Chapter 4

In the Lair of the Cycle-Eaters

Chapter 4

How the PC Hardware Devours Code Performance

This chapter, adapted from my earlier book, *Zen of Assembly Language* located on the companion CD-ROM, goes right to the heart of my philosophy of optimization: Understand where the time really goes when your code runs. That may sound ridiculously simple, but, as this chapter makes clear, it turns out to be a challenging task indeed, one that at times verges on black magic. This chapter is a long-time favorite of mine because it was the first—and to a large extent only—work that I know of that discussed this material, thereby introducing a generation of PC programmers to pedal-to-the-metal optimization.

This chapter focuses almost entirely on the first popular x86-family processor, the 8088. Some of the specific features and results that I cite in this chapter are no longer applicable to modern x86-family processors such as the 486 and Pentium, as I'll point out later on when we discuss those processors. Nonetheless, the overall theme of this chapter—that understanding dimly-seen and poorly-documented code gremlins called cycle-eaters that lurk in your system is essential to performance programming—is every bit as valid today. Also, later chapters often refer back to the basic cycle-eaters described in this chapter, so this chapter is the foundation for the discussions of x86-family optimization to come. What's more, the Zen timer remains an excellent tool with which to flush out and examine cycle-eaters, as we'll see in later chapters, and this chapter is as good an illustration of how to use the Zen timer as you're likely to find.

So, don't take either the absolute or the relative execution times presented in this chapter as gospel for newer processors, and read on to later chapters to see how the

cycle-eaters and optimization rules have changed over time, but do take the time to at least skim through this chapter to give yourself a good start on the material in the rest of this book.

Cycle-Eaters

Programming has many levels, ranging from the familiar (high-level languages, DOS calls, and the like) down to the esoteric things that lie on the shadowy edge of hardware-land. I call these *cycle-eaters* because, like the monsters in a bad 50s horror movie, they lurk in those shadows, taking their share of your program's performance without regard to the forces of goodness or the U.S. Army. In this chapter, we're going to jump right in at the lowest level by examining the cycle-eaters that live beneath the programming interface; that is, beneath your application, DOS, and BIOS—in fact, beneath the instruction set itself.

Why start at the lowest level? Simply because cycle-eaters affect the performance of all assembler code, and yet are almost unknown to most programmers. A full understanding of code optimization requires an understanding of cycle-eaters and their implications. That's no simple task, and in fact it is in precisely that area that most books and articles about assembly programming fall short.

Nearly all literature on assembly programming discusses only the programming interface: the instruction set, the registers, the flags, and the BIOS and DOS calls. Those topics cover the functionality of assembly programs most thoroughly—but it's performance above all else that we're after. No one ever tells you about the raw stuff of performance, which lies *beneath* the programming interface, in the dimly-seen realm—populated by instruction prefetching, dynamic RAM refresh, and wait states—where software meets hardware. This area is the domain of hardware engineers, and is almost never discussed as it relates to code performance. And yet it is only by understanding the mechanisms operating at this level that we can fully understand and properly improve the performance of our code.

Which brings us to cycle-eaters.

The Nature of Cycle-Eaters

Cycle-eaters are gremlins that live on the bus or in peripherals (and sometimes within the CPU itself), slowing the performance of PC code so that it doesn't execute at full speed. Most cycle-eaters (and all of those haunting the older Intel processors) live outside the CPU's Execution Unit, where they can *only* affect the CPU when the CPU performs a bus access (a memory or I/O read or write). Once your code and data are already inside the CPU, those cycle-eaters can no longer be a problem. Only on the 486 and Pentium CPUs will you find cycle-eaters inside the chip, as we'll see in later chapters.

The nature and severity of the cycle-eaters vary enormously from processor to processor, and (especially) from memory architecture to memory architecture. In order to understand them all, we need first to understand the simplest among them, those that haunted the original 8088-based IBM PC. Later on in this book, I'll be better able to explain the newer generation of cycle-eaters in terms of those ancestral cycle-eaters—but we have to get the groundwork down first.

The 8088's Ancestral Cycle-Eaters

Internally, the 8088 is a 16-bit processor, capable of running at full speed at all times—unless external data is required. External data must traverse the 8088's external data bus and the PC's data bus one byte at a time to and from peripherals, with cycle-eaters lurking along every step of the way. What's more, external data includes not only memory operands *but also instruction bytes*, so even instructions with no memory operands can suffer from cycle-eaters. Since some of the 8088's fastest instructions are register-only instructions, that's important indeed.

The major cycle-eaters are:

- The 8088's 8-bit external data bus.
- The prefetch queue.
- Dynamic RAM refresh.
- Wait states, notably display memory wait states and, in the AT and 80386 computers, system memory wait states.

The locations of these cycle-eaters in the primordial 8088-based PC are shown in Figure 4.1. We'll cover each of the cycle-eaters in turn in this chapter. The material won't be easy since cycle-eaters are among the most subtle aspects of assembly programming. By the same token, however, this will be one of the most important and rewarding chapters in this book. Don't worry if you don't catch everything in this chapter, but do read it all even if the going gets a bit tough. Cycle-eaters play a key role in later chapters, so some familiarity with them is highly desirable.

