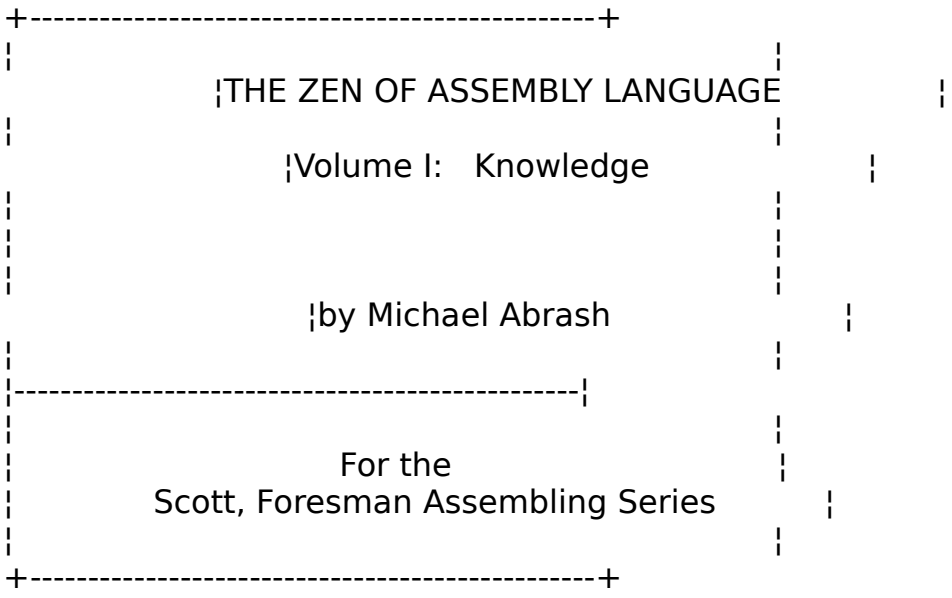

THE ZEN OF ASSEMBLY LANGUAGE
Volume I: Knowledge
by Michael Abrash
For the
Scott, Foresman Assembling Series

Michael Abrash
1599 Bittern Drive
Sunnyvale, CA 94087
(408) 733-3945 (H)
(415) 361-8883 (W)

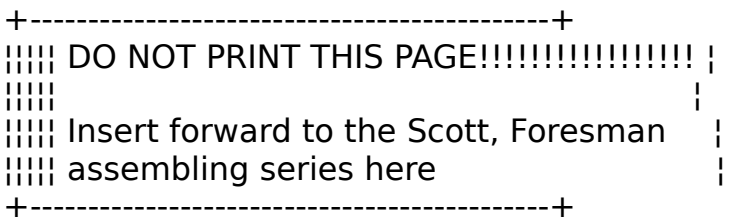
For Shay and Emily

ACKNOWLEDGEMENTS

Special thanks to Jeff Duntemann, who made it all happen, to Noah Oremland, who proofed every word, encouraged me, and even laughed at my jokes, and to Amy Davis of Scott, Foresman & Co., who made this book possible. Thanks also to Tom Blakeslee, John T. Cockerham, Dan Gochnauer, Dan Illowsky, Bob Jervis, Dave Miller, Ted Mirecki, Phil Mummah, Kent Porter, and Tom Wilson for information, feedback and encouragement. Finally, thanks to Orion Instruments for the use of an OmniLab.

+--+
¦¦¦¦¦ DO NOT PRINT THIS PAGE!!!!!!!!!!!!!!!! ¦
¦¦¦¦¦ ¦
¦¦¦¦¦ The following are or may be trademarks ¦
¦¦¦¦¦ or copyrights. ¦
+--+

Term	Company
OS/2	Microsoft Corp.
Microsoft Macro Assembler	Microsoft Corp.
Microsoft Linker	Microsoft Corp.
Symdeb	Microsoft Corp.
CodeView	Microsoft Corp.
Turbo C	Borland International, Inc.
Turbo Debugger	Borland International, Inc.
Turbo Linker	Borland International, Inc.
Turbo Assembler	Borland International, Inc.
OPTASM	SLR Systems
OmniLab	Orion Instruments, Inc.
IBM	International Business Machines Corp.
PC	International Business Machines Corp.
AT	International Business Machines Corp.
PS/2	International Business Machines Corp.
PCjr	International Business Machines Corp.
Unix	AT&T
MS-DOS	Microsoft Corp.
Sidekick	Borland International, Inc.
Wordstar	Micropro International
Visicalc	Visicorp?, Lotus Development Corp?
MAD Magazine	???


