New!
CodeView 4.X
Update Included

Assembly Language Programming

for the IBM PC Family

```
mov dx, OFFSET Msg
mov ah, 09h
int 21h
```


William B. Jones

Assembly Language for the IBM PC Family

William B. Jones
California State University
Dominguez Hills

Scott/Jones Inc.
Publishers
P.O. Box 696
El Granada, CA 94018

Printed in the United States of America

Microsoft Assembler is the product of Microsoft Corporation.
Turbo Assembler is the product of Borland International.

Book production: Highpoint Graphics
Book Manufacturing: Malloy Lithographing, Inc.

Jones, William B.,
Assembly language for the IBM PC family / William B. Jones.
670 p. cm.
Includes bibliographical references and index.
ISBN 0-9624230-6-8
1. IBM microcomputers—Programming. 2. Assembler language (Computer program language) I. Title.
QA76.8.I259193J66 1992
005.265—dc20 91-42704
CIP

Preface

Why Learn Assembly Language?

Assembly Language is a symbolic form of a computer's own internal language, called machine language. In bygone days when I was first making a living as a programmer, all "hero programmers" wrote nothing but assembly language. The only high-level languages generally available were Fortran and COBOL, and they really weren't capable of the esoteric things that hero programmers wanted to do. Today though, the high-level languages are so good that they will do virtually anything you want to do. Also, studies have shown that programmers are much more productive when using high-level languages, and modern optimizing compilers often produce code that is nearly as good as that produced by experienced assembly language programmers.

So why do we study assembly language? There are several good reasons:

- There are programs where small size and/or high speed are so essential that they are written mostly or entirely in assembly language. Examples are the WordPerfect word processing program and the Lotus 1-2-3 spreadsheet program for the IBM PC. It may also be necessary to write entire programs in assembly language when the computer is special purpose and there is an insufficient market to warrant writing a high-quality compiler.
- There are a few things which simply can't be done in a high-level language, or can only be done with much less efficiency than in assembly language. Also sometimes small sections of code are repeated a large number of times, making it worthwhile to optimize them to the highest degree possible. See Chapter 15 and Section 16.7, as well as the book *Undocumented DOS* (see the bibliography) for examples. In cases where a

small amount of assembly language is necessary or desirable, most of the program is written in a high-level language but selected parts are coded in assembly language as separate subprocedures. The current versions of Turbo Pascal, Turbo C, and Microsoft C also allow the programmer to drop into assembly language to make use of special instructions.

- Writers of compilers may need to know the assembly language of the target machine. The most common type of compiler translates a high-level language such as C or Pascal into machine language. Today compilers are usually written in a high-level language, often in the language they compile! (Many books on compiler construction describe how this is possible.) Even though the compiler isn't *written* in assembly language, the programmers who write the code generation part must *know* assembly language, because that's (more or less) what the compiler is producing.
- It is an unfortunate fact that all large programs, including compilers, have bugs. If a piece of code isn't working and you suspect it is because the compiler is producing bad code, you will have to look at the machine language it generates, perhaps as the sort of assembly language displayed by an interactive debugger (see Chapter 5). I have found bugs in several commercial C compilers this way.
- Even programs which are written entirely in a high-level language and don't produce machine language may require a detailed knowledge of machine architecture. For instance, it would be very difficult to write programs for the IBM PC of even moderate sophistication without knowledge of much of the material in Chapters 13, 17, and 20 of this text. As another example, a knowledge of how operations in a high-level language are actually performed in the machine removes a lot of the mystery surrounding programming and can improve the efficiency of the high-level code written.
- Writing assembly language is fun!

Assuming now that learning assembly language is a worthwhile endeavor, **why learn IBM PC assembly language**? Different computer architectures require different assembly languages. IBM PC assembly language is quite different from, say, assembly language on the Apple Macintosh, or even on IBM mainframes. So why pick the IBM PC?

The IBM PC and its clones (copies) form by far the largest group of personal computers available. Therefore the machines are readily available to learn on and software written for them has a large potential market. Also, the software tools necessary to write assembly language on the PC are excellent, widely available and reasonably priced. The machine has quite a simple architecture in terms of I/O and other operating system-related concepts that often present important reasons for writing assembly language. Finally, the IBM PC assembly language is sufficiently ordinary that most other machine assembly languages are easy to learn once you have learned the PC.