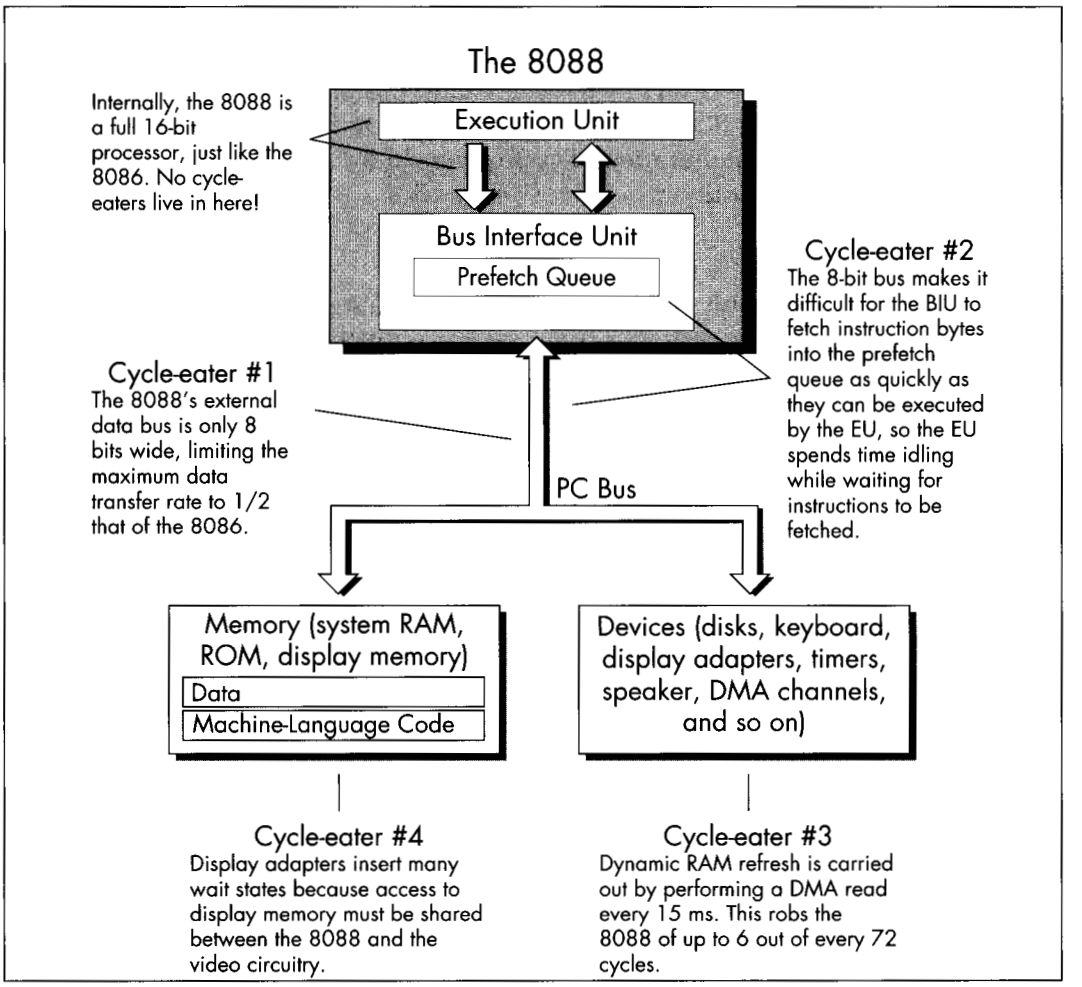
The 8-Bit Bus Cycle-Eater

Look! Down on the motherboard! It's a 16-bit processor! It's an 8-bit processor! It's...

...an 8088!

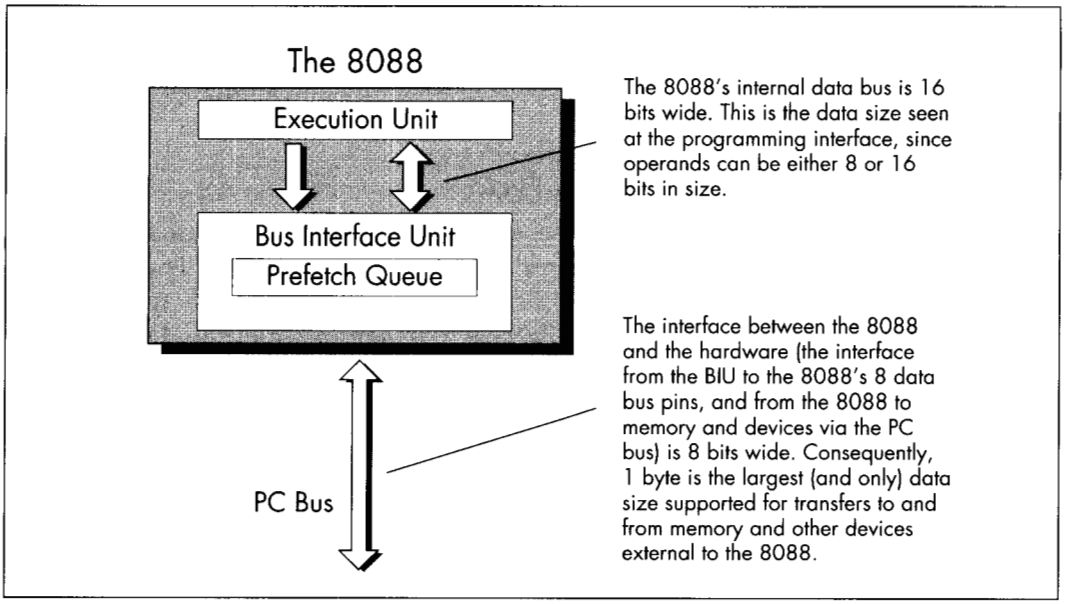
Fans of the 8088 call it a 16-bit processor. Fans of other 16-bit processors call the 8088 an 8-bit processor. The truth of the matter is that the 8088 is a 16-bit processor that often performs like an 8-bit processor.

The 8088 is internally a full 16-bit processor, equivalent to an 8086. (In fact, the 8086 is identical to the 8088, except that it has a full 16-bit bus. The 8088 is basically the poor man's 8086, because it allows a cheaper—albeit slower—system to be built, thanks to the half-sized bus.) In terms of the instruction set, the 8088 is clearly a 16-bit



The location of the major cycle-eaters in the IBM PC.
Figure 4.1

processor, capable of performing any given 16-bit operation—addition, subtraction, even multiplication or division—with a single instruction. Externally, however, the 8088 is unequivocally an 8-bit processor, since the external data bus is only 8 bits wide. In other words, the programming interface is 16 bits wide, but the hardware interface is only 8 bits wide, as shown in Figure 4.2. The result of this mismatch is simple: Word-sized data can be transferred between the 8088 and memory or peripherals at only one-half the maximum rate of the 8086, which is to say one-half the maximum rate for which the Execution Unit of the 8088 was designed.



Internal data bus widths of the 8088.
Figure 4.2

As shown in Figure 4.1, the 8-bit bus cycle-eater lies squarely on the 8088's external data bus. Technically, it might be more accurate to place this cycle-eater in the Bus Interface Unit, which breaks 16-bit memory accesses into paired 8-bit accesses, but it is really the limited width of the external data bus that constricts data flow into and out of the 8088. True, the original PC's bus is also only 8 bits wide, but that's just to match the 8088's 8-bit bus; even if the PC's bus were 16 bits wide, data could still pass into and out of the 8088 chip itself only 1 byte at a time.

Each bus access by the 8088 takes 4 clock cycles, or 0.838 µs in the 4.77 MHz PC, and transfers 1 byte. That means that the maximum rate at which data can be transferred into and out of the 8088 is 1 byte every 0.838 µs. While 8086 bus accesses also take 4 clock cycles, each 8086 bus access can transfer either 1 byte or 1 word, for a maximum transfer rate of 1 word every 0.838 µs. Consequently, for word-sized memory accesses, the 8086 has an effective transfer rate of 1 byte every 0.419 µs. By contrast, every word-sized access on the 8088 requires two 4-cycle-long bus accesses, one for the high byte of the word and one for the low byte of the word. As a result, the 8088 has an effective transfer rate for word-sized memory accesses of just 1 word every 1.676 µs—and that, in a nutshell, is the 8-bit bus cycle-eater.

A related cycle-eater lurks beneath the 386SX chip, which is a 32-bit processor internally with only a 16-bit path to system memory. The numbers are different, but the

way the cycle-eater operates is exactly the same. AT-compatible systems have 16-bit data buses, which can access a full 16-bit word at a time. The 386SX can process 32 bits (a doubleword) at a time, however, and loses a lot of time fetching that doubleword from memory in two halves.

The Impact of the 8-Bit Bus Cycle-Eater

One obvious effect of the 8-bit bus cycle-eater is that word-sized accesses to memory operands on the 8088 take 4 cycles longer than byte-sized accesses. That's why the official instruction timings indicate that for code running on an 8088 an additional 4 cycles are required for every word-sized access to a memory operand. For instance,

```
mov   ax,word ptr [MemVar]
```

takes 4 cycles longer to read the word at address **MemVar** than

```
mov   al,byte ptr [MemVar]
```

takes to read the byte at address **MemVar**. (Actually, the difference between the two isn't very likely to be exactly 4 cycles, for reasons that will become clear once we discuss the prefetch queue and dynamic RAM refresh cycle-eaters later in this chapter.)