DO NOT PRINT THIS PAGE!!!!!!!!!!!!!!!!
Insert forward to the Scott, Foresman
assembling series here

+--+
¦¦¦¦¦   ¦
¦¦¦¦¦ Introduction: Pushing the Envelope ¦
¦¦¦¦¦   ¦
+--+

This is the book I wished for with all my heart seven years ago, when I started programming the IBM PC: the book that unlocks the secrets of writing superb assembly-language code. There was no such book then, so I had to learn the hard way, through experimentation and through trial and error. Over the years, I waited in vain for that book to appear; I looked everywhere without success for a book about advanced assembly- language programming, a book written specifically for assembly- language programmers who want to get better, rather than would-be assembly-language programmers. I'm sure many of you have waited for such a book as well. Well, wait no longer: this is that book.

The Zen of Assembly Language assumes that you're already familiar with assembly language. Not an expert, but at least acquainted with the registers and instructions of the 8088, and with the use of one of the popular PC assemblers. Your familiarity with assembly language will allow us to skip over the droning tutorials about the use of the assembler and the endless explanations of binary arithmetic that take up hundreds of pages in introductory books. We're going to jump into high-performance programming right from the start, and when we come up for air 16 chapters from now, your view of assembly language will be forever altered for the

better. Then we'll leap right back into Volume II, applying our newfound knowledge of assembly language to ever- more-sophisticated programming tasks.

In short, The Zen of Assembler is about nothing less than how to become the best assembly-language programmer you can be.

WHY ASSEMBLY LANGUAGE?

For years, people have been predicting--hoping for--the demise of assembly language, claiming that the world is ready to move on to less primitive approaches to programming...and for years, the best programs around have been written in assembly language. Why is this? Simply because assembly language is hard to work with, but--properly used--produces programs of unparalleled performance. Mediocre programmers have a terrible time working with assembly language; on the other hand, assembly language is, without fail, the language that PC gurus use when they need the best possible code.

Which brings us to you.

Do you want to be a guru? I'd imagine so, if you're reading this book. You've set yourself an ambitious and difficult goal, and your success is far from guaranteed. There's no sure-fire recipe for becoming a guru, any more than there's a recipe for becoming a chess grand master. There is, however, one way you can greatly improve your chances: become an expert assembly language programmer. Assembly language won't by itself make you a guru--but without it you'll never reach your full potential as a

programmer.

Why is assembly language so important in this age of optimizing compilers and program generators? Assembly language is fundamentally different from all other languages, as we'll see throughout The Zen of Assembly Language. Assembly language lets you use every last resource of the PC to push the performance envelope; only in assembly language can you press right up against the inherent limits of the PC.

If you aren't pushing the envelope, there's generally no reason to program in assembler. High-level languages are certainly easier to use, and nowadays most high-level languages let you get at the guts of the PC--display memory, DOS functions, interrupt vectors, and so on--without having to resort to assembler. If, in the other hand, you're striving for the sort of performance that will give your programs snappy interfaces and crackling response times, you'll find assembly language to be almost magical, for no other language even approaches assembler for sheer speed.

Of course, no one tests the limits of the PC with their first assembler program; that takes time and practice. While many PC programmers know something about assembler, few are experts. The typical programmer has typed in the assembler code from an article or two, read a book about assembler programming, and perhaps written a few assembler programs of his own--but doesn't yet feel that he has mastered the language. If you fall into this category, you've surely sensed the remarkable potential of assembler, but you're also keenly aware of how hard it is to write good assembler code and how much you have yet to learn. In

all likelihood, you're not sure how to sharpen your assembler skills and take that last giant step toward mastery of your PC.

This book is for you.

Welcome to the most exciting and esoteric aspect of the IBM PC. The Zen of Assembly Language will teach you how to create blindingly fast code for the IBM PC. More important still, it will teach you how to continue to develop your assembler programming skills on your own. The Zen of Assembly Language will show you a way to learn what you need to know as the need arises, and it is that way of learning that will serve you well for years to come. There are facts and code aplenty in this book and in the companion volume, but it is a way of thinking and learning that lies at the heart of The Zen of Assembly Language.

