Why Would Anyone Learn This Stuff? Forward

Amazing! You're actually reading this. That puts you into one of three categories: a student who is being forced to read this stuff for a class, someone who picked up this book by accident (probably because you have yet to be indoctrinated by the world at large), or one of the few who actually have an interest in learning assembly language.

Egads. What kind of book begins this way? What kind of author would begin the book with a forward like this one? Well, the truth is, *I* considered putting this stuff into the first chapter since most people never bother reading the forward. A discussion of what's right and what's wrong with assembly language is very important and sticking it into a chapter might encourage someone to read it. However, I quickly found that university students can skip Chapter One as easily as they can skip a forward, so this stuff wound up in a forward after all.

So why would anyone learn this stuff, anyway? Well, there are several reasons which come to mind:

- Your major requires a course in assembly language; i.e., you're here against your will.
- A programmer where you work quit. Most of the source code left behind was written in assembly language and you were elected to maintain it.
- Your boss has the audacity to insist that you write your code in assembly against your strongest wishes.
- Your programs run just a little too slow, or are a little too large and you think assembly language might help you get your project under control.
- You want to understand how computers actually work.
- You're interested in learning how to write efficient code.
- You want to try something new.

Well, whatever the reason you're here, welcome aboard. Let's take a look at the subject you're about to study.

1 What's Wrong With Assembly Language

Assembly language has a pretty bad reputation. The common impression about assembly language programmers today is that they are all hackers or misguided individuals who need enlightenment. Here are the reasons people give for *not* using assembly[1]:

- Assembly is hard to learn.
- Assembly is hard to read and understand.
- Assembly is hard to debug.
- Assembly is hard to maintain.
- Assembly is hard to write.
- Assembly language programming is time consuming.
- Improved compiler technology has eliminated the need for assembly language.
- Today, machines are so fast that we no longer need to use assembly.
- If you need more speed, you should use a better algorithm rather than switch to assembly language.
- Machines have so much memory today, saving space using assembly is not important.
- Assembly language is not portable.

1. This text will use the terms "Assembly language" and "assembly" interchangeably.

These are some strong statements indeed!

Given that this is a book which teaches assembly language programming, written for college level students, written by someone who appears to know what he's talking about, your natural tendency is to believe something if it appears in print. Having just read the above, you're starting to assume that assembly must be pretty bad. And that, dear friend, is eighty percent of what's wrong with assembly language. That is, people develop some very strong misconceptions about assembly language based on what they've heard from friends, instructors, articles, and books. Oh, assembly language is certainly not perfect. It does have many real faults. Those faults, however, are blown completely out of proportion by those unfamiliar with assembly language. The next time someone starts preaching about the evils of assembly language, ask, "how many years of assembly language programming experience do you have?" Of course assembly is hard to understand *if you don't know it*. It is surprising how many people are willing to speak out against assembly language based only on conversations they've had or articles they've read.

Assembly language users also use high level languages (HLLs); assembly's most outspoken opponents rarely use anything but HLLs. Who would you believe, an expert well versed in both types of programming languages or someone who has never taken the time to learn assembly language and develop an honest opinion of its capabilities?

In a conversation with someone, I would go to great lengths to address each of the above issues. Indeed, in a rough draft of this chapter I spent about ten pages explaining what is wrong with each of the above statements. However, this book is long enough and I felt that very little was gained by going on and on about these points. Nonetheless, a brief rebuttal to each of the above points is in order, if for no other reason than to keep you from thinking there isn't a decent defense for these statements.

Assembly is hard to learn. So is any language you don't already know. Try learning (really learning) APL, Prolog, or Smalltalk sometime. Once you learn Pascal, learning another language like C, BASIC, FORTRAN, Modula-2, or Ada is fairly easy because these languages are quite similar to Pascal. On the other hand, learning a dissimilar language like Prolog is not so simple. Assembly language is also quite different from Pascal. It will be a little harder to learn than one of the other Pascal-like languages. However, learning assembly isn't much more difficult than learning your first programming language.

