2

SHOULDN'T YOU LEARN ASSEMBLY LANGUAGE?

Although this book will teach you how to write better code without mastering assembly language, the absolute best HLL programmers do know assembly, and that knowledge is one of the reasons they write great code. As mentioned in Chapter 1, although this book can provide a 90 percent solution if you just want to write great HLL code, to fill in that last 10 percent you'll need to learn assembly language. While teaching you assembly language is beyond the scope of this book, it's still an important subject to discuss. To that end, this chapter will explore the following:

- The problem with learning assembly language
- High-level assemblers and how they can make learning assembly language easier

- How you can use real-world products like Microsoft Macro Assembler (MASM), Gas (Gnu Assembler), and HLA (High-Level Assembly) to easily learn assembly language programming
- How an assembly language programmer *thinks* (that is, the assembly language programming paradigm)
- Resources available to help you learn assembly language programming

2.1 Benefits and Roadblocks to Learning Assembly Language

Learning assembly language—*really* learning assembly language—offers two benefits. First, you'll gain a complete understanding of the machine code that a compiler can generate. By mastering assembly language, you'll achieve the 100 percent solution just described and be able to write better HLL code. Second, you'll be able to code critical parts of your application in assembly language when your HLL compiler is incapable, even with your help, of producing the best possible code. Once you've absorbed the lessons of the following chapters to hone your HLL skills, moving on to learn assembly language is a very good idea.

There's one catch to learning assembly language, though. In the past, it's been a long, difficult, and frustrating task. The assembly language programming paradigm is sufficiently different from HLL programming that most people feel like they're starting over from square one when learning it. It's very frustrating when you know how to do something in a programming language like C/C++, Java, Swift, Pascal, or Visual Basic, but you can't yet figure out the solution in assembly language.

Most programmers like being able to apply past experience when learning something new. Unfortunately, traditional approaches to learning assembly language programming tend to force HLL programmers to forget what they've learned in the past. This book, in contrast, offers a way for you to efficiently leverage your existing knowledge while learning assembly language.

2.2 How This Book Can Help

Once you've read through this book, there are three reasons you'll find it much easier to learn assembly language:

- You'll be more motivated to learn it because you'll understand why doing so can help you write better code.
- You'll have had five brief primers on assembly language (80x86, PowerPC, ARM, Java bytecode, and Microsoft IL), so even if you'd never seen it before, you'll have learned some by the time you finish this book.
- You'll have already seen how compilers emit machine code for all the common control and data structures, so you'll have learned one of the most difficult lessons for a beginning assembly programmer—how to achieve things in assembly language that they already know how to do in an HLL.

Though this book won't teach you how to become an expert assembly language programmer, the large number of example programs that demonstrate how compilers translate HLLs into machine code will acquaint you with many assembly language programming techniques. You'll find these useful should you decide to learn assembly language after reading this book.

Certainly, you'll find this book easier to read if you already know assembly language. However, you'll also find assembly language easier to master once you've read this book. Since learning assembly language is probably more time-consuming than reading this book, the more efficient approach is to start with the book.

2.3 High-Level Assemblers to the Rescue

Way back in 1995, I had a discussion with the University of California, Riverside, computer science department chair. I was lamenting the fact that students had to start over when taking the assembly course, spending precious time to relearn so many things. As the discussion progressed, it became clear that the problem wasn't with assembly language, per se, but with the syntax of existing assemblers (like Microsoft Macro Assembler, or MASM). Learning assembly language entailed a whole lot more than learning a few machine instructions. First of all, you have to learn a new programming style. Mastering assembly language involves learning not only the semantics of a few machine instructions but also how to put those instructions together to solve real-world problems. And *that's* the hard part.

Second, *pure* assembly language is not something you can efficiently pick up a few instructions at a time. Writing even the simplest programs requires considerable knowledge and a repertoire of a couple dozen or more machine instructions. When you add that repertoire to all the other machine organization topics students must learn in a typical assembly course, it's often several weeks before they are prepared to write anything other than "spoon-fed" trivial applications in assembly language.

One important feature of MASM back in 1995 was support for HLL-like control statements such as `.if` and `.while`. While these statements are not true machine instructions, they do allow students to use familiar programming constructs early in the course, until they've had time to learn enough low-level machine instructions that they can use them in their applications. By using these high-level constructs early on in the term, students can concentrate on other aspects of assembly language programming and not have to assimilate everything all at once. This allows them to start writing code much sooner in the course and, as a result, they wind up covering more material by the end of the term.

An assembler like MASM (32-bit v6.0 and later) that provides control statements similar to those found in HLLs—in addition to the traditional low-level machine instructions that do the same thing—is called a *high-level assembler*. In theory, with an appropriate textbook that teaches assembly language programming using these high-level assemblers, students could begin writing simple programs during the very first week of the course.

