Assembly Language: Step-by-Step

Jeff Duntemann

John Wiley & Sons, Inc.

New York • Chichester • Brisbane • Toronto • Singapore

For Kathleen M. Duntemann, Godmother

... who gave me books when all I could do was put teeth marks in

It was a good investment.

Library of Congress Cataloging-in-Publication Data

Duntemann, Jeff. 1952 -
Assembly language : step-by-step / Jeff Duntemann.
p. cm. Includes index.
ISBN 0-471-57814-2 (paper : alk. paper) 1. Assembler language (Computer program

language) QA76.73.A8D87 1992 005.265-dc20

I. Title.

92-16665 CIP

Printed in the United States of America

93 10 9876543

Introduction:

Agony in the Key of AX

What astonishes me about learning how to program is not that it's so hard, but that it's so *easy.* Am I nuts? Hardly. It's just that my curse is the curse of a perfect memory, and I remember *piano lessons.* My poor mother paid $600 in 1962 for a beautiful cherrywood spinet, and every week for two years I trucked off to Wilkins School of Music for a five dollar lesson. It wasn't that I was a reluctant student; I love music and I genuinely wanted to master the damned thing. But after two years, the best I could do was play "Camelot" well enough to keep the dog from howling. I can honestly say that nothing I ever tried

and failed to achieve after that (including engineering school and sailboarding) was anything close to that difficult.

That's why I say: if you can play the piano, you can learn to program in assembly language. Even if you can't play the piano, I hold that you can learn to program in assembly language, if:

- You've ever done your own long-form taxes
- You've earned a degree in medicine, law, or engineering
- You've ever put together your kid's swing set
- You've ever cooked a five-course dinner for eight and gotten everything to the table, hot, at all the right times

Still, playing the piano is the acid test. There are a lot more similarities than there are differences. To wit:

In both cases, you sit down in front of a big expensive machine with a keyboard. You try to memorize a system of notation that seems to have originated on Mars. You press the keys according to incomprehensible instructions in stacks of books. Ultimately, you sit there and grit your teeth while making so many mistakes your self-confidence dribbles out of your pores and disappears into the carpet padding. In many cases, it gets so bad that you hurl the books against the wall and stomp off to play Yahtzee with your little brother.

The differences are fewer: mistakes committed while learning assembly language won't make the dog howl. And, more crucially, what takes *years* of agony in front of a piano can be done in a couple of months in front of your average PC.

Furthermore, I'll do my best to help.

That's what this book is for: to get you started as an assembly-language programmer from a dead stop. I'll assume that you know how to run your machine. That is, I won't go through all that nonsense about flipping the big red switch and inserting a disk in a drive and holding down the Ctrl key while pressing the C key. Van Wolverton can teach you all that stuff.

On the other hand, I won't assume that you know *anything* about programming, nor very much about what happens inside the box itself. That means the first few sections will be the kind of necessary groundwork that will start you nodding off if you've been through it already. There's no helping that. Skip to Section 3 or so if you get bored.

I also have to come clean here and admit that this book is not intended to be a complete tutorial on assembly language, or even close to it. What I want to do is get you familiar enough with the jargon and the assumptions of assembly language so that you can pick up your typical "introduction" to assembly language and not get lost by page 6. I specifically recommend Tom Swan's excellent book, *Mastering Turbo Assembler*, which will take

you the rest of the way if you use Borland's assembler. A comparable book devoted to Microsoft's MASM has not yet been written, but even if you use MASM, Tom's book will still be valuable and you'll learn a lot from it. *Mastering Turbo Assembler* can occasionally be found in bookstores, or you can order it by mail through *PC TECHNIQUES* Bookstream.

Assembly language is almost certainly the most difficult kind of computer programming, but keep in mind that we're speaking in relative terms here. Five pushups are harder to do than five jumping jacks—but compared to running the Marathon, both amount to almost nothing. Assembly language is more difficult to learn than Pascal, but compared to raising your average American child from birth to five years, it's a cakewalk.

