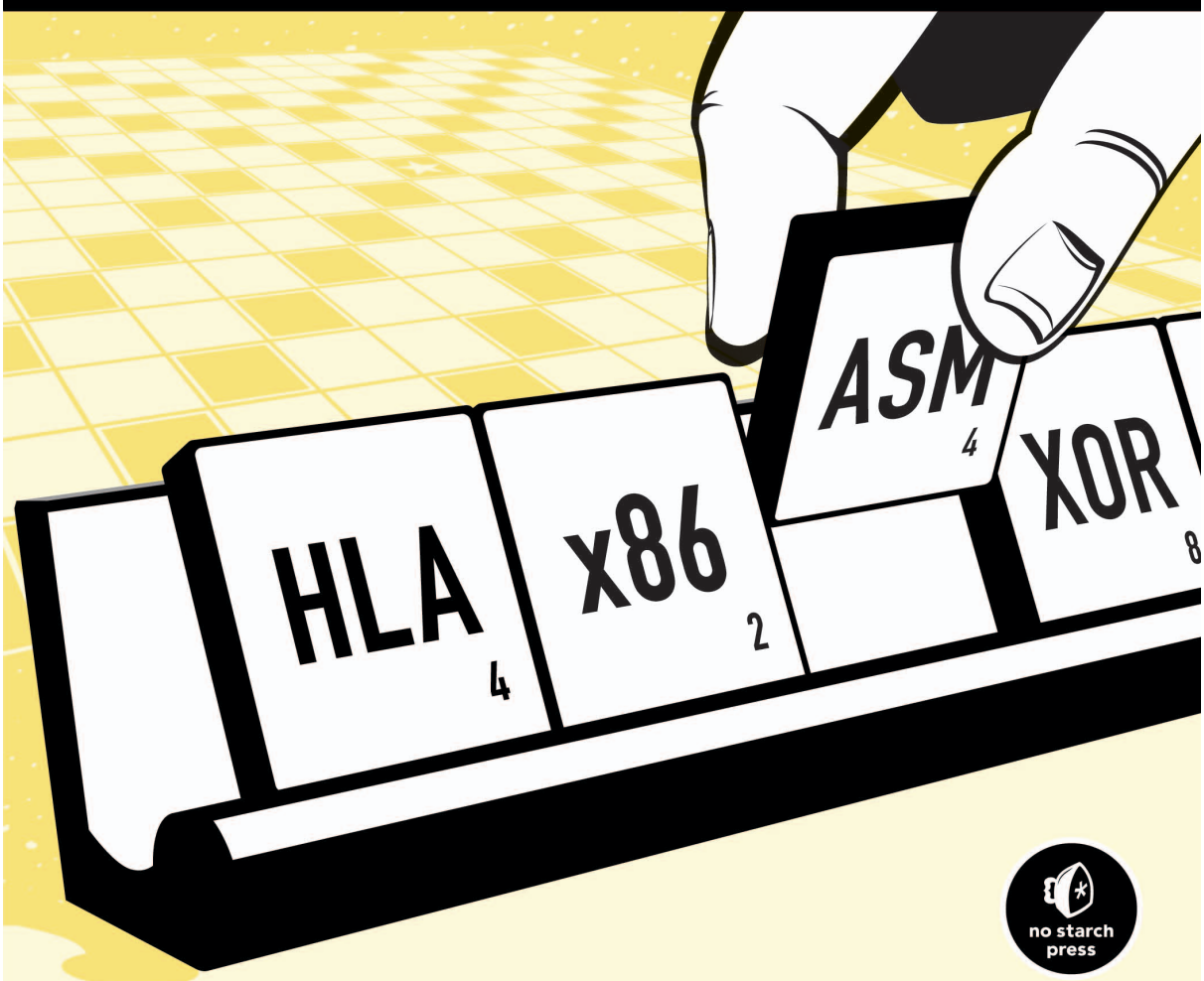
THE ART OF ASSEMBLY LANGUAGE

2ND EDITION

RANDALL HYDE



no starch press

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1027, Page 000001
IPR2024-00627 (Amazon.com, Inc., Amazon.com Services LLC v. Nokia Technologies Oy)

PRAISE FOR THE FIRST EDITION OF *THE ART OF ASSEMBLY LANGUAGE*

"My flat-out favorite book of 2003 was Randall Hyde's *The Art of Assembly Language*."
—SOFTWARE DEVELOPER TIMES

"You would be hard-pressed to find a better book on assembly out there."
—SECURITY-FORUMS.COM

"This is a large book that is comprehensive and detailed. The author and publishers have done a remarkable job of packing so much in without making the explanatory text too terse. If you want to use assembly language, or add it to your list of programming skills, this is the book to have."
—BOOK NEWS (AUSTRALIA)

