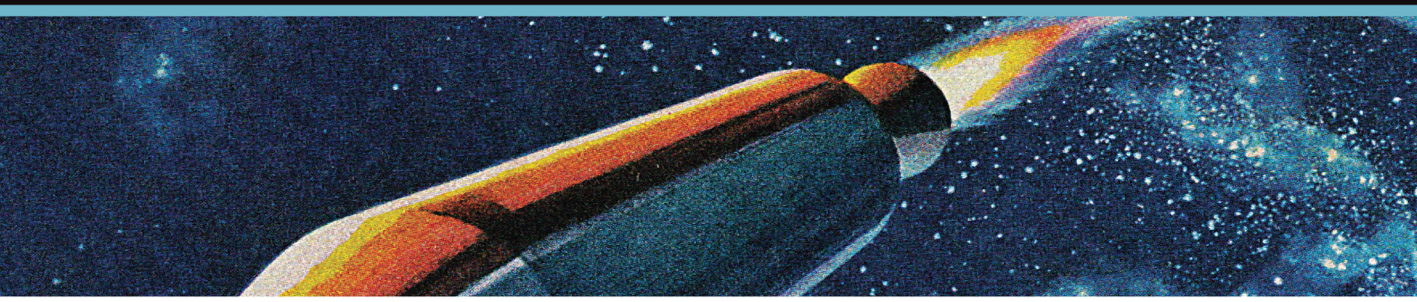
TECH TODAY

x64 ASSEMBLY LANGUAGE STEP-BY-STEP

Programming with Linux®

4TH EDITION

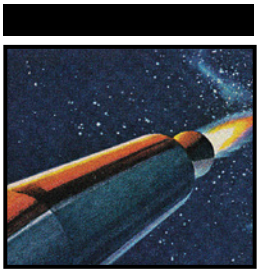


JEFF DUNTEMANN

WILEY

x64 Assembly Language Step-by-Step

4TH Edition



x64 Assembly Language Step-by-Step

Programming with Linux®

4TH Edition

Jeff Duntemann

WILEY

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.
Published simultaneously in Canada and the United Kingdom.

ISBNs: 9781394155248 (Hardback), 9781394155545 (epdf), 9781394155255 (epub)

Library of Congress Control Number: 2023944290

Cover image: © CSA-Printstocks/Getty Images
Cover design: Wiley

To the eternal memory of my father,

Frank W. Duntemann, Engineer

1922–1978

Who said, "When you build 'em right, they fly."

You did. And I do.

About the Author

Jeff Duntemann has had his technical nonfiction and science-fiction work professionally published since 1974. He worked as a programmer for Xerox Corporation and as a technical editor for Ziff-Davis Publishing and Borland International. He launched and edited two print magazines for programmers and has 20 technical books to his credit, including the bestselling *Assembly Language Step-by-Step*. He wrote the "Structured Programming" column in *Dr. Dobb's Journal* for four years and has published dozens of technical articles in many magazines. He has a longstanding interest in "strong" artificial intelligence, and most of his fiction explores the possibilities and consequences of strong AI. His other interests include electronics and amateur radio (callsign K7JPD), telescopes, and kites. Jeff lives in Phoenix, Arizona, with his wife Carol.

About the Technical Editor

David Stafford is an enthusiast of low-level programming in assembly language, from 8-bit processors to modern 64-bit multicore architectures. He lives in the Seattle area and works in the field of artificial intelligence for robotics.

Acknowledgments

Thanks are due to a number of people who helped me out as this edition took shape, in various ways. First, thanks to Jim Minatel and Pete Gaughan of Wiley, who got the project underway and made sure it went to completion. Also thanks to David Stafford, who acted as technical editor and provided a constant stream of invaluable advice.

The event shocked me to the core, but antony-jr of GitHub managed to create a working Linux appimage of the quirky and ancient but very accessible Insight debugger, which was pulled from Linux repositories soon after the third edition of this book hit print in 2009. A very big thanks to him for what was likely a *very* peculiar project. You can find his Insight appimage here: `https://appimage.github.io/Insight`.