The only problem with high-level assemblers like MASM is that they provide just a few HLL control statements and data types. Almost everything else is foreign to someone who is familiar with HLL programming. For example, data declarations in MASM are completely different from data declarations in most HLLs. Beginning assembly programmers still have to relearn a considerable amount of information, despite the presence of HLL-like control statements.

2.4 High-Level Assembly Language

Shortly after the discussion with my department chair, it occurred to me that there is no reason an assembler couldn't adopt a more high-level syntax without changing the semantics of assembly language. For example, consider the following statements in C/C++ and Pascal that declare an integer array variable:

```
int intVar[8]; // C/C++

var intVar: array[0..7] of integer; (* Pascal *)
```

Now consider the MASM declaration for the same object:

```
intVar sdword 8 dup (?) ;MASM
```

While the C/C++ and Pascal declarations differ from each other, the assembly language version is radically different from both. A C/C++ programmer will probably be able to figure out the Pascal declaration even if they have never seen Pascal code before, and vice versa. However, Pascal and C/C++ programmers probably won't be able to make heads or tails of the assembly language declaration. This is but one example of the problems HLL programmers face when first learning assembly language.

The sad part is that there's really no reason a variable declaration in assembly language has to be so radically different from one in an HLL. It makes absolutely no difference in the final executable file which syntax an assembler uses for variable declarations. Given that, why shouldn't an assembler use a more high-level-like syntax so people switching over from HLLs will find the assembler easier to learn? Pondering this question led me to develop a new assembly language, specifically geared toward teaching assembly language programming to students who had already mastered an HLL, called *High-Level Assembly (HLA)*. In HLA, the aforementioned array declaration looks like this:

```
var intVar:int32[8]; // HLA
```

Though the syntax is slightly different from C/C++ and Pascal (actually, it's a combination of the two), most HLL programmers can probably figure out the meaning of this declaration.

The whole purpose of HLA's design is to provide an assembly language programming environment as similar as possible to that of traditional (imperative) high-level programming languages, without sacrificing the capability to write *real* assembly language programs. Those components of the language that have nothing to do with machine instructions use a familiar high-level language syntax, while the machine instructions still map one-to-one to the underlying 80x86 machine instructions.

Making HLA as similar as possible to various HLLs means that students learning assembly language programming don't have to spend as much time assimilating a radically different syntax. Instead, they can apply their existing HLL knowledge, which makes the process of learning assembly language easier and faster.

A comfortable syntax for declarations and a few HLL-like control statements aren't all you need to make learning assembly language as efficient as possible, however. One very common complaint about learning assembly language is that it provides very little support for programmers, who must constantly reinvent the wheel while writing assembly code. For example, when learning assembly language programming using MASM, you'll quickly discover that assembly language doesn't provide useful I/O facilities such as the ability to print integer values as strings to the user's console. Assembly programmers are responsible for writing such code themselves. Unfortunately, writing a decent set of I/O routines requires sophisticated knowledge of assembly language programming. The only way to gain that knowledge is by writing a fair amount of code first, but doing so without having any I/O routines is difficult. Therefore, a good assembly language educational tool also needs to provide a set of I/O routines that allow beginning assembly programmers to do simple I/O tasks, like reading and writing integer values, before they have the programming sophistication to write such routines themselves. HLA accomplishes this with the *HLA Standard Library*, a collection of subroutines and macros that make it very easy to write complex applications.

Because of HLA's popularity and the fact that HLA is a free, open source, and public domain product available for Windows and Linux, this book uses HLA syntax for compiler-neutral examples involving assembly language. Despite the fact that it is now over 20 years old and supports only the 32-bit Intel instruction set, HLA is still an excellent way to learn assembly language programming. Although the latest Intel CPUs directly support 64-bit registers and operations, learning 32-bit assembly language is just as relevant for HLL programmers as 64-bit assembly.

2.5 Thinking High-Level, Writing Low-Level

The goal of HLA is to allow a beginning assembly programmer to think in HLL terms while writing low-level code (in other words, the exact opposite of what this book is trying to teach). For students first approaching assembly language, being able to think in high-level terms is a godsend—they can apply techniques they've already learned in other languages when faced

with a particular assembly language programming problem. Controlling the rate at which a student has to learn new concepts in this way can make the educational process more efficient.

Ultimately, of course, the goal is to learn the low-level programming paradigm. This means gradually giving up HLL-like control structures and writing pure low-level code (that is, "thinking low-level and writing low-level"). Nevertheless, starting out by "thinking high-level while writing low-level" is a great, incremental way to learn assembly language programming.

2.6 The Assembly Programming Paradigm (Thinking Low-Level)

It should be clear now that programming in assembly language is quite different from programming in common HLLs. Fortunately, for this book, you don't need to be able to write assembly language programs from scratch. Nevertheless, if you know how assembly programs are written, you'll be able to understand why a compiler emits certain code sequences. To that end, I'll take some time here to describe how assembly language programmers (and compilers) "think."