So don't let the mystique get you. Assembly-language programmers feel pretty smug about what they've learned to do, but in our workaday lives we are forced to learn and do things that put even assembly language to shame. If you're willing to set aside a couple months' worth of loose moments, you can pick it up too. Give it a shot. Your neighbors will thank you.

And so will the dog.

—-Jeff Duntemann Scottsdale, AZ March 1992

A Note to People Who Have Never Programmed Before

More than anyone else, this book was written for you. Starting with assembly language would not be most people's first choice in a computer language, but it's been done; it *can* be done, and it can be done with less agony than you might think. Still, it's a novel aim for a computer book, and I'd like you to do a little quality control for me and tell me how I'm doing.

While you're going through this book, ask yourself once in a while: is it working? And if

not, why not?
If I lose you somewhere in the discussion, jot a note in the margin. Tell me where I lost you. If possible, tell me why. (And saying, "I just don't get it" is perfectly acceptable, as long as you tell me where in the book you were when you started *not* to get it.)
As with all my books, I hope to keep this one in print well into the 21st century, revising it as need be to hone my technique and follow the technology. Telling me how the book works or doesn't work will, in time, help me make a better book.
Write to me at:
Jeff Duntemann *PC TECHNIQUES* Magazine
7721 E. Gray Road #204
Scottsdale, A2 85260

I can't reply individually to all letters, (not if I ever intend to get another book written!) but you'll have my eternal gratitude nonetheless.

How to Get the Most from this Book

By design, this is a serial-access book. I wrote it to be read like one of those bad/wonderful novels, starting at page one and moving right along to the end. Virtually all of the chapters depend on the chapters that came before them, and if you read a chapter here and a chapter there, there's some danger that the whole thing won't gel.
If you're already familiar with programming, you could conceivably skip Chapters 0,1, and 2. But why not assume there's a hole or two in parts of your experience and a little rust on the rest? Skill is not simply knowledge, but the resonance that comes of seeing how different facets of knowledge reinforce one another.

> Do it all. Get the big picture. (Keep in mind that I've hidden some funny stories in there as bait!)

Contents

Another Pleasant Valley Saturday

Understanding What Computers Really Do

0.1 It's All in the Plan

Quick, get the kids up, it's past 7. Nicky's got Little League at 9 and Dione's got ballet at 10. Mike, give Max his heartworm pill! (We're out of them, ma, remember?) Your father picked a great weekend to go fishing.. .here, let me give you ten 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 K-Mart, and I 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 thirty-odd million suburban home-makers sit down with a pencil and pad at the kitchen table to try and 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 ten 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. Bleed the teller machine at Golf Mill, then cross the parking lot to K-Mart 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 ninety 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 K-Mart
• Drop Dione at ballet (remember back 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's 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!

In 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 (acting as the computer) to execute it completely 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 laundry list shown above. 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) no, 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 taking a U-turn and heading for Rand Park.

In the abstract, a test consists of those 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 took a look at the clock. If it wasn'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 collapse when the program is complete?)

More than Two Ways?

You might object that many or most tests involve more than two alternatives.
Except for totally impulsive behavior, every human decision comes down to the choice of one of 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 Moo Foo Goo 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 Cashew Chicken. Naw, too bland. *That was a test.* You slide down to the next item. Chicken with Black Mushroom. 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 Cashew Chicken, Chicken with Black Mush-rooms, or Kung Pao Chicken. 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."
Many of life's most complicated decisions come about because 99% 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 analog 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 guy 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 very best model for the logic of a computer program is the very 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.

0.2 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'll be using metaphors a lot in this book. A metaphor is a loose comparison drawn between something familiar (like a Saturday morning laundry list) and something unfamiliar (like a computer program.) The idea is to anchor the unfamiliar in the terms of the familiar, so that when I begin tossing facts at you you'll have someplace comfortable to lay them down. The facts don't start until Chapter 1. (That's why I call this Chapter 0: Metaphors only, please.)

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 0. Take these initial metaphors on their own terms. Later on, they'll help a lot.