Abundant thanks also to Dmitriy Manushin, who created SASM, a free assembly language IDE targeted at beginners: `https://dman95.github.io/SASM/english.html`.

When I ran into a weirdness in `glibc`, my wizard crew on Contrapositive Diary helped me figure it out:

- Jim Strickland
- Bill Buhler
- Jason Bucata
- Jonathan O'Neal
- Bruce and Keith (last names not given, and that's OK—the advice was golden)

Finally, and as always, a toast to Carol for her support and sacramental friendship that have enlivened me now for 54 years and enabled me to take on difficult projects like this and see them through to the end, no matter how nuts they made me along the way!

Contents at a Glance

Contents

Introduction

"Why Would You Want to Do That?*"*

It was 1985, and I was in a chartered bus in New York City, heading for a press reception with a bunch of other restless media egomaniacs. I was only beginning my tech journalist career (as technical editor for *PC Tech Journal*), and my first book was still months in the future. I happened to be sitting next to an established programming writer/guru, with whom I was impressed and to whom I was babbling about one thing or another. I would like to eliminate this statement; it adds little to the book, and as annoying as he is, even though we don't name him, I now understand why he's so annoying: He lives and works in a completely different culture than I do.

During our chat, I happened to let slip that I was a Turbo Pascal fanatic, and what I really wanted to do was learn how to write Turbo Pascal programs that made use of the brand new Microsoft Windows user interface. He wrinkled his nose and grimaced wryly, before speaking the Infamous Question:

"Why would you want to do *that*?"

I had never heard the question before (though I would hear it many times thereafter), and it took me aback. Why? Because, well, because. . .I wanted to know how it *worked*.

"Heh. That's what C is for."

Further discussion got me nowhere in a Pascal direction. But some probing led me to understand that you *couldn't* write Windows apps in Turbo Pascal. It was impossible. Or. . .the programming writer/guru didn't know how. Maybe both. I never learned the truth as it stood in 1985. (Delphi answered the question once and for all in 1995.) But I did learn the meaning of the Infamous Question.

Note well: When somebody asks you, "Why would you want to do *that*?" what it really means is this: "You've asked me how to do something that is either

impossible using tools that I favor or completely outside my experience, but I don't want to lose face by admitting it. So. . .how 'bout those Blackhawks?"

I heard it again and again over the years:

Q: How can I set up a C string so that I can read its length without scanning it?

A: Why would you want to do *that?*

Q: How can I write an assembly language subroutine callable from Turbo Pascal?

A: Why would you want to do *that?*

Q: How can I write Windows apps in assembly language?

A: Why would you want to do *that?*

You get the idea. The answer to the Infamous Question is always the same, and if the weasels ever ask it of you, snap back as quickly as possible: *because I want to know how it works*.

That is a completely sufficient answer. It's the answer I've used every single time, except for one occasion a considerable number of years ago, when I put forth that I wanted to write a book that taught people how to program in assembly language as their *first* experience in programming.

Q: Good grief, why would you want to do *that?*

A: Because it's the best way there is to build the skills required to understand how *all the rest* of the programming universe works.

Being a programmer is one thing above all else: It is understanding how things work. Learning to be a programmer, furthermore, is almost entirely a process of learning how things work. This can be done at various levels, depending on the tools you're using. If you're programming in Visual Basic, you have to understand how certain things work, but those things are by and large confined to Visual Basic itself. A great deal of machinery is hidden by the layer that Visual Basic places between the programmer and the computer. (The same is true of Delphi, Lazarus, Java, Python, and many other very high-level programming environments.) If you're using a C compiler, you're a lot closer to the machine, so you see a lot more of that machinery—and must, therefore, understand how it works to be able to use it. However, quite a bit remains hidden, even from the hardened C programmer.

If, on the other hand, you're working in assembly language, you're as close to the machine as you can get. Assembly language hides *nothing*, and withholds no power. The flipside, of course, is that no magical layer between you and the machine will absolve any ignorance and "take care of" things for you. If you don't understand how something works, you're dead in the water—unless you know enough to be able to figure it out on your own.

