set architecture

**•** Elementary Boolean operations

**•** Instruction execution cycle

**•** Memory access and handshaking

**•** Interrupts and polling

**•** Hardware-based I/O

**•** Floating-point binary representation Other topics relate specially to x86 and Intel64 architecture:

**•** Protected memory and paging

**•** Memory segmentation in real-address mode

**•** 16-Bit interrupt handling

**•** MS-DOS and BIOS system calls (interrupts)

**•** Floating-point unit architecture and programming

**•** Instruction encoding Certain examples presented in the book lend themselves to courses that occur later in a computer science curriculum:

**•** Searching and sorting algorithms

**•** High-level language structures

xxiii

xxiv Preface

**•** Finite-state machines

**•** Code optimization examples

**What’s New in the Seventh Edition** In this revision, we increased the discussions of program examples early in the book, added more sup- plemental review questions and key terms, introduced 64-bit programming, and reduced our depen- dence on the book’s subroutine library. To be more specific, here are the details:

**•** Early chapters now include short sections that feature 64-bit CPU architecture and program- ming, and we have created a 64-bit version of the book’s subroutine library named *Irvine64*.

**•** Many of the review questions and exercises have been modified, replaced, and moved from the middle of the chapter to the end of chapters, and divided into two sections: (1) Short answer questions, and (2) Algorithm workbench exercises. The latter exercises require the student to write a short amount of code to accomplish a goal.

**•** Each chapter now has a *Key Terms* section, listing new terms and concepts, as well as new MASM directives and Intel instructions.

**•** New programming exercises have been added, others removed, and a few existing exercises were modified.

**•** There is far less dependency on the author's subroutine libraries in this edition. Students are encouraged to call system functions themselves and use the Visual Studio debugger to step through the programs. The Irvine32 and Irvine64 libraries are available to help students han- dle input/output, but their use is not required.

**•** New tutorial videos covering essential content topics have been created by the author and added to the Pearson website. This book is still focused on its primary goal, to teach students how to write and debug programs at the machine level. It will never replace a complete book on computer architecture, but it does give students the first-hand experience of writing software in an environment that teaches them how a computer works. Our premise is that students retain knowledge better when theory is combined with experience. In an engineering course, students construct prototypes; in a computer architecture course, students should write machine-level programs. In both cases, they have a memorable experi- ence that gives them the confidence to work in any OS/machine-oriented environment.

Protected mode programming is entirely the focus of the printed chapters (1 through 13). As such, students will create 32-bit and 64-bit programs that run under the most recent versions of Microsoft Windows. The remaining four chapters cover 16-bit programming, and are supplied in electronic form. These chapters cover BIOS programming, MS-DOS services, keyboard and mouse input, video programming, and graphics. One chapter covers disk storage fundamentals. Another chapter covers advanced DOS programming techniques. *Subroutine Libraries* We supply three versions of the subroutine library that students use for basic input/output, simulations, timing, and other useful tasks. The Irvine32 and Irvine64 libraries run in protected mode. The 16-bit version (Irvine16.lib) runs in real-address mode and is used only by Chapters 14 through 17. Full source code for the libraries is supplied on the companion website. The link libraries are available only for convenience, not to prevent students from learning how to pro- gram input–output themselves. Students are encouraged to create their own libraries. *Included Software and Examples* All the example programs were tested with Microsoft Macro Assembler Version 11.0, running in Microsoft Visual Studio 2012. In addition, batch files are supplied that permit students to assemble and run applications from the Windows command

Preface xxv

prompt. The 32-bit C++ applications in Chapter 14 were tested with Microsoft Visual C++ .NET. Information Updates and corrections to this book may be found at the Companion Web site, includ- ing additional programming projects for instructors to assign at the ends of chapters.

**Overall Goals** The following goals of this book are designed to broaden the student’s interest and knowledge in topics related to assembly language:

**•** Intel and AMD processor architecture and programming

**•** Real-address mode and protected mode programming

**•** Assembly language directives, macros, operators, and program structure

**•** Programming methodology, showing how to use assembly language to create system-level software tools and application programs

**•** Computer hardware manipulation

**•** Interaction between assembly language programs, the operating system, and other applica- tion programs One of our goals is to help students approach programming problems with a machine-level mind set. It is important to think of the CPU as an interactive tool, and to learn to monitor its operation as directly as possible. A debugger is a programmer’s best friend, not only for catching errors, but as an educational tool that teaches about the CPU and operating system. We encourage stu- dents to look beneath the surface of high-level languages and to realize that most programming languages are designed to be portable and, therefore, independent of their host machines. In addition to the short examples, this book contains hundreds of ready-to-run programs that dem- onstrate instructions or ideas as they are presented in the text. Reference materials, such as guides to MS-DOS interrupts and instruction mnemonics, are available at the end of the book.

*Required Background* The reader should already be able to program confidently in at least one high-level programming language such as Python, Java, C, or C++. One chapter covers C++ interfacing, so it is very helpful to have a compiler on hand. I have used this book in the class- room with majors in both computer science and management information systems, and it has been used elsewhere in engineering courses.

**Features** *Complete Program Listings* The Companion Web site contains supplemental learning mate- rials, study guides, and all the source code from the book’s examples. An extensive link library is supplied with the book, containing more than 30 procedures that simplify user input–output, numeric processing, disk and file handling, and string handling. In the beginning stages of the course, students can use this library to enhance their programs. Later, they can create their own procedures and add them to the library.

*Programming Logic* Two chapters emphasize Boolean logic and bit-level manipulation. A conscious attempt is made to relate high-level programming logic to the low-level details of the machine. This approach helps students to create more efficient implementations and to better understand how compilers generate object code.

xxvi Preface

*Hardware and Operating System Concepts* The first two chapters introduce basic hard- ware and data representation concepts, including binary numbers, CPU architecture, status flags, and memory mapping. A survey of the computer’s hardware and a historical perspective of the Intel processor family helps students to better understand their target computer system.

*Structured Programming Approach* Beginning with Chapter 5, procedures and functional decomposition are emphasized. Students are given more complex programming exercises, requiring them to focus on design before starting to write code.

*Java Bytecodes and the Java Virtual Machine* In Chapters 8 and 9, the author explains the basic operation of Java bytecodes with short illustrative examples. Numerous short examples are shown in disassembled bytecode format, followed by detailed step-by-step explanations.

*Disk Storage Concepts* Students learn the fundamental principles behind the disk storage system on MS-Windows–based systems from hardware and software points of view.

*Creating Link Libraries* Students are free to add their own procedures to the book’s link library and create new libraries. They learn to use a toolbox approach to programming and to write code that is useful in more than one program.

*Macros and Structures* A chapter is devoted to creating structures, unions, and macros, which are essential in assembly language and systems programming. Conditional macros with advanced operators serve to make the macros more professional.

*Interfacing to High-Level Languages* A chapter is devoted to interfacing assembly lan- guage to C and C++. This is an important job skill for students who are likely to find jobs pro- gramming in high-level languages. They can learn to optimize their code and see examples of how C++ compilers optimize code.

*Instructional Aids* All the program listings are available on the Web. Instructors are provided a test bank, answers to review questions, solutions to programming exercises, and a Microsoft PowerPoint slide presentation for each chapter.

*VideoNotes* VideoNotes are Pearson’s new visual tool designed to teach students key pro- gramming concepts and techniques. These short step-by-step videos demonstrate basic assembly language concepts. VideoNotes allow for self-paced instruction with easy navigation including the ability to select, play, rewind, fast-forward, and stop within each VideoNote exercise.

VideoNotes are free with the purchase of a new textbook. To *purchase* access to VideoNotes, go to www.pearsonhighered.com/irvine and click on the VideoNotes under *Student Resources*.

**Chapter Descriptions** Chapters 1 to 8 contain core concepts of assembly language and should be covered in sequence. After that, you have a fair amount of freedom. The following chapter dependency graph shows how later chapters depend on knowledge gained from other chapters.

Preface xxvii

1 through 9

10

15

11 12 13 14 16 17

**1. Basic Concepts:** Applications of assembly language, basic concepts, machine language, and data

representation. **2. x86 Processor Architecture:** Basic microcomputer design, instruction execution cycle, x86 processor architecture, Intel64 architecture, x86 memory management, components of a microcomputer, and the input–output system. **3.Assembly Language Fundamentals:** Introduction to assembly language, linking and

debugging, and defining constants and variables. **4. Data Transfers, Addressing, and Arithmetic:** Simple data transfer and arithmetic instructions, assemble-link-execute cycle, operators, directives, expressions, JMP and LOOP instructions, and indirect addressing. **5. Procedures:** Linking to an external library, description of the book’s link library, stack oper-

ations, defining and using procedures, flowcharts, and top-down structured design. **6. Conditional Processing:** Boolean and comparison instructions, conditional jumps and

loops, high-level logic structures, and finite-state machines. **7. Integer Arithmetic:** Shift and rotate instructions with useful applications, multiplication and division, extended addition and subtraction, and ASCII and packed decimal arithmetic. **8. Advanced Procedures:** Stack parameters, local variables, advanced PROC and INVOKE

directives, and recursion. **9. Strings and Arrays:** String primitives, manipulating arrays of characters and integers, two-

dimensional arrays, sorting, and searching. **10. Structures and Macros:** Structures, macros, conditional assembly directives, and defining

repeat blocks. **11. MS-Windows Programming:** Protected mode memory management concepts, using the

Microsoft-Windows API to display text and colors, and dynamic memory allocation. **12. Floating-Point Processing and Instruction Encoding:** Floating-point binary representa- tion and floating-point arithmetic. Learning to program the IA-32 floating-point unit. Under- standing the encoding of IA-32 machine instructions. **13. High-Level Language Interface:** Parameter passing conventions, inline assembly code, and

linking assembly language modules to C and C++ programs.

**• Appendix A:** MASM Reference

**• Appendix B:** The x86 Instruction Set

**• Appendix C:** Answers to Review Questions

xxviii Preface

The following chapters and appendices are supplied online at the Companion Web site: **14. 16-Bit MS-DOS Programming:** Memory organization, interrupts, function calls, and stan-

dard MS-DOS file I/O services. **15. Disk Fundamentals:** Disk storage systems, sectors, clusters, directories, file allocation

tables, handling MS-DOS error codes, and drive and directory manipulation. **16. BIOS-Level Programming:** Keyboard input, video text, graphics, and mouse programming. **17. Expert MS-DOS Programming:** Custom-designed segments, runtime program structure,

and Interrupt handling. Hardware control using I/O ports.

**• Appendix D:** BIOS and MS-DOS Interrupts

**• Appendix E:** Answers to Review Questions (Chapters 14–17)

**Instructor and Student Resources *Instructor Resource Materials*** The following protected instructor material is available on the Companion Web site:

www.pearsonhighered.com/irvine

For username and password information, please contact your Pearson Representative.

**•** Lecture PowerPoint Slides

**•** Instructor Solutions Manual

***Student Resource Materials*** The student resource materials can be accessed through the publisher’s Web site located at *www.pearsonhighered.com/irvine*. These resources include:

**•** VideoNotes

**•** Online Chapters and Appendices

**•** Chapter 14: *16-Bit MS-DOS Programming*

**•** Chapter 15: *Disk Fundamentals*

**•** Chapter 16: *BIOS-Level Programming*

**•** Chapter 17: *Expert MS-DOS Programming*

**•** Appendix D: *BIOS and MS-DOS Interrupts*

**•** Appendix E: *Answers to Review Questions (Chapters 14–17)* Students must use the access card located in the front of the book to register and access the online chap- ters and VideoNotes. If there is no access card in the front of this textbook, students can purchase access by going to *www.pearsonhighered.com/irvine* and selecting “*Video Notes and Web Chapters.*” Instruc- tors must also register on the site to access this material. Students will also find a link to the author’s Web site. An access card is not required for the following materials, located at *www.asmirvine.com*:

**•** *Getting Started*, a comprehensive step-by-step tutorial that helps students customize Visual Studio for assembly language programming.

**•** Supplementary articles on assembly language programming topics.

**•** Complete source code for all example programs in the book, as well as the source code for the author’s supplementary library.

