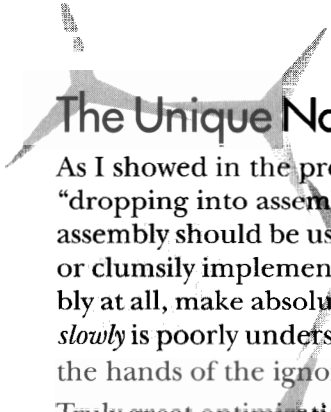
Chapter 2

A World Apart

Chapter 2



The Unique Nature of Assembly Language Optimization

As I showed in the previous chapter, optimization is by no means always a matter of "dropping into assembly." In fact, in performance tuning high-level language code, assembly should be used rarely, and then only after you've made sure a badly chosen or clumsily implemented algorithm isn't eating you alive. Certainly if you use assembly at all, make absolutely sure you use it *right*. The potential of assembly code to run *slowly* is poorly understood by a lot of people, but that potential is great, especially in the hands of the ignorant.

Truly great optimization, however, happens *only* at the assembly level, and it happens in response to a set of dynamics that is totally different from that governing C/C++ or Pascal optimization. I'll be speaking of assembly-level optimization time and again in this book, but when I do, I think it will be helpful if you have a grasp of those assembly specific dynamics.

As usual, the best way to wade in is to present a real-world example.

Instructions: The Individual versus the Collective

Some time ago, I was asked to work over a critical assembly subroutine in order to make it run as fast as possible. The task of the subroutine was to construct a nibble out of four bits read from different bytes, rotating and combining the bits so that they ultimately ended up neatly aligned in bits 3-0 of a single byte. (In case you're curious, the object was to construct a 16-color pixel from bits scattered over 4 bytes.)

I examined the subroutine line by line, saving a cycle here and a cycle there, until the code truly seemed to be optimized. When I was done, the key part of the code looked something like this:

```
LoopTop:
      lodsb             ;get the next byte to extract a bit from
      and   al,ah       ;isolate the bit we want
      rol   al,cl       ;rotate the bit into the desired position
      or    bl,al       ;insert the bit into the final nibble
      dec   cx          ;the next bit goes 1 place to the right
      dec   dx          ;count down the number of bits
      jnz   LoopTop     ;process the next bit, if any
```

Now, it's hard to write code that's much faster than seven instructions, only one of which accesses memory, and most programmers would have called it a day at this point. Still, something bothered me, so I spent a bit of time going over the code again. Suddenly, the answer struck me—the code was rotating each bit into place separately, so that a multibit rotation was being performed every time through the loop, for a total of four separate time-consuming multibit rotations!

While the instructions themselves were individually optimized, the overall approach did not make the best possible use of the instructions.

I changed the code to the following:

```
LoopTop:
      lodsb             ;get the next byte to extract a bit from
      and   al,ah       ;isolate the bit we want
      or    bl,al       ;insert the bit into the final nibble
      rol   bl,1        ;make room for the next bit
      dec   dx          ;count down the number of bits
      jnz   LoopTop     ;process the next bit, if any
      rol   bl,cl       ;rotate all four bits into their final
                        ; positions at the same time
```

This moved the costly multibit rotation out of the loop so that it was performed just once, rather than four times. While the code may not look much different from the original, and in fact still contains exactly the same number of instructions, the performance of the entire subroutine improved by about 10 percent from just this one change. (Incidentally, that wasn't the end of the optimization; I eliminated the **DEC** and **JNZ** instructions by expanding the four iterations of the loop—but that's a tale for another chapter.)

The point is this: To write truly superior assembly programs, you need to know what the various instructions do and which instructions execute fastest...and more. You must also learn to look at your programming problems from a variety of perspectives so that you can put those fast instructions to work in the most effective ways.