The most fundamental aspect of the assembly language programming paradigm—that is, the model for how assembly programming is accomplished—is that large projects are broken up into mini-tasks that the machine can handle. Fundamentally, a CPU can do only one tiny task at a time; this is true even for complex instruction set computers (CISC). Therefore, complex operations, like statements you'll find in an HLL, have to be broken down into smaller components that the machine can execute directly. As an example, consider the following Visual Basic (VB) assignment statement:

```
profits = sales - costOfGoods - overhead - commissions
```

No practical CPU will allow you to execute this entire VB statement as a single machine instruction. Instead, you have to break this assignment statement down to a sequence of machine instructions that compute individual components of it. For example, many CPUs provide a *subtract* instruction that lets you subtract one value from a machine register. Because the assignment statement in this example consists of three subtractions, you'll have to break the assignment operation down into at least three different subtract instructions.

The 80x86 CPU family provides a fairly flexible subtract instruction: `sub()`. This particular instruction allows the following forms (in HLA syntax):

```
sub( constant, reg );       // reg = reg - constant
sub( constant, memory );    // memory = memory - constant
sub( reg1, reg2 );          // reg2 = reg2 - reg1
sub( memory, reg );         // reg = reg - memory
sub( reg, memory );         // memory = memory - reg
```

Assuming that all of the identifiers in the original VB code represent variables, we can use the 80x86 `sub()` and `mov()` instructions to implement the same operation with the following HLA code sequence:

```
// Get sales value into EAX register:

mov( sales, eax );

// Compute sales-costOfGoods (EAX := EAX - costOfGoods)

sub( costOfGoods, eax );

// Compute (sales-costOfGoods) - overhead
// (note: EAX contains sales-costOfGoods)

sub( overhead, eax );

// Compute (sales-costOfGoods-overhead)-commissions
// (note: EAX contains sales-costOfGoods-overhead)

sub( commissions, eax );

// Store result (in EAX) into profits:

mov( eax, profits );
```

This code breaks down the single VB statement into five different HLA statements, each of which does a small part of the total calculation. The secret behind the assembly language programming paradigm is knowing how to break down complex operations like this into a simple sequence of machine instructions. We'll take another look at this process in Chapter 13.

HLL control structures are another big area where complex operations are broken down into simpler statement sequences. For example, consider the following Pascal `if()` statement:

```
if( i = j ) then begin

    writeln( "i is equal to j" );

end;
```

CPUs do not support an `if()` machine instruction. Instead, you compare two values that set *condition-code flags* and then test the result of these condition codes by using *conditional jump* instructions. A common way to translate an HLL `if()` statement into assembly language is to test the opposite condition (`i <> j`) and then jump over the statements that would be executed if the original condition (`i = j`) evaluates to `true`. For example, here's a translation of the former Pascal `if()` statement into HLA (using pure assembly language—that is, no HLL-like constructs):

```
    mov( i, eax );      // Get i's value into eax register
    cmp( eax, j );      // Compare eax to j's value
```

```
    jne skipIfBody;     // Skip body of if statement if i <> j

    << code to print string >>

skipIfBody:
```

As the Boolean expressions in the HLL control structures increase in complexity, the number of corresponding machine instructions also increases. But the process remains the same. Later, we'll take a look at how compilers translate HLL control structures into assembly language (see Chapters 13 and 14).

Passing parameters to a procedure or function, accessing those parameters, and then accessing other data local to that procedure or function is another area where assembly language is quite a bit more complex than typical HLLs. This is an important subject, but it's beyond the scope of this chapter, so we'll revisit it in Chapter 15.

The bottom line is that when converting an algorithm from a high-level language, you have to break down the problem into much smaller pieces in order to code it in assembly language. As noted earlier, the good news is that you don't have to figure out which machine instructions to use when all you're doing is reading assembly code—the compiler (or assembly programmer) that originally created the code will have already done this for you. All you have to do is draw a correspondence between the HLL code and the assembly code. How you accomplish that is the subject of much of the rest of this book.

2.7 For More Information

Bartlett, Jonathan. *Programming from the Ground Up*. Edited by Dominick Bruno, Jr. Self-published, 2004. An older, free version of this book, which teaches assembly language programming using Gas, can be found online at *http://www.plantation-productions.com/AssemblyLanguage/ProgrammingGroundUp-1-0-booksize.pdf*.

Blum, Richard. *Professional Assembly Language*. Indianapolis: Wiley, 2005.

Carter, Paul. *PC Assembly Language*. Self-published, 2019. *http://pacman128.github.io/static/pcasm-book.pdf*.

Duntemann, Jeff. *Assembly Language Step-by-Step*. 3rd ed. Indianapolis: Wiley, 2009.

Hyde, Randall. *The Art of Assembly Language*. 2nd ed. San Francisco: No Starch Press, 2010.

———. "Webster: The Place on the Internet to Learn Assembly." *http://plantation-productions.com/Webster/index.html*.