Preface xxix

**•** *Assembly Language Workbook*, an interactive workbook covering number conversions, address- ing modes, register usage, debug programming, and floating-point binary numbers. Content pages are HTML documents to allow for customization. Help File in Windows Help Format.

**•** Debugging Tools: Tutorials on using the Microsoft Visual Studio debugger.

**Acknowledgments** Many thanks are due to Tracy Johnson, Executive Editor for Computer Science at Pearson Edu- cation, who has provided friendly, helpful guidance over the past few years. Pavithra Jayapaul of Jouve did an excellent job on the book production, along with Greg Dulles as the production editor at Pearson.

***Previous Editions*** I offer my special thanks to the following individuals who were most helpful during the develop- ment of earlier editions of this book:

**•** William Barrett, San Jose State University

**•** Scott Blackledge

**•** James Brink, Pacific Lutheran University

**•** Gerald Cahill, Antelope Valley College

**•** John Taylor

*This page intentionally left blank*

About the Author

Kip Irvine has written five computer programming textbooks, for Intel Assembly Language, C++, Visual Basic (beginning and advanced), and COBOL. His book *Assembly Language for Intel-Based Computers* has been translated into six languages. His first college degrees (B.M., M.M., and doctorate) were in Music Composition, at University of Hawaii and University of Miami. He began programming computers for music synthesis around 1982 and taught pro- gramming at Miami-Dade Community College for 17 years. Kip earned an M.S. degree in Com- puter Science from the University of Miami, and he has been a full-time member of the faculty in the School of Computing and Information Sciences at Florida International University since 2000.

xxxi

*This page intentionally left blank*

**i**

Basic Concepts

1.1 Welcome to Assembly Language

1.1.1 Questions You Might Ask 1.1.2 Assembly Language Applications 1.1.3 Section Review 1.2 Virtual Machine Concept 1.2.1 Section Review 1.3 Data Representation

1.3.1 Binary Integers 1.3.2 Binary Addition 1.3.3 Integer Storage Sizes 1.3.4 Hexadecimal Integers 1.3.5 Hexadecimal Addition 1.3.6 Signed Binary Integers

1.3.7 Binary Subtraction 1.3.8 Character Storage 1.3.9 Section Review 1.4 Boolean Expressions

1.4.1 Truth Tables for Boolean Functions 1.4.2 Section Review 1.5 Chapter Summary 1.6 Key Terms 1.7 Review Questions and Exercises

1.7.1 Short Answer 1.7.2 Algorithm Workbench

This chapter establishes some core concepts relating to assembly language programming. For example, it shows how assembly language fits into the wide spectrum of languages and applica- tions. We introduce the virtual machine concept, which is so important in understanding the rela- tionship between software and hardware layers. A large part of the chapter is devoted to the binary and hexadecimal numbering systems, showing how to perform conversions and do basic arithmetic. Finally, this chapter introduces fundamental boolean operations (AND, OR, NOT, XOR), which will prove to be essential in later chapters.

**1.1 Welcome to Assembly Language** *Assembly Language for x86 Processors* focuses on programming microprocessors compatible with Intel and AMD processors running under 32-bit and 64-bit versions of Microsoft Windows.

1

2 Chapter 1 • Basic Concepts

The latest version of *Microsoft Macro Assembler* (known as *MASM*) should be used with this book. MASM is included with most versions of Microsoft Visual Studio (Pro, Ultimate, Express, . . . ). Please check our web site (*asmirvine.com*) for the latest details about support for MASM in Visual Studio. We also include lots of helpful information about how to set up your software and get started.

Some other well-known assemblers for x86 systems running under Microsoft Windows include TASM (Turbo Assembler), NASM (Netwide Assembler), and MASM32 (a variant of MASM). Two popular Linux-based assemblers are GAS (GNU assembler) and NASM. Of these, NASM’s syntax is most similar to that of MASM.

Assembly language is the oldest programming language, and of all languages, bears the closest resemblance to native machine language. It provides direct access to computer hard- ware, requiring you to understand much about your computer’s architecture and operating system.

*Educational Value* Why read this book? Perhaps you’re taking a college course whose title is similar to one of the following courses that often use our book:

**•** Microcomputer Assembly Language

**•** Assembly Language Programming

**•** Introduction to Computer Architecture

**•** Fundamentals of Computer Systems

**•** Embedded Systems Programming This book will help you learn basic principles about computer architecture, machine lan- guage, and low-level programming. You will learn enough assembly language to test your knowledge on today’s most widely used microprocessor family. You won’t be learning to pro- gram a “toy” computer using a simulated assembler; MASM is an industrial-strength assembler, used by practicing professionals. You will learn the architecture of the Intel processor family from a programmer’s point of view.

If you are planning to be a C or C++ developer, you need to develop an understanding of how memory, address, and instructions work at a low level. A lot of programming errors are not eas- ily recognized at the high-level language level. You will often find it necessary to “drill down” into your program’s internals to find out why it isn’t working.

If you doubt the value of low-level programming and studying details of computer software and hardware, take note of the following quote from a leading computer scientist, Donald Knuth, in discussing his famous book series, *The Art of Computer Programming*:

Some people [say] that having machine language, at all, was the great mistake that I made. I really don’t think you can write a book for serious computer programmers unless you are able to discuss low-level detail.1

Visit this book’s web site to get lots of supplemental information, tutorials, and exercises at **www.asmirvine.com**

1.1 Welcome to Assembly Language 3

**1.1.1 Questions You Might Ask**

*What Background Should I Have?* Before reading this book, you should have programmed in at least one structured high-level language, such as Java, C, Python, or C++. You should know how to use IF statements, arrays, and functions to solve programming problems.

*What Are Assemblers and Linkers?* An *assembler* is a utility program that converts source code programs from assembly language into machine language. A *linker* is a utility program that com- bines individual files created by an assembler into a single executable program. A related utility, called a *debugger*, lets you to step through a program while it’s running and examine registers and memory.

*What Hardware and Software Do I Need?* You need a computer that runs a 32-bit or 64-bit version of Microsoft Windows, along with one of the recent versions of Microsoft Visual Studio.

*What Types of Programs Can Be Created Using MASM?*

**•** *32-Bit Protected Mode:* 32-bit protected mode programs run under all 32-bit versions of Microsoft Windows. They are usually easier to write and understand than real-mode pro- grams. From now on, we will simply call this *32-bit mode*.

**•** *64-Bit Mode:* 64-bit programs run under all 64-bit versions of Microsoft Windows.

**•** *16-Bit Real-Address Mode:* 16-bit programs run under 32-bit versions of Windows and on embedded systems. Because they are not supported by 64-bit Windows, we will restrict dis- cussions of this mode to Chapters 14 through 17. These chapters are in electronic form, avail- able from the publisher’s web site.

*What Supplements Are Supplied with This Book?* The book’s web site (*www.asmirvine.com*) has the following:

**• *Assembly Language Workbook***, a collection of tutorials

**• *Irvine32, Irvine64, and Irvine16 subroutine libraries*** for 64-bit, 32-bit, and 16-bit program- ming, with complete source code

**• *Example programs*** with all source code from the book

**• *Corrections*** to the book

**• *Getting Started***, a detailed tutorial designed to help you set up Visual Studio to use the Microsoft assembler

**• *Articles*** on advanced topics not included in the printed book for lack of space

**• *A link to an online discussion forum***, where you can get help from other experts who use the book

*What Will I Learn?* This book should make you better informed about data representation, debugging, programming, and hardware manipulation. Here’s what you will learn:

**•** Basic principles of computer architecture as applied to x86 processors

**•** Basic boolean logic and how it applies to programming and computer hardware

**•** How x86 processors manage memory, using protected mode and virtual mode

**•** How high-level language compilers (such as C++) translate statements from their language into assembly language and native machine code

4 Chapter 1 • Basic Concepts

**•** How high-level languages implement arithmetic expressions, loops, and logical structures at the machine level

**•** Data representation, including signed and unsigned integers, real numbers, and character data

**•** How to debug programs at the machine level. The need for this skill is vital when you work in languages such as C and C++, which generate native machine code

**•** How application programs communicate with the computer’s operating system via interrupt handlers and system calls

**•** How to interface assembly language code to C++ programs

**•** How to create assembly language application programs

*How Does Assembly Language Relate to Machine Language? Machine language* is a numeric language specifically understood by a computer’s processor (the CPU). All x86 processors understand a common machine language. *Assembly language* consists of statements written with short mnemonics such as ADD, MOV, SUB, and CALL. Assembly language has a *one-to-one* rela- tionship with machine language: Each assembly language instruction corresponds to a single machine-language instruction.

*How Do C++ and Java Relate to Assembly Language?* High-level languages such as Python, C++, and Java have a *one-to-many* relationship with assembly language and machine language. A single statement in C++, for example, expands into multiple assembly language or machine instructions. Most people cannot read raw machine code, so in this book, we examine its closest relative, assembly language. For example, the following C++ code carries out two arithmetic operations and assigns the result to a variable. Assume X and Y are integers:

int Y; int X = (Y + 4) \* 3; Following is the equivalent translation to assembly language. The translation requires multiple statements because each assembly language statement corresponds to a single machine instruction:

mov eax,Y ; move Y to the EAX register add eax,4 ; add 4 to the EAX register mov ebx,3 ; move 3 to the EBX register imul ebx ; multiply EAX by EBX mov X,eax ; move EAX to X (*Registers* are named storage locations in the CPU that hold intermediate results of operations.) The point of this example is not to claim that C++ is superior to assembly language or vice versa, but to show their relationship.

*Is Assembly Language Portable?* A language whose source programs can be compiled and run on a wide variety of computer systems is said to be *portable*. A C++ program, for example, will compile and run on just about any computer, unless it makes specific references to library functions that exist under a single operating system. A major feature of the Java language is that compiled programs run on nearly any computer system.

Assembly language is not portable, because it is designed for a specific processor family. There are a number of different assembly languages widely used today, each based on a processor family.

1.1 Welcome to Assembly Language 5

Some well-known processor families are Motorola 68x00, x86, SUN Sparc, Vax, and IBM-370. The instructions in assembly language may directly match the computer’s architecture or they may be translated during execution by a program inside the processor known as a *microcode interpreter*.

*Why Learn Assembly Language?* If you’re still not convinced that you should learn assembly language, consider the following points:

**•** If you study computer engineering, you may likely be asked to write *embedded* programs. They are short programs stored in a small amount of memory in single-purpose devices such as telephones, automobile fuel and ignition systems, air-conditioning control systems, secu- rity systems, data acquisition instruments, video cards, sound cards, hard drives, modems, and printers. Assembly language is an ideal tool for writing embedded programs because of its economical use of memory.

**•** Real-time applications dealing with simulation and hardware monitoring require precise timing and responses. High-level languages do not give programmers exact control over machine code generated by compilers. Assembly language permits you to precisely specify a program’s executable code.

**•** Computer game consoles require their software to be highly optimized for small code size and fast execution. Game programmers are experts at writing code that takes full advantage of hardware features in a target system. They often use assembly language as their tool of choice because it permits direct access to computer hardware, and code can be hand optimized for speed.

**•** Assembly language helps you to gain an overall understanding of the interaction between computer hardware, operating systems, and application programs. Using assembly language, you can apply and test theoretical information you are given in computer architecture and operating systems courses.

**•** Some high-level languages abstract their data representation to the point that it becomes awk- ward to perform low-level tasks such as bit manipulation. In such an environment, program- mers will often call subroutines written in assembly language to accomplish their goal.

**•** Hardware manufacturers create device drivers for the equipment they sell. *Device drivers* are programs that translate general operating system commands into specific references to hardware details. Printer manufacturers, for example, create a different MS-Windows device driver for each model they sell. Often these device drivers contain significant amounts of assembly language code.

*Are There Rules in Assembly Language?* Most rules in assembly language are based on physical limitations of the target processor and its machine language. The CPU, for example, requires two instruction operands to be the same size. Assembly language has fewer rules than C++ or Java because the latter use syntax rules to reduce unintended logic errors at the expense of low-level data access. Assembly language programmers can easily bypass restrictions charac- teristic of high-level languages. Java, for example, does not permit access to specific memory addresses. One can work around the restriction by calling a C function using JNI (*Java Native Interface*) classes, but the resulting program can be awkward to maintain. Assembly language, on the other hand, can access any memory address. The price for such freedom is high: Assem- bly language programmers spend a lot of time debugging!