Don't take the title to mean that this is a mystical book in any way. In the context of assembly-language programming, Zen is a technique that brings intuition and non-obvious approaches to bear on difficult problems and puzzles. If you would rather think of high-performance assembler programming as something more mundane, such as right-brained thinking or plain old craftsmanship, go right ahead; good assembler programming is a highly individualized process.

The Zen of Assembly Language is specifically about assembly language for the IBM PC (and, by definition, compatible computers). In particular, the bulk of this volume will focus on the capabilities of the 8088 processor that lies at the heart of the PC. However, many of the findings and almost all of the techniques I'll discuss can also be applied to assembly-

language programming for the other members of Intel's 808X processor family, including the 80286 and 80386 processors, as we'll see toward the end of this volume. The Zen of Assembly Language doesn't much apply to computers built around other processors, such as the 68XXX family, the Z80, the 8080, or the 6502, since a great deal of the Zen of assembly language in the case of the IBM PC derives from the highly unusual architecture of the 808X family. (In fact, the processors in the 808X family lend themselves beautifully to assembly language, much more so than other currently-popular processors.)

While I will spend a chapter looking specifically at the 80286 found in the AT and PS/2 Models 50 and 60 and at the 80386 found in the PS/2 Model 80, I'll concentrate primarily on the 8088 processor found in the IBM PC and XT, for a number of reasons. First, there are at least 15,000,000 8088-based computers around, ensuring that good 8088 code isn't going to go out of style anytime soon. Second, the 8088 is far and away the slowest of the processors used in IBM-compatible computers, so no matter how carefully code is tailored to the subtleties of the 8088, it's still going to run much faster on an 80286 or 80386. Third, many of the concepts I'll present regarding the 8088 apply to the 80286 and 80386 as well, but to a different degree. Given that there are simply too many processors around to cover in detail (and the 80486 on the way), I'd rather pay close attention to the 8088, the processor for which top-quality code is most critical, and provide you with techniques that will allow you to learn on your own how best to program other processors.

We'll return to this topic in Chapter 15, when we will in fact discuss other 808X-family processors, but for now, take my word for it: when it comes to optimization, the 8088 is the processor of choice.

WHAT YOU'LL NEED

The tools you'll need to follow this book are simple: a text editor to create ASCII program files, the Microsoft Macro Assembler version 5.0 or a compatible assembler (Turbo Assembler is fine) to assemble programs, and the Microsoft Linker or a compatible linker to link programs into an executable form.

There are several types of reference material you should have available as you pursue assembler mastery. You will certainly want a general reference on 8088 assembler. The 8086 Book, written by Rector and Alexy and published by Osborne/McGraw-Hill, is a good reference, although you should beware of its unusually high number of typographic errors. Also useful is the spiral-bound reference manual that comes with MASM, which contains an excellent summary of the instruction sets of the 8088, 8086, 80186, 80286, and 80386. IBM's hardware, BIOS, and DOS technical reference manuals are also useful references, containing as they do detailed information about the resources available to assembler programmers.

If you're the type who digs down to the hardware of the PC in the pursuit of knowledge, you'll find Intel's handbooks and reference manuals to be invaluable (albeit none too easy to read), since Intel manufactures the

8088 and many of the support chips used in the PC. There's simply no way to understand what a hardware component is capable of doing in the context of the PC without a comprehensive description of everything that part can do, and that's exactly what Intel's literature provides.

Finally, keep an eye open for articles on assembly-language programming. Articles provide a steady stream of code from diverse sources, and are your best sources of new approaches to assembler programming.

By the way, the terms "assembler" and "assembly-language" are generally interchangeable. While "assembly-language" is perhaps technically more accurate, since "assembler" also refers to the software that assembles assembly-language code, "assembler" is a widely-used shorthand that I'll use throughout this book. Similarly, I'll use "the Zen of assembler" as shorthand for "the Zen of assembly language."