Distinctive Features of this Book

- Both major assemblers, MASM and TASM, along with their debuggers, are treated.
- Correct programming style is introduced early and used constantly. For instance, use of EQUates is introduced early and emphasized throughout. Allowing the assembler to do the computing is encouraged: a loop to be executed for subscripts 30 to 40 is executed 40 – 30 + 1 times, rather than the more mysterious 11.
- As early as practicable, examples are given using the kind of applications that real-world programmers use assembly language for. Some themes appear over and over in the text, as the early crude solutions of various problems are refined and varied, emphasizing that in assembly language, there are many solutions to most problems.
- Detailed algorithms are presented for turning high-level pseudo-code for "if" statements and loops into correct low-level assembly code. They involve "decorating" the pseudo-code with arrows showing parts of the code skipped under various conditions, and then straightforward methods for turning such diagrams into code.
- Subprograms for doing numeric I/O are included in a library on the disk that comes with this book. Their sources are also included.
- All major DOS calls are covered, together with selected BIOS calls. An INCLUDE file of macros for the DOS calls is on the disk accompanying this book.
- Use of modern source-level interactive debuggers is introduced early. Features are introduced throughout the text where needed, and exercises are given to walk the student through these features. Usage of two different debuggers is separated so that the student need only consider one of them. If one wishes to avoid debuggers entirely (for religious reasons or because another system not using the debuggers covered here is being used) this is easily done.
- Exercises are provided in a wide range of difficulties. These vary from large numbers of exercises of the "what does this instruction do in this situation?" form to short programming exercises which may require substantial thought. Representative answers are given. In addition, most chapters end with several carefully specified programming problems in varying degrees of difficulty.
- Programs are developed incrementally. Code which is at all complex is first developed in pseudo-code. Modifications to code are indicated clearly: **bold** for new text and ~~strikeout~~ for text that has been removed. Code is shown mostly in lowercase, with capital letters for legibility. Types of operation codes are distinguished using capitalization: all lowercase for machine operations, all uppercase for pseudo-operations, and mixed upper- and lowercase for macros.
- High-level language constructs are related to the assembly language into which they translate.

In addition to the above, the reader should note the following:

- Each chapter will begin with a preview of its contents and a list of the topics to be covered and ends with a summary of the material covered.

- Exercises or parts of exercises marked ✓ have answers in the Answer section at the end of the book.
- Pieces of code which you should emulate only at your peril are indicated by the universal Don't-do-this symbol in the margin:

```
Play    with, matches
```

Organization of this Text

There is much too much material in this book to be covered in a one semester course. I would suggest that there is a central core of the book which should be covered, pretty much in the order given. The remaining time can then be given over to special topics, laboratory exercises, and student projects as suits the interests of instructors and their students.

Central Core

Chapters 1–4 (`_GetStr` in §3.5 may be delayed until Chapter 11; §4.2 (byte swapping) may be de-emphasized.
Chapters 6–7 (arithmetic and decisions), §§ 8.2–8.3 (subprocedures and separate assembly.
§§9.1 (program testing), 9.2 (simple macros) and 9.4 (repetition and = pseudo-ops)
§§10.1–10.2 (logical and shift operations)
Chapter 11 (arrays)
§§13.1–13.5 (segments—very important for the (peculiar) IBM PC architecture; less so for more typical architectures. Only §13.1 is required for the remaining core.)
§14.1 (procedures and the stack)
§§16.1–16.2 (Interrupts)

Alternate Themes

In addition to this core material, various other topics or emphases can be arranged into various *threads* or themes, and pursued to varying degrees depending on the interests and orientation of the reader.

Interactive Debuggers (possibly lab exercises): Chapter 5, §§ 6.3, 7.4, 8.6, and 13.6
I/O Conversion Routines: §§ 8.1, 8.4–8.5, 10.3, 12.1
Relationships to High-Level Languages (chiefly C and Pascal): 10.4, 11.4, 12.2–12.4, 13.7 (pointers), 14.2–14.4
Macros: §§ 9.2–9.4, 11.3, Chapter 17, §18.3, §§20.2–20.3
Systems Programming Stuff: §§13.5, 13.8, Chapter 16, §§18.4–18.5
Raisons d'être for Assembly Language: Chapter 15
File Processing: Chapter 18
Special String Operations: Chapter 19
Video: Chapter 20
Advanced Processors (80286, 80286, 80486, and 80X87): Chapter 21

What Do You Need for this Book?