6 Chapter 1 • Basic Concepts

**1.1.2 Assembly Language Applications** In the early days of programming, most applications were written partially or entirely in assem- bly language. They had to fit in a small area of memory and run as efficiently as possible on slow processors. As memory became more plentiful and processors dramatically increased in speed, programs became more complex. Programmers switched to high-level languages such as C, FORTRAN, and COBOL that contained a certain amount of structuring capability. More recently, object-oriented languages such as Python, C++, C#, and Java have made it possible to write complex programs containing millions of lines of code.

It is rare to see large application programs coded completely in assembly language because they would take too much time to write and maintain. Instead, assembly language is used to opti- mize certain sections of application programs for speed and to access computer hardware. Table 1-1 compares the adaptability of assembly language to high-level languages in relation to various types of applications.

Table 1-1 Comparison of Assembly Language to High-Level Languages.

**Type of Application High-Level Languages Assembly Language**

Commercial or scientific appli- cation, written for single plat- form, medium to large size.

The C and C++ languages have the unique quality of offering a compromise between high- level structure and low-level details. Direct hardware access is possible but completely nonport- able. Most C and C++ compilers allow you to embed assembly language statements in their code, providing access to hardware details.

**1.1.3 Section Review**

1. How do assemblers and linkers work together? 2. How will studying assembly language enhance your understanding of operating systems?

Formal structures make it easy to orga- nize and maintain large sections of code.

Minimal formal structure, so one must be imposed by programmers who have varying levels of experi- ence. This leads to difficulties main- taining existing code.

Hardware device driver. The language may not provide for direct hardware access. Even if it does, awk- ward coding techniques may be required, resulting in maintenance difficulties.

Hardware access is straightforward and simple. Easy to maintain when pro- grams are short and well documented.

Commercial or scientific appli- cation written for multiple platforms (different operating systems).

Usually portable. The source code can be recompiled on each target operating system with minimal changes.

Must be recoded separately for each platform, using an assembler with a different syntax. Difficult to maintain.

Embedded systems and com- puter games requiring direct hardware access.

May produce large executable files that exceed the memory capacity of the device.

Ideal, because the executable code is small and runs quickly.

1.2 Virtual Machine Concept 7

3. What is meant by a *one-to-many relationship* when comparing a high-level language to

machine language? 4. Explain the concept of *portability* as it applies to programming languages. 5. Is the assembly language for x86 processors the same as those for computer systems such as

the Vax or Motorola 68x00? 6. Give an example of an embedded systems application. 7. What is a device driver? 8. Do you suppose type checking on pointer variables is stronger (stricter) in assembly lan-

guage, or in C and C++? 9. Name two types of applications that would be better suited to assembly language than a

high-level language. 10. Why would a high-level language not be an ideal tool for writing a program that directly

accesses a printer port? 11. Why is assembly language not usually used when writing large application programs? 12. *Challenge:* Translate the following C++ expression to assembly language, using the example

presented earlier in this chapter as a guide: X (Y \* 4) 3.

**1.2 Virtual Machine Concept** An effective way to explain how a computer’s hardware and software are related is called the *virtual machine concept*. A well-known explanation of this model can be found in Andrew Tanenbaum’s book, *Structured Computer Organization.* To explain this concept, let us begin with the most basic function of a computer, executing programs.

A computer can usually execute programs written in its native *machine language*. Each instruction in this language is simple enough to be executed using a relatively small number of electronic circuits. For simplicity, we will call this language **L0**.

Programmers would have a difficult time writing programs in L0 because it is enormously detailed and consists purely of numbers. If a new language, **L1**, could be constructed that was easier to use, programs could be written in L1. There are two ways to achieve this:

**•** *Interpretation:* As the L1 program is running, each of its instructions could be decoded and executed by a program written in language L0. The L1 program begins running immediately, but each instruction has to be decoded before it can execute.

**•** *Translation:* The entire L1 program could be converted into an L0 program by an L0 program specifically designed for this purpose. Then the resulting L0 program could be executed directly on the computer hardware.

***Virtual Machines*** Rather than using only languages, it is easier to think in terms of a hypothetical computer, or *vir- tual machine*, at each level. Informally, we can define a virtual machine as a software program that emulates the functions of some other physical or virtual computer. The virtual machine

8 Chapter 1 • Basic Concepts

**VM1**, as we will call it, can execute commands written in language L1. The virtual machine **VM0** can execute commands written in language L0:

Virtual Machine VM1

Virtual Machine VM0

Each virtual machine can be constructed of either hardware or software. People can write pro- grams for virtual machine VM1, and if it is practical to implement VM1 as an actual computer, programs can be executed directly on the hardware. Or programs written in VM1 can be inter- preted/translated and executed on machine VM0.

Machine VM1 cannot be radically different from VM0 because the translation or interpreta- tion would be too time-consuming. What if the language VM1 supports is still not programmer- friendly enough to be used for useful applications? Then another virtual machine, VM2, can be designed that is more easily understood. This process can be repeated until a virtual machine VM*n* can be designed to support a powerful, easy-to-use language.

The Java programming language is based on the virtual machine concept. A program written in the Java language is translated by a Java compiler into *Java byte code*. The latter is a low-level language quickly executed at runtime by a program known as a *Java virtual machine (JVM)*. The JVM has been implemented on many different computer systems, making Java programs rela- tively system independent.

***Specific Machines*** Let us relate this to actual computers and languages, using names such as **Level 2** for VM2 and **Level 1** for VM1, shown in Figure 1-1. A computer’s digital logic hardware represents machine Level 1. Above this is Level 2, called the *instruction set Architecture (ISA).* This is the first level at which users can typi- cally write programs, although the programs consist of binary values called *machine language.*

*Instruction Set Architecture (Level 2)* Computer chip manufacturers design into the proces- sor an instruction set to carry out basic operations, such as move, add, or multiply. This set of instructions is also referred to as *machine language*. Each machine-language instruction is exe- cuted either directly by the computer’s hardware or by a program embedded in the microprocessor chip called a *microprogram*. A discussion of microprograms is beyond the scope of this book, but you can refer to Tanenbaum for more details.

*Assembly Language (Level 3)* Above the ISA level, programming languages provide trans- lation layers to make large-scale software development practical. Assembly language, which appears at Level 3, uses short mnemonics such as ADD, SUB, and MOV, which are easily trans- lated to the ISA level. Assembly language programs are translated (assembled) in their entirety into machine language before they begin to execute.

1.3 Data Representation 9

Figure 1–1 Virtual machine levels.

Level 4 High-level language

Level 3

Assembly language

Level 2

Instruction set architecture (ISA)

Level 1 Digital logic *High-Level Languages (Level 4)* At Level 4 are high-level programming languages such as C, C++, and Java. Programs in these languages contain powerful statements that translate into multiple assembly language instructions. You can see such a translation, for example, by exam- ining the listing file output created by a C++ compiler. The assembly language code is automati- cally assembled by the compiler into machine language.

**1.2.1 Section Review**

1. In your own words, describe the *virtual machine* concept. 2. Why do you suppose translated programs often execute more quickly than interpreted ones? 3. *(True/False):* When an interpreted program written in language L1 runs, each of its instruc-

tions is decoded and executed by a program written in language L0. 4. Explain the importance of translation when dealing with languages at different virtual

machine levels. 5. At which level does assembly language appear in the virtual machine example shown in this

section? 6. What software utility permits compiled Java programs to run on almost any computer? 7. Name the four virtual machine levels named in this section, from lowest to highest. 8. Why don’t programmers write applications in machine language? 9. Machine language is used at which level of the virtual machine shown in Figure 1-1? 10. Statements at the assembly language level of a virtual machine are translated into state-

ments at which other level?

**1.3 Data Representation** Assembly language programmers deal with data at the physical level, so they must be adept at examining memory and registers. Often, binary numbers are used to describe the contents of computer memory; at other times, decimal and hexadecimal numbers are used. You must develop

10 Chapter 1 • Basic Concepts

a certain fluency with number formats, so you can quickly translate numbers from one format to another.

Each numbering format, or system, has a *base*, or maximum number of symbols that can be assigned to a single digit. Table 1-2 shows the possible digits for the numbering systems used most commonly in hardware and software manuals. In the last row of the table, hexadecimal numbers use the digits 0 through 9 and continue with the letters A through F to represent deci- mal values 10 through 15. It is quite common to use hexadecimal numbers when showing the contents of computer memory and machine-level instructions.

**1.3.1 Binary Integers** A computer stores instructions and data in memory as collections of electronic charges. Representing these entities with numbers requires a system geared to the concepts of *on* and *off* or *true* and *false*. *Binary numbers* are base 2 numbers, in which each binary digit (called a *bit*) is either 0 or 1. ***Bits*** are numbered sequentially starting at zero on the right side and increasing toward the left. The bit on the left is called the *most significant bit* (MSB), and the bit on the right is the *least significant bit* (LSB). The MSB and LSB bit numbers of a 16-bit binary number are shown in the following figure:

MSB LSB

1 0 1 1 0 0 1 0 1 0 0 1 1 1 0 0

150 Bit number Table 1-2 Binary, Octal, Decimal, and Hexadecimal Digits.

**System Base Possible Digits**

Binary 2 0 1

Octal 8 0 1 2 3 4 5 6 7

Decimal 10 0 1 2 3 4 5 6 7 8 9

Hexadecimal 16 0 1 2 3 4 5 6 7 8 9 A B C D E F

Binary integers can be signed or unsigned. A signed integer is positive or negative. An unsigned integer is by default positive. Zero is considered positive. When writing down large binary numbers, many people like to insert a dot every 4 bits or 8 bits to make the numbers eas- ier to read. Examples are 1101.1110.0011.1000.0000 and 11001010.10101100.

***Unsigned Binary Integers*** Starting with the LSB, each bit in an unsigned binary integer represents an increasing power of 2. The following figure contains an 8-bit binary number, showing how powers of two increase from right to left:

11 1 1 1 1 1 1

27

26 25 24 23 22 21 20

Table 1-3 lists the decimal values of 20 through 215.

1.3 Data Representation 11

Table 1-3 Binary Bit Position Values.

**2*n* Decimal Value 2*n* Decimal Value**

20 1 28 256

21 2 29 512

22 4 210 1024

23 8 211 2048

24 16 212 4096

25 32 213 8192

26 64 214 16384

27 128 215 32768

***Translating Unsigned Binary Integers to Decimal*** *Weighted positional notation* represents a convenient way to calculate the decimal value of an unsigned binary integer having *n* digits:

dec (*Dn* 1 2*n* 1) (*Dn* 2 2*n* 2) (*D*1 21) (*D*0 20) *D* indicates a binary digit. For example, binary 00001001 is equal to 9. We calculate this value by leaving out terms equal to zero: (1 23) (1 20) 9 The same calculation is shown by the following figure:

8 19

10010000

***Translating Unsigned Decimal Integers to Binary*** To translate an unsigned decimal integer into binary, repeatedly divide the integer by 2, saving each remainder as a binary digit. The following table shows the steps required to translate decimal 37 to binary. The remainder digits, starting from the top row, are the binary digits *D*0, *D*1, *D*2, *D*3, *D*4, and *D*5:

**Division Quotient Remainder**

37 / 2 18 1

18 / 2 9 0

9 / 2 4 1

4 / 2 2 0

2 / 2 1 0

1 / 2 0 1

12 Chapter 1 • Basic Concepts

We can concatenate the binary bits from the remainder column of the table in reverse order (*D*5, *D*4, . . .) to produce binary 100101. Because computer storage always consists of binary numbers whose lengths are multiples of 8, we fill the remaining two digit positions on the left with zeros, producing 00100101.

***Tip:*** How many bits? There’s a simple formula to find *b*, the number of binary bits you need to represent the unsigned decimal 4.087463, which when raised to value the smallest *n*. It is *b* = following ceiling ( integer, log2 *n*). equals If *n* = 17, 5. Most for example, calculators log2 don’t 17 =

have a log base 2 operation, but you can find web pages that will calculate it for you.