What's more, in some cases one instruction can perform multiple word-sized accesses, incurring that 4-cycle penalty on each access. For example, adding a value to a word-sized memory variable requires two word-sized accesses—one to read the destination operand from memory prior to adding to it, and one to write the result of the addition back to the destination operand—and thus incurs not one but two 4-cycle penalties. As a result

```
add   word ptr [MemVar],ax
```

takes about 8 cycles longer to execute than:

```
add   byte ptr [MemVar],al
```

String instructions can suffer from the 8-bit bus cycle-eater to a greater extent than other instructions. Believe it or not, a single **REP MOVSW** instruction can lose as much as 131,070 word-sized memory accesses x 4 cycles, or *524,280 cycles* to the 8-bit bus cycle-eater! In other words, one 8088 instruction (admittedly, an instruction that does a great deal) can take over one-tenth of a second longer on an 8088 than on an 8086, simply because of the 8-bit bus. *One-tenth of a second!* That's a phenomenally long time in computer terms; in one-tenth of a second, the 8088 can perform more than 50,000 additions and subtractions.

The upshot of all this is simply that the 8088 can transfer word-sized data to and from memory at only half the speed of the 8086, which inevitably causes performance problems when coupled with an Execution Unit that can process word-sized data

every bit as quickly as an 8086. These problems show up with any code that uses word-sized memory operands. More ominously, as we will see shortly, the 8-bit bus cycle-eater can cause performance problems with other sorts of code as well.

What to Do about the 8-Bit Bus Cycle-Eater?

The obvious implication of the 8-bit bus cycle-eater is that byte-sized memory variables should be used whenever possible. After all, the 8088 performs *byte-sized* memory accesses just as quickly as the 8086. For instance, Listing 4.1, which uses a byte-sized memory variable as a loop counter, runs in 10.03 µs per loop. That's 20 percent faster than the 12.05 µs per loop execution time of Listing 4.2, which uses a word-sized counter. Why the difference in execution times? Simply because each word-sized **DEC** performs 4 byte-sized memory accesses (two to read the word-sized operand and two to write the result back to memory), while each byte-sized **DEC** performs only 2 byte-sized memory accesses in all.

LISTING 4.1 LST4-1.ASM

```
; Measures the performance of a loop which uses a
; byte-sized memory variable as the loop counter.
;
      jmp   Skip
;
Counter     db    100
;
Skip:
      call  ZTimerOn
LoopTop:
      dec   [Counter]
      jnz   LoopTop
      call  ZTimerOff
```

LISTING 4.2 LST4-2.ASM

```
; Measures the performance of a loop which uses a
; word-sized memory variable as the loop counter.
;
      jmp   Skip
;
Counter     dw    100
;
Skip:
      call  ZTimerOn
LoopTop:
      dec   [Counter]
      jnz   LoopTop
      call  ZTimerOff
```

I'd like to make a brief aside concerning code optimization in the listings in this book. Throughout this book I've modeled the sample code after working code so that the timing results are applicable to real-world programming. In Listings 4.1 and 4.2, for example, I could have shown a still greater advantage for byte-sized operands simply by performing 1,000 **DEC** instructions in a row, with no branching at all. However, **DEC**

instructions don't exist in a vacuum, so in the listings I used code that both decremented the counter and tested the result. The difference is that between decrementing a memory location (simply an instruction) and using a loop counter (a functional instruction sequence). If you come across code in this book that seems less than optimal, it's simply due to my desire to provide code that's relevant to real programming problems. On the other hand, optimal code is an elusive thing indeed; by no means should you assume that the code in this book is ideal! Examine it, question it, and improve upon it, for an inquisitive, skeptical mind is an important part of the Zen of assembly optimization.

Back to the 8-bit bus cycle-eater. As I've said, in 8088 work you should strive to use byte-sized memory variables whenever possible. That does *not* mean that you should use 2 byte-sized memory accesses to manipulate a word-sized memory variable in preference to 1 word-sized memory access, as, for instance,

```
mov   dl,byte ptr [MemVar]
mov   dh,byte ptr [MemVar+1]
```

versus:

```
mov   dx,word ptr [MemVar]
```

Recall that every access to a memory byte takes at least 4 cycles; that limitation is built right into the 8088. The 8088 is also built so that the second byte-sized memory access to a 16-bit memory variable takes just those 4 cycles and no more. There's no way you can manipulate the second byte of a word-sized memory variable faster with a second separate byte-sized instruction in less than 4 cycles. As a matter of fact, you're bound to access that second byte much more slowly with a separate instruction, thanks to the overhead of instruction fetching and execution, address calculation, and the like.

For example, consider Listing 4.3, which performs 1,000 word-sized reads from memory. This code runs in 3.77 µs per word read on a 4.77 MHz 8088. That's 45 percent faster than the 5.49 µs per word read of Listing 4.4, which reads the same 1,000 words as Listing 4.3 but does so with 2,000 byte-sized reads. Both listings perform exactly the same number of memory accesses—2,000 accesses, each byte-sized, as all 8088 memory accesses must be. (Remember that the Bus Interface Unit must perform two byte-sized memory accesses in order to handle a word-sized memory operand.) However, Listing 4.3 is considerably faster because it expends only 4 additional cycles to read the second byte of each word, while Listing 4.4 performs a second **LODSB**, requiring 13 cycles, to read the second byte of each word.

LISTING 4.3 LST4-3.ASM

```
; Measures the performance of reading 1,000 words
; from memory with 1,000 word-sized accesses.
;
     sub   si,si
```

```
        mov     cx,1000
        call    ZTimerOn
        rep     lodsw
        call    ZTimerOff
```

LISTING 4.4 LST4-4.ASM

```
; Measures the performance of reading 1000 words
; from memory with 2,000 byte-sized accesses.
;
        sub     si,si
        mov     cx,2000
        call    ZTimerOn
        rep     lodsb
        call    ZTimerOff
```

In short, if you must perform a 16-bit memory access, let the 8088 break the access into two byte-sized accesses for you. The 8088 is more efficient at that task than your code can possibly be.

Word-sized variables should be stored in registers to the greatest feasible extent, since registers are inside the 8088, where 16-bit operations are just as fast as 8-bit operations because the 8-bit cycle-eater can't get at them. In fact, it's a good idea to keep as many variables of all sorts in registers as you can. Instructions with register-only operands execute very rapidly, partially because they avoid both the time-consuming memory accesses and the lengthy address calculations associated with memory operands.

There is yet another reason why register operands are preferable to memory operands, and it's an unexpected effect of the 8-bit bus cycle-eater. Instructions with only register operands tend to be shorter (in terms of bytes) than instructions with memory operands, and when it comes to performance, shorter is usually better. In order to explain why that is true and how it relates to the 8-bit bus cycle-eater, I must diverge for a moment.

For the last few pages, you may well have been thinking that the 8-bit bus cycle-eater, while a nuisance, doesn't seem particularly subtle or difficult to quantify. After all, any instruction reference tells us exactly how many cycles each instruction loses to the 8-bit bus cycle-eater, doesn't it?

Yes and no. It's true that in general we know approximately how much longer a given instruction will take to execute with a word-sized memory operand than with a byte-sized operand, although the dynamic RAM refresh and wait state cycle-eaters (which I'll cover a little later) can raise the cost of the 8-bit bus cycle-eater considerably. However, *all* word-sized memory accesses lose 4 cycles to the 8-bit bus cycle-eater, and there's one sort of word-sized memory access we haven't discussed yet: instruction fetching. The ugliest manifestation of the 8-bit bus cycle-eater is in fact the prefetch queue cycle-eater.

