Chapter 5: Night of the Cycle-Eaters

When sorrows come, they come not single spies,
But in battalions.

-- William Shakespeare, _Hamlet_

Thus far we've explored what might be called the science of assembler programming. We've dissected in considerable detail the many factors that affect code performance, increasing our understanding of the PC greatly in the process. We've approached the whole business in a logical fashion, measuring 8 cycles here, accounting for 6 cycles there, always coming up with reasonable explanations for the phenomena we've observed. In short, we've acted as if assembler programming for the PC can be reduced to a well-understood, cut-and-dried cookbook discipline once we've learned enough.

I'm here to tell you it ain't so.

Assembler programming for the PC can't be reduced to a science, and the cycle-eaters are the reasons why. The 8-bit bus and prefetch queue cycle-eaters give every code sequence on the PC unique and hard-to-predict performance characteristics. Throw in the DRAM refresh and display adapter cycle-eaters and you've got virtually infinite possibilities not only for the performance of different code sequences but also for the performance of the _same_ code sequence at different times! There is simply _no way_ to know in advance exactly how fast a specific instance of an instruction will execute, and there's no way to be sure what code is the fastest for a particular purpose. Instead, what we must do is use the Zen timer to gain experience and develop rules of thumb, write code by feel as much as by prescription, and measure the actual performance of what we write.

In other words, we must become Zen programmers.

As you read this, you may understand but not believe. Surely, you think, there must be a way to know what the best code is for a given task. How can it not be possible to come up with a purely rational solution to a problem that involves that most rational of man's creations, the computer?

The answer lies in the nature of the computer in question. While it's true that it's not impossible to understand the exact performance of a given piece of code on the IBM PC, because of the 8-bit bus and prefetch queue cycle-eaters it _is_ extremely complex and requires expensive hardware. And then, when you fully understand the performance of that piece of code, what have you got? Only an understanding of one out of millions of possible code sequences, each of which is a unique problem requiring just as much analysis as did the first code sequence.

That's bad enough, but the two remaining cycle-eaters make the problem of understanding code performance more complex still. The DRAM refresh and display adapter cycle-eaters don't affect the execution of each instruction equally; they occur periodically and have varying impacts on performance when they do occur, thereby causing instruction performance to vary as a function not only of the sequence of instructions but also of time. In other words, the understanding you gain of a particular code sequence <u>may not even be valid the next time that code runs</u>, thanks to the varying effects of the DRAM refresh and display adapter cycle-eaters.

In short, it is true that the exact performance of assembler code is indeed a solvable problem in the classic sense, since everything about the performance of a given execution of a given chunk of code is knowable given enough time, effort, and expensive hardware. It is equally true, however, that the exact performance of assembler code over time is such a complex problem that it might as well be unsolvable, since that hard-won knowledge would be so specific as to be of no use. We are going to spend the rest of this chapter proving that premise. First we'll look at some of the interactions between the cycle-eaters; those interactions make the prediction of code performance still more complex than we've seen thus far. After that we'll look at every detail of 170 cycles in the life of the PC. What we'll find is that if we set out to understand the exact performance of an entire assembler program, we could well spend the rest of our lives at that task--and would be no better off than we were before.

The object of this chapter is to convince you that when it comes to writing assembler code there's no complete solution, no way to understand every detail or get precise, unvarying answers about performance. We <u>can</u> come close, though, by understanding the basic operation of the PC, developing our intuition, following rules of thumb such as keeping instructions short, and always measuring code performance. Those approaches are precisely what this book is about, and are the foundation of the Zen of assembler.

NO, WE'RE NOT IN KANSAS ANYMORE

You may be feeling a bit lost at this point. That's certainly understandable, for the last two chapters have covered what is surely the most esoteric aspect of assembler programming. I must tell you that this chapter will be more of the same.

Follow along as best you can, but don't be concerned if some of the material is outside your range right now. Both the following chapters and experience will give you a better feel for what this chapter is about. It's important that you be exposed to these concepts now, though, so you can recognize them when you run into them later. The key concept to come away from this chapter with is that the cycle-eaters working together make for such enormous variability of code performance that there's no point in worrying about exactly what's happening in the execution of a given instruction or sequence of instructions. Instead, we must use rules of thumb and a programming feel developed with experience, and we must focus on overall performance as measured with the Zen timer.