**1.3.2 Binary Addition** When adding two binary integers, proceed bit by bit, starting with the low-order pair of bits (on the right) and add each subsequent pair of bits. There are four ways to add two binary digits, as shown here:

0 0 0 0 1 1

1 0 1 1 1 10

When adding 1 to 1, the result is 10 binary (think of it as the decimal value 2). The extra digit generates a carry to the next-highest bit position. In the following figure, we add binary 00000100 to 00000111:

Carry:

1

00 0 0 0 1 0

0(4)

00 0 0 0 1 1

1(7)

00 0 0 1 0 1

1(11)

Bit position: 7

56 01234 Beginning with the lowest bit in each number (bit position 0), we add 0 1, producing a 1 in the bottom row. The same happens in the next highest bit (position 1). In bit position 2, we add 1 1, generating a sum of zero and a carry of 1. In bit position 3, we add the carry bit to 0 0, producing 1. The rest of the bits are zeros. You can verify the addition by adding the decimal equivalents shown on the right side of the figure (4 7 11).

Sometimes a carry is generated out of the highest bit position. When that happens, the size of the storage area set aside becomes important. If we add 11111111 to 00000001, for exam- ple, a 1 carries out of the highest bit position, and the lowest 8 bits of the sum equal all zeros. If the storage location for the sum is at least 9 bits long, we can represent the sum as 100000000. But if the sum can only store 8 bits, it will equal to 00000000, the lowest 8 bits of the calculated value.

1.3 Data Representation 13

**1.3.3 Integer Storage Sizes** The basic storage unit for all data in an x86 computer is a *byte*, containing 8 bits. Other storage sizes are *word* (2 bytes), *doubleword* (4 bytes), and *quadword* (8 bytes). In the following figure, the number of bits is shown for each size:

Byte

8

Word

16

Doubleword

32

Quadword 64 Double quadword 128 Table 1-4 shows the range of possible values for each type of unsigned integer.

*Large Measurements* A number of large measurements are used when referring to both memory and disk space:

**•** One *kilobyte* is equal to 210, or 1024 bytes.

**•** One *megabyte* (1 MByte) is equal to 220, or 1,048,576 bytes.

**•** One *gigabyte* (1 GByte) is equal to 230, or 10243, or 1,073,741,824 bytes.

**•** One *terabyte* (1 TByte) is equal to 240, or 10244, or 1,099,511,627,776 bytes.

**•** One *petabyte* is equal to 250, or 1,125,899,906,842,624 bytes.

**•** One *exabyte* is equal to 260, or 1,152,921,504,606,846,976 bytes.

**•** One *zettabyte* is equal to 270 bytes.

**•** One *yottabyte* is equal to 280 bytes.

Table 1-4 Ranges and Sizes of Unsigned Integer Types.

**Type Range**

**1.3.4 Hexadecimal Integers** Large binary numbers are cumbersome to read, so hexadecimal digits offer a convenient way to represent binary data. Each digit in a hexadecimal integer represents four binary bits, and two hexadecimal digits together represent a byte. A single hexadecimal digit represents decimal 0 to 15, so letters A to F represent decimal values in the range 10 through 15. Table 1-5 shows how each sequence of four binary bits translates into a decimal or hexadecimal value.

**Storage Size in Bits**

Unsigned byte 0 to 28 − 1 8

Unsigned word 0 to 216 − 1 16

Unsigned doubleword 0 to 232 − 1 32

Unsigned quadword 0 to 264 − 1 64

Unsigned double quadword 0 to 2128− 1 128

14 Chapter 1 • Basic Concepts

Table 1-5 Binary, Decimal, and Hexadecimal Equivalents.

**Binary Decimal Hexadecimal Binary Decimal Hexadecimal**

0000 0 0 1000 8 8

0001 1 1 1001 9 9

0010 2 2 1010 10 A

0011 3 3 1011 11 B

0100 4 4 1100 12 C

0101 5 5 1101 13 D

0110 6 6 1110 14 E

0111 7 7 1111 15 F

The following example shows how binary 0001 0110 1010 0111 1001 0100 is equivalent to hexadecimal 16A794:

1 6 A 7 9 4

0001 0110 1010 0111 1001 0100

***Converting Unsigned Hexadecimal to Decimal*** In hexadecimal, each digit position represents a power of 16. This is helpful when calculating the decimal value of a hexadecimal integer. Suppose we number the digits in a four-digit hexadecimal integer with subscripts as D3D2D1D0. The following formula calculates the integer’s decimal value:

*dec* (D3 163) (D2 162) (D1 161) (D0 160) The formula can be generalized for any *n*-digit hexadecimal integer:

*dec* (D*n* 1 16*n* 1) (D*n* 2 16*n* 2) (D1 161) (D0 160)

In general, you can convert an *n*-digit integer in any base B to decimal using the following formula: *dec* = (D*n* 1 B*n* 1) (D*n* 2 B*n* 2) (D1 × B1) (D0 B0).

For example, hexadecimal 1234 is equal to (1 163) (2 162) (3 161) (4 160), or decimal 4660. Similarly, hexadecimal 3BA4 is equal to (3 163) (11 162) (10 161) (4 160), or decimal 15,268. The following figure shows this last calculation:

3 × 163 12,288

11 × 162 2,816

10 × 161 160

4 × 160 4

4AB3

Total: 15,268

1.3 Data Representation 15

Table 1-6 lists the powers of 16 from 160 to 167.

Table 1-6 Powers of 16 in Decimal.

**16*n* Decimal Value 16*n* Decimal Value**

160 1 164 65,536

161 16 165 1,048,576

162 256 166 16,777,216

163 4096 167 268,435,456

***Converting Unsigned Decimal to Hexadecimal*** To convert an unsigned decimal integer to hexadecimal, repeatedly divide the decimal value by 16 and retain each remainder as a hexadecimal digit. For example, the following table lists the steps when converting decimal 422 to hexadecimal:

**Division Quotient Remainder**

422 / 16 26 6

26 / 16 1 A

1 / 16 0 1

The resulting hexadecimal number is assembled from the digits in the remainder column, start- ing from the last row and working upward to the top row. In this example, the hexadecimal rep- resentation is **1A6**. The same algorithm was used for binary integers in Section 1.3.1. To convert from decimal into some other number base other than hexadecimal, replace the divisor (16) in each calculation with the desired number base.

**1.3.5 Hexadecimal Addition** Debugging utility programs (known as *debuggers*) usually display memory addresses in hexa- decimal. It is often necessary to add two addresses in order to locate a new address. Fortu- nately, hexadecimal addition works the same way as decimal addition, if you just change the number base.

Suppose we want to add two numbers X and Y, using numbering base *b*. We will number their digits from the lowest position (*x*0) to the highest. If we add digits *xi* and *yi* in X and Y, we produce the value *si*. If s*i* ≥

*b* , we recalculate *si* (*si* MOD *b*) and generate a carry value of 1. When we move to the next pair of digits *xi*+1 and *yi*+1, we add the carry value to their sum.

For example, let’s add the hexadecimal values 6A2 and 49A. In the lowest digit position, 2 A decimal 12, so there is no carry and we use C to indicate the hexadecimal sum digit. In the next position, A 9 decimal 19, so there is a carry because 19 ≥

16 , the num- ber base. We calculate 19 MOD 16 3, and carry a 1 into the third digit position. Finally, we add 1 6 4 decimal 11, which is shown as the letter B in the third position of the sum. The hexadecimal sum is B3C.

16 Chapter 1 • Basic Concepts

**Carry** 1

X 6 A 2

Y 4 9 A

S B 3 C

**1.3.6 Signed Binary Integers** Signed binary integers are positive or negative. For x86 processors, the MSB indicates the sign: 0 is positive and 1 is negative. The following figure shows examples of 8-bit negative and positive integers:

Sign bit

1

1 1 1 0 1 1 0

Negative

0 0 0 0 1 0 1 0

Positive

***Two’s-Complement Representation*** Negative integers use *two’s-complement* representation, using the mathematical principle that the two’s complement of an integer is its additive inverse. (If you add a number to its additive inverse, the sum is zero.)

Two’s-complement representation is useful to processor designers because it removes the need for separate digital circuits to handle both addition and subtraction. For example, if presented with the expression *A B*, the processor can simply convert it to an addition expression: *A* ( *B*).

The two’s complement of a binary integer is formed by inverting (complementing) its bits and adding 1. Using the 8-bit binary value 00000001, for example, its two’s complement turns out to be 11111111, as can be seen as follows:

Starting value 00000001

Step 1: Reverse the bits 11111110

Step 2: Add 1 to the value from Step 1 11111110 +00000001

Sum: Two’s-complement representation 11111111

11111111 is the two’s-complement representation of 1. The two’s-complement operation is reversible, so the two’s complement of 11111111 is 00000001.

*Hexadecimal Two’s Complement* To create the two’s complement of a hexadecimal integer, reverse all bits and add 1. An easy way to reverse the bits of a hexadecimal digit is to subtract the digit from 15. Here are examples of hexadecimal integers converted to their two’s complements:

6A3D --> 95C2 + 1 --> 95C3 95C3 --> 6A3C + 1 --> 6A3D

1.3 Data Representation 17

*Converting Signed Binary to Decimal* Use the following algorithm to calculate the decimal equivalent of a signed binary integer:

**•** If the highest bit is a 1, the number is stored in two’s-complement notation. Create its two’s complement a second time to get its positive equivalent. Then convert this new number to decimal as if it were an unsigned binary integer.

**•** If the highest bit is a 0, you can convert it to decimal as if it were an unsigned binary integer. For example, signed binary 11110000 has a 1 in the highest bit, indicating that it is a negative integer. First we create its two’s complement, and then convert the result to decimal. Here are the steps in the process:

Starting value 11110000

Step 1: Reverse the bits 00001111

Step 2: Add 1 to the value from Step 1 00001111 + 1

Step 3: Create the two’s complement 00010000

Step 4: Convert to decimal 16

Because the original integer (11110000) was negative, we know that its decimal value is −16.

*Converting Signed Decimal to Binary* To create the binary representation of a signed deci- mal integer, do the following:

1. Convert the absolute value of the decimal integer to binary. 2. If the original decimal integer was negative, create the two’s complement of the binary num-

ber from the previous step. For example, −43 decimal is translated to binary as follows:

1. The binary representation of unsigned 43 is 00101011. 2.Because the original value was negative, we create the two’s complement of 00101011,

which is 11010101. This is the representation of −43 decimal.

*Converting Signed Decimal to Hexadecimal* To convert a signed decimal integer to hexa- decimal, do the following:

1. Convert the absolute value of the decimal integer to hexadecimal. 2. If the decimal integer was negative, create the two’s complement of the hexadecimal number

from the previous step.

*Converting Signed Hexadecimal to Decimal* To convert a signed hexadecimal integer to decimal, do the following:

1.If the hexadecimal integer is negative, create its two’s complement; otherwise, retain the

integer as is. 2. Using the integer from the previous step, convert it to decimal. If the original value was nega-

tive, attach a minus sign to the beginning of the decimal integer.

You can tell whether a hexadecimal integer is positive or negative by inspecting its most signifi- cant (highest) digit. If the digit is ≥ 8, the number is negative; if the digit is ≤ 7, the number is pos- itive. For example, hexadecimal 8A20 is negative and 7FD9 is positive.

18 Chapter 1 • Basic Concepts

***Maximum and Minimum Values*** A signed integer of *n* bits uses only *n* 1 bits to represent the number’s magnitude. Table 1-7 shows the minimum and maximum values for signed bytes, words, doublewords, and quadwords.

Table 1-7 Ranges and Sizes of Signed Integer Types.