"Allows the reader to focus on what's really important, writing programs without hitting the proverbial brick wall that dooms many who attempt to learn assembly language to failure. . . . Topics are discussed in detail and no stone is left unturned."
—MAINE LINUX USERS GROUP-CENTRAL

"The text is well authored and easy to understand. The tutorials are thoroughly explained, and the example code segments are superbly commented."
—TECHIMO

"This big book is a very complete treatment [of assembly language]."
—MSTATION.ORG

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1027, Page 000003

THE ART OF ASSEMBLY LANGUAGE, 2ND EDITION

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1027, Page 000005

THE ART OF ASSEMBLY LANGUAGE

2ND EDITION

by Randall Hyde

no starch press

San Francisco

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1027, Page 000007

THE ART OF ASSEMBLY LANGUAGE, 2ND EDITION. Copyright © 2010 by Randall Hyde.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

14 13 12 11 10 1 2 3 4 5 6 7 8 9

Printed in Canada

ISBN-10: 1-59327-207-3
ISBN-13: 978-1-59327-207-4

Publisher: William Pollock
Production Editor: Riley Hoffman
Cover and Interior Design: Octopod Studios
Developmental Editor: William Pollock
Technical Reviewer: Nathan Baker
Copyeditor: Linda Recktenwald
Compositor: Susan Glinert Stevens
Proofreader: Nancy Bell

For information on book distributors or translations, please contact No Starch Press, Inc. directly:

No Starch Press, Inc.
555 De Haro Street, Suite 250, San Francisco, CA 94107
phone: 415.863.9900; fax: 415.863.9950; info@nostarch.com; www.nostarch.com

Library of Congress Cataloging-in-Publication Data

```
Hyde, Randall.
 The art of Assembly language / by Randall Hyde. -- 2nd ed.
      p. cm.
 ISBN 978-1-59327-207-4 (pbk.)
1.  Assembler language (Computer program language) 2.  Programming languages (Electronic computers)  I. Title.
QA76.73.A8H97 2010
 005.13'6--dc22
                                                                2009040777
```

No Starch Press and the No Starch Press logo are registered trademarks of No Starch Press, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The information in this book is distributed on an "As Is" basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

BRIEF CONTENTS

Acknowledgments xix

Chapter 1: Hello, World of Assembly Language 1

Chapter 2: Data Representation 53

Chapter 3: Memory Access and Organization 111

Chapter 4: Constants, Variables, and Data Types 155

Chapter 5: Procedures and Units 255

Chapter 6: Arithmetic 351

Chapter 7: Low-Level Control Structures 413

Chapter 8: Advanced Arithmetic 477

Chapter 9: Macros and the HLA Compile-Time Language 551

Chapter 10: Bit Manipulation 599

Chapter 11: The String Instructions 633

Chapter 12: Classes and Objects 651

Appendix: ASCII Character Set 701

Index 705

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1027, Page 000009

CONTENTS IN DETAIL

ACKNOWLEDGMENTS **xix**

1
HELLO, WORLD OF ASSEMBLY LANGUAGE **1**

1.1 The Anatomy of an HLA Program ... 2
1.2 Running Your First HLA Program ... 4
1.3 Some Basic HLA Data Declarations ... 5
1.4 Boolean Values ... 7
1.5 Character Values ... 8
1.6 An Introduction to the Intel 80x86 CPU Family ... 8
1.7 The Memory Subsystem ... 11
1.8 Some Basic Machine Instructions ... 14
1.9 Some Basic HLA Control Structures ... 17
1.9.1 Boolean Expressions in HLA Statements ... 18
1.9.2 The HLA if..then..elseif..else..endif Statement ... 20
1.9.3 Conjunction, Disjunction, and Negation in Boolean Expressions ... 22
1.9.4 The while..endwhile Statement ... 24
1.9.5 The for..endfor Statement ... 25
1.9.6 The repeat..until Statement ... 26
1.9.7 The break and breakif Statements ... 27
1.9.8 The forever..endfor Statement ... 27
1.9.9 The try..exception..endtry Statement ... 28
1.10 Introduction to the HLA Standard Library ... 32
1.10.1 Predefined Constants in the stdio Module ... 33
1.10.2 Standard In and Standard Out ... 34
1.10.3 The stdout.newln Routine ... 35
1.10.4 The stdout.puti*X* Routines ... 35
1.10.5 The stdout.puti*X*Size Routines ... 35
1.10.6 The stdout.put Routine ... 37
1.10.7 The stdin.getc Routine ... 38
1.10.8 The stdin.geti*X* Routines ... 39
1.10.9 The stdin.readLn and stdin.flushInput Routines ... 40
1.10.10 The stdin.get Routine ... 41
1.11 Additional Details About try..endtry ... 42
1.11.1 Nesting try..endtry Statements ... 43
1.11.2 The unprotected Clause in a try..endtry Statement ... 45
1.11.3 The anyexception Clause in a try..endtry Statement ... 48
1.11.4 Registers and the try..endtry Statement ... 48
1.12 High-Level Assembly Language vs. Low-Level Assembly Language ... 50
1.13 For More Information ... 51

