Why Learning Assembly Language Is Still a Good Idea

by Randall Hyde, author of Write Great Code (No Starch)
05/06/2004

The world is full of case studies outlining software engineering disasters. Almost every programmer has had to work on a project involving "less than stellar" source code that was difficult to read and maintain. On rare occasion, some programmers get the opportunity to work on a well-designed system, an awe-inspiring piece of craftsmanship that usually produces the exclamation, "This is truly great code!"

Clearly, professional software engineers should strive to achieve this level of greatness in all their code. But the real question is, "What makes code great?" Simply "meeting specifications" is not how one writes great code. True, in today's software environment, some might actually believe that simply meeting the specifications sets an application apart, as many development projects fail to meet their basic design goals.

However, in other areas greatness is rarely defined by doing the expected and succeeding; greatness is defined by going above and beyond what is expected. Software engineers should expect no less from great software--it should go above and beyond the standard conventions for software development.

Efficiency Is the Key

Because greatness is a multifaceted attribute, a short article such as this one cannot begin to describe all the possible components of a great piece of software. Instead, this article will describe one component of writing great code that has been neglected in recent years as computer systems have increased in capacity and power: efficiency.

Anyone who has been around the computer industry for a decade or more is well aware of this phenomenon: machines are getting exponentially more powerful per unit cost, yet users do not perceive this improvement in the applications that they purchase. For example, while word processors are clearly faster today than they were 21 years ago, they aren't 16,384 times faster as Moore's Law [1] would suggest. Part of the problem, of course, is that some of the additional processing power has been employed to support new features (such as a bitmapped display), but a ***large part of the reason software users aren't seeing an increase in speed is because many of today's programmers don't take the time to write efficient software, or they simply don't know how to write fast software.***

Outrageous software development schedules that don't give programmers enough time to develop efficient code is certainly a problem, but many of today's programmers have grown up with fast CPUs, whose speed has made up for poor coding habits and, as such, many of these programmers have never had to learn how to write fast code.

Unfortunately, when software performance is less than optimal, these programmers generally don't know how to correct the problems with their software. They'll often spout things like "The 90-10 rule," or "I'll just use a profiler to correct the performance problems," but the truth is they don't really know how to improve the performance of their underperforming applications. It's all well and good to say, "I'll just find a better algorithm!" However, finding and deploying that algorithm, if one actually exists, is another matter.

Most of the time you can achieve very good performance boosts by simply improving the implementation of an existing algorithm. A computer scientist may argue that a constant improvement in performance isn't as good as, say, going from an algorithm with O(n^2) performance to one with O(n lg n) performance, but the truth is that most of the time a constant factor of two or three times

improvement, applied throughout a piece of software, can make the difference between a practical application and one that is simply too slow to comfortably use. And it is exactly this type of optimization with which most modern programmers have little experience.

Unfortunately, writing efficient software is a skill, one that must be practiced to learn and one that must be practiced to maintain. Programmers who never practice this skill will never be able to apply it the day they discover that their software is running too slow. Even if a programmer has mastered the skill of writing efficient software, the programmer must practice them on a regular basis. **So, there are two reasons why some programmers don't write efficient (and great) software today: they never learned how to write efficient code in the first place, or they've allowed their programming skills to atrophy to the point that they no longer write efficient code as a matter of course.**

Practice Your Skills

For programmers who have simply allowed their skills to falter from lack of use, the solution is obvious--practice writing efficient code, even when the project doesn't absolutely require it. This doesn't mean, of course, that a practicing engineer should sacrifice project schedules, readable and maintainable code, or other important software attributes for the sake of efficiency.

What it does mean is that the software engineer should keep efficiency in mind while designing and implementing the software. T**he programmer should make a conscious decision to choose a less efficient implementation over a more efficient implementation based on economic or engineering concerns, rather than simply utilizing the first implementation that comes to mind. Just as often as not, this simple consideration of different (and possibly more efficient) implementations is all that is necessary to produce great code. After all, sometimes the more efficient implementation is no more difficult to create than an inefficient one. All an experienced engineer may need are multiple options from which to choose.**

Unfortunately, unrealistic software development schedules have led many professional engineers to shortcut the careful consideration of software development and implementation. The end result is that many professional programmers have gotten out of the habit of writing great code. Fortunately, this process is easy to reverse by practicing good software development methodologies, such as considering multiple algorithms and their implementations, as often as possible.

Learn Assembly Language

What about the programmer who has never learned to write efficient code in the first place? How does one learn how to efficiently implement an application? Unfortunately, colleges and universities today largely take the attitude that if you choose a good algorithm, you don't have to worry about the implementation of that algorithm. Far too many students come out of their data structures and algorithms courses with the attitude that if you can only achieve a constant (that is, O(1)) performance improvement, you've really achieved nothing at all, and that attempts at improvement are a waste of time.