Assembly is hard to read and understand. It sure is, if you don't know it. Most people who make this statement simply don't know assembly. Of course, it's very easy to write impossible-to-read assembly language programs. It's also quite easy to write impossible-to-read C, Prolog, and APL programs. With experience, you will find assembly as easy to read as other languages.

Assembly is hard to debug. Same argument as above. If you don't have much experience debugging assembly language programs, it's going to be hard to debug them. Remember what it was like finding bugs in your first Pascal (or other HLL) programs? Anytime you learn a new programming language you'll have problems debugging programs in that language until you gain experience.

Assembly is hard to maintain. C programs are hard to maintain. Indeed, *programs* are hard to maintain period. Inexperienced assembly language programmers tend to write hard to maintain programs. Writing maintainable programs isn't a talent. It's a skill you develop through experience.

Assembly language is hard. This statement actually has a ring of truth to it. For the longest time assembly language programmers wrote their programs completely from scratch, often "re-inventing the wheel." HLL programmers, especially C, Ada, and Modula-2 programmers, have long enjoyed the benefits of a *standard library* package which solves many common programming problems. Assembly language programmers, on the other hand, have been known to rewrite an integer output routine every time they need one. This book does *not* take that approach. Instead, it takes advantage of some work done at the University of California, Riverside: the *UCR Standard Library for 80x86 Assembly Language Programmers*. These subroutines simplify assembly language just as the C standard library aids C programmers. The library source listings are available electronically via Internet and various other communication services as well as on a companion diskette.

Assembly language programming is time consuming. Software engineers estimate that developers spend only about thirty percent of their time coding a solution to a problem. Even if it took twice as

much time to write a program in assembly versus some HLL, there would only be a fifteen percent difference in the total project completion time. In fact, *good* assembly language programmers do not need twice as much time to implement something in assembly language. It is true using a HLL will save *some* time; however, the savings is insufficient to counter the benefits of using assembly language.

Improved compiler technology has eliminated the need for assembly language. This isn't true and probably never will be true. Optimizing compilers are getting better every day. However, assembly language programmers get better performance by writing their code *differently* than they would if they were using some HLL. If assembly language programmers wrote their programs in C and then translated them manually into assembly, a good C compiler would produce equivalent, or even better, code. Those who make this claim about compiler technology are comparing their *hand*-compiled code against that produced by a compiler. Compilers do a much better job of compiling than humans. Then again, you'll never catch an assembly language programmer writing "C code with MOV instructions." After all, that's why you use C compilers.

Today, machines are so fast that we no longer need to use assembly. It is amazing that people will spend lots of money to buy a machine slightly faster than the one they own, but they won't spend any extra time writing their code in assembly so it runs faster on the same hardware. There are many raging debates about the speed of machines versus the speed of the software, but one fact remains: users always want more speed. On any given machine, the fastest possible programs will be written in assembly language[2].

If you need more speed, you should use a better algorithm rather than switch to assembly language. Why can't you use this better algorithm in assembly language? What if you're already using the best algorithm you can find and it's still too slow? This is a totally bogus argument against assembly language. Any algorithm you can implement in a HLL you can implement in assembly. On the other hand, there are many algorithms you can implement in assembly which you cannot implement in a HLL[3].

Machines have so much memory today, saving space using assembly is not important. If you give someone an inch, they'll take a mile. Nowhere in programming does this saying have more application than in program memory use. For the longest time, programmers were quite happy with 4 Kbytes. Later, machines had 32 or even 64 Kilobytes. The programs filled up memory accordingly. Today, many machines have 32 or 64 *megabytes* of memory installed and some applications use it all. There are lots of technical reasons why programmers should strive to write shorter programs, though now is not the time to go into that. Let's just say that space *is* important and programmers should strive to write programs as short as possible regardless of how much main memory they have in their machine.