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1027, Page 000011

2
DATA REPRESENTATION 53

2.1 Numbering Systems ... 54
2.1.1 A Review of the Decimal System ... 54
2.1.2 The Binary Numbering System ... 54
2.1.3 Binary Formats ... 55
2.2 The Hexadecimal Numbering System ... 56
2.3 Data Organization ... 58
2.3.1 Bits ... 58
2.3.2 Nibbles ... 59
2.3.3 Bytes ... 60
2.3.4 Words ... 61
2.3.5 Double Words ... 62
2.3.6 Quad Words and Long Words ... 63
2.4 Arithmetic Operations on Binary and Hexadecimal Numbers ... 64
2.5 A Note About Numbers vs. Representation ... 65
2.6 Logical Operations on Bits ... 67
2.7 Logical Operations on Binary Numbers and Bit Strings ... 70
2.8 Signed and Unsigned Numbers ... 72
2.9 Sign Extension, Zero Extension, Contraction, and Saturation ... 76
2.10 Shifts and Rotates ... 80
2.11 Bit Fields and Packed Data ... 85
2.12 An Introduction to Floating-Point Arithmetic ... 89
2.12.1 IEEE Floating-Point Formats ... 93
2.12.2 HLA Support for Floating-Point Values ... 96
2.13 Binary-Coded Decimal Representation ... 100
2.14 Characters ... 101
2.14.1 The ASCII Character Encoding ... 101
2.14.2 HLA Support for ASCII Characters ... 105
2.15 The Unicode Character Set ... 109
2.16 For More Information ... 110

3
MEMORY ACCESS AND ORGANIZATION 111

3.1 The 80x86 Addressing Modes ... 112
3.1.1 80x86 Register Addressing Modes ... 112
3.1.2 80x86 32-Bit Memory Addressing Modes ... 113
3.2 Runtime Memory Organization ... 119
3.2.1 The code Section ... 120
3.2.2 The static Section ... 122
3.2.3 The readonly Data Section ... 123
3.2.4 The storage Section ... 123
3.2.5 The @nostorage Attribute ... 124
3.2.6 The var Section ... 125
3.2.7 Organization of Declaration Sections Within Your Programs ... 126
3.3 How HLA Allocates Memory for Variables ... 127
3.4 HLA Support for Data Alignment ... 128

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1027, Page 000012

3.5 Address Expressions ... 131
3.6 Type Coercion ... 133
3.7 Register Type Coercion ... 136
3.8 The stack Segment and the push and pop Instructions ... 137
3.8.1 The Basic push Instruction ... 137
3.8.2 The Basic pop Instruction ... 138
3.8.3 Preserving Registers with the push and pop Instructions ... 140
3.9 The Stack Is a LIFO Data Structure ... 140
3.9.1 Other push and pop Instructions ... 143
3.9.2 Removing Data from the Stack Without Popping It ... 144
3.10 Accessing Data You've Pushed onto the Stack Without Popping It ... 146
3.11 Dynamic Memory Allocation and the Heap Segment ... 147
3.12 The inc and dec Instructions ... 152
3.13 Obtaining the Address of a Memory Object ... 152
3.14 For More Information ... 153

4
CONSTANTS, VARIABLES, AND DATA TYPES 155

4.1 Some Additional Instructions: intmul, bound, into ... 156
4.2 HLA Constant and Value Declarations ... 160
4.2.1 Constant Types ... 164
4.2.2 String and Character Literal Constants ... 165
4.2.3 String and Text Constants in the const Section ... 167
4.2.4 Constant Expressions ... 169
4.2.5 Multiple const Sections and Their Order in an HLA Program ... 171
4.2.6 The HLA val Section ... 172
4.2.7 Modifying val Objects at Arbitrary Points in Your Programs ... 173
4.3 The HLA Type Section ... 173
4.4 enum and HLA Enumerated Data Types ... 174
4.5 Pointer Data Types ... 175
4.5.1 Using Pointers in Assembly Language ... 177
4.5.2 Declaring Pointers in HLA ... 178
4.5.3 Pointer Constants and Pointer Constant Expressions ... 179
4.5.4 Pointer Variables and Dynamic Memory Allocation ... 180
4.5.5 Common Pointer Problems ... 180
4.6 Composite Data Types ... 185
4.7 Character Strings ... 185
4.8 HLA Strings ... 188
4.9 Accessing the Characters Within a String ... 194
4.10 The HLA String Module and Other String-Related Routines ... 196
4.11 In-Memory Conversions ... 208
4.12 Character Sets ... 209
4.13 Character Set Implementation in HLA ... 210
4.14 HLA Character Set Constants and Character Set Expressions ... 212
4.15 Character Set Support in the HLA Standard Library ... 213
4.16 Using Character Sets in Your HLA Programs ... 217
4.17 Arrays ... 218
4.18 Declaring Arrays in Your HLA Programs ... 219