**Type Range Storage Size in Bits**

Signed byte –27 to +27– 1 8

Signed word –215 to +215– 1 16

Signed doubleword –231 to +231– 1 32

Signed quadword –263 to +263– 1 64

Signed double quadword –2127 to +2127– 1 128

**1.3.7 Binary Subtraction** Subtracting a smaller unsigned binary number from a large one is easy if you go about it in the same way you handle decimal subtraction. Here’s an example:

0 1 1 0 1 (decimal 13) - 0 0 1 1 1 (decimal 7) ----------- Subtracting the bits in position 0 is straightforward:

0 1 1 0 1 - 0 0 1 1 1 ----------- 0 In the next position (0 – 1), we are forced to borrow a 1 from the next position to the left. Here’s the result of subtracting 1 from 2:

0 1 0 0 1 - 0 0 1 1 1 ----------- 1 0

In the next bit position, we again have to borrow a bit from the column just to the left and sub- tract 1 from 2:

0 0 0 1 1 - 0 0 1 1 1 ----------- 1 1 0 Finally, the two high-order bits are zero minus zero:

0 0 0 1 1 - 0 0 1 1 1 -----------

0 0 1 1 0 (decimal 6)

1.3 Data Representation 19

A simpler way to approach binary subtraction is to reverse the sign of the value being subtracted, and then add the two values. This method requires you to have an extra empty bit to hold the number’s sign. Let’s try it with the same problem we just calculated: (01101 minus 00111). First, we negate 00111 by inverting its bits (11000) and adding 1, producing 11001. Next, we add the binary values and ignore the carry out of the highest bit:

0 1 1 0 1 (+13) 1 1 0 0 1 (-7) --------- 0 0 1 1 0 (+6)

The result, +6, is exactly what we expected.

**1.3.8 Character Storage** If computers only store binary data, how do they represent characters? They use a *character set*, which is a mapping of characters to integers. In earlier times, character sets used only 8 bits. Even now, when running in character mode (such as MS-DOS), IBM-compatible microcomputers use the***ASCII*** (pronounced “askey”) character set. ASCII is an acronym for *American Standard Code for Information Interchange*. In ASCII, a unique 7-bit integer is assigned to each character. Because ASCII codes use only the lower 7 bits of every byte, the extra bit is used on various com- puters to create a proprietary character set. On IBM-compatible microcomputers, for example, values 128 through 255 represent graphic symbols and Greek characters.

*ANSI Character Set* The American National Standards Institute (ANSI) defines an 8-bit character set that represents up to 256 characters. The first 128 characters correspond to the letters and symbols on a standard U.S. keyboard. The second 128 characters represent spe- cial characters such as letters in international alphabets, accents, currency symbols, and fractions. Early version of Microsoft Windows used the ANSI character set.

*Unicode Standard* Today, computers must be able to represent a wide variety of international languages in computer software. As a result, the *Unicode* standard was created as a universal way of defining characters and symbols. It defines numeric codes (called *code points*) for char- acters, symbols, and punctuation used in all major languages, as well as European alphabetic scripts, Middle Eastern right-to-left scripts, and many scripts of Asia. Three transformation for- mats are used to transform code points into displayable characters:

**• UTF-8** is used in HTML, and has the same byte values as ASCII.

**• UTF-16** is used in environments that balance efficient access to characters with economical use of storage. Recent versions of Microsoft Windows, for example, use UTF-16 encoding. Each character is encoded in 16 bits.

**• UTF-32** is used in environments where space is no concern and fixed-width characters are required. Each character is encoded in 32 bits.

*ASCII Strings* A sequence of one or more characters is called a *string*. More specifically, an *ASCII string* is stored in memory as a succession of bytes containing ASCII codes. For example, the numeric codes for the string “ABC123” are 41h, 42h, 43h, 31h, 32h, and 33h. A *null-terminated* string is a string of characters followed by a single byte containing zero. The C and C++ languages use null-terminated strings, and many Windows operating system functions require strings to be in this format.

20 Chapter 1 • Basic Concepts

*Using the ASCII Table* A table on the inside back cover of this book lists ASCII codes used when running in Windows Console mode. To find the hexadecimal ASCII code of a character, look along the top row of the table and find the column containing the character you want to translate. The most significant digit of the hexadecimal value is in the second row at the top of the table; the least significant digit is in the second column from the left. For example, to find the ASCII code of the letter **a**, find the column containing the **a** and look in the second row: The first hexadecimal digit is 6. Next, look to the left along the row containing **a** and note that the second column contains the digit 1. Therefore, the ASCII code of **a** is 61 hexadecimal. This is shown as follows in simplified form:

*ASCII Control Characters* Character codes in the range 0 through 31 are called *ASCII control characters.* If a program writes these codes to standard output (as in C++), the con- trol characters will carry out predefined actions. Table 1-8 lists the most commonly used characters in this range, and a complete list may be found in the inside front cover of this book.

6

1 aTable 1-8 ASCII Control Characters.

**ASCII Code (Decimal) Description**

8 Backspace (moves one column to the left)

9 Horizontal tab (skips forward *n* columns)

10 Line feed (moves to next output line)

12 Form feed (moves to next printer page)

13 Carriage return (moves to leftmost output column)

27 Escape character

*Terminology for Numeric Data Representation* It is important to use precise terminology when describing the way numbers and characters are represented in memory and on the display screen. Decimal 65, for example, is stored in memory as a single binary byte as 01000001. A debugging program would probably display the byte as “41,” which is the number’s hexadeci- mal representation. If the byte were copied to video memory, the letter “**A**” would appear on the screen because 01000001 is the ASCII code for the letter **A**. Because a number’s interpretation can depend on the context in which it appears, we assign a specific name to each type of data representation to clarify future discussions:

**•** A *binary integer* is an integer stored in memory in its raw format, ready to be used in a calcu- lation. Binary integers are stored in multiples of 8 bits (such as 8, 16, 32, or 64).

1.3 Data Representation 21

**•** A *digit string* is a string of ASCII characters, such as “123” or “65.” This is simply a repre- sentation of the number and can be in any of the formats shown for the decimal number 65 in Table 1-9:

**1.3.9 Section Review**

1. Explain the term *least significant bit* (LSB).

2. What is the decimal representation of each of the following unsigned binary integers?

a. 11111000 b. 11001010 c. 11110000

3. What is the sum of each pair of binary integers?

a. 00001111 00000010 b. 11010101 01101011 c. 00001111 00001111

4. How many bytes are contained in each of the following data types?

a. word b. doubleword c. quadword d. double quadword

5. What is the minimum number of binary bits needed to represent each of the following

unsigned decimal integers? a. 65 b. 409 c. 16385

6. What is the hexadecimal representation of each of the following binary numbers?

a. 0011 0101 1101 1010 b. 1100 1110 1010 0011 c. 1111 1110 1101 1011

7. What is the binary representation of the following hexadecimal numbers?

a. A4693FBC b. B697C7A1 c. 2B3D9461

Table 1-9 Types of Digit Strings.

**Format Value**

Binary digit string “01000001”

Decimal digit string “65”

Hexadecimal digit string “41”

Octal digit string “101”

22 Chapter 1 • Basic Concepts

**1.4 Boolean Expressions** *Boolean algebra* defines a set of operations on the values **true** and **false**. It was invented by George Boole, a mid-nineteenth-century mathematician. When early digital computers were invented, it was found that Boole’s algebra could be used to describe the design of digital circuits. At the same time, boolean expressions are used in computer programs to express logical operations.

A *boolean expression* involves a boolean operator and one or more operands. Each boolean expression implies a value of true or false. The set of operators includes the following:

**•** NOT: notated as ¬ or ~ or ’

**•** AND: notated as ∧ or •

**•** OR: notated as ∨ or The NOT operator is unary, and the other operators are binary. The operands of a boolean expression can also be boolean expressions. The following are examples:

**Expression Description**

¬X NOT X

X ∧ Y X AND Y

X ∨ Y X OR Y ¬X ∨ Y (NOT X) OR Y ¬(X ∧ Y) NOT (X AND Y) X ∧¬Y X AND (NOT Y)

*NOT* The NOT operation reverses a boolean value. It can be written in mathematical notation as ¬X, where X is a variable (or expression) holding a value of true (T) or false (F). The follow- ing truth table shows all the possible outcomes of NOT using a variable **X**. Inputs are on the left side and outputs (shaded) are on the right side:

**X** ¬**X**

F T

T F

A truth table can use 0 for false and 1 for true.

*AND* The Boolean AND operation requires two operands, and can be expressed using the notation X∧ Y. The following truth table shows all the possible outcomes (shaded) for the values of X and Y:

**X Y X** ∧ **Y**

F F F

F T F

T F F

T T T

1.4 Boolean Expressions 23

The output is true only when both inputs are true. This corresponds to the logical AND used in compound boolean expressions in C++ and Java.

The AND operation is often carried out at the bit level in assembly language. In the following example, each bit in X is ANDed with its corresponding bit in Y:

X: 11111111 Y: 00011100 X ∧ Y: 00011100 As Figure 1-2 shows, each bit of the resulting value, 00011100, represents the result of ANDing the corresponding bits in X and Y.

Figure 1–2 ANDing the bits of two binary integers.

X:1 1 1 1 1 1 1 1

AND AND AND AND AND AND AND AND

Y:

0 0 0 1 1 1 0 0

X^Y: 0

0 0 1 1 1 0 0

*OR* The Boolean OR operation requires two operands, and is often expressed using the nota- tion **X** ∨ **Y**. The following truth table shows all the possible outcomes (shaded) for the values of X and Y:

**X Y X** ∨ **Y**

F F F

F T T

T F T

T T T

The output is false only when both inputs are false. This truth table corresponds to the logical OR used in compound boolean expressions in C++ and Java.

The OR operation is often carried out at the bit level. In the following example, each bit in X is ORed with its corresponding bit in Y, producing 11111100:

X: 11101100 Y: 00011100 X ∨ Y: 11111100

24 Chapter 1 • Basic Concepts

As shown in Figure 1-3, the bits are ORed individually, producing a corresponding bit in the result.

Figure 1–3 ORing the bits in two binary integers.

X:1 1 1 0 1 1 0 0

OR OR OR OR OR OR OR OR

Y:

0 0 0 1 1 1 0 0

X ^

Y: 1 1 1 1 1 1 0 0

*Operator Precedence Operator precedence rules* are used to indicate which operators exe- cute first in expressions involving multiple operators. In a boolean expression involving more than one operator, precedence is important. As shown in the following table, the NOT operator has the highest precedence, followed by AND and OR. You can use parentheses to force the ini- tial evaluation of an expression:

**Expression Order of Operations**

¬X ∨ Y NOT, then OR

¬(X ∨ Y) OR, then NOT

X ∨ (Y ∧ Z) AND, then OR

**1.4.1 Truth Tables for Boolean Functions** A *boolean function* receives boolean inputs and produces a boolean output. A truth table can be constructed for any boolean function, showing all possible inputs and outputs. The following are truth tables representing boolean functions having two inputs named X and Y. The shaded col- umn on the right is the function’s output:

1.4 Boolean Expressions 25

**Example 1:** ¬**X** ∨ **Y**

**Example 2: X** ∧ ¬**Y Example 3: (Y** ∧ **S)** ∨ **(X** ∧ ¬**S)X** ¬**X Y** ¬**X** ∨ **Y** F T F T

F T T T

T F F F

T F T T

**X Y** ¬**Y X** ∧¬**Y**

F F T F

F T F F

T F T T

T T F F

**X Y S Y** ∧ **S** ¬**S X** ∧¬**S (Y** ∧ **S)** ∨ **(X** ∧ ¬**S)**

F F F F T F F

F T F F T F F

T F F F T T T

T T F F T T T

F F T F F F F

F T T T F F T

T F T F F F F

T T T T F F T

26 Chapter 1 • Basic Concepts

The boolean function in Example 3 describes a *multiplexer*, a digital component that uses a selector bit (S) to select one of two outputs (X or Y). If S false, the function output (Z) is the same as X. If S true, the function output is the same as Y. Here is a block diagram of a multiplexer:

S

Xmux

Z Y

**1.4.2 Section Review**

1. Describe the following boolean expression: ¬X ∨ Y. 2. Describe the following boolean expression: (X ∧ Y). 3. What is the value of the boolean expression (T ∧ F) ∨ T ? 4. What is the value of the boolean expression ¬(F ∨ T) ? 5. What is the value of the boolean expression ¬F ∨ ¬T ?

**1.5 Chapter Summary** This book focuses on programming x86 processors, using the MS-Windows platform. We cover basic principles about computer architecture, machine language, and low-level programming. You will learn enough assembly language to test your knowledge on today’s most widely used microprocessor family.

Before reading this book, you should have completed a single college course or equivalent in computer programming.

An *assembler* is a program that converts source-code programs from assembly language into machine language. A companion program, called a linker, combines individual files created by an assembler into a single executable program. A third program, called a debugger, provides a way for a programmer to trace the execution of a program and examine the contents of memory.