Assembly language is not portable. This is an undeniable fact. An 80x86 assembly language program written for an IBM PC will not run on an Apple Macintosh[4]. Indeed, assembly language programs written for the Apple Macintosh will not run on an Amiga, even though they share the same 680x0 microprocessor. If you need to run your program on different machines, you'll have to think long and hard about using assembly language. Using C (or some other HLL) is no guarantee that your program will be portable. C programs written for the IBM PC won't compile and run on a Macintosh. And even if they did, most Mac owners wouldn't accept the result.

Portability is probably the biggest complaint people have against assembly language. They refuse to use assembly because it is not portable, and then they turn around and write equally non-portable programs in C.

Yes, there are lots of lies, misconceptions, myths, and half-truths concerning assembly language. Whatever you do, make sure you learn assembly language before forming your own opinions[5]. Speaking

2. That is not to imply that assembly language programs are always faster than HLL programs. A poorly written assembly language program can run much slower than an equivalent HLL program. On the other hand, if a program is written in an HLL it is certainly possible to write a faster one in assembly.

3. We'll see some of these algorithms later in the book. They deal with instruction sequencing and other tricks based on how the processor operates.

4. Strictly speaking, this is not true. There is a program called SoftPC which emulates an IBM PC using an 80286 *interpreter*. However, 80x86 assembly language programs will not run in native mode on the Mac's 680x0 microprocessor.

out in ignorance may impress others who know less than you do, but it won't impress those who know the truth.

2 What's Right With Assembly Language?

An old joke goes something like this: "There are three reasons for using assembly language: speed, speed, and more speed." Even those who absolutely hate assembly language will admit that if speed is your primary concern, assembly language is the way to go. Assembly language has several benefits:

- Speed. Assembly language programs are generally the fastest programs around.
- Space. Assembly language programs are often the smallest.
- Capability. You can do things in assembly which are difficult or impossible in HLLs.
- Knowledge. Your knowledge of assembly language will help you write better programs, even when using HLLs.

Assembly language is the uncontested speed champion among programming languages. An expert assembly language programmer will almost always produce a faster program than an expert C programmer[6]. While certain programs may not benefit much from implementation in assembly, you can speed up many programs by a factor of five or ten over their HLL counterparts by careful coding in assembly language; even greater improvement is possible if you're not using an optimizing compiler. Alas, speedups on the order of five to ten times are generally not achieved by beginning assembly language programmers. However, if you spend the time to learn assembly language really well, you too can achieve these impressive performance gains.

Despite some people's claims that programmers no longer have to worry about memory constraints, there are many programmers who need to write smaller programs. Assembly language programs are often less than one-half the size of comparable HLL programs. This is especially impressive when you consider the fact that data items generally consume the same amount of space in both types of programs, and that data is responsible for a good amount of the space used by a typical application. Saving space saves money. Pure and simple. If a program requires 1.5 megabytes, it will not fit on a 1.44 Mbyte floppy. Likewise, if an application requires 2 megabytes RAM, the user will have to install an extra megabyte if there is only one available in the machine[7]. Even on big machines with 32 or more megabytes, writing gigantic applications isn't excusable. Most users put more than eight megabytes in their machines so they can run *multiple* programs from memory at one time. The bigger a program is, the fewer applications will be able to coexist in memory with it. Virtual memory isn't a particularly attractive solution either. With virtual memory, the bigger an application is, the slower the system will run as a result of that program's size.

Capability is another reason people resort to assembly language. HLLs are an abstraction of a typical machine architecture. They are designed to be independent of the particular machine architecture. As a result, they rarely take into account any special features of the machine, features which are available to assembly language programmers. If you want to use such features, you will need to use assembly language. A really good example is the input/output instructions available on the 80x86 microprocessors. These instructions let you directly access certain I/O devices on the computer. In general, such access is not part of any high level language. Indeed, some languages like C pride themselves on not supporting

5. Alas, a typical ten-week course is rarely sufficient to learn assembly language well enough to develop an informed opinion on the subject. Probably three months of eight-hour days using the stuff would elevate you to the point where you could begin to make some informed statements on the subject. Most people wouldn't be able to consider themselves "good" at assembly language programs until they've been using the stuff for at least a year.