The Prefetch Queue Cycle-Eater

In an 8088 context, here's the prefetch queue cycle-eater in a nutshell: The 8088's 8-bit external data bus keeps the Bus Interface Unit from fetching instruction bytes as fast as the 16-bit Execution Unit can execute them, so the Execution Unit often lies idle while waiting for the next instruction byte to be fetched.

Exactly why does this happen? Recall that the 8088 is an 8086 internally, but accesses word-sized memory data at only one-half the maximum rate of the 8086 due to the 8088's 8-bit external data bus. Unfortunately, instructions are among the word-sized data the 8086 fetches, meaning that the 8088 can fetch instructions at only one-half the speed of the 8086. On the other hand, the 8086-equivalent Execution Unit of the 8088 can *execute* instructions every bit as fast as the 8086. The net result is that the Execution Unit burns up instruction bytes much faster than the Bus Interface Unit can fetch them, and ends up idling while waiting for instructions bytes to arrive.

The BIU can fetch instruction bytes at a maximum rate of one byte every 4 cycles—*and that 4-cycle per instruction byte rate is the ultimate limit on overall instruction execution time, regardless of EU speed.* While the EU may execute a given instruction that's already in the prefetch queue in less than 4 cycles per byte, over time the EU can't execute instructions any faster than they can arrive—and they can't arrive faster than 1 byte every 4 cycles.

Clearly, then, the prefetch queue cycle-eater is nothing more than one aspect of the 8-bit bus cycle-eater. 8088 code often runs at less than the Execution Unit's maximum speed because the 8-bit data bus can't keep up with the demand for instruction bytes. That's straightforward enough—so why all the fuss about the prefetch queue cycle-eater?

What makes the prefetch queue cycle-eater tricky is that it's undocumented and unpredictable. That is, with a word-sized memory access, such as

```
mov   [bx],ax
```

it's well-documented that an extra 4 cycles will always be required to write the upper byte of AX to memory. Not so with the prefetch queue cycle-eater lurking nearby. For instance, the instructions

```
shr   ax,1
shr   ax,1
shr   ax,1
shr   ax,1
shr   ax,1
```

should execute in 10 cycles, since each **SHR** takes 2 cycles to execute, according to Intel's specifications. Those specifications contain Intel's official instruction execution times, but in this case—and in many others—the specifications are drastically wrong. Why? Because they describe execution time *once an instruction reaches the prefetch*

queue. They say nothing about whether a given instruction will be in the prefetch queue when it's time for that instruction to run, or how long it will take that instruction to reach the prefetch queue if it's not there already. Thanks to the low performance of the 8088's external data bus, that's a glaring omission—but, alas, an unavoidable one. Let's look at why the official execution times are wrong, and why that can't be helped.

Official Execution Times Are Only Part of the Story

The sequence of 5 **SHR** instructions in the last example is 10 bytes long. That means that it can never execute in less than 24 cycles even if the 4-byte prefetch queue is full when it starts, since 6 instruction bytes would still remain to be fetched, at 4 cycles per fetch. If the prefetch queue is empty at the start, the sequence *could* take 40 cycles. In short, thanks to instruction fetching, the code won't run at its documented speed, and could take up to four times longer than it is supposed to.

Why does Intel document Execution Unit execution time rather than overall instruction execution time, which includes both instruction fetch time and Execution Unit (EU) execution time? Well, instruction fetching isn't performed as part of instruction execution by the Execution Unit, but instead is carried on in parallel by the Bus Interface Unit (BIU) whenever the external data bus isn't in use or whenever the EU runs out of instruction bytes to execute. Sometimes the BIU is able to use spare bus cycles to prefetch instruction bytes before the EU needs them, so in those cases instruction fetching takes no time at all, practically speaking. At other times the EU executes instructions faster than the BIU can fetch them, and instruction fetching then becomes a significant part of overall execution time. As a result, *the effective fetch time for a given instruction varies greatly depending on the code mix preceding that instruction.* Similarly, the state in which a given instruction leaves the prefetch queue affects the overall execution time of the following instructions.

In other words, while the execution time for a given instruction is constant, the fetch time for that instruction depends heavily on the context in which the instruction is executing—the amount of prefetching the preceding instructions allowed—and can vary from a full 4 cycles per instruction byte to no time at all.

As we'll see later, other cycle-eaters, such as DRAM refresh and display memory wait states, can cause prefetching variations even during different executions of the *same* code sequence. Given that, it's meaningless to talk about the prefetch time of a given instruction except in the context of a specific code sequence.

So now you know why the official instruction execution times are often wrong, and why Intel can't provide better specifications. You also know now why it is that you must time your code if you want to know how fast it really is.

There Is No Such Beast as a True Instruction Execution Time

The effect of the code preceding an instruction on the execution time of that instruction makes the Zen timer trickier to use than you might expect, and complicates the interpretation of the results reported by the Zen timer. For one thing, the Zen timer is best used to time code sequences that are more than a few instructions long; below 10μs or so, prefetch queue effects and the limited resolution of the clock driving the timer can cause problems.

Some slight prefetch queue-induced inaccuracy usually exists even when the Zen timer is used to time longer code sequences, since the calls to the Zen timer usually alter the code's prefetch queue from its normal state. (Branches—jumps, calls, returns and the like—empty the prefetch queue.) Ideally, the Zen timer is used to measure the performance of an entire subroutine, so the prefetch queue effects of the branches at the start and end of the subroutine are similar to the effects of the calls to the Zen timer when you're measuring the subroutine's performance.

Another way in which the prefetch queue cycle-eater complicates the use of the Zen timer involves the practice of timing the performance of a few instructions over and over. I'll often repeat one or two instructions 100 or 1,000 times in a row in listings in this book in order to get timing intervals that are long enough to provide reliable measurements. However, as we just learned, the actual performance of any 8088 instruction depends on the code mix preceding any given use of that instruction, which in turn affects the state of the prefetch queue when the instruction starts executing. Alas, the execution time of an instruction preceded by dozens of identical instructions reflects just one of many possible prefetch states (and not a very likely state at that), and some of the other prefetch states may well produce distinctly different results.

For example, consider the code in Listings 4.5 and 4.6. Listing 4.5 shows our familiar **SHR** case. Here, because the prefetch queue is always empty, execution time should work out to about 4 cycles per byte, or 8 cycles per **SHR**, as shown in Figure 4.3. (Figure 4.3 illustrates the relationship between instruction fetching and execution in a simplified way, and is not intended to show the exact timings of 8088 operations.) That's quite a contrast to the official 2-cycle execution time of **SHR**. In fact, the Zen timer reports that Listing 4.5 executes in 1.81μs per byte, or slightly *more* than 4 cycles per byte. (The extra time is the result of the dynamic RAM refresh cycle-eater, which we'll discuss shortly.) Going by Listing 4.5, we would conclude that the "true" execution time of **SHR** is 8.64 cycles.