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1027, Page 000013

4.19 HLA Array Constants ... 220
4.20 Accessing Elements of a Single-Dimensional Array ... 221
4.21 Sorting an Array of Values ... 222
4.22 Multidimensional Arrays ... 224
4.22.1 Row-Major Ordering ... 225
4.22.2 Column-Major Ordering ... 228
4.23 Allocating Storage for Multidimensional Arrays ... 229
4.24 Accessing Multidimensional Array Elements in Assembly Language ... 231
4.25 Records ... 233
4.26 Record Constants ... 235
4.27 Arrays of Records ... 236
4.28 Arrays/Records as Record Fields ... 237
4.29 Aligning Fields Within a Record ... 241
4.30 Pointers to Records ... 242
4.31 Unions ... 243
4.32 Anonymous Unions ... 246
4.33 Variant Types ... 247
4.34 Namespaces ... 248
4.35 Dynamic Arrays in Assembly Language ... 251
4.36 For More Information ... 254

5
PROCEDURES AND UNITS 255

5.1 Procedures ... 255
5.2 Saving the State of the Machine ... 258
5.3 Prematurely Returning from a Procedure ... 262
5.4 Local Variables ... 262
5.5 Other Local and Global Symbol Types ... 268
5.6 Parameters ... 268
5.6.1 Pass by Value ... 269
5.6.2 Pass by Reference ... 273
5.7 Functions and Function Results ... 275
5.7.1 Returning Function Results ... 276
5.7.2 Instruction Composition in HLA ... 277
5.7.3 The HLA @returns Option in Procedures ... 280
5.8 Recursion ... 282
5.9 Forward Procedures ... 286
5.10 HLA v2.0 Procedure Declarations ... 287
5.11 Low-Level Procedures and the call Instruction ... 288
5.12 Procedures and the Stack ... 290
5.13 Activation Records ... 293
5.14 The Standard Entry Sequence ... 296
5.15 The Standard Exit Sequence ... 298
5.16 Low-Level Implementation of Automatic (Local) Variables ... 299
5.17 Low-Level Parameter Implementation ... 301
5.17.1 Passing Parameters in Registers ... 301
5.17.2 Passing Parameters in the Code Stream ... 304
5.17.3 Passing Parameters on the Stack ... 307

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1027, Page 000014

5.18 Procedure Pointers 329
5.19 Procedural Parameters 333
5.20 Untyped Reference Parameters 334
5.21 Managing Large Programs 335
5.22 The #include Directive 336
5.23 Ignoring Duplicate #include Operations 337
5.24 Units and the external Directive 338
5.24.1 Behavior of the external Directive 343
5.24.2 Header Files in HLA 344
5.25 Namespace Pollution 345
5.26 For More Information 348

6
ARITHMETIC 351

6.1 80x86 Integer Arithmetic Instructions 351
6.1.1 The mul and imul Instructions 352
6.1.2 The div and idiv Instructions 355
6.1.3 The cmp Instruction 357
6.1.4 The setcc Instructions 362
6.1.5 The test Instruction 364
6.2 Arithmetic Expressions 365
6.2.1 Simple Assignments 366
6.2.2 Simple Expressions 366
6.2.3 Complex Expressions 369
6.2.4 Commutative Operators 374
6.3 Logical (Boolean) Expressions 375
6.4 Machine and Arithmetic Idioms 377
6.4.1 Multiplying without mul, imul, or intmul 378
6.4.2 Division Without div or idiv 379
6.4.3 Implementing Modulo-N Counters with and 380
6.5 Floating-Point Arithmetic 380
6.5.1 FPU Registers 380
6.5.2 FPU Data Types 387
6.5.3 The FPU Instruction Set 389
6.5.4 FPU Data Movement Instructions 389
6.5.5 Conversions 391
6.5.6 Arithmetic Instructions 394
6.5.7 Comparison Instructions 399
6.5.8 Constant Instructions 402
6.5.9 Transcendental Instructions 402
6.5.10 Miscellaneous Instructions 404
6.5.11 Integer Operations 405
6.6 Converting Floating-Point Expressions to Assembly Language 406
6.6.1 Converting Arithmetic Expressions to Postfix Notation 407
6.6.2 Converting Postfix Notation to Assembly Language 409
6.7 HLA Standard Library Support for Floating-Point Arithmetic 411
6.8 For More Information 411