ODDS AND ENDS

I'd like to identify the manufacturers of the products I'll refer to in this volume. Microsoft makes the Microsoft Macro Assembler (MASM), the Microsoft Linker (LINK), CodeView (CV), and Symdeb (SYMDEB). Borland International makes Turbo Assembler (TASM), Turbo C (TC), Turbo Link (TLINK), and Turbo Debugger (TD). SLR Systems makes OPTASM, an assembler. Finally, Orion Instruments makes OmniLab, which integrates high-performance oscilloscope, logic analyzer, stimulus generator, and disassembler instrumentation in a single PC-based package.

In addition, I'd like to point out that while I've made every effort to ensure that the code in this volume works as it should, no one's perfect. Please let me know if you find bugs. Also, please let me know what works for you and what doesn't in this book; teaching is not a one-way street. You can write me at:

1599 Bittern Drive
Sunnyvale, CA 94087

THE PATH TO THE ZEN OF ASSEMBLER

The Zen of Assembly Language consists of four major parts, contained in two volumes. Parts I and II are in this volume, Volume I, while Parts III and IV are in Volume II, The Zen of Assembly Language: The Flexible Mind. While the book you're reading stands on its own as a tutorial in high-performance assembler code, the two volumes together cover the whole of superior assembler programming, from hardware to implementation. I strongly recommend that you read both. The four parts of The Zen of Assembly Language are organized as follows.

Part I introduces the concept of the Zen of assembler, and presents the tools we'll use to delve into assembler code performance.

Part II covers various and sundry pieces of knowledge about assembler programming, examines the resources available when programming the PC, and probes fundamental hardware aspects that affect

code performance.

Part III (in Volume II) examines the process of creating superior code, combining the detailed knowledge of Part II with varied and often unorthodox coding approaches.

Part IV (also in Volume II) illustrates the Zen of assembler in the form of a working animation program.

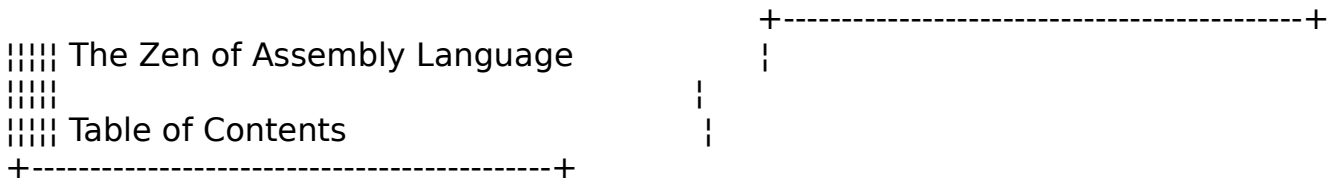
In general, Parts I and II discuss the raw stuff of performance, while Parts III and IV show how to integrate that raw performance with algorithms and applications, although there is considerable overlap. The four parts together teach all aspects of the Zen of assembler: concept, knowledge, the flexible mind, and implementation. Together, we will follow that path down the road to mastery of the IBM PC.

Shall we begin?

-- Michael Abrash

Sunnyvale, CA

May 29, 1989



PART I: THE ZEN OF ASSEMBLER

Chapter 1: Zen?

The Zen of assembler in a nutshell
Assembler is fundamentally different from other languages
Knowledge
The flexible mind
Where to begin?

Chapter 2: Assume Nothing

The Zen timer
The Zen timer is a means, not an end
Starting the Zen timer
Time and the PC
Stopping the Zen timer
Reporting timing results
Notes on the Zen timer
A sample use of the Zen timer
The long-period Zen timer
Stopping the clock
A sample use of the long-period Zen timer
Further reading
Armed with the Zen timer, onward and upward

PART II: KNOWLEDGE

Chapter 3: Context

From the bottom up
The traditional model
Cycle-eaters
Code is data
Inside the 8088
Stepchild of the 8086
Which model to use?

Chapter 4: Things Mother Never Told You:
Under the Programming Interface

Cycle-eaters revisited
The 8-bit bus cycle-eater
The impact of the 8-bit bus cycle-eater
What to do about the 8-bit bus cycle-eater?
The prefetch queue cycle-eater
Official execution times are only part of the story
There is no such beast as a true instruction execution time
Approximating overall execution times
What to do about the prefetch queue cycle-eater?
Holding up the 8088
Dynamic RAM refresh: the invisible hand
How DRAM refresh works in the PC
The impact of DRAM refresh
What to do about the DRAM refresh cycle-eater?
Wait states
The display adapter cycle-eater
The impact of the display adapter cycle-eater
What to do about the display adapter cycle-eater?
Cycle-eaters: a summary
What does it all mean?