You will create 32-bit and 64-bit programs for the most part, and 16-bit programs if you focus on the last four chapters.

You will learn the following concepts from this book: basic computer architecture applied to x86 (and Intel 64) processors; elementary boolean logic; how x86 processors manage memory; how high-level language compilers translate statements from their language into assembly lan- guage and native machine code; how high-level languages implement arithmetic expressions, loops, and logical structures at the machine level; and the data representation of signed and unsigned integers, real numbers, and character data.

Assembly language has a *one-to-one* relationship with machine language, in which a single assembly language instruction corresponds to one machine language instruction. Assembly lan- guage is not portable because it is tied to a specific processor family.

1.6 Key Terms 27

Programming languages are tools that you can use to create individual applications or parts of applications. Some applications, such as device drivers and hardware interface routines, are more suited to assembly language. Other applications, such as multiplatform commercial and scientific applications, are more easily written in high-level languages.

The *virtual machine* concept is an effective way of showing how each layer in a computer architecture represents an abstraction of a machine. Layers can be constructed of hardware or software, and programs written at any layer can be translated or interpreted by the next-lowest layer. The virtual machine concept can be related to real-world computer layers, including digi- tal logic, instruction set architecture, assembly language, and high-level languages.

Binary and hexadecimal numbers are essential notational tools for programmers working at the machine level. For this reason, you must understand how to manipulate and translate between number systems and how character representations are created by computers.

The following boolean operators were presented in this chapter: NOT, AND, and OR. A bool- ean expression combines a boolean operator with one or more operands. A truth table is an effective way to show all possible inputs and outputs of a boolean function.

**1.6 Key Terms** ASCII ASCII control characters ASCII digit string assembler assembly language binary digit string binary integer bit boolean algebra boolean expression boolean function character set code interpretation code point (Unicode) code translation debugger device driver digit string embedded systems application exabyte gigabyte hexadecimal digit string hexadecimal integer

high-level language instruction set architecture (ISA) Java Native Interface (JNI) kilobyte language portability least significant bit (LSB) machine language megabyte microcode interpreter microprogram Microsoft Macro Assembler (MASM) most significant bit (MSB) multiplexer null-terminated string octal digit string one-to-many relationship operator precedence petabyte registers signed binary integer terabyte Unicode Unicode Transformation Format (UTF)

28 Chapter 1 • Basic Concepts

unsigned binary integer UTF-8 UTF-16 UTF-32 virtual machine (VM)

virtual machine concept Visual Studio yottabyte zettabyte

**1.7 Review Questions and Exercises**

**1.7.1 Short Answer**

1. In an 8-bit binary number, which is the most significant bit (MSB)? 2. What is the decimal representation of each of the following unsigned binary integers?

a. 00110101 b. 10010110 c. 11001100 3. What is the sum of each pair of binary integers?

a. 10101111 + 11011011 b. 10010111 + 11111111 c. 01110101 + 10101100 4. Calculate binary 00001101 minus 00000111. 5. How many bits are used by each of the following data types?

a. word b. doubleword c. quadword d. double quadword 6. What is the minimum number of binary bits needed to represent each of the following

unsigned decimal integers? a. 4095 b. 65534 c. 42319 7. What is the hexadecimal representation of each of the following binary numbers?

a. 0011 0101 1101 1010 b. 1100 1110 1010 0011 c. 1111 1110 1101 1011 8. What is the binary representation of the following hexadecimal numbers?

a. 0126F9D4 b. 6ACDFA95 c. F69BDC2A 9. What is the unsigned decimal representation of each of the following hexadecimal integers?

a. 3A b. 1BF c. 1001

1.7 Review Questions and Exercises 29

10. What is the unsigned decimal representation of each of the following hexadecimal integers?

a. 62 b. 4B3 c. 29F 11. What is the 16-bit hexadecimal representation of each of the following signed decimal integers?

a.–24 b. –331 12. What is the 16-bit hexadecimal representation of each of the following signed decimal integers?

a.–21 b. –45 13. The following 16-bit hexadecimal numbers represent signed integers. Convert each to

decimal. a. 6BF9 b. C123 14. The following 16-bit hexadecimal numbers represent signed integers. Convert each to

decimal. a. 4CD2 b. 8230 15. What is the decimal representation of each of the following signed binary numbers?

a. 10110101 b. 00101010 c. 11110000 16. What is the decimal representation of each of the following signed binary numbers?

a. 10000000 b. 11001100 c. 10110111 17. What is the 8-bit binary (two’s-complement) representation of each of the following signed

decimal integers? a.5–b.–42 c. –16 18. What is the 8-bit binary (two’s-complement) representation of each of the following signed

decimal integers? a.–72 b.–98 c. – 26 19. What is the sum of each pair of hexadecimal integers?

a. 6B4 + 3FE b. A49 + 6BD

30 Chapter 1 • Basic Concepts

20. What is the sum of each pair of hexadecimal integers?

a. 7C4 3BE b. B69 7AD 21. What are the hexadecimal and decimal representations of the ASCII character capital B? 22. What are the hexadecimal and decimal representations of the ASCII character capital G? 23. *Challenge:* What is the largest decimal value you can represent, using a 129-bit unsigned

integer? 24. *Challenge:* What is the largest decimal value you can represent, using a 86-bit signed

integer? 25. Create a truth table to show all possible inputs and outputs for the boolean function

described by ¬( A ∨B ). 26. Create a truth table to show all possible inputs and outputs for the boolean function

described by ( A¬ B¬∧

). How would you describe the rightmost column of this table in relation to the table from question number 25? Have you heard of *De Morgan’s Theorem?* 27. If a boolean function has four inputs, how many rows are required for its truth table? 28. How many selector bits are required for a four-input multiplexer?

**1.7.2 Algorithm Workbench** Use any high-level programming language you wish for the following programming exercises. Do not call built-in library functions that accomplish these tasks automatically. (Examples are sprintf and sscanf from the Standard C library.)

1. Write a function that receives a string containing a 16-bit binary integer. The function must

return the string’s integer value. 2. Write a function that receives a string containing a 32-bit hexadecimal integer. The function

must return the string’s integer value. 3. Write a function that receives an integer. The function must return a string containing the

binary representation of the integer. 4. Write a function that receives an integer. The function must return a string containing the

hexadecimal representation of the integer. 5. Write a function that adds two digit strings in base *b*, where 2 ≤ *b* ≤

10 . Each string may contain as many as 1,000 digits. Return the sum in a string that uses the same number base. 6. Write a function that adds two hexadecimal strings, each as long as 1,000 digits. Return a

hexadecimal string that represents the sum of the inputs. 7. Write a function that multiplies a single hexadecimal digit by a hexadecimal digit string as

long as 1,000 digits. Return a hexadecimal string that represents the product.

1.7 Review Questions and Exercises 31

8. Write a Java program that contains the calculation shown below. Then, use the *javap –c* command to disassemble your code. Add comments to each line that provide your best guess as to its purpose.

int Y; int X = (Y + 4) \* 3;

9. Devise a way of subtracting unsigned binary integers. Test your technique by subtracting binary 00000101 from binary 10001000, producing 10000011. Test your technique with at least two other sets of integers, in which a smaller value is always subtracted from a larger one.

**Chapter End Notes** 1. Donald Knuth, MMIX, *A RISC Computer for the New Millennium*, Transcript of a lecture given at the Mas-

sachusetts Institute of Technology, December 30, 1999.

**2**

x86 Processor Architecture

2.1 General Concepts

2.1.1 Basic Microcomputer Design 2.1.2 Instruction Execution Cycle 2.1.3 Reading from Memory 2.1.4 Loading and Executing a Program 2.1.5 Section Review 2.2 32-Bit x86 Processors

2.2.1 Modes of Operation 2.2.2 Basic Execution Environment 2.2.3 x86 Memory Management 2.2.4 Section Review 2.3 64-Bit x86-64 Processors

2.3.1 64-Bit Operation Modes 2.3.2 Basic 64-Bit Execution Environment

2.4 Components of a Typical x86 Computer

2.4.1 Motherboard 2.4.2 Memory 2.4.3 Section Review 2.5 Input-Output System

2.5.1 Levels of I/O Access 2.5.2 Section Review 2.6 Chapter Summary 2.7 Key Terms 2.8 Review Questions

This chapter focuses on the underlying hardware associated with x86 assembly language. It may be said that assembly language is the ideal software tool for communicating directly with a machine. If that is true, then assembly programmers must be intimately familiar with the proces- sor’s internal architecture and capabilities. We will discuss some of the basic operations that take place inside the processor when instructions are executed. We will talk about how programs are loaded and executed by the operating system. A sample motherboard layout will give some insight into the hardware environment of x86 systems, and the chapter ends with a discussion of how layered input/output works between application programs and operating systems. All of the topics in this chapter provide the hardware foundation for you to begin writing assembly lan- guage programs.

32

2.1 General Concepts 33

**2.1 General Concepts** This chapter describes the architecture of the x86 processor family and its host computer sys- tem from a programmer’s point of view. Included in this group are all Intel IA-32 and Intel 64 processors, such as the Intel Pentium and Core-Duo, as well as the Advanced Micro Devices (AMD) processors, such as Athlon, Phenom, Opteron, and AMD64. Assembly language is a great tool for learning how a computer works, and it requires you to have a working knowledge of com- puter hardware. To that end, the concepts and details in this chapter will help you to understand the assembly language code you write.

We strike a balance between concepts applying to all microcomputer systems and specifics about x86 processors. You may work on various processors in the future, so we expose you to broad concepts. To avoid giving you a superficial understanding of machine architecture, we focus on specifics of the x86, which will give you a solid grounding when programming in assembly language.

If you want to learn more about the Intel IA-32 architecture, read *the Intel 64 and IA-32 Architec- tures Software Developer’s Manual, Volume 1: Basic Architecture.* It’s a free download from the Intel web site (www.intel.com).

**2.1.1 Basic Microcomputer Design** Figure 2-1 shows the basic design of a hypothetical microcomputer. The *central processor unit* (CPU), where calculations and logical operations take place, contains a limited number of storage locations named *registers*, a high-frequency clock, a control unit, and an arithmetic logic unit.

**•** The *clock* synchronizes the internal operations of the CPU with other system components.

**•** The *control unit* (CU) coordinates the sequencing of steps involved in executing machine instructions.

**•** The *arithmetic logic unit* (ALU) performs arithmetic operations such as addition and subtrac- tion and logical operations such as AND, OR, and NOT. The CPU is attached to the rest of the computer via pins attached to the CPU socket in the computer’s motherboard. Most pins connect to the data bus, the control bus, and the address bus. The *memory storage unit* is where instructions and data are held while a computer program is running. The storage unit receives requests for data from the CPU, transfers data from random access memory (RAM) to the CPU, and transfers data from the CPU into memory. All process- ing of data takes place within the CPU, so programs residing in memory must be copied into the CPU before they can execute. Individual program instructions can be copied into the CPU one at a time, or groups of instructions can be copied together.

A *bus* is a group of parallel wires that transfer data from one part of the computer to another. A computer system usually contains four bus types: data, I/O, control, and address. The *data bus* transfers instructions and data between the CPU and memory. The I/O bus transfers data

34 Chapter 2 • x86 Processor Architecture

Figure 2–1 Block diagram of a microcomputer.

Data bus, I/O bus

Registers

**Central processor unit (CPU)**

Memory storage unit

I/O device #1

Control bus

Address bus

between the CPU and the system input/output devices. The *control bus* uses binary signals to synchronize actions of all devices attached to the system bus. The *address bus* holds the addresses of instructions and data when the currently executing instruction transfers data between the CPU and memory.

*Clock* Each operation involving the CPU and the system bus is synchronized by an internal clock pulsing at a constant rate. The basic unit of time for machine instructions is a *machine cycle* (or *clock cycle*). The length of a clock cycle is the time required for one complete clock pulse. In the following figure, a clock cycle is depicted as the time between one falling edge and the next:

The duration of a clock cycle is calculated as the reciprocal of the clock’s speed, which in turn is measured in oscillations per second. A clock that oscillates 1 billion times per second (1 GHz), for example, produces a clock cycle with a duration of one billionth of a second (1 nanosecond).