CYCLE-EATERS BY THE BATTALION

Taken individually the cycle-eaters are formidable, as we saw in the last chapter. Cycle-eaters don't line up neatly and occur one at a time, though. They're like the proverbial 900- pound gorilla--they occur whenever they want. Frequently one cycle-eater will occur during the course of another cycle-eater, with compound (and complex) effects.

For example, it's perfectly legal to put code in display memory and execute that code. However, as the instruction bytes of that code are fetched they'll be subjected to the display adapter cycle-eater, meaning that each instruction byte could easily take twice as long as usual to fetch. Naturally, this will worsen the already serious effects of the prefetch queue cycle-eater. (Remember that the prefetch queue cycle-eater is simply the inability of the 8088 to fetch instruction bytes quickly enough.) In this case, the display adapter and prefetch queue cycle-eaters together could make overall execution times five to ten times longer than the times listed in Appendix A!

As another example, the DRAM refresh and 8-bit bus cycle- eaters can work together to increase the variability of code performance. When DRAM refresh occurs during an instruction that accesses a word-sized memory operand, the instruction's memory accesses are held up until the DRAM refresh is completed. However, the exact amount by which the instruction's accesses are delayed (and which access is delayed, as well) depends on exactly where in the course of execution the instruction was when the DRAM refresh came along. If the DRAM refresh happens just as the 8088 was about to begin a bus access, the 8088 can be held up for a long time. If, however, the DRAM refresh happens while the 8088 is performing internal operations, such as address calculations or instruction decoding, the impact on performance will be less.

The point is not, Lord knows, that you should understand how every cycle-eater affects every other cycle-eater and how together and separately they affect each instruction in your code. Quite the opposite, in fact. *I* certainly don't understand all the interactions between cycle-eaters and code performance, and frankly I don't ever expect (or want) to. Rather, what I'm telling you (again) is that a complete understanding of the performance of a given code sequence is so complex and varies so greatly with context that there's no point worrying about it. As a result, high-performance assembler code comes from programming by intuition and experience and then measuring performance, not from looking up execution times and following rigid rules. In a way that's all to the good: experienced, intuitive assembler programmers are worth a great deal, because no compiler can rival a good assembler programmer's ability to deal with cycle-eaters and the complexity of code execution on the 8088.

One fallout of the near-infinite variability of code performance is that the exact performance of a given instruction is for all intents and purposes undefined. There are so many factors affecting performance, and those factors can vary so easily with time and context, that there's just no use to trying to tag a given instruction with a single execution time. In other words...

...THERE'S STILL NO SUCH BEAST AS A TRUE EXECUTION TIME

Thanks to the combined efforts of the cycle-eaters, it's more true than ever that there's no such thing as a single "true" execution time for a given instruction. As you'll recall, I said that in the last chapter. Why do I keep bringing it up? Because I don't want you to look at the times reported by our tests of 1000 repetitions of the same instruction and think that those times are the true execution times of that instruction--they aren't, any more than the official cycle times in Appendix A are the true times. <u>There is no such thing as a true execution time on the 8088.</u> There are only execution times in context.

Do you remember the varying performances of **shr** in different contexts in Chapter 4? Well, that was just for repeated instances of one or two instructions. Imagine how much variation cycle-eaters could induce in the performance of a sequence of ten or twenty different instructions, especially if some of the instructions accessed word-sized display memory operands. You should always bear in mind that the times reported by the Zen timer are accurate only for the particular code sequence you've timed, not for all instances of a given instruction in all code sequences. There's just no way around it: <u>you must measure the performance of your code to know how fast it is.</u> Yes, I know--it would be awfully nice just to be able to look up instruction execution times and be done with it. That's not the way the 8088 works, though--and the odd architecture of the 8088 is what the Zen of assembler is all about.

170 CYCLES IN THE LIFE OF A PC

Next, we're going to examine every detail of instruction execution on the PC over a period of 170 cycles. One reason for doing this is to convince any of you who may still harbor the notion that there must be some way to come up with hard-and-fast execution times that you're on a fool's quest. Another reason is to illustrate many of the concepts we've developed over the last two chapters.