Chapter 5: Night of the Cycle-Eaters

No, we're not in Kansas anymore
Cycle-eaters by the battalion
...there's still no such beast as a true instruction time

170 cycles in the life of a PC
The test set-up
The results
Code execution isn't all that exciting
The 8088 really does coprocess
When does an instruction execute?
The true nature of instruction execution
Variability
You never know unless you measure (in context!)
The longer the better
Odds and ends
Back to the programming interface

Chapter 6: The 8088

An overview of the 8088
Resources of the 8088
Registers The 8088's register set
The general-purpose registers
The AX register
The BX register
The CX register
The DX register
The SI register
The DI register
The BP register
The SP register
The segment registers
The CS register
The DS register
The ES register
The SS register
The instruction pointer
The FLAGS register
The Carry flag (CF)
The Parity flag (PF)
The Auxiliary Carry flag (AF)
The Zero flag (ZF)
The Sign flag (SF)
The Overflow flag (OF)
The Interrupt flag (IF)

The Direction flag (DF)
The Trap flag (TF)
There's more to life than registers

Chapter 7: Memory Addressing

Definitions
Square brackets mean memory addressing
The memory architecture of the 8088
Segments and offsets
Segment:offset pairs aren't unique
Good news and bad news
More good news
Notes on optimization
A final word on segment:offset addressing
Segment handling
What can you do with segment registers? Not much
Using segment registers for temporary storage
Setting and copying segment registers
Loading 20-bit pointers with **lds** and **les**
Loading doublewords with **les**
Segment:offset and byte ordering in memory
Loading SS
Extracting segment values with the **seg** directive
Joining segments
Segment override prefixes **assume** and segment override prefixes
Offset handling
Loading offsets
mod-reg-rm addressing
What's mod-reg-rm addressing good for?
Displacements and sign-extension
Naming the mod-reg-rm addressing modes
Direct addressing
Miscellaneous information about memory addressing
mod-reg-rm addressing: the dark side
Why memory accesses are slow
Some mod-reg-rm memory accesses are slower than others
Performance implications of effective address calculations
mod-reg-rm addressing: slow, but not quite as slow as you think
The importance of addressing well
The 8088 is faster at memory address calculations than you are

Calculating effective addresses with **lea**
Offset wrapping at the ends of segments
Non-mod-reg-rm memory addressing
Special forms of common instructions
The string instructions
Immediate addressing
Sign-extension of immediate operands
mov doesn't sign-extend immediate operands
Don't **mov** immediate operands to memory if you can help it
Stack addressing
An example of avoiding **push** and **pop**
Miscellaneous notes about stack addressing
Stack frames
When stack frames are useful
Tips on stack frames
Stack frames are often in DS
Use BP as a normal register if you must
The many ways of specifying mod-reg-rm addressing
xlat
Memory is cheap: you could look it up
Five ways to double bits
Table look-ups to the rescue
There are many ways to approach any task
Initializing memory
A brief note on I/O addressing
Video programming and I/O
Avoid memory!

Chapter 8: Strange Fruit of the 8080

The 8080 legacy
More than a passing resemblance
Accumulator-specific instructions
Accumulator-specific direct-addressing instructions Looks aren't everything
How fast are they?
When should you use them?
Accumulator-specific immediate-operand instructions
An accumulator-specific example
Other accumulator-specific instructions
The accumulator-specific version of **test**

The AX-specific version of **xchg**
Pushing and popping the 8080 flags
lahf and **sahf**: an example
A brief digression on optimization
Onward through the instruction set

Chapter 9: Around and About the Instruction Set

Shortcuts for handling zero and constants
Making zero
Initializing constants from the registers
Initializing two bytes with a single **mov**
More fun with zero
inc and **dec**
Using 16-bit **inc** and **dec** instructions for 8-bit operations
How **inc** and **add** (and **dec** and **sub**) differ--and why
Carrying results along in a flag
Byte-to-word and word-to-doubleword conversion
xchg is handy when registers are tight
Destination: register
neg and **not**
Shifting and rotating memory
Rotates
Shifts
Signed division with **sar**
Bit-doubling made easy
ASCII and decimal adjust
daa, **das**, and packed BCD arithmetic
aam, **aad**, and unpacked BCD arithmetic
Notes on **mul** and **div**
aaa, **aas**, and decimal ASCII arithmetic
Mnemonics that cover multiple instructions
On to the string instructions