That's a key point: My goal in creating this book is not entirely to teach you assembly language *per se*. If this book has a prime directive at all, it is to impart a certain disciplined curiosity about the underlying machine, along with some basic context from which you can begin to explore the machine at its very lowest levels—that, and the confidence to give it your best shot. This is difficult stuff, but it's nothing you can't master given some concentration, patience, and the time it requires—which, I caution, may be considerable.

In truth, what I'm really teaching you here is how to learn.

What You'll Need

To program as I intend to teach, you're going to need a 64-bit Intel computer running a 64-bit distribution of Linux. The one I used in preparing this book is Linux Mint Cinnamon V20. 3 Una. "Una" here is a code name for this version of Linux Mint. It's nothing more than a short way of saying "Linux Mint 20.3." I recommend Mint; it's thrown me fewer curves than any other distro I've ever used—and I've used Linux here and there ever since it first appeared. I don't think which graphical shell you use matters a great deal. I like Cinnamon, but you can use whatever you like or are familiar with.

You need to be reasonably proficient with Linux at the user level. I can't teach you how to install, configure, and run Linux in this book. If you're not already familiar with Linux, get a tutorial text and work through it. There are many such online.

You'll need a piece of free software called SASM, which is a simple interactive development environment (IDE) for programming in assembly. Basically, it consists of an editor, a build system, and a front end to the standard Linux debugger `gdb`. You'll also need a free assembler called NASM.

You don't have to know how to download, install, and configure these tools in advance because, at the appropriate times, I'll cover all necessary tool installation and configuration.

Do note that other Unix implementations not based on the Linux kernel may not function precisely the same way under the hood. BSD Unix uses different conventions for making system calls, for example, and other Unix versions like Solaris are outside my experience.

Remember that *this book is about the x64 architecture*. To the extent that x64 contains x86, I will also be teaching elements of the x86 architecture. The gulf between 32-bit x86 and 64-bit x64 is a *lot* narrower than the gulf between 16-bit x86 and 32-bit x86. If you already have a firm grounding in 32-bit x86, you'll breeze through most of this book at a gallop. If you can do that, cool—just please remember that the book is for those who are just starting out in programming on Intel CPUs.

Also remember that this book is limited in size by its publisher: Paper, ink, and cover stock aren't free. That means I have to narrow the scope of what I teach and explain within those limits. I wish I had the space to cover the AVX math subsystem. I don't. But I'll bet that once you go through this book, you can figure much of it out by yourself.

The Master Plan

This book starts at the beginning, and I mean the *beginning*. Maybe you're already there, or well past it. I respect that. I still think that it wouldn't hurt to start at the first chapter and read through all the chapters in order. Review is useful, and hey—you may realize that you didn't know *quite* as much as you thought you did. (Happens to me all the time!)

But if time is at a premium, here's the cheat sheet:

- If you already understand the fundamental ideas of computer programming, skip Chapter 1.
- If you already understand the ideas behind number bases other than decimal (especially hexadecimal and binary), skip Chapter 2.
- If you already have a grip on the nature of computer internals (memory, CPU architectures, and so on) skip Chapter 3.
- If you already understand x64 memory addressing, skip Chapter 4.
- No. Stop. Scratch that. Even if you already understand x64 memory addressing, *read Chapter 4*.

The last bullet is there, and emphatic, for a reason: *Assembly language programming is about memory addressing*. If you don't understand memory addressing, nothing else you learn in assembly will help you one. . .bit. So, don't skip Chapter 4 no matter what else you know or think you know. Start from there, and see it through to the end. Memory addressing comes up regularly throughout the rest of the book. It's really the heart of the topic.

Load every example program, assemble each one, and run them all. Strive to understand every single line in every program. Take nothing on faith. Furthermore, don't stop there. Change the example programs as things begin to make sense to you. Try different approaches. Try things that I don't mention. Be audacious. Nay, go nuts—bits don't have feelings, and the worst thing that can happen is that Linux throws a segmentation fault, which may hurt your program but does not hurt Linux. The only catch is that when you do try something, strive to understand why it *doesn't* work as clearly as you understand all the other things that do. Single-step your way through a program in the SASM debugger, even when the program works. *Take notes*.