A third reason is simple curiosity. We'll spend most of this book measuring instruction execution times and inferring how cycle-eaters and instruction execution are interacting. Why not take a look at the real thing? It won't answer any fundamental questions, but it will give us a feel for what's going on under the programming interface.

THE TEST SET-UP

The code we'll observe is shown in Listing 5-1. This code is an endless loop in which the value stored in the variable **i** is copied to the variable **j** over and over by way of AH. The **DS:** override prefixes on the variables, while not required, make it clear that both variables are accessed by way of DS.

The detailed performance of the code in Listing 5-1 was monitored with the logic analyzer capability of the OmniLab multipurpose electronic test instrument manufactured by Orion Instruments. (Not coincidentally, I was part of the team that developed the OmniLab software.) OmniLab's probes were hooked up to a PC's 8088 and bus, Listing 5-1 was started, and a snapshot of code execution was captured and studied.

By the way, OmniLab, a high-performance but relatively low-priced instrument, costs (circa 1989) about $9,000. Money is one reason why you probably won't want to analyze code performance in great detail yourself!

The following lines of the 8088 were monitored with OmniLab: the 16 lines that carry addresses, 8 of which also carry data, the READY line (used to hold the 8088 up during DRAM refresh), and the QS1 and QS0 lines (which signal transfers of instruction bytes from the prefetch queue to the Execution Unit). The /MEMR and /MEMW lines on the PC bus were monitored in order to observe memory accesses. The 8088 itself provides additional information about bus cycle timing and type, but the lines described above will show us program execution in plenty of detail for our purposes.

Odds are that you, the reader, are not a hardware engineer. After all, this is a book about software, however far it may seem to stray at times. Consequently, I'm not going to show the execution of Listing 5-1 in the form of the timing diagrams of which hardware engineers are so fond. Timing diagrams are fine for observing the state of a single line, but are hard to follow at an overall level, which is precisely what we want to see. Instead, I've condensed the information I collected with OmniLab into an event time-line, shown in Figure 5-1.

THE RESULTS

Figure 5-1 shows 170 consecutive 8088 cycles. To the left of the cycle time-line Figure 5-1 shows the timing of instruction byte transfers from the prefetch queue to the Execution Unit. This information was provided by the QS1 and QS0 pins of the 8088. To the right of the cycle time-line Figure 5-1 shows the timing of bus read and write accesses. The timing of these accesses was provided by the /MEMR and /MEMW lines of the PC bus, and the data and addresses were provided by the address/data lines of the 8088. One note for the technically oriented: since bus accesses take 4 cycles from start to finish, I considered the read and write accesses to complete on the last cycle during which /MEMR or /MEMW was active.

Take a minute to look Figure 5-1 over, before we begin our discussion. Bear in mind that Figure 5-1 is actually a simplified, condensed version of the information that actually appeared on the 8088's pins. In other words, if you choose to analyze cycle-by-cycle performance yourself, the data will be considerably <u>harder</u> to interpret than Figure 5-1!

CODE EXECUTION ISN'T ALL THAT EXCITING

The first thing that surely strikes you about Figure 5-1 is that it's awfully tedious, even by assembler standards. During the entire course of the figure only seven instructions are executed--not much to show for all the events listed. The monotony of picking apart code execution is one reason why such a detailed level of understanding of code performance isn't desirable.

THE 8088 REALLY DOES COPROCESS

The next notable aspect of Figure 5-1 is that you can truly see the two parts of the 8088--the Execution Unit and the Bus Interface Unit--coprocessing. The left side of the time-line shows the times at which the EU receives instruction bytes to execute, indicating the commencement and continuation of instruction execution. The right side of the time-line shows the times at which the BIU reads or writes bytes from or to memory, indicating instruction fetches and accesses to memory operands.