I make the assumption that you already know some high-level language, such as Pascal or C. Only occasionally will any detailed knowledge of these languages be required, and those situations will be encapsulated so that you can skip them if you wish. A description of the Pascal-like pseudo-language used for comments, exercises, etc., is given in section 1.4.

You can't learn to program in a vacuum. You need to write and debug programs, and for that you need the following hardware and software:

- An IBM PC or PC Clone running the MS- or PCDOS operating system, version 2.1 or later, and preferably version 3.0 or later. You should have a hard disk (highly preferable) or two floppy disk drives and at least 512K of RAM memory. Any model (XT, AT, PS-2, etc.) will work. A mouse is desirable.
- An Assembler and related software, a linker and a debugger—either the Microsoft Assembler package (version 5.1, or 6.0; version 5.0 differs only slightly from version 5.1) or the Turbo Assembler package (version 1.0 or later; version 2.0 or later preferred). The important parts of the Microsoft package are MASM, the assembler; LINK, the linker; and CV, the CodeView debugger. The important parts of the Turbo package are TASM, the assembler; TLINK, the linker; and TD, the Turbo Debugger. The assemblers and linkers in these two packages are virtually functionally identical. For a comparison of features of the debuggers, see Section 5.6.
- A text editor. The editor M which comes with MASM 5.1 or the editor ED which comes on the disk that accompanies this book are both acceptable. MASM [6.0] comes with an all-encompassing programming environment called *Programmer's Work Bench* (PWB for short) which includes an editor. Unfortunately, PWB is so slow even on the author's 16 MHz 80386 as to be virtually unusable.
- The disk that accompanies this book. In addition to the editor mentioned above, it contains a library of useful routines (chiefly, for numeric I/O), a library of useful macros, the source files of the library routines, the source files for the various debugger exercises in the book, and the source files of all complete programs in the text.

Note: As you saw above, we will mark passages specific to MASM 6.0 with the symbol [6.0]. We will mark passages specific to TASM 2.0 or later (mostly having to do with the Turbo Debugger) with [2.0].

Pedagogy and Politics—Some choices which may prove controversial

Every textbook author makes many choices. There are three I have made that I would like to explain in advance, in hopes that the more sophisticated reader who may disagree with them will at least understand my side of the argument. In brief, the three decisions I will try to justify are

1. I ignore the rather elaborate **procedure declaration and call** mechanisms available in both assemblers,
2. Almost from the beginning, I use **macros** to effect DOS calls, and
3. I will use a quick-and-dirty solution to a rather subtle problem with **assumed segment register values** rather than a more elaborate and more "correct" mechanism.

The first and second can be explained rather easily on an elementary level, while the third requires an excursion into concepts which will not be fully understood until the material in section 13.8 and Chapter 16 is mastered.

1. Both assemblers have procedure declaration mechanisms which resemble those of high-level languages. They take some of the pain out of manipulating parameters and local variables. I have three reasons for ignoring these mechanisms. First, I believe that it is important to understand in detail how the stack-frame works. To do this one must learn the details of manipulating it. Second, my publisher assures me that assembly language textbooks must be noticeably shorter than Calculus texts. I simply don't have the space to cover the stack manipulations both with and without the syntactic sugar. Thirdly, the two assemblers use three separate syntaxes for procedure declaration, and other assemblers for other machines don't have any such syntactic sugar at all, which exacerbates the second reason.
2. I introduce macros for DOS calls (contained on the disk accompanying this book) early on and use them almost exclusively. You might think that this contradicts my effort with procedure declarations to get close to the hardware. However I just don't think it takes very many times to get the general idea of putting stuff in the randomly chosen registers expected by DOS, putting a funny number in AH, and executing `int 21h`, an instruction which will remain mysterious until Chapter 16 anyway.
3. The problem with assumed segment register values is as follows: the MASM and TASM abbreviated segment definitions tacitly group the stack and .DATA segment, and thus assume that SS = DS. This in fact is not the case when DOS starts your program. In order to make SS = DS happen, each program must start out with about ten lines of non-obvious, highly sophisticated code which alters SS and SP.