0.3 Do Not Pass GO

"There's a reason *bored* and *board* are homonyms," said my best friend Art one evening, as we sat (two super-sophisticated twelve-year-olds) playing some game in his basement. (He may have been unhappy because he was losing.) Was it Mille Bornes? Or Stratego? Or Monopoly? Or something else entirely? I confess I don't remember. I simply recall hopping some little piece of plastic shaped like a pregnant bowling pin up and down a series of colored squares that told me to do dumb things like go back two spaces or put $100 in the pot or nuke Outer Mongolia.

Outer Mongolia notwithstanding, there are strong parallels to be drawn between that peculiar American obsession, the board game, and assembly-language programming. First of all, everything we said before still holds: Board games, by and large, consist of a progression of steps and tests. In some games, like Trivial Pursuit, *every* step on the board is a test: To see if you can answer, or not answer, a question on a card. In other board games, each little square on the board contains some sort of instruction: Lose One Turn; Go Back Two Squares; Take a Card from Community Chest; and, of course, Go to Jail. Certain board games made for some lively arguments between Art and me (it was that or be bored, as it were) concerning what it meant to Go Forward or Backward Five Steps. It seemed to me that you should count the square you were already on. Art, traditionalist always, thought you should start counting with the first step in the direction you had to go. This made a difference in the game, of course. (I conveniently forgot to press my point when doing so would land me on something like Park Place with fifteen of Art's hotels on it...)

The Game of Big Bux

To avoid getting in serious trouble, I have invented my own board game to continue with this particular metaphor. In the sense that art mirrors life, the Game of Big Bux mirrors life in Silicon Valley, where money seems to be spontaneously created (generally in

somebody else's pocket) and the three big Money Black Holes are fast cars, California real estate, and messy divorces.

A portion of the Big Bux game board is shown on the following page. The line of rectangles on the left side of the page continues all the way around the board. In the middle of the board are cubbyholes to store your play money and game pieces; stacks of cards to be read occasionally; and short "detours" with names like Messy Divorce and Start a Business, which are brief sequences of the same sort of action rectangles as those forming the path around the edge of the board.

Unlike many board games, you don't throw dice to determine how many steps around the board you take. Big Bux requires that you move *one* step forward on each turn, *unless* the square you land on instructs you to move forward or backward or go somewhere else, like through a detour. This makes for a considerably less random game. In fact, Big Bux is a pretty deterministic game, meaning that whether you win or lose is far less important than just going through the ringer and coming out the other side. (Again, this mirrors Silicon Valley, where you come out either bankrupt or ready to flee to Peoria and open a hardware store. That *other kind* of hardware.)

There is some math involved. You start out with one house, a cheap car, and $50,000 in cash. You can buy CDs at a given interest rate, payable each time you make it once around the board. You can invest in stocks and other securities whose value is determined by a changeable index in economic indicators, which fluctuates based on cards chosen from the stack called

Fickle Finger of Fate. You can sell cars on a secondary market, buy and sell houses, and wheel and deal with the other players. Each time you make it once around the board you have to recalculate your net worth. All of this involves some addition, subtraction, multiplication, and division, but there's no math more complex than compound interest. Most of Big Bux involves nothing more than taking a step and following the instructions at each step. Is this starting to sound familiar?