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1027, Page 000015

7
LOW-LEVEL CONTROL STRUCTURES 413

7.1 Low-Level Control Structures ... 414
7.2 Statement Labels ... 414
7.3 Unconditional Transfer of Control (jmp) ... 416
7.4 The Conditional Jump Instructions ... 418
7.5 "Medium-Level" Control Structures: jt and jf ... 421
7.6 Implementing Common Control Structures in Assembly Language ... 422
7.7 Introduction to Decisions ... 422
7.7.1 if..then..else Sequences ... 424
7.7.2 Translating HLA if Statements into Pure Assembly Language ... 427
7.7.3 Implementing Complex if Statements Using Complete Boolean Evaluation ... 432
7.7.4 Short-Circuit Boolean Evaluation ... 433
7.7.5 Short-Circuit vs. Complete Boolean Evaluation ... 435
7.7.6 Efficient Implementation of if Statements in Assembly Language ... 437
7.7.7 switch/case Statements ... 442
7.8 State Machines and Indirect Jumps ... 452
7.9 Spaghetti Code ... 455
7.10 Loops ... 456
7.10.1 while Loops ... 457
7.10.2 repeat..until Loops ... 458
7.10.3 forever..endfor Loops ... 459
7.10.4 for Loops ... 460
7.10.5 The break and continue Statements ... 461
7.10.6 Register Usage and Loops ... 465
7.11 Performance Improvements ... 466
7.11.1 Moving the Termination Condition to the End of a Loop ... 466
7.11.2 Executing the Loop Backwards ... 469
7.11.3 Loop-Invariant Computations ... 470
7.11.4 Unraveling Loops ... 471
7.11.5 Induction Variables ... 472
7.12 Hybrid Control Structures in HLA ... 473
7.13 For More Information ... 476

8
ADVANCED ARITHMETIC 477

8.1 Multiprecision Operations ... 478
8.1.1 HLA Standard Library Support for Extended-Precision Operations ... 478
8.1.2 Multiprecision Addition Operations ... 480
8.1.3 Multiprecision Subtraction Operations ... 483
8.1.4 Extended-Precision Comparisons ... 485
8.1.5 Extended-Precision Multiplication ... 488
8.1.6 Extended-Precision Division ... 492
8.1.7 Extended-Precision neg Operations ... 501
8.1.8 Extended-Precision and Operations ... 503
8.1.9 Extended-Precision or Operations ... 503

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1027, Page 000016

8.1.10 Extended-Precision xor Operations ... 504
8.1.11 Extended-Precision not Operations ... 504
8.1.12 Extended-Precision Shift Operations ... 504
8.1.13 Extended-Precision Rotate Operations ... 508
8.1.14 Extended-Precision I/O ... 509
8.2 Operating on Different-Size Operands ... 530
8.3 Decimal Arithmetic ... 532
8.3.1 Literal BCD Constants ... 533
8.3.2 The 80x86 daa and das Instructions ... 534
8.3.3 The 80x86 aaa, aas, aam, and aad Instructions ... 535
8.3.4 Packed Decimal Arithmetic Using the FPU ... 537
8.4 Tables ... 539
8.4.1 Function Computation via Table Lookup ... 539
8.4.2 Domain Conditioning ... 544
8.4.3 Generating Tables ... 545
8.4.4 Table Lookup Performance ... 548
8.5 For More Information ... 549

9
MACROS AND THE HLA COMPILE-TIME LANGUAGE 551

9.1 Introduction to the Compile-Time Language (CTL) ... 551
9.2 The #print and #error Statements ... 553
9.3 Compile-Time Constants and Variables ... 555
9.4 Compile-Time Expressions and Operators ... 555
9.5 Compile-Time Functions ... 558
9.5.1 Type-Conversion Compile-Time Functions ... 559
9.5.2 Numeric Compile-Time Functions ... 561
9.5.3 Character-Classification Compile-Time Functions ... 561
9.5.4 Compile-Time String Functions ... 561
9.5.5 Compile-Time Symbol Information ... 562
9.5.6 Miscellaneous Compile-Time Functions ... 563
9.5.7 Compile-Time Type Conversions of Text Objects ... 564
9.6 Conditional Compilation (Compile-Time Decisions) ... 565
9.7 Repetitive Compilation (Compile-Time Loops) ... 570
9.8 Macros (Compile-Time Procedures) ... 573
9.8.1 Standard Macros ... 574
9.8.2 Macro Parameters ... 576
9.8.3 Local Symbols in a Macro ... 582
9.8.4 Macros as Compile-Time Procedures ... 585
9.8.5 Simulating Function Overloading with Macros ... 586
9.9 Writing Compile-Time "Programs" ... 592
9.9.1 Constructing Data Tables at Compile Time ... 592
9.9.2 Unrolling Loops ... 596
9.10 Using Macros in Different Source Files ... 598
9.11 For More Information ... 598