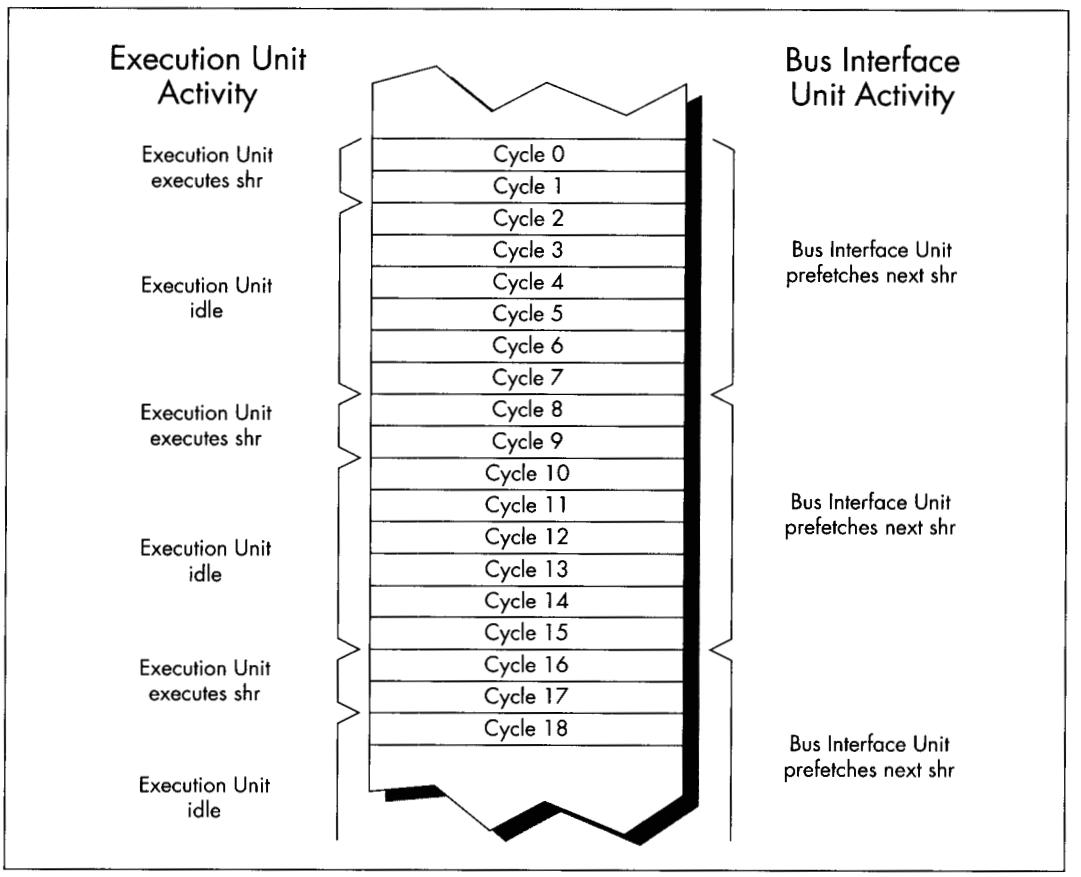
LISTING 4.5 LST4-5.ASM

```
; Measures the performance of 1,000 SHR instructions
; in a row. Since SHR executes in 2 cycles but is
; 2 bytes long, the prefetch queue is always empty,
; and prefetching time determines the overall
; performance of the code.
;
     call  ZTimerOn
     rept  1000
```

```
        shr   ax,1
        endm
        call  ZTimerOff
```

LISTING 4.6 LST4-6.ASM

```
; Measures the performance of 1,000 MUL/SHR instruction
; pairs in a row. The lengthy execution time of MUL
; should keep the prefetch queue from ever emptying.
;
        mov   cx,1000
        sub   ax,ax
        call  ZTimerOn
        rept  1000
        mul   ax
        shr   ax,1
        endm
        call  ZTimerOff
```



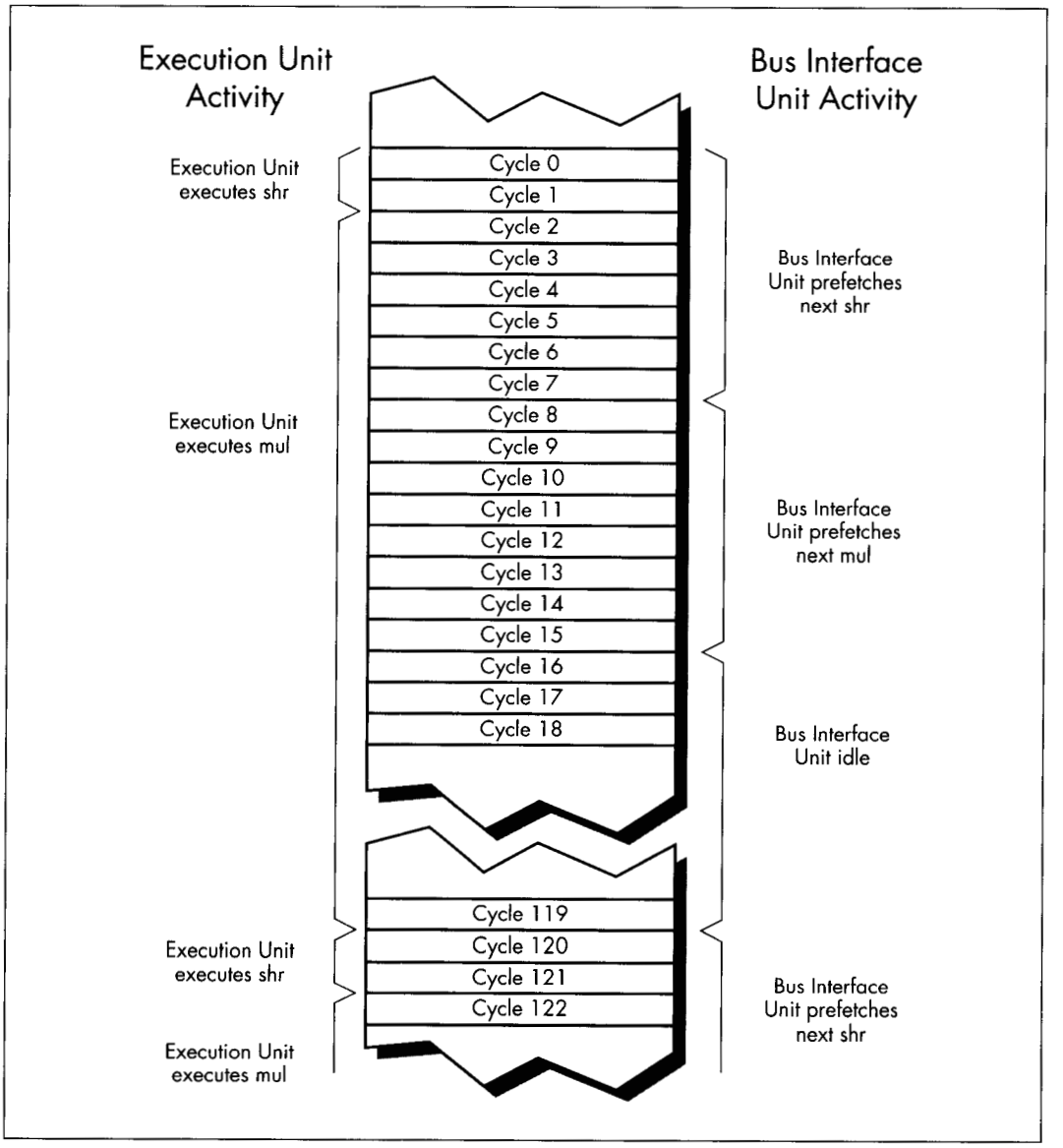
Execution and instruction prefetching sequence for Listing 4.5.
Figure 4.3

Now let's examine Listing 4.6. Here each **SHR** follows a **MUL** instruction. Since **MUL** instructions take so long to execute that the prefetch queue is always full when they finish, each **SHR** should be ready and waiting in the prefetch queue when the preceding **MUL** ends. As a result, we'd expect that each **SHR** would execute in 2 cycles; together with the 118-cycle execution time of multiplying 0 times 0, the total execution time should come to 120 cycles per **SHR/MUL** pair, as shown in Figure 4.4. And, by God, when we run Listing 4.6 we get an execution time of 25.14 µs per **SHR/MUL** pair, or *exactly* 120 cycles! According to these results, the "true" execution time of **SHR** would seem to be 2 cycles, quite a change from the conclusion we drew from Listing 4.5.