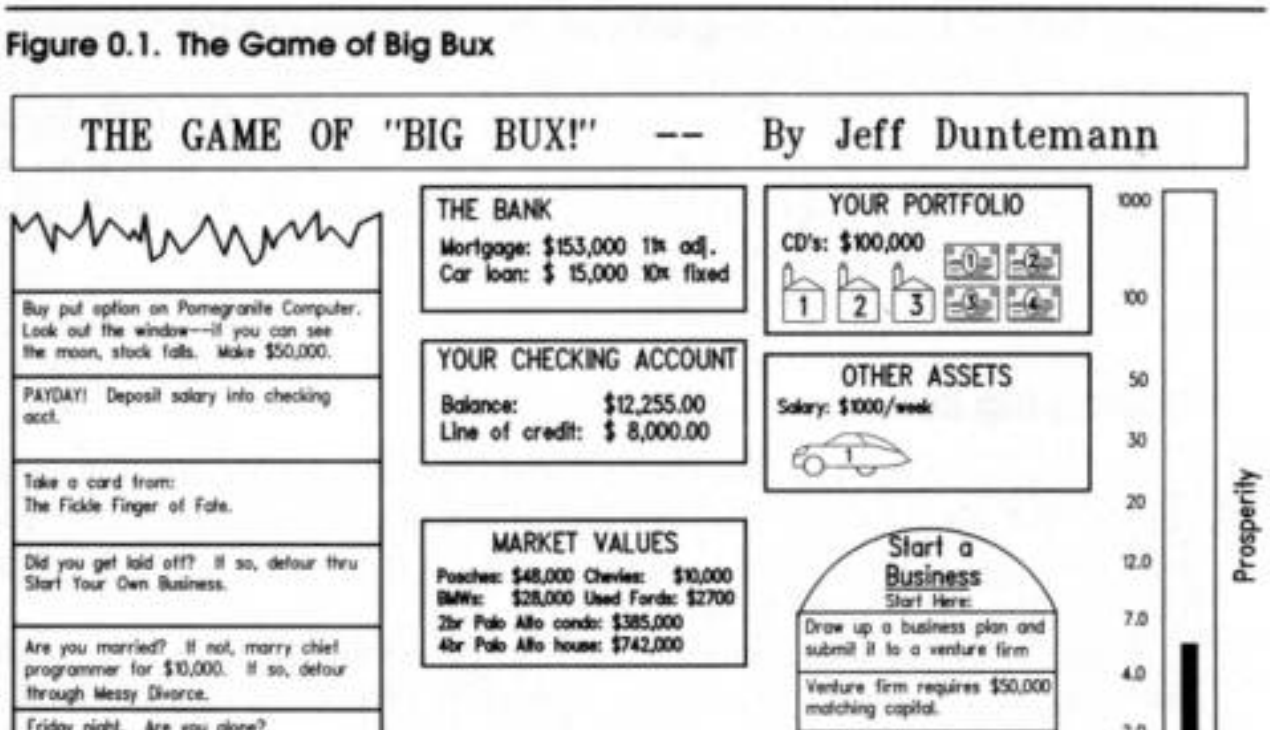
Playing Big Bux

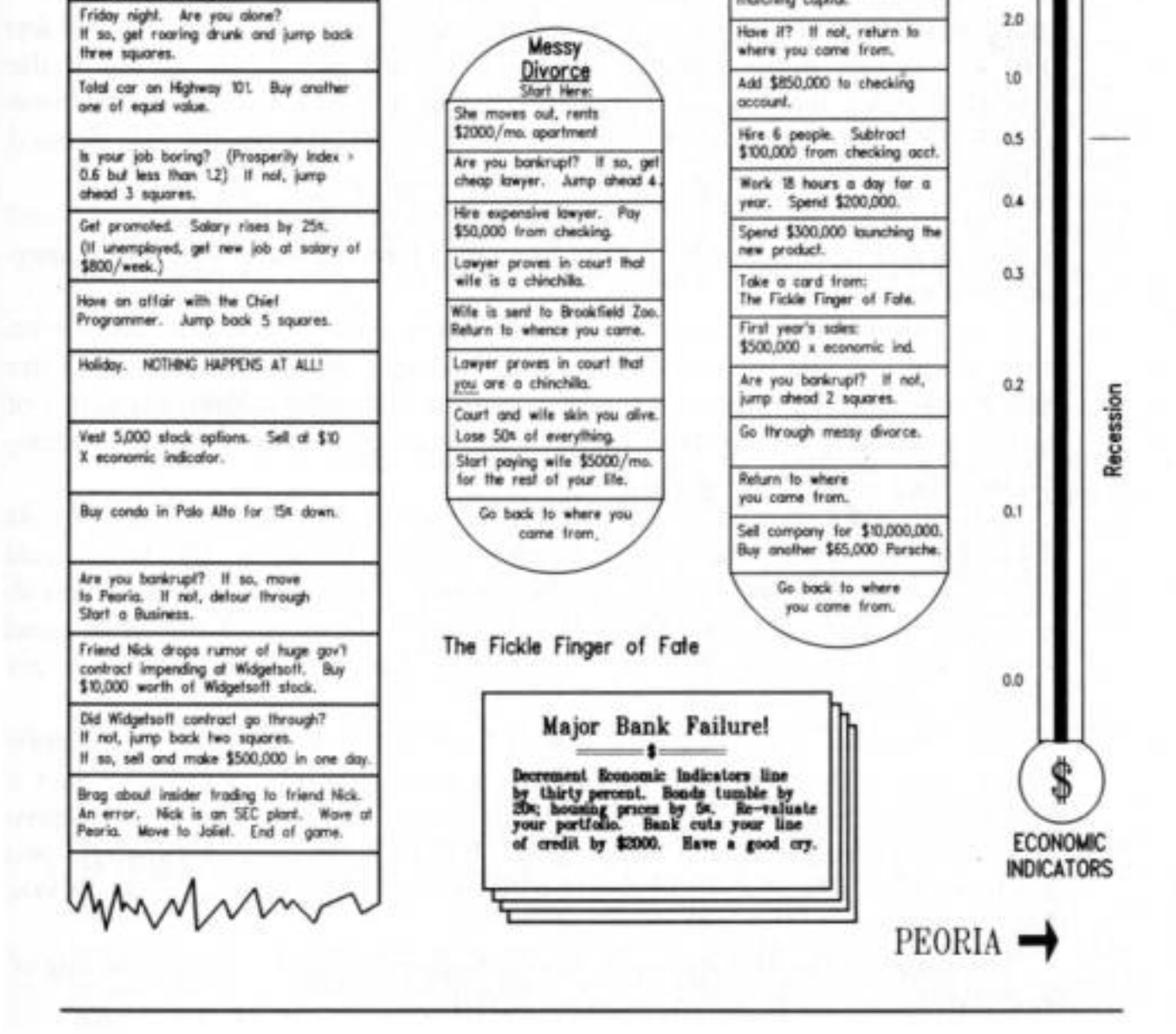
At one corner of the Big Bux board is the legend Move In, since that's how people start life in California—no one is actually *born* there. Once you're moved in, you begin working your way around the board, square by square, following the instructions in the squares.

Some of the squares simply tell you to do something, like *Buy condo in Palo Alto for 5% down.* Many of the squares involve a test of some kind. For example, one square reads: *Is your job boring? (Prosperity Index 0.3 but less than 4.0) If not, jump ahead 3 squares.*

The test is actually to see if the Prosperity Index has a value between 0.3 and 4.0. Any value outside those bounds (i.e., runaway prosperity or Four Horsemen class recession) are defined as Interesting Times, and cause a jump ahead by three squares.
You always move one step forward at each turn, unless the square you land on directs you to do something else, like jump forward three squares or jump back five squares.
The notion of taking a detour is an interesting one. Two detours are shown in the portion of the board I've provided. Taking a detour means leaving the main run around the edge of the game board and stepping through a series of squares elsewhere on the board. The detours involve some specific process; i.e., starting a business or getting divorced.
You can work through a detour, step by step, until you hit the bottom. At that point you simply pick up your journey around the board right where you left it. You may also find that one of the squares in the detour instructs you to go back to where you came from. Depending on the logic of the game (and your luck and finances) you may completely run through a detour, or get thrown out somewhere in the middle.
Also note that you can take a detour from within a detour. If you detour through Start a Business and your business goes bankrupt, you leave Start a Business temporarily and detour through Messy Divorce. Once you leave Messy Divorce you return to where you left Start a Business. Ultimately, you also leave Start a Business and return to wherever it was you were when you took the detour.
The same detour (for example, Start a Business) can be taken from any of several different places along the game board.