The two sides of the time-line overlap considerably. For example, at cycle 10 the EU receives the opcode byte of **mov ds:[j],ah** from the prefetch queue at the same time that the BIU prefetches the mod-reg-rm byte for the same instruction. (We'll discuss mod-reg-rm bytes in detail in Chapter 7.) Clearly, the two parts of the 8088 are processing independently during cycle 10. The EU and BIU aren't always able the process independently, however. The EU spends a considerable amount of time waiting for the BIU to provide the next instruction byte, thanks to the prefetch queue cycle-eater. This is apparent during cycles 129 through 135, where the EU must wait 6 cycles for the mod-reg-rm byte of **mov ah,ds:[i]** to arrive. Back at cycle 84, the EU only had to wait 1 cycle for the same byte to arrive. Why the difference?

The difference is the result of the DRAM refresh that occurred at cycle 118, preempting the bus and delaying prefetching so that the mod-reg-rm byte of **mov ah,ds:[i]** wasn't available until cycle 135. What's particularly interesting is that this variation occurs even though the sequence of instructions is exactly the same at cycle 83 as at cycle 129. In this case, it's the DRAM refresh cycle-eater that causes identical instructions in identical code sequences to execute at different speeds. Another time, it might be the display adapter cycle-eater that causes the variation, or the prefetch queue cycle-eater, or a combination of the three. This is an important lesson in the true nature of code execution: the same instruction sequence may execute at different speeds at different times.

WHEN DOES AN INSTRUCTION EXECUTE?

One somewhat startling aspect of Figure 5-1 is that it makes it clear that there is no such thing as the time interval during which a given instruction--and only that instruction--executes. There is the time at which a given byte of an instruction is prefetched, there is a time at which a given byte of an instruction is sent to the EU, and there is a time at which each memory operand byte of an instruction is accessed. None of those times really marks the start or end of an instruction, though, and the instruction fetches and memory accesses of one instruction usually overlap those of other instructions. Figure 5-2 illustrates the full range of action of each of the instructions in Figure 5-1. (In Figure 5-2, and in Figure 5-3 as well, the two sides of the time-line are equivalent; there is no specific meaning to text on, say, the left side as there is in Figure 5-1. I simply alternate sides in order to keep one instruction from running into the next.)

For example, at cycle 143 the last instruction byte of **mov ah,ds:[i]** is sent to the EU. At cycle 144 the opcode of the next instruction, **mov ds:[j],ah**, is prefetched. Not until cycle 150 is the operand of **mov ah,ds:[i]** read, and not until cycle 154 is the opcode byte of **mov ds:[j],ah** sent to the EU. Which instruction is executing between cycles 143 and 154?

It's easiest to consider execution to start when the opcode byte of an instruction is sent to the EU and end when the opcode byte of the next instruction is sent to the EU, as shown in Figure 5-3. Under this approach, the current instruction is charged with any instruction fetch time for the opcode byte of the next instruction that isn't overlapped with EU execution of the current instruction. This is consistent with our conclusion in Chapter 4 that execution time is, practically speaking, EU execution time plus any instruction fetch time that's not overlapped with the EU execution time of another instruction. Therefore, **mov ah,ds:[i]** executes during cycles 129 through 153.

In truth, though, the first hint of **mov ah,ds:[i]** occurs at cycle 122, when the opcode byte is fetched. In fact, since read accesses to memory take 4 cycles, the 8088 must have begun fetching the opcode byte earlier still. Figure 5-2 assumes that the 8088 starts bus accesses 2 cycles before the cycle during which /MEMR or /MEMW becomes inactive. That assumption may be off by a cycle, but none of our conclusions would be altered if that were the case. Consequently, the instruction **mov ah,ds:[i]** occupies the attention of at least some part of the 8088 from around cycle 120 up through cycle 153, or 34 cycles, as shown in Figure 5-2.

Figure 5-3 shows that **mov ah,ds:[i]** doesn't take 34 cycles to execute, however. The instruction fetching that occurs during cycles 120 through 128 is overlapped with the execution of the preceding instruction, so those cycles aren't counted against the execution time of **mov ah,ds:[i]**. The instruction does take 25 cycles to execute, though, illustrating the power of the cycle- eaters: according to Appendix A, **mov ah,ds:[i]** should execute in 14 cycles, so just two of the cycle-eaters, the prefetch queue and DRAM refresh, have nearly doubled the actual execution time of the instruction in this context.