That is, ultimately, what I'm after: to show you the way to understand what every however distant corner of your machine is doing and how all its many pieces work together. This doesn't mean I'll explain every corner of it myself—no one will live long enough to do that because computing isn't simple anymore—but if you develop the discipline of patient research and experimentation, you can probably work it out for yourself. Ultimately, that's the only way to learn it: by yourself. The guidance you find—in friends, on the Net, in books like this—is only guidance and grease on the axles. You have to decide who's to be the master, you or the machine, and make it so. Assembly programmers are the only programmers who can truly claim to be masters, which is a truth worth meditating on.

A Note on Capitalization Conventions

Assembly language is peculiar among programming languages in that there is no universal standard for case-sensitivity. In the C language, all identifiers are case-sensitive, and I have seen assemblers that do not recognize differences in case at all. NASM, the assembler I'm presenting in this book, is case-sensitive only for programmer-defined identifiers. The instruction mnemonics and the names of registers, however, are *not* case sensitive.

There are customs in the literature on assembly language, and one of those customs is to treat CPU instruction mnemonics as uppercase in the chapter text and in lowercase in source code files and code snippets interspersed within the text. I'll be following that custom here. Within discussion text, I'll speak of `MOV` and `CALL` and `CMP`. In example code, it will be `mov` and `call` and `cmp`. Code snippets and listings will be in a monospace Courier-style font. When mentioned in the text, registers will be in uppercase but not in the Courier font and lowercase in snippets and listings.

There are two reasons for this:

- In text discussions, the mnemonics need to stand out. It's too easy to lose track of them amid a torrent of ordinary mixed-case words.
- To read and learn from existing documents and source code outside of this one book, you need to be able to easily read assembly language whether it's in uppercase, lowercase, or mixed case. Getting comfortable with different ways of expressing the same things is important.

Remember Why You're Here

Anyway. Wherever you choose to start the book, it's time to get underway. Just remember that whatever gets in your face, be it the weasels, the machine, or

your own inexperience, the thing to keep in the forefront of your mind is this: *You're in it to figure out how it works*.

Let's go.

Jeff Duntemann
Scottsdale, Arizona
May 24, 2023

x64 Assembly Language Step-by-Step

4TH Edition

CHAPTER

1

It's All in the Plan

Understanding What Computers Really Do

Another Pleasant Valley Saturday

"Quick, Mike, get your sister and brother up; it's past 7. Nicky's got Little League at 9, and Dione's got ballet at 10. Give Max his heartworm pill! (We're out of them, Mom, remember?) Your father picked a great weekend to go fishing . . . . Here, let me give you 10 bucks and go get more pills at the vet's . . . . My God, that's right, Hank needed gas money and left me broke. There's a teller machine over by Kmart, and if I go there, I can take that stupid toilet seat back and get the right one. "I guess I'd better make a list . . . ."

It's another Pleasant Valley Saturday, and 30-odd million suburban homemakers sit down with a pencil and pad at the kitchen table to try to make sense of a morning that would kill and pickle any lesser being. In her mind she thinks of the dependencies and traces the route:

"Drop Nicky at Rand Park, go back to Dempster, and it's about 10 minutes to Golf Mill Mall. Do I have gas? I'd better check first—if not, stop at Del's Shell or I won't make it to Milwaukee Avenue. Milk the teller machine at Golf Mill; then cross the parking lot to Kmart to return the toilet seat that Hank

bought last weekend without checking what shape it was. Gotta remember to throw the toilet seat in back of the van—write that at the top of the list.

"By then it'll be half past, maybe later. Ballet is all the way down Greenwood in Park Ridge. No left turn from Milwaukee—but there's the sneak path around behind the mall. I have to remember not to turn right onto Milwaukee like I always do—jot that down. While I'm in Park Ridge, I can check and see if Hank's new glasses are in—should call, but they won't even be open until 9:30. Oh, and groceries—can do that while Dione dances. On the way back I can cut over to Oakton and get the dog's pills."