6. There is absolutely no reason why an assembly language programmer would produce a *slower* program since that programmer could look at the output of the C compiler and copy whatever code runs faster than the hand produced code. HLL programmers don't have an equivalent option.

7. You can substitute any numbers here you like. One fact remains though, programmers are famous for assuming users have more memory than they really do.

any specific I/O operations[8]. In assembly language you have no such restrictions. Anything you can do on the machine you can do in assembly language. This is definitely *not* the case with most HLLs.

Of course, another reason for learning assembly language is just for the knowledge. Now some of you may be thinking, "Gee, that would be wonderful, but I've got lots to do. My time would be better spent writing code than learning assembly language." There are some practical reasons for learning assembly, even if you never intend to write a single line of assembly code. If you know assembly language well, you'll have an appreciation for the compiler, and you'll know exactly what the compiler is doing with all those HLL statements. Once you see how compilers translate seemingly innocuous statements into a ton of machine code, you'll want to search for better ways to accomplish the same thing. Good assembly language programmers make better HLL programmers because they understand the limitations of the compiler and they know what it's doing with their code. Those who don't know assembly language will accept the poor performance their compiler produces and simply shrug it off.

Yes, assembly language is definitely worth the effort. The only scary thing is that once you learn it really well, you'll probably start using it far more than you ever dreamed you would. That is a common malady among assembly language programmers. Seems they can't stand what the compilers are doing with their programs.

3 Organization of This Text and Pedagogical Concerns

This book is divided into seven main sections: a section on machine organization and architecture, a section on basic assembly language, a section on intermediate assembly language, a section on interrupts and resident programs, a section covering IBM PC hardware peculiarities, a section on optimization, and various appendices. It is doubtful that any single (even year-long) college course could cover all this material, the final chapters were included to support compiler design, microcomputer design, operating systems, and other courses often found in a typical CS program.

Developing a text such as this one is a very difficult task. First of all, different universities have different ideas about how this course should be taught. Furthermore, different schools spend differing amounts of time on this subject (one or two quarters, a semester, or even a year). Furthermore, different schools cover different material in the course. For example, some schools teach a "Machine Organization" course that emphasizes hardware concepts and presents the assembly language instruction set, but does not expect students to write real assembly language programs (that's the job of a compiler). Other schools teach a "Machine Organization and Assembly Language" course that combines hardware and software issues together into one course. Still others teach a "Machine Organization" or "Digital Logic" course as a prerequisite to an "Assembly Language" course. Still others teach "Assembly Language Programming" as a course and leave the hardware for a "Computer Architecture" course later in the curriculum. Finally, let us not forget that some people will pick up this text and use it to learn machine organization or assembly language programming on their own, without taking a formal course on the subject. A good *textbook* in this subject area must be adaptable to the needs of the course, instructor, and student. These requirements place enough demands on an author, but I wanted more for this text. Many textbooks teach a particular subject well, but once you've read and understood them, they do not serve well as a reference guide. Given the cost of textbooks today, it is a real shame that many textbooks' value diminishes once the course is complete. I sought to create a textbook that will explain many difficult concepts in as friendly a manner as possible *and* will serve as a reference guide once you've mastered the topic. By moving advanced material you probably won't cover in a typical college course into later chapters and by organizing this text so you can continue using it once the course is over, I hope to provide you with an excellent value in this text.

Since this volume attempts to satisfy the requirements of several different courses, as well as provide an excellent reference, you will probably find that it contains far more material than any single course

8. Certain languages on the PC support extensions to access the I/O devices since this is such an obvious limitation of the language. However, such extensions are not part of the actual language.

would actually cover. For example, the first section of this text covers machine organization. If you've already covered this material in a previous course, your instructor may elect to skip the first four chapters or so. For those courses that teach only assembly language, the instructor may decide to skip chapters two and three. Schools operating on a ten-week quarter system may cover the material in each chapter only briefly (about one week per chapter). Other schools may cover the material in much greater depth because they have more time.