Advances in computer architecture have exacerbated this problem--for example, you might hear a programmer say, "If this program needs to be a little faster, just wait a year or so and CPUs will be twice as fast; there's no need to worry about it." And this attitude, probably more than any other, is why software performance doesn't keep pace with CPU performance.

With every new application, the programmer writes the software slower than it ought to run, on whatever current CPU they're using, believing that future CPU performance boosts will solve their problems. Of course, by the time the CPUs are fast enough to execute their software, the programmer has "enhanced" the software, and is now depending on yet another future version of the CPU. The cycle repeats almost endlessly, with CPU performance never really catching up with the demands of

the software, until finally, the software's life comes to an end and the programmer begins the cycle anew with a different application.

The truth is, it is possible to write software that executes efficiently on contemporary processors. Programmers were doing great things with software back in the days when their applications were running on eight-bit 5MHz 8088 PCs; the same techniques they used to squeeze every last bit of performance out of those low-end CPUs provides the key to high-performance applications today. So, how did they achieve reasonable performance on such low-end processors? The answer is not a secret--**they understood how the underlying hardware operated and they wrote their code accordingly.** That same knowledge, of the underlying hardware, is the key to writing efficient software today.

Often, you'll hear old-time programmers make the comment that truly efficient software is written in assembly language. However, the reason such software is efficient isn't because the implementation language imparts some magical efficiency properties to that software -- it's perfectly possible to write inefficient software in assembly language. No, the real reason assembly language programs tend to be more efficient than programs written in other languages is because assembly language forces the programmer to consider how the underlying hardware operates with each machine instruction they write. And this is the key to learning how to write efficient code -- keeping one's eye on the low-level capabilities of the machine.

Those same old-time programmers who claim that truly efficient software is written in assembly language also offer another common piece of advice -- if you want to learn how to write great high-level language code, learn how to program in assembly language.

This is very good advice. After all, high-level compilers translate their high-level source statements into low-level machine code. So if you know assembly language for your particular machine, you'll be able to correlate high-level language constructs with the machine language sequences that a compiler generates. And with this understanding, you'll be able to choose better high-level language statements based on your understanding of how compilers translate those statements into machine code.

All too often, high-level language programmers pick certain high-level language sequences without any knowledge of the execution costs of those statements. Learning assembly language forces the programmer to learn the costs associated with various high-level constructs. So even if the programmer never actually writes applications in assembly language, the knowledge makes the programmer aware of the problems with certain inefficient sequences so they can avoid them in their high-level code.

Learning assembly language, like learning any new programming language, requires considerable effort. The problem is that assembly language itself is deceptively simple. You can learn the 20 or 30 machine instructions found in common assembly applications in just a few days. You can even learn how to put those machine instructions together to solve problems the same way you'd solve those same problems in a high-level language in just a few short weeks.

Unfortunately, this isn't the kind of knowledge that a high-level language programmer will find useful when attempting to write efficient high-level code. To reap the benefits of knowing assembly language, a programmer has to learn to think in assembly language. Then, such a programmer can write very efficient high-level language code while thinking in assembly and writing high-level language statements. Though code written in this manner is truly great, there is one slight problem with this approach -- it takes considerable effort to achieve this level. That's one of the reasons such code is great -- because so few practitioners are capable of producing it.

Assembly Isn't Dead

Assembly language, of course, developed a very bad reputation throughout the 1990s. Advances in compiler technology, improved CPU performance, and the "software crisis" all conspired to suggest that assembly language was a "dead" language that was no longer needed. As assembly language was a bit more difficult to learn than traditional high-level programming languages, students (and instructors!) gladly embraced this brave new high-level world, abandoning difficult-to-learn assembly in favor of higher and higher level languages.

The only problem with the demise of assembly language is that as its popularity waned, so did the percentage of programmers who understood the low-level ramifications of the code they were writing. Those programmers who were claiming that assembly language was dead already knew how to think in assembly language and how to apply that low-level thinking to their high-level code; in effect, enjoying the benefits of assembly language while writing high-level language code. However, as new programmers worked their way into the system, without the benefits of having written several applications in assembly, the efficiency of software applications began to decline.

Though it would be foolish to start claiming that programmers should begin writing commercial applications in assembly language, it is clear today that the demise of assembly language's popularity has had a big impact on the efficiency of modern software. To reverse this trend, one of two things must happen: programmers must once again begin studying assembly language, or they must somehow pick up this low-level programming knowledge some other way.