Chapter 10: String Instructions: The Magic Elixir

A quick tour of the string instructions
Reading memory: **lods**
Writing memory: **stos**
Moving memory: **movs**

Scanning memory: **scas**
Notes on loading segments for string instructions Comparing memory: **cmps**
Hither and yon with the string instructions
Data size, advancing pointers, and the Direction flag
The **rep** prefix
rep = no instruction fetching + no branching
repz and **repnz**
rep is a prefix, not an instruction
Of counters and flags
Of data size and counters
Handling very small and very large blocks
Words of caution
Segment overrides: sometimes you can, sometimes you can't
The good and the bad of segment overrides
...leave ES and/or DS set for as long as possible
rep and segment prefixes don't mix
On to string instruction applications

Chapter 11: String Instruction Applications

String handling with **lods** and **stos**
Block handling with **movs**
Searching with **scas**
scas and zero-terminated strings
More on **scas** and zero-terminated strings
Using repeated **scasw** on byte-sized data
scas and look-up tables
Consider your options
Comparing memory to memory with **cmps**
String searching
cmps without **rep**
A note about returning values
Putting string instructions to work in unlikely places
Animation basics
String instruction-based animation
Notes on the animation implementations
A note on handling blocks larger than 64 K bytes
Conclusion

Chapter 12: Don't Jump!

How slow is it?
Branching and calculation of the target address
Branching and the prefetch queue
The prefetch queue empties when you branch
Branching instructions do prefetch
Branching and the second byte of the branched-to instruction
Don't jump!
Now that we know why not to branch...

Chapter 13: Not-Branching

Think functionally
rep: looping without branching
Look-up tables: calculating without branching
Take the branch less travelled by
Put the load on the unimportant case
Yes, Virginia, there is a faster 32-bit negate!
How 32-bit negation works
How fast 32-bit negation works
Arrange your code to eliminate branches
Preloading the less common case
Use the Carry flag to replace some branches
Never use two jumps when one will do
Jump to the land of no return
Don't be afraid to duplicate code
Inside loops is where branches really hurt
Two loops can be better than one
Make up your mind once and for all
Don't come calling
Smaller isn't always better
loop may not be bad, but Lord knows it's not good: in-line code
Branched-to in-line code: flexibility needed and found
Partial in-line code
Partial in-line code: limitations and workarounds
Partial in-line code and strings: a good match
Labels and in-line code
A note on self-modifying code
Conclusion

Chapter 14: If You Must Branch...

Don't go far
How to avoid far branches
Odds and ends on branching far
Replacing **call** and **ret** with **jmp**
Flexibility ad infinitum
Tinkering with the stack in a subroutine
Beware of letting DOS do the work
Forward references can waste time and space
The right assembler can help
Saving space with branches
Multiple entry points
A brief Zen exercise in branching (and not-branching)
Double-duty tests
Using loop counters as indexes
The looping instructions
loopz and **loopnz**
How you loop matters more than you might think Only **jcxz** can test and branch in a single bound
Jump and call tables
Partial jump tables
Generating jump table indexes
Jump tables, macros, and branched-to in-line code
Forward references rear their collective ugly head once more
Still and all...don't jump!
This concludes our tour of the 8088's instruction set

Chapter 15: Other Processors

Why optimize for the 8088?
Which processors matter?
The 80286 and the 80386
Things Mother never told you, part II
System wait states
Data alignment
Code alignment
Alignment and the 80386
Alignment and the stack
The DRAM refresh cycle-eater: still an act of God
The display adapter cycle-eater

New instructions and features: the 80286
New instructions and features: the 80386
Optimization rules: the more things change...
Detailed optimization
Don't sweat the details
popf and the 80286
Coprocessors and peripherals
A brief note on the 8087
Conclusion

Chapter 16: Onward to the Flexible Mind

A taste of what you've learned
Zenning
Knowledge and beyond

APPENDICES

Appendix A: An 8088 instruction set reference

Appendix B: ASCII table and PC character set