The key point is this: We've seen one code sequence in which **SHR** took 8-plus cycles to execute, and another in which it took only 2 cycles. Are we talking about two different forms of **SHR** here? Of course not—the difference is purely a reflection of the differing states in which the preceding code left the prefetch queue. In Listing 4.5, each **SHR** after the first few follows a slew of other **SHR** instructions which have sucked the prefetch queue dry, so overall performance reflects instruction fetch time. By contrast, each **SHR** in Listing 4.6 follows a **MUL** instruction which leaves the prefetch queue full, so overall performance reflects Execution Unit execution time.

Clearly, either instruction fetch time *or* Execution Unit execution time—or even a mix of the two, if an instruction is partially prefetched—can determine code performance. Some people operate under a rule of thumb by which they assume that the execution time of each instruction is 4 cycles times the number of bytes in the instruction. While that's often true for register-only code, it frequently doesn't hold for code that accesses memory. For one thing, the rule should be 4 cycles times the number of *memory accesses*, not instruction bytes, since all accesses take 4 cycles on the 8088-based PC. For another, memory-accessing instructions often have slower Execution Unit execution times than the 4 cycles per memory access rule would dictate, because the 8088 isn't very fast at calculating memory addresses. Also, the 4 cycles per instruction byte rule isn't true for register-only instructions that are already in the prefetch queue when the preceding instruction ends.

The truth is that it never hurts performance to reduce either the cycle count or the byte count of a given bit of code, but there's no guarantee that one or the other will improve performance either. For example, consider Listing 4.7, which consists of a series of 4-cycle, 2-byte **MOV AL,0** instructions, and which executes at the rate of 1.81 µs per instruction. Now consider Listing 4.8, which replaces the 4-cycle **MOV AL,0** with the 3-cycle (but still 2-byte) **SUB AL,AL**. Despite its 1-cycle-per-instruction advantage, Listing 4.8 runs at exactly the same speed as Listing 4.7. The reason: Both instructions are 2 bytes long, and in both cases it is the 8-cycle instruction fetch time, not the 3 or 4-cycle Execution Unit execution time, that limits performance.



Execution and instruction prefetching sequence for Listing 4.6.
Figure 4.4

LISTING 4.7 LST4-7.ASM

```
; Measures the performance of repeated MOV AL,0 instructions,
; which take 4 cycles each according to Intel's official
; specifications.
;
      sub   ax,ax
      call  ZTimerOn
      rept  1000
      mov   al,0
      endm
      call  ZTimerOff
```

LISTING 4.8 LST4-8.ASM

```
; Measures the performance of repeated SUB AL,AL instructions,
; which take 3 cycles each according to Intel's official
; specifications.
;
      sub   ax,ax
      call  ZTimerOn
      rept  1000
      sub   al,al
      endm
      call  ZTimerOff
```

As you can see, it's easy to be drawn into thinking you're saving cycles when you're not. You can only improve the performance of a specific bit of code by reducing the factor—either instruction fetch time or execution time, or sometimes a mix of the two—that's limiting the performance of that code.

In case you missed it in all the excitement, the variability of prefetching means that our method of testing performance by executing 1,000 instructions in a row by no means produces "true" instruction execution times, any more than the official execution times in the Intel manuals are "true" times. The fact of the matter is that a given instruction takes *at least* as long to execute as the time given for it in the Intel manuals, but may take as much as 4 cycles per byte longer, depending on the state of the prefetch queue when the preceding instruction ends.

The only true execution time for an instruction is a time measured in a certain context, and that time is meaningful only in that context.

What we *really* want is to know how long useful working code takes to run, not how long a single instruction takes, and the Zen timer gives us the tool we need to gather that information. Granted, it would be easier if we could just add up neatly documented instruction execution times—but that's not going to happen. Without actually measuring the performance of a given code sequence, you simply don't know how fast it is. For crying out loud, even the people who *designed* the 8088 at Intel couldn't tell you exactly how quickly a given 8088 code sequence executes on the PC just by looking at it! Get used to the idea that execution times are only meaningful in context, learn the rules of thumb in this book, and use the Zen timer to measure your code.

Approximating Overall Execution Times

Don't think that because overall instruction execution time is determined by both instruction fetch time and Execution Unit execution time, the two times should be added together when estimating performance. For example, practically speaking, each **SHR** in Listing 4.5 does not take 8 cycles of instruction fetch time plus 2 cycles of Execution Unit execution time to execute. Figure 4.3 shows that while a given **SHR** is executing, the fetch of the next **SHR** is starting, and since the two operations are overlapped for 2 cycles, there's no sense in charging the time to both instructions. You could think of the extra instruction fetch time for **SHR** in Listing 4.5 as being 6 cycles, which yields an overall execution time of 8 cycles when added to the 2 cycles of Execution Unit execution time.

Alternatively, you could think of each **SHR** in Listing 4.5 as taking 8 cycles to fetch, and then executing in effectively 0 cycles while the next **SHR** is being fetched. Whichever perspective you prefer is fine. The important point is that the time during which the execution of one instruction and the fetching of the next instruction overlap should only be counted toward the overall execution time of one of the instructions. For all intents and purposes, one of the two instructions runs at no performance cost whatsoever while the overlap exists.

As a working definition, we'll consider the execution time of a given instruction in a particular context to start when the first byte of the instruction is sent to the Execution Unit and end when the first byte of the next instruction is sent to the EU.

What to Do about the Prefetch Queue Cycle-Eater?

Reducing the impact of the prefetch queue cycle-eater is one of the overriding principles of high-performance assembly code. How can you do this? One effective technique is to minimize access to memory operands, since such accesses compete with instruction fetching for precious memory accesses. You can also greatly reduce instruction fetch time simply by your choice of instructions: *Keep your instructions short.* Less time is required to fetch instructions that are 1 or 2 bytes long than instructions that are 5 or 6 bytes long. Reduced instruction fetching lowers minimum execution time (minimum execution time is 4 cycles times the number of instruction bytes) and often leads to faster overall execution.

While short instructions minimize overall prefetch time, ironically they actually often suffer more from the prefetch queue bottleneck than do long instructions. Short instructions generally have such fast execution times that they drain the prefetch queue despite their small size. For example, consider the **SHR** of Listing 4.5, which runs at only 25 percent of its Execution Unit execution time even though it's only 2 bytes long, thanks to the prefetch queue bottleneck. Short instructions are nonetheless generally faster than long instructions, thanks to the combination of fewer instruction bytes and faster Execution Unit execution times, and should be used as much as possible—just don't expect them to run at their "official" documented speeds.

More than anything, the above rules mean using the registers as heavily as possible, both because register-only instructions are short and because they don't perform memory accesses to read or write operands. However, using the registers is a rule of thumb, not a commandment. In some circumstances, it may actually be *faster* to access memory. (The look-up table technique is one such case.) What's more, the performance of the prefetch queue (and hence the performance of each instruction) differs from one code sequence to the next, and can even differ during different executions of the *same* code sequence.