THE TRUE NATURE OF INSTRUCTION EXECUTION

Figure 5-1 makes it perfectly clear that at the lowest level code execution is really nothing more than two parallel chains of execution, one taking place in the EU and one taking place in the BIU. What's more, the BIU interleaves instruction fetches for one instruction with memory operand accesses for another instruction. Thus, instruction execution really consists of three interleaved streams of events.

Unfortunately, assembler itself tests the limits of human comprehension of processor actions. Thinking in terms of the interleaved streams of events shown in Figure 5-1 is too much for any mere mortal. It's ridiculous to expect that an assembler programmer could visualize interleaved instruction fetches, EU execution, and memory operand fetches as he writes code, and in fact no one even tries to do so.

And that is yet another reason why an understanding of code performance at the level shown in Figure 5-1 isn't desirable.

VARIABILITY

This brings us to an excellent illustration of the variability of performance, even for the same instruction in the same code sequence executing at different times. As we just discovered, the **mov ah,ds:[i]** instruction that starts at cycle 129 takes 25 cycles to execute. However, the same instruction starting at cycle 33 takes 27 cycle to execute. Starting at cycle 83, **mov ah,ds:[i]** takes just 21 cycles to execute. <u>That's three significantly different times for the same instruction executing in the same</u>

instruction sequence!

How can this be? In this case it's the DRAM refresh cycle- eater that's stirring things up by periodically holding up 8088 bus accesses for 4 cycles or more. This alters the 8088's prefetching and memory access sequence, with a resultant change in execution time. As we discussed earlier, the DRAM refresh read at cycle 118 takes up valuable bus access time, keeping the 8088 from fetching the mod-reg-rm byte of **mov ah,ds:[i]** ahead of time and thereby pushing the succeeding bus accesses a few cycles later in time.

The DRAM refresh and display adapter cycle-eaters can cause almost any code sequence to vary in much the same way over time. That's why the Zen timer reports fractional times. It is also the single most important reason why a micro-analysis of code performance of the sort done in Figure 5-1 is not only expensive and time-consuming but also pointless. If a given instruction following the same sequence of instructions can vary in performance by 20%, 50%, 100% or even more from one execution to the next, what sort of performance number can you give that instruction other than as part of the overall instruction sequence? What point is there in trying to understand the instruction's exact performance during any one of those executions?

The answer, briefly stated, is: no point at all.

YOU NEVER KNOW UNLESS YOU MEASURE (IN CONTEXT!)

I hope I've convinced you that the actual performance of 8088 code is best viewed as the interaction of many highly variable forces, the net result of which is measurable but hardly predictable. But just in case I haven't, consider this...

Figure 5-1 illustrates the execution of one of the simplest imaginable code sequences. The exact pattern of code execution repeats after 144 cycles, so even with DRAM refresh we have an execution pattern that repeats after only 6 instructions. That's not likely to be the case with real code, which rarely features the endless alternation of two instructions. In real code the code mix changes endlessly, so DRAM refresh and the prefetch queue cycle-eater normally result in a far greater variety of execution sequences than in Figure 5-1.

Also, only two of the four cycle-eaters are active in Figure 5-1. Since Listing 5-1 uses no word-sized operands, the 8-bit bus cycle-eater has no effect other than slowing instruction prefetching. Likewise, Listing 5-1 doesn't access display memory, so the display adapter cycle-eater doesn't affect performance. Imagine if we threw those cycle-eaters into Figure 5-1 as well!

Worse still, in the real world interrupts occur often and asynchronously, flushing the prefetch queue and often changing the fetching, execution, and memory operand access patterns of whatever code happens to be running. Most notable among these interrupts is the timer interrupt, which occurs every 54.9 ms. Because the timer interrupt may occur after any instruction and doesn't always take the same amount of time, it can cause execution to settle into new patterns. For example, after I captured the sequence shown in Figure 5-1 I took another snapshot of the execution of Listing 5-1. <u>The second snapshot did not match the first.</u> The timer interrupt had kicked execution into a different pattern, in which the same instructions were executed, with the same results--but not at exactly the same speeds.