Assembly Is Fundamentally Different

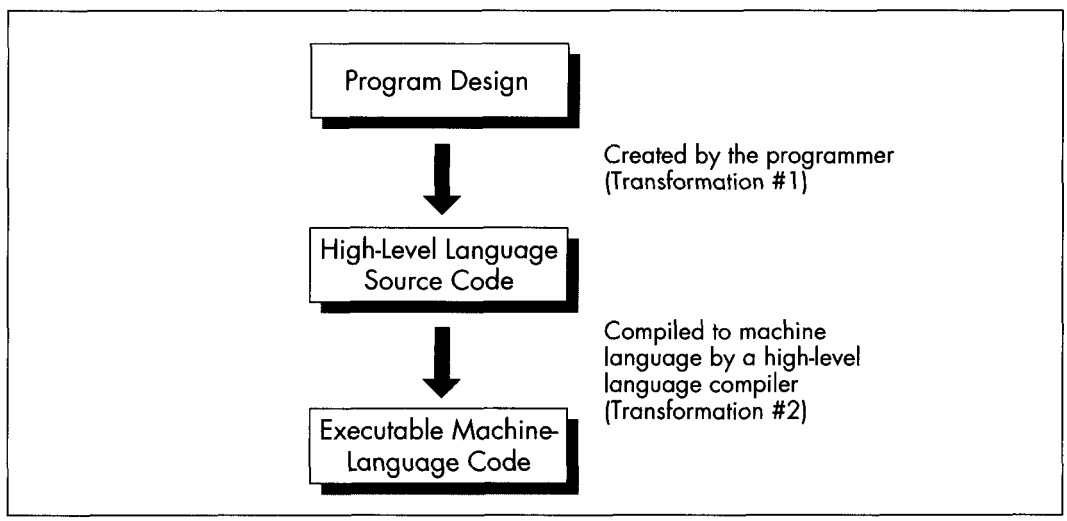
Is it really so hard as all that to write good assembly code for the PC? Yes! Thanks to the decidedly quirky nature of the x86 family CPUs, assembly language differs fundamentally from other languages, and is undeniably harder to work with. On the other hand, the potential of assembly code is much greater than that of other languages, as well.

To understand why this is so, consider how a program gets written. A programmer examines the requirements of an application, designs a solution at some level of abstraction, and then makes that design come alive in a code implementation. If not handled properly, the transformation that takes place between conception and implementation can reduce performance tremendously; for example, a programmer who implements a routine to search a list of 100,000 sorted items with a linear rather than binary search will end up with a disappointingly slow program.

Transformation Inefficiencies

No matter how well an implementation is derived from the corresponding design, however, high-level languages like C/C++ and Pascal inevitably introduce additional transformation inefficiencies, as shown in Figure 2.1.

The process of turning a design into executable code by way of a high-level language involves two transformations: one performed by the programmer to generate source code, and another performed by the compiler to turn source code into machine



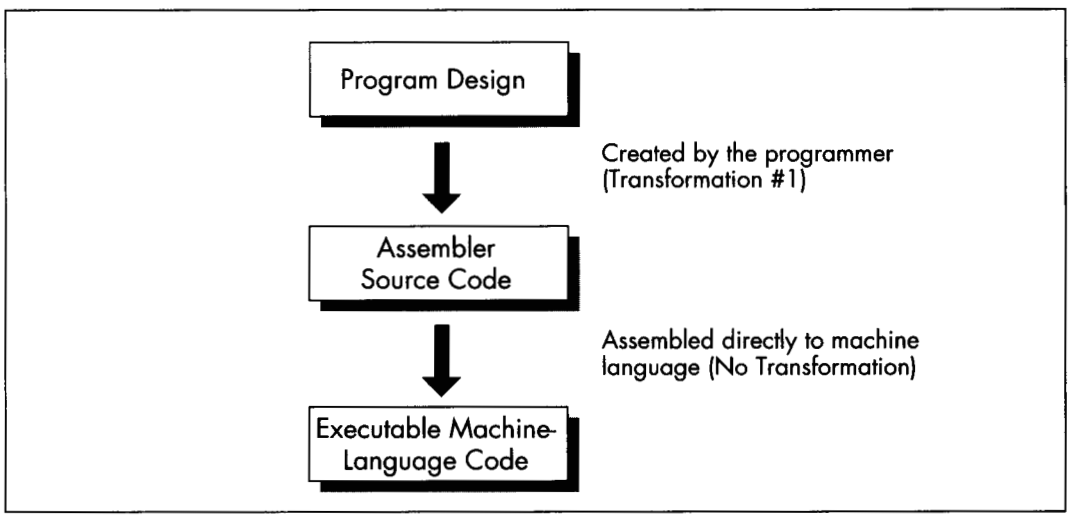
The high-level language transformation inefficiencies.
Figure 2.1

language instructions. Consequently, the machine language code generated by compilers is usually less than optimal given the requirements of the original design.