All in all, writing good assembler code is as much an art as a science. As a result, you should follow the rules of thumb described here—and then time your code to see how fast it really is. You should experiment freely, but always remember that actual, measured performance is the bottom line.

Holding Up the 8088

In this chapter I've taken you further and further into the depths of the PC, telling you again and again that you must understand the computer at the lowest possible level in order to write good code. At this point, you may well wonder, "Have we gotten low enough?"

Not quite yet. The 8-bit bus and prefetch queue cycle-eaters are low-level indeed, but we've one level yet to go. Dynamic RAM refresh and wait states—our next topics—together form the lowest level at which the hardware of the PC affects code performance. Below this level, the PC is of interest only to hardware engineers.

Before we begin our discussion of dynamic RAM refresh, let's step back for a moment to take an overall look at this lowest level of cycle-eaters. In truth, the distinctions between wait states and dynamic RAM refresh don't much matter to a programmer. What is important is that you understand this: *Under certain circumstances, devices on the PC bus can stop the CPU for 1 or more cycles, making your code run more slowly than it seemingly should.*

Unlike all the cycle-eaters we've encountered so far, wait states and dynamic RAM refresh are strictly external to the CPU, as was shown in Figure 4.1. Adapters on the PC's bus, such as video and memory cards, can insert wait states on any bus access, the idea being that they won't be able to complete the access properly unless the access is stretched out. Likewise, the channel of the DMA controller dedicated to dynamic RAM refresh can request control of the bus at any time, although the CPU must relinquish the bus before the DMA controller can take over. This means that your code can't directly control wait states or dynamic RAM refresh. However, code *can* sometimes be designed to minimize the effects of these cycle-eaters, and even when the cycle-eaters slow your code without there being a thing in the world you can do about it, you're still better off understanding that you're losing performance and knowing why your code doesn't run as fast as it's supposed to than you were programming in ignorance.

Let's start with DRAM refresh, which affects the performance of every program that runs on the PC.

Dynamic RAM Refresh: The Invisible Hand

Dynamic RAM (DRAM) refresh is sort of an act of God. By that I mean that DRAM refresh invisibly and inexorably steals a certain fraction of all available memory access time from your programs, when they are accessing memory for code and data. (When they are accessing cache on more recent processors, theoretically the DRAM refresh cycle-eater doesn't come into play, but there are other cycle-eaters waiting to prey on cache-bound programs.) While you *could* stop DRAM refresh, you wouldn't want to since that would be a sure prescription for crashing your computer. In the end, thanks to DRAM refresh, almost all code runs a bit slower on the PC than it otherwise would, and that's that.

A bit of background: A static RAM (SRAM) chip is a memory chip that retains its contents indefinitely so long as power is maintained. By contrast, each of several blocks of bits in a dynamic RAM (DRAM) chip retains its contents for only a short time after it's accessed for a read or write. In order to get a DRAM chip to store data for an extended period, each of the blocks of bits in that chip must be accessed regularly, so that the chip's stored data is kept refreshed and valid. So long as this is done often enough, a DRAM chip will retain its contents indefinitely.

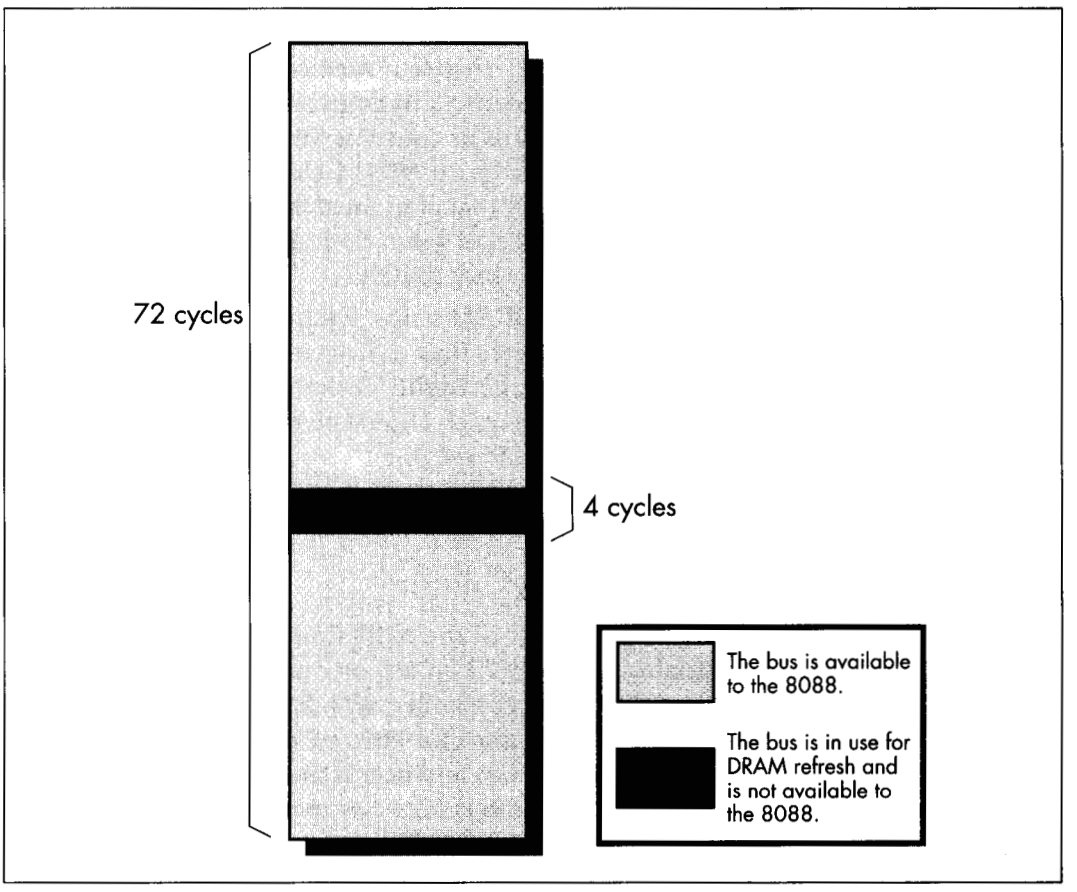
All of the PC's system memory consists of DRAM chips. Each DRAM chip in the PC must be completely refreshed about once every four milliseconds in order to ensure the integrity of the data it stores. Obviously, it's highly desirable that the memory in the PC retain the correct data indefinitely, so each DRAM chip in the PC *must* always be refreshed within 4 ms of the last refresh. Since there's no guarantee that a given program will access each and every DRAM block once every 4 ms, the PC contains special circuitry and programming for providing DRAM refresh.

How DRAM Refresh Works in the PC

On the original 8088-based IBM PC, timer 1 of the 8253 timer chip is programmed at power-up to generate a signal once every 72 cycles, or once every 15.08μs. That signal goes to channel 0 of the 8237 DMA controller, which requests the bus from the 8088 upon receiving the signal. (DMA stands for *direct memory access*, the ability of a device other than the 8088 to control the bus and access memory directly, without any help from the 8088.) As soon as the 8088 is between memory accesses, it gives control of the bus to the 8237, which in conjunction with special circuitry on the PC's motherboard then performs a single 4-cycle read access to 1 of 256 possible addresses, advancing to the next address on each successive access. (The read access is only for the purpose of refreshing the DRAM; the data that is read isn't used.)