   For the first twelve chapters of this book, my solution to this problem is the same as that of TASM and MASM-before-version-6.0: I ignore it! You only get into real trouble with the ignorance-is-bliss solution when you use multiple data segments (starting in Chapter 13).* I know from bitter experience, however, that such a non-solution can ultimately give rise to bugs which are *very* difficult to find.

   One could of course publicize the problem and rely on the programmer to avoid the tricky situations, but it is better to come up with a more *reliable* solution. Once

* If you ever use DS for something other than the .DATA segment address and try to address *something in the* .DATA segment, both assemblers will use SS, *even if you have put the* .DATA *segment address in ES and* ASSUME*d it there!*

again, there are three possibilities, with various associated plusses and minuses.

First there is the MASM 6.0 solution in which a macro is supplied which correctly sets SS = DS. I could have supplied a version of this macro but for various reasons, rejected that idea. The two-instruction startup I *do* use isn't completely explained until Chapter 13, but the student knows virtually from the beginning what the code does, even if the reason for doing it is mysterious. Similarly for the code used in DOS macros. I don't like the idea, though, of forcing the reader to use a large amount of code from the very beginning—even as a macro—that he or she won't understand until late in the book (if ever).

Second, we could always tell the assembler that stack and .DATA are *not* grouped. This can be done in MASM 6.0 by the statement

```
.MODEL  SMALL, FARSTACK ; MASM 6.0 only
```

and in TASM 2.0 by the (apparently undocumented) statement

```
.MODEL  SMALL, NOLANGUAGE, FARSTACK ; TASM 2.0 only
```

In my opinion these are the best solutions because they actually describe the real world when a program starts execution. They were discarded because there is *no* such solution in some versions of the assemblers, and the solutions differ markedly in versions where they do exist. Also it goes without saying that it is always dangerous to use an undocumented solution (it may change in later versions of the software).

Therefore I was left with the third solution, the one I actually chose. In programs with multiple data segments, use

```
ASSUME  SS : NOTHING ;      The chosen method
```

at the beginning of the code segment. This is a little mysterious, but goes to the heart of the problem and has the advantage of working in all assemblers in all versions. I start using it where necessary from Chapter 13 on.

This choice is not without a little accompanying guilt. For one thing, it appears inconsistent. In the first place, if I remained true to my principles, I would have used such a solution from the beginning of the text. Also I would have used compact model (one code segment, multiple data segments) instead of small model since I am tacitly assuming from the start that .DATA and stack are separate segments. Though these things bother me, I believe they are outweighed by pedagogical considerations: Most documentation mentions small model, and also it seemed unnecessarily confusing to switch from small to compact later on. In addition, the assemblers don't seem to think there is anything wrong with separate .DATA and stack in small model (witness the FARSTACK solution above). Finally, the only difference between small and compact models in the code generated is when the fancy procedure declaration syntax is used, and I rejected that in number 1 above.

Acknowledgments

Class Testing

Any teacher will confirm that they don't really know if a book will do the job until they've taught out of it. In an effort to provide more than the usual quality control in the development of this text, a preliminary edition was taught from twice by Maria Kolatis at County College of Morris in New Jersey and by Reza Ahmadnia at California State University Dominguez Hills (as well as several times by myself). Their feedback has been of great help.

Further, prompted by my publisher's offer of a free published text to anyone reporting a dozen errors or even unclear explanations of the class-test edition, their students were not at all hesitant to inform us of problems. They really helped improve the quality of this book.

Reviews

Portions of this book were reviewed by various colleagues. Their comments and criticisms have improved the book immensely:

Richard Easton
Indiana State University

Cay Horstman
San Jose State University

Kerry Hays
San Jose City Collge

Arthur Geis
College of DuPage

Thomas Abromovich
Black Hawk College

Guy Pollock
Mountain View College

Eric Lundstrom
Diablo Valley College

Gary Lippman
CSU Hayward

John Crenshaw
Western Kentucky University

John Chapman
Johnson County Community College

David Williamson
Indiana University/Purdue University

Ray Bell
University of Texas El Paso

Russell Hollingsworth
Tarrant County Junior College

Nita Caftori
Northeastern Illinois University

In addition the reviews of the entire manuscript by Paul LeCoq of Spokane Community College, including a person-to-person chat when he was visiting Southern California, have consistently been "above-and-beyond-the-call."