A machine instruction requires at least one clock cycle to execute, and a few require in excess of 50 clocks (the multiply instruction on the 8088 processor, for example). Instructions requiring memory access often have empty clock cycles called *wait states* because of the differences in the speeds of the CPU, the system bus, and memory circuits.

**2.1.2 Instruction Execution Cycle** A single machine instruction does not just magically execute all at once. The CPU has to go through a predefined sequence of steps to execute a machine instruction, called the *instruction execution cycle*. Let’s assume that the instruction pointer register holds the address of the instruction we want to execute. Here are the steps to execute it:

I/O device #2

ALU CU

Clock One cycle

10

2.1 General Concepts 35

1. First, the CPU has to **fetch the instruction** from an area of memory called the *instruction*

*queue*. Right after doing this, it increments the instruction pointer. 2. Next, the CPU **decodes** the instruction by looking at its binary bit pattern. This bit pattern

might reveal that the instruction has operands (input values). 3. If operands are involved, the CPU **fetches the operands** from registers and memory. Some-

times, this involves address calculations. 4. Next, the CPU **executes** the instruction, using any operand values it fetched during the earlier

step. It also updates a few status flags, such as Zero, Carry, and Overflow. 5. Finally, if an output operand was part of the instruction, the CPU **stores the result** of its exe-

cution in the operand. We usually simplify this complicated-sounding process to three basic steps: **Fetch**, **Decode**, and **Execute**. An *operand* is a value that is either an input or an output to an opera- tion. For example, the expression Z = X + Y has two input operands (X and Y) and a single output operand (Z).

A block diagram showing data flow within a typical CPU is shown in Figure 2-2. The diagram helps to show relationships between components that interact during the instruction execution cycle. In order to read program instructions from memory, an address is placed on the address bus. Next, the memory controller places the requested code on the data bus, making the code available inside the code cache. The instruction pointer’s value determines which instruction will be executed next. The instruction is analyzed by the instruction decoder, causing the appropriate

Figure 2–2 Simplified CPU block diagram.

Memory

s uba taDs

ubs serddACode cache

Code

Instruction decoder

Data

Registers ALU

Instruction pointer

Control unit

Floating-point unit

Data cache

36 Chapter 2 • x86 Processor Architecture

digital signals to be sent to the control unit, which coordinates the ALU and floating-point unit. Although the control bus is not shown in this figure, it carries signals that use the system clock to coordinate the transfer of data between the different CPU components.

**2.1.3 Reading from Memory** As a rule, computers read memory much more slowly than they access internal registers. This is because reading a single value from memory involves four separate steps:

1. Place the address of the value you want to read on the address bus. 2. Assert (change the value of) the processor’s RD *(read)* pin. 3. Wait one clock cycle for the memory chips to respond. 4. Copy the data from the data bus into the destination operand.

Each of these steps generally requires a single *clock cycle*, a measurement of time based on a clock that ticks inside the processor at a regular rate. Computer CPUs are often described in terms of their clock speeds. A speed of *1.2 GHz*, for example, means the clock ticks, or oscil- lates, 1.2 billion times per second. So, 4 clock cycles go by fairly fast, considering each one lasts for only 1/1,200,000,000th of a second. Still, that’s much slower than the CPU registers, which are usually accessed in only one clock cycle.

Fortunately, CPU designers figured out a long time ago that computer memory creates a speed bottleneck because most programs have to access variables. They came up with a clever way to reduce the amount of time spent reading and writing memory—they store the most recently used instructions and data in high-speed memory called *cache*. The idea is that a pro- gram is more likely to want to access the same memory and instructions repeatedly, so cache keeps these values where they can be accessed quickly. Also, when the CPU begins to execute a program, it can look ahead and load the next thousand instructions (for example) into cache, on the assumption that these instructions will be needed fairly soon. If there happens to be a loop in that block of code, the same instructions will be in cache. When the processor is able to find its data in cache memory, we call that a *cache hit*. On the other hand, if the CPU tries to find some- thing in cache and it’s not there, we call that a *cache miss*.

Cache memory for the x86 family comes in two types. *Level-1 cache* (or *primary cache*) is stored right on the CPU. *Level-2 cache* (or *secondary cache*) is a little bit slower, and attached to the CPU by a high-speed data bus. The two types of cache work together in an optimal way.

There’s a reason why cache memory is faster than conventional RAM—it’s because cache memory is constructed from a special type of memory chip called *static RAM*. It’s expensive, but it does not have to be constantly refreshed in order to keep its contents. On the other hand, con- ventional memory, known as *dynamic RAM*, must be refreshed constantly. It’s much slower, but cheaper.

**2.1.4 Loading and Executing a Program** Before a program can run, it must be loaded into memory by a utility known as a *program loader*. After loading, the operating system must point the CPU to the program’s *entry point*, which is the address at which the program is to begin execution. The following steps break this process down in more detail:

2.2 32-Bit x86 Processors 37

**•** The operating system (OS) searches for the program’s filename in the current disk directory. If it cannot find the name there, it searches a predetermined list of directories (called *paths*) for the filename. If the OS fails to find the program filename, it issues an error message.

**•** If the program file is found, the OS retrieves basic information about the program’s file from the disk directory, including the file size and its physical location on the disk drive.

**•** The OS determines the next available location in memory and loads the program file into mem- ory. It allocates a block of memory to the program and enters information about the program’s size and location into a table (sometimes called a *descriptor table*). Additionally, the OS may adjust the values of pointers within the program so they contain addresses of program data.

**•** The OS begins execution of the program’s first machine instruction (its entry point). As soon as the program begins running, it is called a *process*. The OS assigns the process an identifi- cation number (*process ID*), which is used to keep track of it while running.

**•** The *process* runs by itself. It is the OS’s job to track the execution of the process and to respond to requests for system resources. Examples of resources are memory, disk files, and input-output devices.

**•** When the process ends, it is removed from memory.

***Tip:*** If you’re using any version of Microsoft Windows, press *Ctrl-Alt-Delete* and select the *Task Manager* item. The Task Manager window lets you view lists of Applications and Pro- cesses. Applications are the names of complete programs currently running, such as Windows Explorer or Microsoft Visual C++. When you click on the *Processes* tab, you see a long list of process names. Each of those processes is a small program running independently of all the others. You can continuously track the amount of CPU time and memory used by each pro- cess. In some cases, you can shut down a process by selecting its name and pressing the *Delete* key.

**2.1.5 Section Review**

1. The central processor unit (CPU) contains registers and what other basic elements? 2. The central processor unit is connected to the rest of the computer system using what three

buses? 3. Why does memory access take more machine cycles than register access? 4. What are the three basic steps in the instruction execution cycle? 5. Which two additional steps are required in the instruction execution cycle when a memory

operand is used?

**2.2 32-Bit x86 Processors** In this section, we focus on the basic architectural features of all x86 processors. This includes members of the Intel IA-32 family as well as all 32-bit AMD processors.

**2.2.1 Modes of Operation** x86 processors have three primary modes of operation: protected mode, real-address mode, and system management mode. A sub-mode, named *virtual-8086*, is a special case of protected mode. Here are short descriptions of each:

38 Chapter 2 • x86 Processor Architecture

*Protected Mode* Protected mode is the native state of the processor, in which all instructions and features are available. Programs are given separate memory areas named *segments*, and the processor prevents programs from referencing memory outside their assigned segments.

*Virtual-8086 Mode* While in protected mode, the processor can directly execute real-address mode software such as MS-DOS programs in a safe environment. In other words, if a program crashes or attempts to write data into the system memory area, it will not affect other programs running at the same time. A modern operating system can execute multiple separate virtual-8086 sessions at the same time.

*Real-Address Mode* Real-address mode implements the programming environment of an early Intel processor with a few extra features, such as the ability to switch into other modes. This mode is useful if a program requires direct access to system memory and hardware devices.

*System Management Mode* System management mode (SMM) provides an operating sys- tem with a mechanism for implementing functions such as power management and system secu- rity. These functions are usually implemented by computer manufacturers who customize the processor for a particular system setup.

**2.2.2 Basic Execution Environment**

***Address Space*** In 32-bit protected mode, a task or program can address a linear address space of up to 4 GBytes. Beginning with the P6 processor, a technique called *extended physical addressing* allows a total of 64 GBytes of physical memory to be addressed. Real-address mode programs, on the other hand, can only address a range of 1 MByte. If the processor is in protected mode and running multiple programs in virtual-8086 mode, each program has its own 1-MByte memory area.

Figure 2–3 Basic program execution registers.

**32-Bit General-Purpose Registers**

EAX

EBP

EBX

ESP

ECX

ESI

EDX

EDI

**16-Bit Segment Registers**

EFLAGS

CSESSSFSEIP DS GS

2.2 32-Bit x86 Processors 39

***Basic Program Execution Registers*** *Registers* are high-speed storage locations directly inside the CPU, designed to be accessed at much higher speed than conventional memory. When a processing loop is optimized for speed, for example, loop counters are held in registers rather than variables. Figure 2-3 shows the *basic program execution registers*. There are eight general-purpose registers, six segment registers, a processor status flags register (EFLAGS), and an instruction pointer (EIP).

*General-Purpose Registers* The *general-purpose registers* are primarily used for arith- metic and data movement. As shown in Figure 2-4, the lower 16 bits of the EAX register can be referenced by the name AX.

Figure 2–4 General-purpose registers.

Portions of some registers can be addressed as 8-bit values. For example, the AX register has an 8-bit upper half named AH and an 8-bit lower half named AL. The same overlapping relationship exists for the EAX, EBX, ECX, and EDX registers:

The remaining general-purpose registers can only be accessed using 32-bit or 16-bit names, as shown in the following table:

8

AX

EAX

8

AH AL

8 bits 8 bits

16 bits

32 bits

**32-Bit 16-Bit 8-Bit (High) 8-Bit (Low)**

EAX AX AH AL

EBX BX BH BL

ECX CX CH CL

EDX DX DH DL

**32-Bit 16-Bit**

ESI SI

EDI DI

EBP BP

ESP SP

40 Chapter 2 • x86 Processor Architecture

*Specialized Uses* Some general-purpose registers have specialized uses:

**•** EAX is automatically used by multiplication and division instructions. It is often called the *extended accumulator* register.

**•** The CPU automatically uses ECX as a loop counter.

**•** ESP addresses data on the stack (a system memory structure). It is rarely used for ordinary arithmetic or data transfer. It is often called the *extended stack pointer* register.

**•** ESI and EDI are used by high-speed memory transfer instructions. They are sometimes called the *extended source index* and *extended destination index* registers.

**•** EBP is used by high-level languages to reference function parameters and local variables on the stack. It should not be used for ordinary arithmetic or data transfer except at an advanced level of programming. It is often called the *extended frame pointer* register.

*Segment Registers* In real-address mode, 16-bit segment registers indicate base addresses of preassigned memory areas named *segments*. In protected mode, segment registers hold pointers to segment descriptor tables. Some segments hold program instructions (code), others hold vari- ables (data), and another segment named the *stack segment* holds local function variables and function parameters.

*Instruction Pointer* The EIP, or *instruction pointer*, register contains the address of the next instruction to be executed. Certain machine instructions manipulate EIP, causing the program to branch to a new location.

*EFLAGS Register* The EFLAGS (or just *Flags*) register consists of individual binary bits that control the operation of the CPU or reflect the outcome of some CPU operation. Some instructions test and manipulate individual processor flags.

A flag is *set* when it equals 1; it is *clear* (or reset) when it equals 0.

*Control Flags* Control flags control the CPU’s operation. For example, they can cause the CPU to break after every instruction executes, interrupt when arithmetic overflow is detected, enter virtual-8086 mode, and enter protected mode.

Programs can set individual bits in the EFLAGS register to control the CPU’s operation. Examples are the *Direction* and *Interrupt* flags.

*Status Flags* The status flags reflect the outcomes of arithmetic and logical operations per- formed by the CPU. They are the Overflow, Sign, Zero, Auxiliary Carry, Parity, and Carry flags. Their abbreviations are shown immediately after their names:

**•** The **Carry** flag (CF) is set when the result of an *unsigned* arithmetic operation is too large to fit into the destination.

**•** The **Overflow** flag (OF) is set when the result of a *signed* arithmetic operation is too large or too small to fit into the destination.