Learning assembly language still remains the best way to learn the basic organization of the underlying machine. Those programmers who take the effort to master assembly language become some of the very best high-level language programmers around. Their ability to choose appropriate high-level constructs to produce efficient code, their ability to read disassembled high-level language code and detect heinous bugs in a system, and their understanding of how the whole system operates elevates them to near legendary status among their peers. These are the programmers everyone goes to when they have questions how to implement something. These are the engineers who garner the respect of everyone around them. They are the ones other programmers want to emulate. These are the programmers who write great code.

If knowing assembly language helps make programmers great, a obvious question is "Why don't more programmers learn assembly language?" Part of the problem is prejudice: many college and university instructors that teach assembly programming begin their course with a statement like, "No one really needs to know this stuff, and you'll never use it, but it is required by this program so we've got to struggle through the next several weeks studying this material." After four years of this type of attitude from their instructors, it's no surprise that students really want nothing whatsoever at all to do with assembly language programming.

Still, once it becomes obvious to a coder that the truly great programmers are the ones who've mastered assembly language programming, you might ask why more programmers don't pick up this valuable knowledge. The only problem is that, traditionally, most programmers have found it difficult to master assembly language. Assembly is radically different than most high-level languages, so learning assembly language is almost as much work as learning programming from scratch.

To someone attempting to learn assembly, it often seems as though none of their past programming experience is of any help. All too often, an engineer learning assembly becomes frustrated with the fact that they know how to achieve a goal in a high-level language but they cannot figure out how to achieve the same thing in assembly. For many programmers, switching from "thinking in a high-level language" to "thinking in an assembly language" becomes an insurmountable problem.

As an instructor teaching assembly language for over a decade at the University of California, I was quite aware of the problems students have making the transition from the high- level programming paradigm to the low-level programming paradigm.

In the early 1990s, Microsoft provided a solution with the introduction of the Microsoft Macro Assembler (MASM) v6.0 -- the inclusion of high-level control structures in an assembly language translator. While these new statements are definitely not true assembly language, they do provide a nice transition path from traditional, imperative, high-level programming languages to assembly. A

programmer can continue to use statements like IF, WHILE, and FOR while learning other aspects of assembly language programs. This lets the programmer learn assembly language programming in graduated steps rather than having to make the plunge all at once.

Master Low-Level Statements

Of course, a programmer cannot truly call themselves an assembly language programmer until they've mastered the equivalent low-level statements. Nevertheless, these high-level control structures provide an excellent bridge between high-level languages and assembly language, allowing the student to leverage their existing high-level programming knowledge to learn assembly language. Alas, there are few, if any, appropriate textbooks that teach assembly language programming using this high-level to low-level approach using the high-level control structures that MASM provides.

Another problem with the high-level to low-level transition is that most high-level languages provide large libraries of routines to handle mundane tasks such as input/output, numeric conversions, and string operations. A big problem that beginning assembly programmers face is that they typically need the ability to input and output numeric quantities or do numeric-to-string conversions (and vice versa) in order to write and test very simple programs. Unfortunately, most assembly language development systems leave it up to the programmer to provide this functionality for their applications. This presents a Catch-22 situation: it is difficult to learn assembly language without these functions, but you can't really write such functions until you learn assembly language.

These roadblocks effectively prevent all but the most determined programmers from mastering assembly language. As such, throughout the 1990s and early 2000s the use of assembly language continued to wane. Seeing the decline in software efficiency and quality that seemed to track the decline of the use of assembly language, I set out on a crusade in the mid-1990s to encourage programmers to learn assembly language programming.

...

Most importantly, while it is difficult to sell the idea of learning assembly language to the current generation of programmers (two decades of anti-assembly propaganda have assured this), most programmers realize that if they just "learned a little more about how the underlying hardware works" they would be able to write better code.

...

To write great code requires one to write efficient code. Writing efficient code requires good algorithm choices and a good implementation of those algorithms. The best implementations are going to be written by those who've mastered assembly language or fully understand the low-level implementation of the high-level language statements they're choosing. This doesn't mean that we'll return to the days when major commercial applications were written entirely in assembly language. However, to reverse the trend of software running slower and slower even though CPUs are running faster and faster is going to require programmers to begin considering the low-level implications of the code that they write.

So, even if you never intend to write a single line of assembly language code again, learning assembly language, and learning to think in assembly language, is one of the best things you can do to learn how to write better assembly code.

[1] Moore's Law states that semiconductor technology doubles in capacity or performance at a given price point about every 18 months.

Randall Hyde is the author of the recently released "Write Great Code" and "The Art of Assembly Language" (both from No Starch Press), one of the most highly recommended resources on assembly.