Production and Development

Sheryl Rose copyedited the manuscript for this textbook, and Sheryl Strauss served as a proofreader for page proof. They both did great jobs. I did the artwork for this text using MacDraw II and Adobe Illustrator. The manuscript was conscientiously transformed into a text by Highpoint Type and Graphics. Finally, while developing this project over the past two years, my publisher Richard Jones (no relation), has alternately made suggestions; bitten his tongue; and mostly knew when to do which.

Bill Jones
Hermosa Beach, CA
1992

Contents

Preface ix

1. A Simple Program 1
1.1 The Program 2
1.2 The Decimal, Binary, and Hexadecimal Numbering Systems 7
1.2.1 The Binary Number System 8
1.2.2 Conversions between Binary and Decimal 8
1.2.3 The Hexadecimal Number System 9
1.2.4 Conversions Involving Hexadecimal 10
1.2.5 Addition in Binary and Hex 12
1.3 The ASCII Character Set 13
1.4 A Pascal-like Pseudoprogramming Language 15

2. Assembler Overview 19
2.1 Hardware Overview 20
2.1.1 Level I: General Properties of Digital Computers 20
2.1.2 Level II: The IBM PC Hardware 24
2.1.3 Level III: Particular Characteristics of the IBM PC 25
2.2 Structure of an IBM PC Assembly Language Program 27
2.2.1 Global Program Structure 30
2.2.2 The .DATA Segment 32
2.2.3 The .CODE Segment in the First Program 36
2.2.4 Comments on Comments 41

3. Input/Output and Such 45
3.1 Macros 46
3.2 The DOS Display Character Call 49
3.3 Magic Numbers 50
3.4 Numeric I/O 51
3.5 Getting Input from the Keyboard 55

4. Data Movement Instructions 63
4.1 More on the `Mov` Instruction 64
4.2 Byte Swapping 68
4.3 The Stack 72
4.4 The Data Exchange (`Xchg`) Instruction 78

5. Introduction to Debuggers 81
5.1 Common Points in CV and TD 82
5.2 CodeView Introduction 86
5.2.1 All Versions 86
5.2.2 Special Information for MASM 6.0 CodeView 89
5.3 Turbo Debugger Introduction 91
5.4 Debugger Exercises 96
5.5 Automating Assembly 99
5.6 Some Strengths and Weaknesses of the Two Debuggers 99

6. Arithmetic 103
6.1 Negative Numbers; 2's Complement Arithmetic 104
6.1.1 Sign-Magnitude Representation 104
6.1.2 1's Complement Representation 104
6.1.3 2's Complement Representation 105
6.2 Arithmetic 111
6.2.1 Addition and Subtraction 111
6.2.2 Multiplication and Division 113
6.3 A Debugging Example 120

7. Comparing and Branching 125
7.1 Decision Making in Assembly Language 126
7.2 Loops 136
7.3 Arithmetic and Conditional Jumps 145
7.4 Debugging with Breakpoints 147
7.4.1 Turbo Debugger Breakpoints 147
7.4.2 CodeView Breakpoints 148
7.4.3 Debugger Example 149
7.5 Some Technical Details on Compares and Jumps 150
7.5.1 The Internal Format of Jump Instructions 150
7.5.2 Unsigned Conditional Jumps 152

8. Applying Assembly I: I/O Number Conversions and Subprocedures 157
8.1 Displaying Unsigned Numbers 158
8.2 Procedures 162
8.2.1 The `Call` and `Ret` Instructions 162
8.2.2 Writing `Call`able Procedures 163
8.2.3 Turning Our Code Into a `Call`able Procedure: `PutUDec` 164
8.2.4 The Complete Program 165
8.3 Placing Procedures in Separate Files 166
8.3.1 Using Separately Translated Procedures 167
8.3.2 Writing Separately Translated Procedures 169
8.4 Turning `PutUDec` into `PutDec` 173
8.5 Reading a Number from the Keyboard 176
8.6 Debuggers: The Stack and Separately Translated Procedures 180
8.6.1 CodeView 180
8.6.2 Turbo Debugger 181
8.6.3 Debugger Example 5 183
8.6.4 Debugger Example 6 185

9. Introduction to Writing Macros; Program Testing 191
9.1 Program Testing 192
9.2 Simple Macros 193
9.3 Fancier Macros 200
9.4 Pseudomacros for Repetition 204