High-level languages provide artificial environments that lend themselves relatively well to human programming skills, in order to ease the transition from design to implementation. The price for this ease of implementation is a considerable loss of efficiency in transforming source code into machine language. This is particularly true given that the x86 family in real and 16-bit protected mode, with its specialized memory-addressing instructions and segmented memory architecture, does not lend itself particularly well to compiler design. Even the 32-bit mode of the 386 and its successors, with their more powerful addressing modes, offer fewer registers than compilers would like.

Assembly, on the other hand, is simply a human-oriented representation of machine language. As a result, assembly provides a difficult programming environment—the bare hardware and systems software of the computer—*but properly constructed assembly programs suffer no transformation loss*, as shown in Figure 2.2.

Only one transformation is required when creating an assembler program, and that single transformation is completely under the programmer's control. Assemblers perform no transformation from source code to machine language; instead, they merely map assembler instructions to machine language instructions on a one-to-one basis. As a result, the programmer is able to produce machine language code that's precisely tailored to the needs of each task a given application requires.



Properly constructed assembly programs suffer no transformation loss.
Figure 2.2

The key, of course, is the programmer, since in assembly the programmer must essentially perform the transformation from the application specification to machine language entirely on his or her own. (The assembler merely handles the *direct* translation from assembly to machine language.)

Self-Reliance

The first part of assembly language optimization, then, is self-reliance. An assembler is nothing more than a tool to let you design machine-language programs without having to think in hexadecimal codes. So assembly language programmers—unlike all other programmers—must take full responsibility for the quality of their code. Since assemblers provide little help at any level higher than the generation of machine language, the assembly programmer must be capable both of coding any programming construct directly and of controlling the PC at the lowest practical level—the operating system, the BIOS, even the hardware where necessary. High-level languages handle most of this transparently to the programmer, but in assembly everything is fair—and necessary—game, which brings us to another aspect of assembly optimization: knowledge.

Knowledge

In the PC world, you can never have enough knowledge, and every item you add to your store will make your programs better. Thorough familiarity with both the operating system APIs and BIOS interfaces is important; since those interfaces are well-documented and reasonably straightforward, my advice is to get a good book or two and bring yourself up to speed. Similarly, familiarity with the PC hardware is required. While that topic covers a lot of ground—display adapters, keyboards, serial ports, printer ports, timer and DMA channels, memory organization, and more—most of the hardware is well-documented, and articles about programming major hardware components appear frequently in the literature, so this sort of knowledge can be acquired readily enough.

The single most critical aspect of the hardware, and the one about which it is hardest to learn, is the CPU. The x86 family CPUs have a complex, irregular instruction set, and, unlike most processors, they are neither straightforward nor well-documented regarding true code performance. What's more, assembly is so difficult to learn that most articles and books that present assembly code settle for code that just works, rather than code that pushes the CPU to its limits. In fact, since most articles and books are written for inexperienced assembly programmers, there is very little information of any sort available about how to generate high-quality assembly code for the x86 family CPUs. As a result, knowledge about programming them effectively is by far the hardest knowledge to gather. A good portion of this book is devoted to seeking out such knowledge.

Be forewarned, though: No matter how much you learn about programming the PC in assembly, there's always more to discover.

The Flexible Mind

Is the never-ending collection of information all there is to the assembly optimization, then? Hardly. Knowledge is simply a necessary base on which to build. Let's take a moment to examine the objectives of good assembly programming, and the remainder of the forces that act on assembly optimization will fall into place.