**•** The **Sign** flag (SF) is set when the result of an arithmetic or logical operation generates a negative result.

**•** The **Zero** flag (ZF) is set when the result of an arithmetic or logical operation generates a result of zero.

2.2 32-Bit x86 Processors 41

**•** The **Auxiliary Carry** flag (AC) is set when an arithmetic operation causes a carry from bit 3 to bit 4 in an 8-bit operand.

**•** The **Parity** flag (PF) is set if the least-significant byte in the result contains an even number of 1 bits. Otherwise, PF is clear. In general, it is used for error checking when there is a possi- bility that data might be altered or corrupted.

***MMX Registers*** MMX technology improves the performance of Intel processors when implementing advanced multimedia and communications applications. The eight 64-bit MMX registers support special instructions called SIMD (*Single-Instruction, Multiple-Data*). As the name implies, MMX instructions operate in parallel on the data values contained in MMX registers. Although they appear to be separate registers, the MMX register names are in fact aliases to the same registers used by the floating-point unit.

***XMM Registers*** The x86 architecture also contains eight 128-bit registers called XMM registers. They are used by streaming SIMD extensions to the instruction set.

*Floating-Point Unit* The *floating-point unit* (FPU) performs high-speed floating-point arith- metic. At one time a separate coprocessor chip was required for this. From the Intel486 onward, the FPU has been integrated into the main processor chip. There are eight floating-point data registers in the FPU, named ST(0), ST(1), ST(2), ST(3), ST(4), ST(5), ST(6), and ST(7). The remaining control and pointer registers are shown in Figure 2-5.

Figure 2–5 Floating-point unit registers.

**80-Bit Data Registers**

**48-Bit Pointer Registers** ST(0)

ST(1)

ST(2)

FPU data pointer

ST(3)

ST(4)

ST(5)

Tag register

ST(6)

Control register

ST(7)

Status register

**2.2.3 x86 Memory Management** x86 processors manage memory according to the basic modes of operation discussed in Section 2.2.1. Protected mode is the most robust and powerful, but it does restrict application programs from directly accessing system hardware.

FPU instruction pointer

Opcode register

**16-Bit Control Registers**

42 Chapter 2 • x86 Processor Architecture

In ***real-address*** mode, only 1 MByte of memory can be addressed, from hexadecimal 00000 to FFFFF. The processor can run only one program at a time, but it can momentarily interrupt that program to process requests (called *interrupts*) from peripherals. Application programs are permitted to access any memory location, including addresses that are linked directly to system hardware. The MS-DOS operating system runs in real-address mode, and Windows 95 and 98 can be booted into this mode.

In ***protected*** mode, the processor can run multiple programs at the same time. It assigns each process (running program) a total of 4 GByte of memory. Each program can be assigned its own reserved memory area, and programs are prevented from accidentally accessing each other’s code and data. MS-Windows and Linux run in protected mode.

In ***virtual-8086*** mode, the computer runs in protected mode and creates a virtual-8086 machine with its own 1-MByte address space that simulates an 80x86 computer running in real- address mode. Windows NT and 2000, for example, create a virtual-8086 machine when you open a *Command* window. You can run many such windows at the same time, and each is pro- tected from the actions of the others. Some MS-DOS programs that make direct references to computer hardware will not run in this mode under Windows NT, 2000, and XP.

Chapter 11 explains many more details of both real-address mode and protected mode.

**2.2.4 Section Review**

1. What are the x86 processor’s three basic modes of operation? 2. Name all eight 32-bit general-purpose registers. 3. Name all six segment registers. 4. What special purpose does the ECX register serve?

**2.3 64-Bit x86-64 Processors** In this section, we focus on the basic architectural details of all 64-bit processors that use the x86-64 instruction set. This group the Intel 64 and AMD64 processor families. The instruction set is a 64-bit extension of the x86 instruction set we’ve already looked at. Here are some of the essential features:

1. It is backward-compatible with the x86 instruction set. 2. Addresses are 64 bits long, allowing for a virtual address space of size 264 bytes. In current

chip implementations, only the lowest 48 bits are used. 3.It can use 64-bit general-purpose registers, allowing instructions to have 64-bit integer

operands. 4. It uses eight more general-purpose registers than the x86. 5. It uses a 48-bit physical address space, which supports up to 256 terabytes of RAM.

On the other hand, when running in native 64-bit mode, these processors do not support 16-bit real mode or virtual-8086 mode. (There is a *legacy mode* that still supports 16-bit pro- gramming, but it is not available in 64-bit versions of Microsoft Windows.)

***Note:*** Although *x86-64* refers to an instruction set, we will from this point on treat it as a processor type. For the purpose of learning assembly language, it is not necessary to consider hardware implementation differences between processors that support x86-64.

2.3 64-Bit x86-64 Processors 43

The first Intel processor to use x86-64 was the Xeon, followed by a host of other processors, including Core i5 and Core i7 processors. Examples of AMD’s processors that use x86-64 are Opteron and Athlon 64.

You might also have heard of another 64-bit architecture from Intel known as *IA-64*, later renamed to *Itanium*. The IA-64 instruction set is completely different from x86 and x86-64. Ita- nium processors are often used for high-performance database and network servers.

**2.3.1 64-Bit Operation Modes** The Intel 64 architecture introduces a new mode named *IA-32e*. Technically it contains two sub- modes, named *compatibility mode* and *64-bit mode*. But it’s easier to refer to these as modes rather than submodes, so we will do that from now on.

***Compatibility Mode*** When running in compatibility mode, existing 16-bit and 32-bit applications can usually run without being recompiled. However, 16-bit Windows (Win16) and DOS applications will not run in 64-bit Microsoft Windows. Unlike earlier versions of Windows, 64-bit Windows does not have a virtual DOS machine subsystem to take advantage of the processor’s ability to switch into virtual-8086 mode.

***64-Bit Mode*** In 64-bit mode, the processor runs applications that use the 64-bit linear address space. This is the native mode for 64-bit Microsoft Windows. This mode enables 64-bit instruction operands.

**2.3.2 Basic 64-Bit Execution Environment** In 64-bit mode, addresses can theoretically be as large as 64-bits, although processors currently only support 48 bits for addresses. In terms of registers, the following are the most important differences from 32-bit processors:

**•** Sixteen 64-bit general purpose registers (in 32-bit mode, you have only eight general-purpose registers)

**•** Eight 80-bit floating-point registers

**•** A 64-bit status flags register named RFLAGS (only the lower 32 bits are used)

**•** A 64-bit instruction pointer named RIP As you may recall, the 32-bit flags and instruction pointers are named EFLAGS and EIP. In addition, there are some specialized registers for multimedia processing we mentioned when talking about the x86 processor:

**•** Eight 64-bit MMX registers

**•** Sixteen 128-bit XMM registers (in 32-bit mode, you have only 8 of these)

***General-Purpose Registers*** The general-purpose registers, introduced when we described 32-bit processors, are the basic operands for instructions that do arithmetic, move data, and loop through data. The general- purpose registers can access 8-bit, 16-bit, 32-bit, or 64-bit operands (with a special prefix).

In 64-bit mode, the default operand size is 32 bits and there are eight general-purpose regis- ters. By adding the REX (register extension) prefix to each instruction, however, the operands can be 64 bits long and a total of 16 general-purpose registers become available. You have all the

44 Chapter 2 • x86 Processor Architecture

same registers as in 32-bit mode, plus eight numbered registers, R8 through R15. Table 2-1 shows which registers are available when the REX prefix is enabled.

Table 2-1 Operand Sizes in 64-Bit Mode When REX Is Enabled.

**Operand Size Available Registers**

8 bits AL, BL, CL, DL, DIL, SIL, BPL, SPL, R8L, R9L, R10L, R11L, R12L, R13L, R14L, R15L

16 bits AX, BX, CX, DX, DI, SI, BP, SP, R8W, R9W, R10W, R11W, R12W, R13W, R14W, R15W

32 bits EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D, R9D, R10D, R11D, R12D, R13D,

R14D, R15D

64 bits RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8, R9, R10, R11, R12, R13, R14, R15

Here are a few more details to remember:

**•** In 64-bit mode, a single instruction cannot access both a high-byte register, such as AH, BH, CH, and DH, and at the same time, the low byte of one of the new byte registers (such as DIL).

**•** The 32-bit EFLAGS register is replaced by a 64-bit RFLAGS register in 64-bit mode. The two registers share the same lower 32 bits, and the upper 32 bits of RFLAGS are not used.

**•** The status flags are the same in 32-bit mode and 64-bit mode.

**2.4 Components of a Typical x86 Computer** Let us look at how the x86 integrates with other components by examining a typical mother- board configuration and the set of chips that surround the CPU. Then we will discuss memory, I/O ports, and common device interfaces. Finally, we will show how assembly language pro- grams can perform I/O at different levels of access by tapping into system hardware, firmware, and by calling functions in the operating system.

**2.4.1 Motherboard** The heart of a microcomputer is its *motherboard*, a flat circuit board onto which are placed the computer’s CPU, supporting processors (*chipset*), main memory, input-output connectors, power supply connectors, and expansion slots. The various components are connected to each other by a *bus*, a set of wires etched directly on the motherboard. Dozens of motherboards are available on the PC market, varying in expansion capabilities, integrated components, and speed. The following components have traditionally been found on PC motherboards:

**•** A CPU socket. Sockets are different shapes and sizes, depending on the type of processor they support

**•** Memory slots (SIMM or DIMM), holding small plug-in memory boards

**•** BIOS (*basic input–output system*) computer chips, holding system software

**•** CMOS RAM, with a small circular battery to keep it powered

**•** Connectors for mass-storage devices such as hard drives and CD-ROMs

**•** USB connectors for external devices

**•** Keyboard and mouse ports

2.4 Components of a Typical x86 Computer 45

**•** PCI bus connectors for sound cards, graphics cards, data acquisition boards, and other input– output devices The following components are optional:

**•** Integrated sound processor

**•** Parallel and serial device connectors

**•** Integrated network adapter

**•** AGP bus connector for a high-speed video card Following are some important support processors in a typical system:

**•** The *Floating-Point Unit* (FPU) handles floating-point and extended integer calculations.

**•** The 8284/82C284 *Clock Generator*, known simply as the *clock*, oscillates at a constant speed. The clock generator synchronizes the CPU and the rest of the computer.

**•** The 8259A *Programmable Interrupt Controller* (PIC) handles external interrupts from hard- ware devices, such as the keyboard, system clock, and disk drives. These devices interrupt the CPU and make it process their requests immediately.

**•** The 8253 *Programmable Interval Timer/Counter* interrupts the system 18.2 times per second, updates the system date and clock, and controls the speaker. It is also responsible for con- stantly refreshing memory because RAM memory chips can remember their data for only a few milliseconds.

**•** The 8255 *Programmable Parallel Port* transfers data to and from the computer using the IEEE Parallel Port interface. This port is commonly used for printers, but it can be used with other input–output devices as well.

***PCI and PCI Express Bus Architectures*** The **PCI** (*Peripheral Component Interconnect*) bus provides a connecting bridge between the CPU and other system devices such as hard drives, memory, video controllers, sound cards, and network controllers. More recently, the *PCI Express* bus provides two-way serial connections between devices, memory, and the processor. It carries data in packets, similar to networks, in separate “lanes.” It is widely supported by graphics controllers, and can transfer data at very high speeds.

***Motherboard Chipset*** A *motherboard chipset* is a collection of processor chips designed to work together on a specific type of motherboard. Various chipsets have features that increase processing power, multimedia capabilities, or reduce power consumption. The *Intel P965 Express Chipset* can be used as an example. It is used in desktop PCs, with either an Intel Core 2 Duo or a Pentium D processor. Here are some of its features:

**•** Intel *Fast Memory Access* uses an updated Memory Controller Hub (MCH). It can access dual-channel DDR2 memory, at an 800 MHz clock speed.

**•** An I/O Controller Hub (Intel ICH8/R/DH) uses Intel Matrix Storage Technology (MST) to support multiple Serial ATA devices (disk drives).

**•** Support for multiple USB ports, multiple PCI express slots, networking, and Intel Quiet Sys- tem technology.

**•** A high definition audio chip provides digital sound capabilities.