In about 90 seconds flat the list is complete:

- Throw toilet seat in van.
- Check gas—if empty, stop at Del's Shell.
- Drop Nicky at Rand Park.
- Stop at Golf Mill teller machine.
- Return toilet seat at Kmart.
- Drop Dione at ballet (remember the sneak path to Greenwood).
- See if Hank's glasses are at Pearle Vision—if they are, make double sure they remembered the extra scratch coating.
- Get groceries at Jewel.
- Pick up Dione.
- Stop at vet for heartworm pills.
- Drop off groceries at home.
- If it's time, pick up Nicky. If not, collapse for a few minutes; then pick up Nicky.
- Collapse!

What we often call a "laundry list" (whether it involves laundry or not) is the perfect metaphor for a computer program. Without realizing it, our intrepid homemaker has written herself a computer program and then set out (with herself acting as the computer) to execute it and be done before noon.

Computer programming is nothing more than this: you the programmer write a list of steps and tests. The computer then performs each step and test in sequence. When the list of steps has been executed, the computer stops.

A computer program is a list of steps and tests, nothing more.

Steps and Tests

Think for a moment about what I call a test in the preceding laundry list. A *test* is the sort of either/or decision we make dozens or hundreds of times on even the most placid of days, sometimes nearly without thinking about it.

Our homemaker performed a test when she jumped into the van to get started on her adventure. She looked at the gas gauge. The gas gauge would tell her one of two things: (1) she has enough gas, or (2) she doesn't. If she has enough gas, she takes a right and heads for Rand Park. If she doesn't have enough gas, she takes a left down to the corner and fills the tank at Del's Shell. (Del takes credit cards.) Then, with a full tank, she continues the program by making a U-turn and heading for Rand Park.

In the abstract, a test consists of these two parts:

- First, you take a look at something that can go one of two ways.
- Then you do one of two things, depending on what you saw when you took a look.

Toward the end of the program, our homemaker got home, took the groceries out of the van, and looked at the clock. If it isn't time to get Nicky back from Little League, she has a moment to collapse on the couch in a nearly empty house. If it *is* time to get Nicky, there's no rest for the ragged: she sprints for the van and heads back to Rand Park.

(Any guesses as to whether she really gets to rest when the program finishes running?)

More Than Two Ways?

You might object, saying that many or most tests involve more than two alternatives. Sorry, you're wrong—in every case. Read this twice: except for totally impulsive or psychotic behavior, every human decision comes down to the choice between two alternatives.

What you have to do is look a little more closely at what goes through your mind when you make decisions. The next time you buzz down to Chow Now for fast Chinese, observe yourself while you're poring over the menu. The choice might seem, at first, to be of one item out of 26 Cantonese main courses. Not so—the choice, in fact, is between choosing one item and *not* choosing that one item. Your eyes rest on chicken with cashews. Naw, too bland. *That was a test*. You slide down to the next item. Chicken with black mushrooms. Hmmm, no, had that last week. *That was another test*. Next item: kung pao chicken. Yeah, that's it! *That was a third test*.

The choice was not among chicken with cashews, chicken with black mushrooms, and chicken with kung pao. Each dish had its moment, poised before the critical eye of your mind, and you turned thumbs up or thumbs down on it, individually. Eventually, one dish won, but it won in that same game of "to eat or not to eat."

Let me give you another example. Many of life's most complicated decisions come about because 99.99867 percent of us are not nudists. You've been there: you're standing in the clothes closet in your underwear, flipping through your rack of pants. The tests come thick and fast. This one? No. This one? No. This one? No. This one? Yeah. You pick a pair of blue pants, say. (It's a Monday, after all, and blue would seem an appropriate color.) Then you stumble over to your sock drawer and take a look. Whoops, no blue socks. *That was a test*. So you stumble back to the clothes closet, hang your blue pants back on the pants rack, and start over. This one? No. This one? No. This one? Yeah. This time it's brown pants, and you toss them over your arm and head back to the sock drawer to take another look. Nertz, out of brown socks, too. So it's back to the clothes closet . . . .