Figure 0.1. The Game of Big Bux





Assembly Language Programming as a Board Game

Now that you're thinking in terms of board games, take a look at Figure 0.2. What I've drawn is actually a fair approximation of assembly language as it was used on some of our simpler microprocessors about ten or twelve years ago. The **PROGRAM INSTRUCTIONS** column 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.

Everything else is odds and ends in the middle of the board that serve the game in progress. 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'll 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 the innermost 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 should caution you that the Game of Assembly Language represents no real computer processor like the 8088. Also, I've made the names of instructions more clearly understandable than the names of the instructions in 86 assembly language. In the real world, instruction names are typically things like **STOSB, DAA, BVC, SBB,** and other crypticisms that cannot be understood without considerable explanation. We're easing into this stuff sidewise, and in this chapter I have to sugar-coat certain things a little to draw the metaphors clearly.

Code and Data

Like most board games (including Big Bux), 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: The 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.*

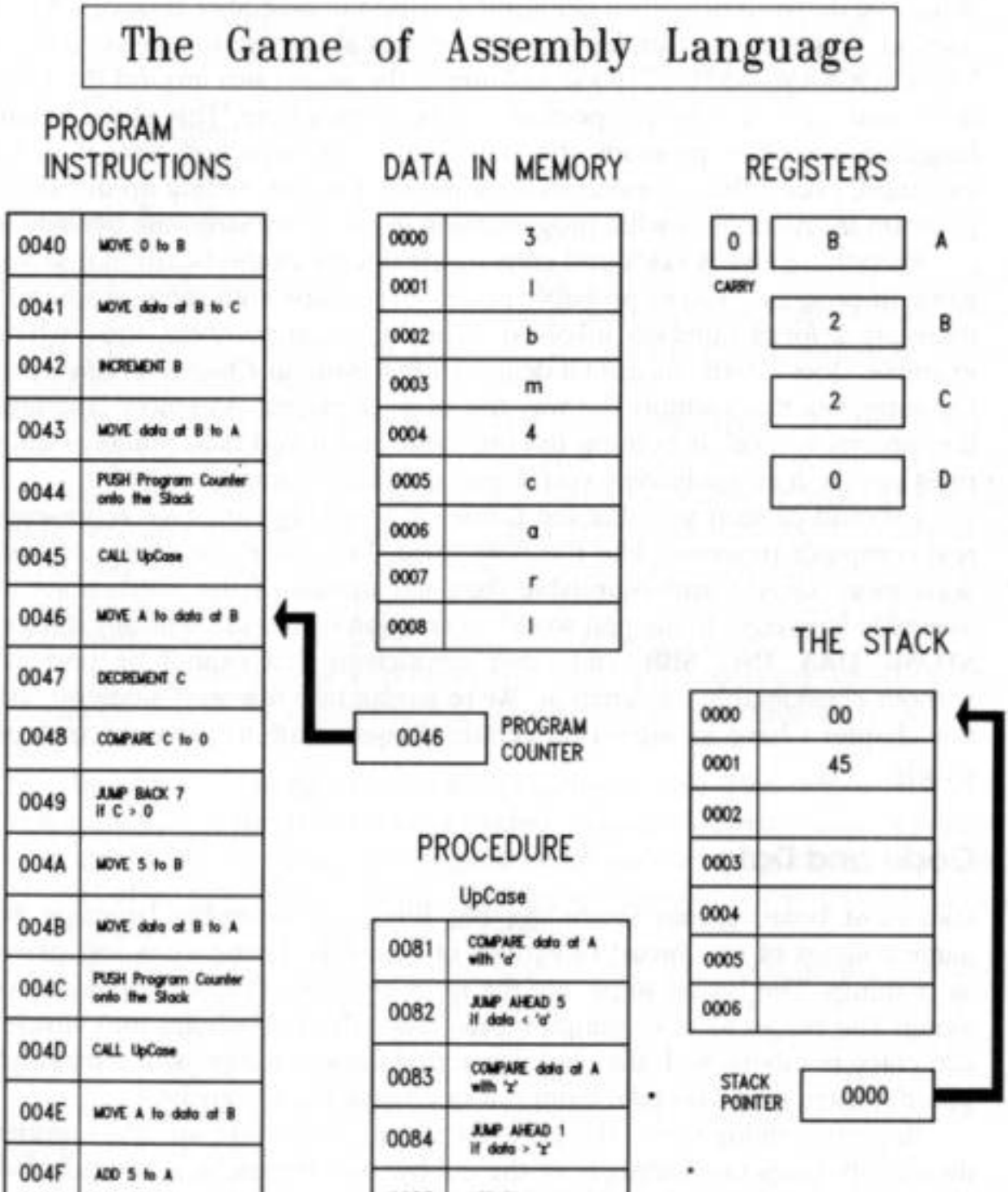
The Game of Big Bux works the same way. Look back to Figure 0.1 and note that in the **Start a Business** detour, there is an instruction that reads **Add $850,000 to checking account.** The checking account is one of several different kinds of storage in this game, and money values are a type of data. It's no different conceptually from an instruction in the Game of Assembly Language that reads **AJDLJ 5 to Register A**. An **ADD** instruction in the code alters a data value stored in a cubbyhole named Register **A**.