When writing this text, I choose to pick a subject and cover it in depth before proceeding to the next topic. This pedagogy (teaching method) is unusual. Most assembly language texts jump around to different topics, lightly touching on each one and returning to them as further explanation is necessary. Unfortunately, such texts make poor references; trying to lookup information in such a book is difficult, at best, because the information is spread throughout the book. Since I want this text to serve as a reasonable reference manual, such an organization was unappealing.

The problem with a straight reference manual is three-fold. First, reference manuals are often organized in a manner that makes it easy to look something up, not in a logical order that makes the material easy to learn. For example, most assembly language reference manuals introduce the instruction set in alphabetical order. However, you do not learn the instruction set in this manner. The second problem with a (good) reference manual is that it presents the material in far greater depth than most beginners can handle; this is why most texts keep returning to a subject, they add a little more depth on each return to the subject. Finally, reference texts can present material in any order. The author need not ensure that a discussion only include material appearing earlier in the text. Material in the early chapters of a reference manual can refer to later chapters; a typical college textbook should *not* do this.

To receive maximum benefit from this text, you need to read it understanding its organization. This is *not* a text you read from front to back, making sure you understand each and every little detail before proceeding to the next. I've covered many topics in this text in considerable detail. Someone learning assembly language for the first time will become overwhelmed with the material that appears in each chapter. Typically, you will read over a chapter once to learn the basic essentials and then refer back to each chapter learning additional material as you need it. Since it is unlikely that you will know which material is basic or advanced, I've taken the liberty of describing which sections are basic, intermediate, or advanced at the beginning of each chapter. A ten-week course, covering this entire text for example, might only deal with the basic topics. In a semester course, there is time to cover the intermediate material as well. Depending on prerequisites and length of course, the instructor can elect to teach this material at any level of detail (or even jump around in the text).

In the past, if a student left an assembly language class and could actually implement an algorithm in assembly language, the instructor probably considered the course a success. However, compiler technology has progressed to the point that simply "getting something to work" in assembly language is pure folly. If you don't write your code efficiently in assembly language, you may as well stick with HLLs. They're easy to use, and the compiler will probably generate faster code than you if you're careless in the coding process.

This text spends a great deal of time on machine and data organization. There are two important reasons for this. First of all, to write efficient code on modern day processors requires an intimate knowledge of what's going on in the hardware. Without this knowledge, your programs on the 80486 and later could run at less than half their possible speed. To write the best possible assembly language programs you must be familiar with how the hardware operates. Another reason this text emphasizes computer organization is that most colleges and universities are more interested in teaching machine organization than they are a particular assembly language. While the typical college student won't have much need for assembly language during the four years as an undergraduate, the machine organization portion of the class is useful in several upper division classes. Classes like data structures and algorithms, computer architecture, operating systems, programming language design, and compilers all benefit from an introductory course in computer organization. That's why this text devotes an entire section to that subject.

4 Obtaining Program Source Listings and Other Materials in This Text

All of the software appearing in this text is available on the companion diskette. The material for this text comes in two parts: source listings of various examples presented in this text and the code for the *UCR Standard Library for 80x86 Assembly Language Programmers*. The UCR Standard Library is also available electronically from several different sources (including Internet, BIX, and other on-line services).

You may obtain the files electronically via ftp from the following Internet address:

ftp.cs.ucr.edu

Log onto ftp.cs.ucr.edu using the anonymous account name and any password. Switch to the "/pub/pc/ibmpcdir" subdirectory (this is UNIX so make sure you use lowercase letters). You will find the appropriate files by searching through this directory.

The exact filename(s) of this material may change with time, and different services use different names for these files. Generally posting a message enquiring about the UCR Standard Library or this text will generate appropriate responses.