10. Bit Operations 211
10.1 Boolean Operations 212
10.2 Shift Operations 219
10.3 An Application: Code to Implement `PutHex` 226
10.4 A Little Something Extra: Bit Instructions in High-Level Languages 227
10.5 A Little Something Extra: Assembler Record Structures 230

11. Arrays 237
11.1 Address Arithmetic and Arrays 238
11.2 Using Arrays 246
11.3 An Important Application: Conversion Tables 258
11.4 Arrays in High-Level Languages 263

12. Applying Assembly II: Using Arrays 273
12.1 The `PutDec` Procedure Revisited 274
12.2 C-Type Variable-Length Character Strings 278
12.2.1 Using Debuggers with C Strings 282
12.3 Sets 284
12.4 Multiway Branching; Pascal `case` and C `switch` Statements 287

13. Segments 293
13.1 Segments and Offsets 294
13.2 Segment Registers 297
13.3 Defining Segments 299
13.4 The ASSUME Statement 300
13.5 The Program Segment Prefix (PSP) 311
13.6 Debuggers and Segments 320
13.6.1 Turbo Debugger and Segments 320
13.6.2 CodeView and Segments 320
13.6.3 Debug Example 7 321
13.7 A Little Something Extra: Pointers 322
13.8 Memory Models 325

14. Procedures and High-Level Languages 335
14.1 Procedures and the Stack 336
14.2 A Little Something Extra: Communicating with High-Level Languages 349
14.2.1 Turbo Pascal 350
14.2.2 Microsoft C 352
14.3 A Little Something Extra: Inline Assembly Language in High-Level Languages 356
14.4 A Little Something Extra: Parameters and Pointers 358

15. Applying Assembly III: Multiple-Precision and Decimal Arithmetic 367
15.1 Multiple-Precision Arithmetic 368
15.2 The "Minimal Standard" Random Number Generator 382
15.3 Decimal Arithmetic 387

16. Interrupts 395
16.1 Generalities About Interrupts 396
16.2 Interrupt Processing on the 80X86 398
16.3 Applications: Timing and Debuggers 401
16.3.1 Reading the Clock and Timing Operations 401
16.3.2 Interrupts and Debuggers 403
16.4 Interrupt Handlers 405
16.5 A Little Something Extra: A TSR (Terminate and Stay Resident) Program 411
16.6 A Little Something Extra: Caveats for Interrupt Hackers 421
16.7 A Little Something Extra: Simultaneously Executing Programs 422
16.7.1 Race Conditions 422
16.7.2 Semaphores: An Abstract Method of Synchronization 426
16.7.3 Using Semaphores to Solve Race Conditions 426
16.7.4 Implementing Semaphores on the 80X86 427
16.8 A Little Something Extra: A Short History of Interrupts 429

17. Conditional Assembly and More on Macros 433
17.1 Generally Applicable IFs 435
17.2 IFs Usable Only in Macros 440

18. File Processing 449
18.1 Handles and Opening, Creating, and Closing Files 451
18.2 Basic File Operations 454
18.3 Random File Accessing 459
18.3.1 The _LSeek Macro 460
18.3.2 Applications of _LSeek I: Log, a Message Logger 460
18.3.3 Applications of _LSeek II: Tail, an End-of-File Displayer 464
18.3.4 Writing the _LSeek Macro 466
18.4 A Little Something Extra: Redirecting STDERR 469
18.5 The Utility Routines ParseCmd, CCheck, and WCheck 479

19. String Processing Instructions 489
19.1 The String Operations 490
19.2 The REP Instruction Prefixes 496
19.3 The Tail Program Revisited 501

20. Video Basics 505
20.1 Display Hardware 507
20.2 Text Mode 509
20.2.1 Int 10h BIOS Calls for Text Mode 510
20.2.2 Direct Access to Display Memory 514
20.2.3 Hardware I/O: CGA Snow 519
20.3 Simple VGA Graphics 522

21. Other CPUs: 80286, 80386, 80486, and Coprocessors 537
21.1 The 80286 538
21.2 The 80386 and 80386 SX 544
21.2.1 The 80386 Register Set 545
21.2.2 Addressing 545
21.2.3 New Instructions 549
21.3 The 80486 559
21.4 Numeric Coprocessor Chips 559

Answers to Selected Exercises 567

Appendix A The IBM Extended ASCII Character Set 605

Appendix B 80X86 Instructions 608

Appendix C The ED Editor 629