Basically, there are only two possible objectives to high-performance assembly programming: Given the requirements of the application, keep to a minimum either the number of processor cycles the program takes to run, or the number of bytes in the program, or some combination of both. We'll look at ways to achieve both objectives, but we'll more often be concerned with saving cycles than saving bytes, for the PC generally offers relatively more memory than it does processing horsepower. In fact, we'll find that two-to-three times performance improvements *over already tight assembly code* are often possible if we're willing to spend additional bytes in order to save cycles. It's not always desirable to use such techniques to speed up code, due to the heavy memory requirements—but it is almost always *possible.*

You will notice that my short list of objectives for high-performance assembly programming does not include traditional objectives such as easy maintenance and speed of development. Those are indeed important considerations—to persons and companies that develop and distribute software. People who actually *buy* software, on the other hand, care only about how well that software performs, not how it was developed nor how it is maintained. These days, developers spend so much time focusing on such admittedly important issues as code maintainability and reusability, source code control, choice of development environment, and the like that they often forget rule #1: From the user's perspective, *performance is fundamental.*

Comment your code, design it carefully, and write non-time-critical portions in a high-level language, if you wish—but when you write the portions that interact with the user and/or affect response time, performance must be your paramount objective, and assembly is the path to that goal.

Knowledge of the sort described earlier is absolutely essential to fulfilling either of the objectives of assembly programming. What that knowledge doesn't do by itself is meet the need to write code that both performs to the requirements of the application at hand and also operates as efficiently as possible in the PC environment. Knowledge makes that possible, but your programming instincts make it happen. And it is that intuitive, on-the-fly integration of a program specification and a sea of facts about the PC that is the heart of the Zen-class assembly optimization.

As with Zen of any sort, mastering that Zen of assembly language is more a matter of learning than of being taught. You will have to find your own path of learning, although I will start you on your way with this book. The subtle facts and examples I provide will help you gain the necessary experience, but you must continue the journey on your own. Each program you create will expand your programming horizons and increase the options available to you in meeting the next challenge. The ability of your mind to find surprising new and better ways to craft superior code from a concept—the flexible mind, if you will—is the linchpin of good assembler code, and you will develop this skill only by doing.

Never underestimate the importance of the flexible mind. Good assembly code is better than good compiled code. Many people would have you believe otherwise, but they're wrong. That doesn't mean that high-level languages are useless; far from it. High-level languages are the best choice for the majority of programmers, and for the bulk of the code of most applications. When the *best* code—the fastest or smallest code possible—is needed, though, assembly is the only way to go.

Simple logic dictates that no compiler can know as much about what a piece of code needs to do or adapt as well to those needs as the person who wrote the code. Given that superior information and adaptability, an assembly language programmer can generate better code than a compiler, all the more so given that compilers are constrained by the limitations of high-level languages and by the process of transformation from high-level to machine language. Consequently, carefully optimized assembly is not just the language of choice but the *only* choice for the 1percent to 10 percent of code—usually consisting of small, well-defined subroutines—that determines overall program performance, and it is the only choice for code that must be as compact as possible, as well. In the run-of-the-mill, non-time-critical portions of your programs, it makes no sense to waste time and effort on writing optimized assembly code—concentrate your efforts on loops and the like instead; but in those areas where you need the finest code quality, accept no substitutes.

Note that I said that an assembly programmer *can* generate better code than a compiler, not *will* generate better code. While it is true that good assembly code is better than good compiled code, it is also true that bad assembly code is often much worse than bad compiled code; since the assembly programmer has so much control over the program, he or she has virtually unlimited opportunities to waste cycles and bytes. The sword cuts both ways, and good assembly code requires more, not less, forethought and planning than good code written in a high-level language.

The gist of all this is simply that good assembly programming is done in the context of a solid overall framework unique to each program, and the flexible mind is the key to creating that framework and holding it together.

Where to Begin?

To summarize, the skill of assembly language optimization is a combination of knowledge, perspective, and a way of thought that makes possible the genesis of absolutely the fastest or the smallest code. With that in mind, what should the first step be? Development of the flexible mind is an obvious step. Still, the flexible mind is no better than the knowledge at its disposal. The first step in the journey toward mastering optimization at that exalted level, then, would seem to be learning how to learn.