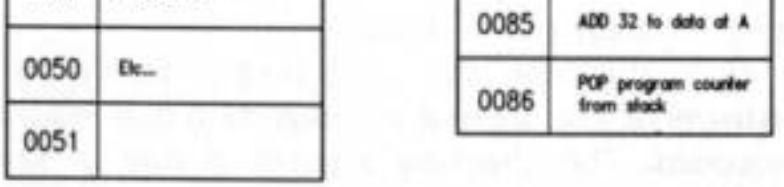
Code and data are two very different kinds of critters, but they interact in ways that make the game interesting. The code includes steps that place data into storage **(MOVE** instructions) and steps that alter data that is already in storage **(INCREMENT and DECREMENT** instructions.) Most of the time 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 **(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 a separate copy everywhere it is needed.

Figure 0.2 The Game of Assembly Language





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 (we say, "incremented") each time an instruction is performed *unless the instruction tells the program counter to do something else.*
Notice the **JUMP BACK** 7 instruction at address **0049**. When this instruction is performed, the program counter will back up by seven counts. 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 0, and we're still in metaphor territory. People who have had some exposure to computers will recognize and understand more of what Figure 0.2 is doing. (There's a real, traceable program going on in there—I dare you to figure out what it does—and how!) 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 byte from one storage cubbyhole to another, 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 that...
• *It takes a lot of steps to do anything useful.* You can often write a useful program in Pascal or BASIC in five or six lines. A useful assembly language program cannot be implemented in fewer than about fifty lines, and anything challenging takes hundreds or

thousands of lines. The skill of assembly language programming lies in structuring these hundreds or thousands of instructions so that the program can be read and understood. And finally,

- *The key to assembly language is understanding memory addresses.* In lan-guages like Pascal and BASIC, the compiler takes care of *where* something is located—you simply have to give that something a name, and call it by that name when you want it. In assembly language, you must *always* be cognizant of where things are in your computer's memory. So in working through this book, pay special attention to the concept of *addressing,* which is nothing more than the art of specifying where something is. The Game of Assembly Language is peppered with addresses and instructions that work with addresses. (Such as **MOVE data at B to C,** which means move the data stored at the address specified by register **B** to the address specified by register **C**.) Addressing is by far the trickiest part of assembly language, but master it and you've got the whole thing in your hip pocket.

Everything I've said so far has been orientation. I've tried to give you a taste of the big picture of assembly language and how its fundamental principles relate to the life you've been living all along. Life is a sequence of steps and tests, and so are board games—and so is assembly language. Keep those metaphors in mind as we proceed to "get real" by confronting the nature of computer numbers.