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1027, Page 000017

10
BIT MANIPULATION **599**

10.1 What Is Bit Data, Anyway? ... 600
10.2 Instructions That Manipulate Bits ... 601
10.3 The Carry Flag as a Bit Accumulator ... 609
10.4 Packing and Unpacking Bit Strings ... 609
10.5 Coalescing Bit Sets and Distributing Bit Strings ... 612
10.6 Packed Arrays of Bit Strings ... 615
10.7 Searching for a Bit ... 617
10.8 Counting Bits ... 620
10.9 Reversing a Bit String ... 623
10.10 Merging Bit Strings ... 625
10.11 Extracting Bit Strings ... 626
10.12 Searching for a Bit Pattern ... 627
10.13 The HLA Standard Library Bits Module ... 628
10.14 For More Information ... 631

11
THE STRING INSTRUCTIONS **633**

11.1 The 80x86 String Instructions ... 634
11.1.1 How the String Instructions Operate ... 634
11.1.2 The rep/repe/repz and repnz/repne Prefixes ... 635
11.1.3 The Direction Flag ... 636
11.1.4 The movs Instruction ... 638
11.1.5 The cmps Instruction ... 644
11.1.6 The scas Instruction ... 647
11.1.7 The stos Instruction ... 648
11.1.8 The lods Instruction ... 648
11.1.9 Building Complex String Functions from lods and stos ... 649
11.2 Performance of the 80x86 String Instructions ... 650
11.3 For More Information ... 650

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1027, Page 000018

12
CLASSES AND OBJECTS **651**

12.1 General Principles ... 652
12.2 Classes in HLA ... 654
12.3 Objects ... 657
12.4 Inheritance ... 659
12.5 Overriding ... 660
12.6 Virtual Methods vs. Static Procedures ... 661
12.7 Writing Class Methods and Procedures ... 663
12.8 Object Implementation ... 668
12.8.1 Virtual Method Tables ... 671
12.8.2 Object Representation with Inheritance ... 673
12.9 Constructors and Object Initialization ... 677
12.9.1 Dynamic Object Allocation Within the Constructor ... 679
12.9.2 Constructors and Inheritance ... 681
12.9.3 Constructor Parameters and Procedure Overloading ... 685
12.10 Destructors ... 686
12.11 HLA's _initialize_ and _finalize_ Strings ... 687
12.12 Abstract Methods ... 693
12.13 Runtime Type Information ... 696
12.14 Calling Base Class Methods ... 698
12.15 For More Information ... 699

APPENDIX
ASCII CHARACTER SET **701**

INDEX **705**

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1027, Page 000019

and instruction, the result is 0. True and true should produce true, not false. Although you can account for situations like this, it usually requires a few extra instructions and is somewhat less efficient when computing boolean operations.

A system that uses nonzero values to represent true and 0 to represent false is an *arithmetic logical system*. A system that uses the two distinct values like 0 and 1 to represent false and true is called a *boolean logical system*, or simply a boolean system. You can use either system, as convenient. Consider again the boolean expression

```
b := ((x = y) and (a <= d)) or ((z - a) <> 5);
```

The simple expressions resulting from this expression might be:

```
mov( x, eax );
cmp( y, eax );
sete( al );       // al := x = y;

mov( a, ebx );
cmp( ebx, d );
setle( bl );     // bl := a <= d;
and( al, bl );   // bl := (x = y) and (a <= d);

mov( z, eax );
sub( a, eax );
cmp( eax, 5 );
setne( al );
or( bl, al );     // al := ((x = y) and (a <= d)) or ((z - a) <> 5);
mov( al, b );
```

When working with boolean expressions don't forget that you might be able to optimize your code by simplifying those boolean expressions. You can use algebraic transformations to help reduce the complexity of an expression. In the chapter on control structures, you'll also see how to use control flow to calculate a boolean result. This is generally quite a bit more efficient than using *complete boolean evaluation* as the examples in this section teach.

6.4 Machine and Arithmetic Idioms

An idiom is an idiosyncrasy. Several arithmetic operations and 80x86 instructions have idiosyncrasies that you can take advantage of when writing assembly language code. Some people refer to the use of machine and arithmetic idioms as "tricky programming" that you should always avoid in well-written programs. While it is wise to avoid tricks just for the sake of tricks, many machine and arithmetic idioms are well known and commonly found in assembly language programs. Some of them are little more than tricks, but a good number of them are simply "tricks of the trade." This text cannot even begin to present all of the idioms in common use today; they are too numerous and the list is constantly changing. Nevertheless, there are some very important idioms that you will see all the time, so it makes sense to discuss those.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1027, Page 000020

6.4.1 Multiplying without mul, imul, or intmul

When multiplying by a constant, you can sometimes write faster code by using shifts, additions, and subtractions in place of multiplication instructions.