What you might consider a single decision, or perhaps two decisions inextricably tangled (like picking pants and socks of the same color, given stock on hand), is actually a series of small decisions, always binary in nature: pick 'em or don't pick 'em. Find 'em or don't find 'em. The Monday morning episode in the clothes closet is a good analogy of a programming structure called a *loop*: you keep doing a series of things until you get it right, and then you stop (assuming you're not the kind of geek who wears blue socks with brown pants). But whether you get everything right always comes down to a sequence of simple either/or decisions.

Computers Think Like Us

I can almost hear what you're thinking: "Sure, it's a computer book, and he's trying to get me to think like a computer." Not at all. Computers think like *us*. We designed them; how else could they think? No, what I'm trying to do is get you to take a long, hard look at how *you* think. We run on automatic for so much of our lives that we literally do most of our thinking without really thinking about it.

The best model for the logic of a computer program is the same logic we use to plan and manage our daily affairs. No matter what we do, it comes down to a matter of confronting two alternatives and picking one. What we might think of as a single large and complicated decision is nothing more than a messy tangle of many smaller decisions. The skill of looking at a complex decision and seeing all the little decisions in its tummy will serve you well in learning how to

program. Observe yourself the next time you have to decide something. Count up the little decisions that make up the big one. You'll be surprised.

And, surprise! You'll be a programmer.

Had This Been the Real Thing . . .

Do not be alarmed. What you have just experienced was a metaphor. It was not the real thing. (The real thing comes later.)

I use metaphors a lot in this book. A metaphor is a loose comparison drawn between something familiar (such as a Saturday morning laundry list) and something unfamiliar (such as a computer program). The idea is to anchor the unfamiliar in terms of the familiar so that when I begin tossing facts at you, you'll have someplace comfortable to lay them down.

The most important thing for you to do right now is keep an open mind. If you know a little bit about computers or programming, don't pick nits. Yes, there are important differences between a homemaker following a scribbled laundry list and a computer executing a program. I'll mention those differences all in good time.

For now, it's still Chapter 1. Take these initial metaphors on their own terms. Later, they'll help a lot.

Assembly Language Programming As a Square Dance

Carol and I have a certain fondness for "called" dances, the most prevalent type being square dances. There are others, like New England contra dances, which are a lot like square dances but with better music. In a called dance, the caller person at the front of the hall calls out movements, and the dancers perform those movements. The music provides a beat, like the ticking of a clock. The sequence of movements taken together is the dance, and the dance usually has a name.

The first time Carol and I attended a contra dance, I was poleaxed: *this was like assembly language programming!* The caller called out "allemande left," and we performed the movement known as "allemande left." The caller called out "forward and back," and we executed the "forward and back" movement. The caller called out "box the gnat," and, well, we boxed the gnat. (I am not making this up!) There are a reasonable number of movements, and to be good at that sort of dancing, you have to memorize them all by name. Otherwise, if the caller calls a movement that you don't know, the dance might stumble or grind to a halt. (Bluescreen!)

At its deepest level, a computer understands a collection of individual operations called *instructions*. These perform arithmetic, execute logic like AND and OR, move data around, and do many other things. Each instruction is performed inside the CPU chip. Just as a set of dance movements are the individual atoms of motion making up a square dance, instructions are the atoms of a computer program. The program is like the dance as a whole: a sequence of instructions executed in order. The couples taking part in the dance execute the dance/program as the caller moves down the list of movements, calling out each one in turn. The couples, then, are the computer on which the dance runs.

That's about as far as the square dance metaphor goes. Once you get the knack of assembly language, hey, go take square dance or contra dance lessons somewhere and see if you don't come to the same conclusion that I did.

Assembly Language Programming As a Board Game

Board games were a really big deal when I was a kid, when board games were actually printed on a species of board. (OK, cardboard.) Monopoly was one that almost everybody had. There was a sort of pathway around the edge of the board divided into squares. You had a game piece that advanced from square to square according to dice throws, and when your piece landed on a square, you could do one of several things: buy property that hadn't been bought yet, pay rent on property owned by other players, pull a card from the Chance stack, or—eek!—go to jail. You had a pile of Monopoly money to spend, and when another player had to pay rent, you got more.