Other interrupts, such as those from keyboards, mice, and serial ports, can similarly alter program performance. Of course, interrupts and cycle-eaters don't change the <u>effects</u> of code--**add ax,1** always adds 1 to AX, and so on--but they can drastically change the performance of a given instruction in a given context. That's why we focus on the overall performance of code sequences in context, as measured with the Zen timer, rather than on the execution times of individual instructions.

THE LONGER THE BETTER

Now is a good time to point out that the longer the instruction sequence you measure, the less variability you'll normally get from one execution to the next. Over time, the perturbations caused by the DRAM refresh cycle-eater tend to average out, since DRAM refresh occurs on a regular basis. Similarly, a lengthy code sequence that accesses display memory multiple times will tend to suffer a fairly consistent loss of performance to the display adapter cycle-eater. By contrast, a short code sequence that accesses display memory just once may vary greatly in performance from one run to the next, depending on how many wait states occur on the one access during a given run.

In short, you should time either long code sequences or repeated executions of shorter code sequences. While there's no strict definition of "long" in this context, the effects of the DRAM refresh and display adapter cycle-eaters should largely even out in sequences longer than about 100 us. While you can certainly use the Zen timer to measure shorter intervals, you should take multiple readings in such cases to make sure that the variations the cycle-eaters can cause from one run to the next aren't skewing your readings.

ODDS AND ENDS

There are a few more interesting observations to be made about Figure 5-1. For one thing, we can clearly see that while bus accesses are sometimes farther apart than 4 cycles, they are never any closer together. This confirms our earlier observation that bus cycles take a minimum of 4 cycles.

On the other hand, instruction bytes can be transferred from the prefetch queue to the Execution Unit at a rate of 1 byte per cycle when the EU needs them that quickly. This reinforces the notion that the EU can use up instruction bytes faster than the BIU can fetch them. In fact, we can see the EU waiting for an instruction byte fetch from cycle 130 to cycle 135, as discussed earlier. It's worth noting that after the instruction byte transfer to the EU at cycle 135, the next two instruction byte transfers occur at cycles 139 and 143, each occurring 4 cycles after the previous transfer. That mimics the 4 cycles separating the fetches of those instruction bytes, and that's no coincidence. During these cycles the EU does nothing but wait for the BIU to fetch instruction bytes--the most graphic demonstration yet of the prefetch queue cycle-eater.

The prefetch queue cycle-eater can be observed in another way in Figure 5-1. A careful reading of Figure 5-1 will make it apparent that the prefetch queue never contains more than 2 bytes at any time. In other words, the prefetch queue not only never fills, it never gets more than 2 bytes ahead of the Execution Unit. Moreover, we can see at cycles 33 and 34 that the EU can empty those 2 bytes from the prefetch queue in just 2 cycles. There's no doubt but what the BIU often fights a losing battle in trying to keep the EU supplied with instruction bytes.

BACK TO THE PROGRAMMING INTERFACE

It's not important that you grasp everything in this chapter, so long as you understand that the factors affecting the performance of an instruction in a given context are complex and vary with time. These complex and varied factors make it virtually impossible to know beforehand at what speed code will actually run. They also make it both impractical and pointless to understand exactly--down to the cycles--why a particular instruction or code sequence performs as well or poorly as it does.

As a result, high-performance assembler programming must be an intuitive art, rather than a cut-and-dried cookbook process. That's why this book is called The Zen of Assembly Language, not The Assembly Language Programming Reference Guide. That's also why you must time your code if you want to know how fast it is.

Cycle-eaters underlie the programming interface, the topic we'll tackle next. Together, cycle-eaters and the programming interface constitute the knowledge aspect of the Zen of assembler. Ultimately, the concept of the flexible mind rests on knowledge, and algorithms and implementation rest on the flexible mind. In short, cycle-eaters are the foundation of the Zen of assembler, and as such they will pop up frequently in the following chapters in a variety of contexts. The constant application of our understanding of the various cycle-eaters to working code should clear up any uncertainties you may still have about the cycle-eaters.

Next, we'll head up out of the land of the cycle-eaters to the programming interface, the far more familiar domain of registers, instructions, memory addressing, DOS calls and the like. After our journey to the land of the cycle-eaters, however, don't be surprised if the programming interface looks a little different. Assembler code never looks quite the same to a programmer who understands the true nature of performance.