Remember, a `shl` instruction computes the same result as multiplying the specified operand by 2. Shifting to the left two bit positions multiplies the operand by 4. Shifting to the left three bit positions multiplies the operand by 8. In general, shifting an operand to the left n bits multiplies it by 2^n. You can multiply any value by some constant using a series of shifts and additions or shifts and subtractions. For example, to multiply the AX register by 10, you need only multiply it by 8 and then add in two times the original value. That is, `10 * ax = 8 * ax + 2 * ax`. The code to accomplish this is:

```
shl( 1, ax );          // Multiply ax by two.
mov( ax, bx);          // Save 2*ax for later.
shl( 2, ax );          // Multiply ax by eight (*4 really,
                       // but ax contains *2).
add( bx, ax );         // Add in ax*2 to ax*8 to get ax*10.
```

Many x86 processors can multiply the AX register (or just about any register, for that matter) by various constant values much faster by using `shl` than by using the `mul` instruction. This may seem hard to believe because it takes only one instruction to compute this product:

```
intmul( 10, ax );
```

However, if you look at the instruction timings, the shift and add example above requires fewer clock cycles on many processors in the 80x86 family than the `mul` instruction. Of course, the code is somewhat larger (by a few bytes), but the performance improvement is usually worth it.

You can also use subtraction with shifts to perform a multiplication operation. Consider the following multiplication by 7:

```
mov( eax, ebx );           // Save eax * 1
shl( 3, eax );             // eax = eax * 8
sub( ebx, eax );           // eax * 8 - eax * 1 is eax * 7
```

A common error beginning assembly language programmers make is subtracting or adding 1 or 2 rather than `eax * 1` or `eax * 2`. The following does not compute `eax * 7`:

```
shl( 3, eax );
sub( 1, eax );
```

It computes `(8 * eax) - 1`, something entirely different (unless, of course, EAX = 1). Beware of this pitfall when using shifts, additions, and subtractions to perform multiplication operations.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1027, Page 000021

You can also use the lea instruction to compute certain products. The trick is to use the scaled index addressing modes. The following examples demonstrate some simple cases:

```
lea( eax, [ecx][ecx] );       // eax := ecx * 2
lea( eax, [eax][eax*2] );     // eax := eax * 3
lea( eax, [eax*4] );          // eax := eax * 4
lea( eax, [ebx][ebx*4] );     // eax := ebx * 5
lea( eax, [eax*8] );          // eax := eax * 8
lea( eax, [edx][edx*8] );     // eax := edx * 9
```

6.4.2 Division Without div or idiv

Just as the shl instruction is useful for simulating a multiplication by a power of 2, the shr and sar instructions can simulate a division by a power of 2. Unfortunately, you cannot easily use shifts, additions, and subtractions to perform a division by an arbitrary constant. Therefore, keep in mind that this trick is useful only when dividing by powers of 2. Also, don't forget that the sar instruction rounds towards negative infinity rather than toward 0; this is not the way the idiv instruction operates (it rounds toward 0).

Another way to perform division is to use the multiply instructions. You can divide by some value by multiplying by its reciprocal. Because the multiply instruction is faster than the divide instruction, multiplying by a reciprocal is usually faster than division.

Now you're probably wondering, "How does one multiply by a reciprocal when the values we're dealing with are all integers?" The answer, of course, is that we must cheat to do this. If you want to multiply by 1/10, there is no way you can load the value 1/10 into an 80x86 integer register prior to performing the multiplication. However, we could multiply 1/10 by 10, perform the multiplication, and then divide the result by 10 to get the final result. Of course, this wouldn't buy you anything; in fact, it would make things worse because you're now doing a multiplication by 10 as well as a division by 10. However, suppose you multiply 1/10 by 65,536 (6,553), perform the multiplication, and then divide by 65,536. This would still perform the correct operation, and, as it turns out, if you set up the problem correctly, you can get the division operation for free. Consider the following code that divides AX by 10:

```
mov( 6554, dx );          // 6,554 = round( 65,536/10 )
mul( dx, ax );
```

This code leaves AX/10 in the DX register.

To understand how this works, consider what happens when you multiply AX by 65,536 ($1_0000). This simply moves AX into DX and sets AX to 0 (a multiply by $1_0000 is equivalent to a shift left by 16 bits). Multiplying by 6,554 (65,536 divided by 10) puts AX divided by 10 into the DX register. Because mul is faster than div, this technique runs a little faster than using a division.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1027, Page 000022

Multiplying by a reciprocal works well when you need to divide by a constant. You could even use it to divide by a variable, but the overhead to compute the reciprocal pays off only if you perform the division many, many times (by the same value).