The specifics of the Monopoly game aren't important here. What matters is that you progress through a series of steps, and at each step, something happens. Your pile of money grows or shrinks. Assembly language is a little like that: a program is like the game board. Each step in the program does something. There are places where you can store numbers. The numbers change as you move through the program.

Now that you're thinking in terms of board games, take a look at Figure 1.1. What I've drawn is actually a fair approximation of assembly language as it was used on some of our simpler computers 50 or 60 years ago. The column marked "Program Instructions" is the main path around the edge of the board, of which only a portion can be shown here. This is the assembly language computer program, the actual series of steps and tests that, when executed, causes the computer to do something useful. Setting up this series of program instructions is what programming in assembly language actually *is*.

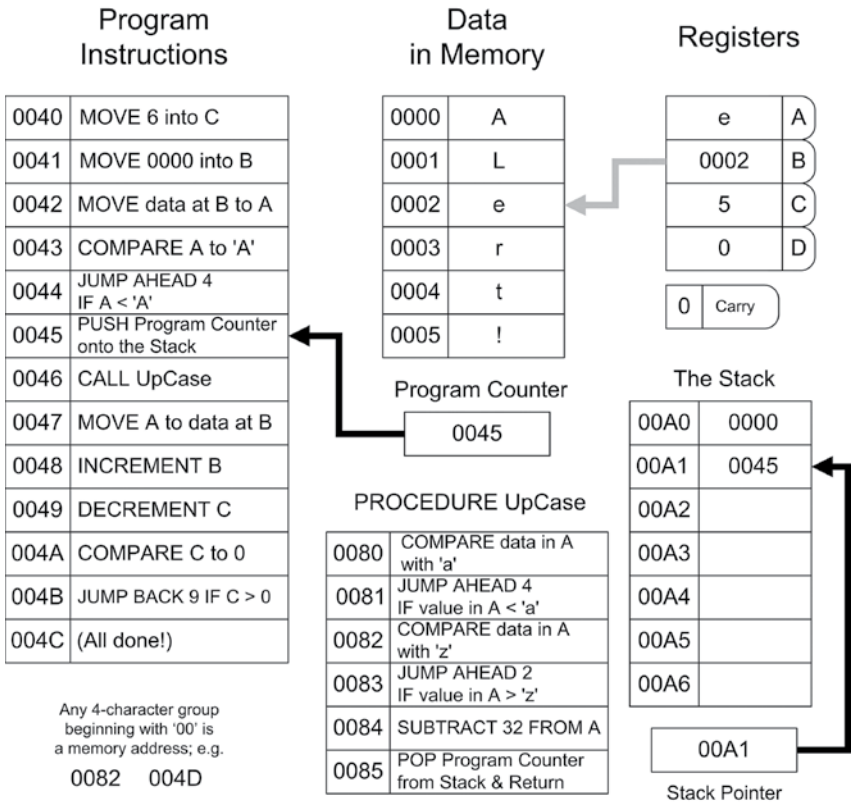


Figure 1.1: The Game of Assembly Language

Everything else is odds and ends in the middle of the board that serve the game in progress. Most of these are storage locations that contain your data. You're probably noticing (perhaps with sagging spirits) that there are a *lot* of numbers involved. (They're weird numbers, too. What, for example, does "004B" mean? I deal with that issue in Chapter 2, "Alien Bases.") I'm sorry, but that's simply the way the game is played. Assembly language, at its deepest level, is nothing *but* numbers, and if you hate numbers the way most people hate anchovies, you're going to have a rough time of it. (I like anchovies, which is part of my legend. Learn to like numbers. They're not as salty.) Higher-level programming languages such as Pascal or Python disguise the numbers by treating them symbolically. But assembly language, well, it's just you and the numbers.

I should caution you that the Game of Assembly Language in Figure 1.1 represents no real computer processor, like the Intel Core i5. Also, I've made the names of instructions more clearly understandable than the names of the instructions in Intel assembly language actually are. In the real world, instruction names

are typically short things like **LAHF**, **STC**, **INC**, **SHRX**, and other crypticisms that cannot be understood without considerable explanation. We're easing into this stuff sidewise, and in this chapter I have to sugarcoat certain things a little to draw the metaphors clearly.

Code and Data

Like most board games, the assembly language board game consists of two broad categories of elements: game steps and places to store things. The "game steps" are the steps and tests I've been speaking of all along. The places to store things are just that: cubbyholes into which you can place numbers, with the confidence that those numbers will remain where you put them until you take them out or change them somehow.

In programming terms, the game steps are called *code*, and the numbers in their cubbyholes (as distinct from the cubbyholes themselves) are called *data*. The cubbyholes themselves are usually called *storage*. (The difference between the places you store information and the information you store in them is crucial. Don't confuse them.) Consider an instruction in the Game of Assembly Language that says **ADD 32 to A**. An **ADD** instruction in the code alters a data value stored in a cubbyhole named Register A.

Code and data are two very different kinds of critters, but they interact in ways that make the game interesting. The code includes steps that place data into storage (**MOVE** instructions) and steps that alter data that is already in storage (**INCREMENT** and **DECREMENT** instructions, and **ADD** instructions, among others). Most of the time you'll think of code as being the master of data, in that the code writes data values into storage. Data does influence code as well, however. Among the tests that the code makes are tests that examine data in storage, the **COMPARE** instructions. If a given data value exists in storage, the code may do one thing; if that value does not exist in storage, the code will do something else, as in the **JUMP BACK** and **JUMP AHEAD** instructions.

The short block of instructions marked **PROCEDURE** is a detour off the main stream of instructions. At any point in the program you can duck out into the procedure, perform its steps and tests, and then return to the very place from which you left. This allows a sequence of steps and tests that is generally useful and used frequently to exist in only one place rather than exist as separate copies everywhere it's needed.

Addresses

Another critical concept lies in the funny numbers at the left side of the program step locations and data locations. Each number is unique, in that a location tagged with that number appears only *once* inside the computer. This location is called

an *address*. Data is stored and retrieved by specifying the data's address in the machine. Procedures are called by specifying the address at which they begin.

The little box (which is also a storage location) marked "Program Counter" keeps the address of the next instruction to be performed. The number inside the program counter is increased by one (*incremented*) each time an instruction is performed *unless the instruction tells the program counter to do something else*. For example, notice the **JUMP BACK 9** instruction at address 004B. When this instruction is performed, the program counter will "back up" by nine locations. This is analogous to the "go back three spaces" concept in most board games.

Metaphor Check!

That's about as much explanation of the Game of Assembly Language as I'm going to offer for now. This is still Chapter 1, and we're still in metaphor territory. People who have had some exposure to computers will recognize and understand some of what Figure 1.1 is doing. People with no exposure to computer innards at all shouldn't feel left behind for being utterly lost. I created the Game of Assembly Language solely to put across the following points:

- *The individual steps are very simple*. One single instruction rarely does more than move a single value from one storage cubbyhole to another, perform very elementary arithmetic like addition or subtraction, or compare the value contained in one storage cubbyhole to a value contained in another. This is good news, because it allows you to concentrate on the simple task accomplished by a single instruction without being overwhelmed by complexity. The bad news, however, is the next point.
- *It takes a lot of steps to do anything useful*. You can often write a useful program in such languages as Pascal or BASIC in five or six lines. You can actually create useful programs in visual programming systems like Visual Basic, Delphi, or Lazarus *without writing any code at all*. (The code is still there . . . but the code is "canned" and all you're really doing is choosing which chunks of canned code in a collection of many such chunks will run.) A useful assembly language program cannot be implemented in fewer than about 50 lines, and anything challenging takes hundreds or thousands—or tens of thousands—of lines. The skill of assembly language programming lies in structuring these hundreds or thousands of instructions so that the program both operates correctly and can still be read and understood by other programmers—and yourself—six months later.
- *The key to assembly language is understanding memory addresses*. In such languages as Pascal and BASIC, the compiler takes care of where something is located—you simply have to give that something a symbolic name and call it by that name whenever you want to look at it or change it.