6.4.3 Implementing Modulo-N Counters with and

If you want to implement a counter variable that counts up to $2^n - 1$ and then resets to 0, simply use the following code:

```
inc( CounterVar );
and( nBits, CounterVar );
```

where `nBits` is a binary value containing n bits containing ones right justified in the number. For example, to create a counter that cycles between 0 and 15 ($2^4 - 1$), you could use the following:

```
inc( CounterVar );
and( %00001111, CounterVar );
```

6.5 Floating-Point Arithmetic

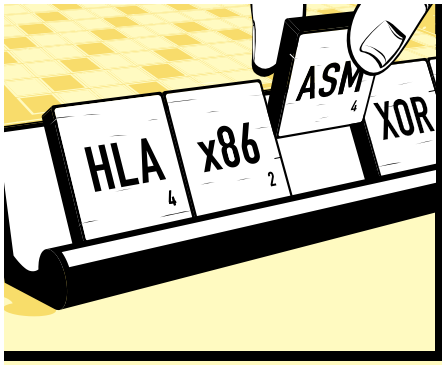
When the 8086 CPU first appeared in the late 1970s, semiconductor technology was not to the point where Intel could put floating-point instructions directly on the 8086 CPU. Therefore, Intel devised a scheme whereby it could use a second chip to perform the floating-point calculations—the floating-point unit (or FPU).[1] By the release of the Intel Pentium chip, semiconductor technology had advanced to the point that the FPU was fully integrated onto the 80x86 CPU. Therefore, almost all modern 80x86 CPU devices fully support floating-point arithmetic directly on the CPU.

6.5.1 FPU Registers

The 80x86 FPUs add 13 registers to the 80x86: eight floating-point data registers, a control register, a status register, a tag register, an instruction pointer, and a data pointer. The data registers are similar to the 80x86's general-purpose register set insofar as all floating-point calculations take place in these registers. The control register contains bits that let you decide how the FPU handles certain degenerate cases like rounding of inaccurate computations; it also contains bits that control precision and so on. The status register is similar to the 80x86's flags register; it contains the condition code bits and several other floating-point flags that describe the state of the FPU. The tag register contains several groups of bits that determine the state of the value in each of the eight floating-point data registers. The instruction and data pointer registers

[1] Intel has also referred to this device as the Numeric Data Processor (NDP), Numeric Processor Extension (NPX), and math coprocessor.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1027, Page 000023



ASSEMBLY FOR THE NON-ASSEMBLY PROGRAMMER

Assembly is a low-level programming language that's one step above a computer's native machine language. Although assembly language is commonly used for writing device drivers, emulators, and video games, many programmers find its somewhat unfriendly syntax intimidating to learn and use.

Since 1996, Randall Hyde's *The Art of Assembly Language* has provided a comprehensive, plain-English, and patient introduction to 32-bit x86 assembly for non-assembly programmers. Hyde's primary teaching tool, High Level Assembler (or HLA), incorporates many of the features found in high-level languages (like C, C++, and Java) to help you quickly grasp basic assembly concepts. HLA lets you write true low-level code while enjoying the benefits of high-level language programming.

As you read *The Art of Assembly Language*, you'll learn the low-level theory fundamental to computer science and turn that understanding into real, functional code.

You'll learn how to:

- Edit, compile, and run HLA programs
- Declare and use constants, scalar variables, pointers, arrays, structures, unions, and namespaces
- Translate arithmetic expressions (integer and floating point)
- Convert high-level control structures

This much anticipated second edition of *The Art of Assembly Language* has been updated to reflect recent changes to HLA and to support Linux, Mac OS X, and FreeBSD. Whether you're new to programming or you have experience with high-level languages, *The Art of Assembly Language, 2nd Edition* is your essential guide to learning this complex, low-level language.

ABOUT THE AUTHOR

Randall Hyde is the author of *Write Great Code, Volumes 1* and *2* (No Starch Press) and the co-author of *MASM 6.0 Bible* (The Waite Group). He has written for *Dr. Dobb's Journal*, *Byte*, and various professional journals. Hyde taught assembly language at the University of California, Riverside for over a decade.

no starch press

THE FINEST IN GEEK ENTERTAINMENT™

www.nostarch.com

OTABIND "I LIE FLAT."

This book uses a lay-flat binding that won't snap shut.

ISBN: 978-1-59327-207-4

9 781593 272074 55995

$59.95 ($74.95 CDN)

6 89145 72073 0

SHELVE IN: COMPUTERS/PROGRAMMING LANGUAGES

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1027, Page 000024