The 256 addresses accessed by the refresh DMA accesses are arranged so that taken together they properly refresh all the memory in the PC. By accessing one of the 256 addresses every 15.08 µs, all of the PC's DRAM is refreshed in 256 x 15.08 µs, or 3.86 ms, which is just about the desired 4 ms time I mentioned earlier. (Only the first 640K of memory is refreshed in the PC; video adapters and other adapters above 640K containing memory that requires refreshing must provide their own DRAM refresh in pre-AT systems.)

Don't sweat the details here. The important point is this: For at least 4 out of every 72 cycles, the original PC's bus is given over to DRAM refresh and is not available to the 8088, as shown in Figure 4.5. That means that as much as 5.56 percent of the PC's already inadequate bus capacity is lost. However, DRAM refresh doesn't necessarily



The PC bus dynamic RAM (DRAM) refresh.
Figure 4.5

stop the 8088 in its tracks for 4 cycles. The Execution Unit of the 8088 can keep processing while DRAM refresh is occurring, unless the EU needs to access memory. Consequently, DRAM refresh can slow code performance anywhere from 0 percent to 5.56 percent (and actually a bit more, as we'll see shortly), depending on the extent to which DRAM refresh occupies cycles during which the 8088 would otherwise be accessing memory.

The Impact of DRAM Refresh

Let's look at examples from opposite ends of the spectrum in terms of the impact of DRAM refresh on code performance. First, consider the series of **MUL** instructions in Listing 4.9. Since a 16-bit **MUL** on the 8088 executes in between 118 and 133 cycles and is only 2 bytes long, there should be plenty of time for the prefetch queue to fill after each instruction, even after DRAM refresh has taken its slice of memory access time. Consequently, the prefetch queue should be able to keep the Execution Unit well-supplied with instruction bytes at all times. Since Listing 4.9 uses no memory operands, the Execution Unit should never have to wait for data from memory, and DRAM refresh should have no impact on performance. (Remember that the Execution Unit can operate normally during DRAM refreshes so long as it doesn't need to request a memory access from the Bus Interface Unit.)

LISTING 4.9 LST4-9.ASM

```
; Measures the performance of repeated MUL instructions,
; which allow the prefetch queue to be full at all times,
; to demonstrate a case in which DRAM refresh has no impact
; on code performance.
;
      sub   ax,ax
      call  ZTimerOn
      rept  1000
      mul   ax
      endm
      call  ZTimerOff
```

Running Listing 4.9, we find that each **MUL** executes in 24.72 µs, or exactly 118 cycles. Since that's the shortest time in which **MUL** can execute, we can see that no performance is lost to DRAM refresh. Listing 4.9 clearly illustrates that DRAM refresh only affects code performance when a DRAM refresh forces the Execution Unit of the 8088 to wait for a memory access.

Now let's look at the series of **SHR** instructions shown in Listing 4.10. Since **SHR** executes in 2 cycles but is 2 bytes long, the prefetch queue should be empty while Listing 4.10 executes, with the 8088 prefetching instruction bytes non-stop. As a result, the time per instruction of Listing 4.10 should precisely reflect the time required to fetch the instruction bytes.

LISTING 4.10 LST4-10.ASM

```
; Measures the performance of repeated SHR instructions,
; which empty the prefetch queue, to demonstrate the
; worst-case impact of DRAM refresh on code performance.
;
      call  ZTimerOn
      rept  1000
      shr   ax,1
      endm
      call  ZTimerOff
```

Since 4 cycles are required to read each instruction byte, we'd expect each **SHR** to execute in 8 cycles, or 1.676 µs, if there were no DRAM refresh. In fact, each **SHR** in Listing 4.10 executes in 1.81 µs, indicating that DRAM refresh is taking 7.4 percent of the program's execution time. That's nearly 2 percent more than our worst-case estimate of the loss to DRAM refresh overhead! In fact, the result indicates that DRAM refresh is stealing not 4, but 5.33 cycles out of every 72 cycles. How can this be?

The answer is that a given DRAM refresh can actually hold up CPU memory accesses for as many as 6 cycles, depending on the timing of the DRAM refresh's DMA request relative to the 8088's internal instruction execution state. When the code in Listing 4.10 runs, each DRAM refresh holds up the CPU for either 5 or 6 cycles, depending on where the 8088 is in executing the current **SHR** instruction when the refresh request occurs. Now we see that things can get even worse than we thought: *DRAM refresh can steal as much as 8.33 percent of available memory access time—6 out of every 72 cycles—from the 8088.*

Which of the two cases we've examined reflects reality? While either case *can* happen, the latter case—significant performance reduction, ranging as high as 8.33 percent—is far more likely to occur. This is especially true for high-performance assembly code, which uses fast instructions that tend to cause non-stop instruction fetching.

What to Do About the DRAM Refresh Cycle-Eater?

Hmmm. When we discovered the prefetch queue cycle-eater, we learned to use short instructions. When we discovered the 8-bit bus cycle-eater, we learned to use byte-sized memory operands whenever possible, and to keep word-sized variables in registers. What can we do to work around the DRAM refresh cycle-eater?

Nothing.

As I've said before, DRAM refresh is an act of God. DRAM refresh is a fundamental, unchanging part of the PC's operation, and there's nothing you or I can do about it. If refresh were any less frequent, the reliability of the PC would be compromised, so tinkering with either timer 1 or DMA channel 0 to reduce DRAM refresh overhead is out. Nor is there any way to structure code to minimize the impact of DRAM refresh. Sure, some instructions are affected less by DRAM refresh than others, but how many multiplies and divides in a row can you really use? I suppose that code *could* conceivably be structured to leave a free memory access every 72 cycles, so DRAM refresh

wouldn't have any effect. In the old days when code size was measured in bytes, not K bytes, and processors were less powerful—and complex—programmers did in fact use similar tricks to eke every last bit of performance from their code. When programming the PC, however, the prefetch queue cycle-eater would make such careful code synchronization a difficult task indeed, and any modest performance improvement that did result could never justify the increase in programming complexity and the limits on creative programming that such an approach would entail. Besides, all that effort goes to waste on faster 8088s, 286s, and other computers with different execution speeds and refresh characteristics. There's no way around it: Useful code accesses memory frequently and at irregular intervals, and over the long haul DRAM refresh always exacts its price.

If you're still harboring thoughts of reducing the overhead of DRAM refresh, consider this. Instructions that tend not to suffer very much from DRAM refresh are those that have a high ratio of execution time to instruction fetch time, and those aren't the fastest instructions of the PC. It certainly wouldn't make sense to use slower instructions just to reduce DRAM refresh overhead, for it's *total* execution time—DRAM refresh, instruction fetching, and all—that matters.

The important thing to understand about DRAM refresh is that it generally slows your code down, and that the extent of that performance reduction can vary considerably and unpredictably, depending on how the DRAM refreshes interact with your code's pattern of memory accesses. When you use the Zen timer and get a fractional cycle count for the execution time of an instruction, that's often the DRAM refresh cycle-eater at work. (The display adapter cycle-eater is another possible culprit, and, on 386s and later processors, cache misses and pipeline execution hazards produce this sort of effect as well.) Whenever you get two timing results that differ less or more than they seemingly should, that's usually DRAM refresh too. Thanks to DRAM refresh, variations of up to 8.33 percent in PC code performance are par for the course.

Wait States

Wait states are cycles during which a bus access by the CPU to a device on the PC's bus is temporarily halted by that device while the device gets ready to complete the read or write. Wait states are well and truly the lowest level of code performance. Everything we have discussed (and will discuss)—even DMA accesses—can be affected by wait states.

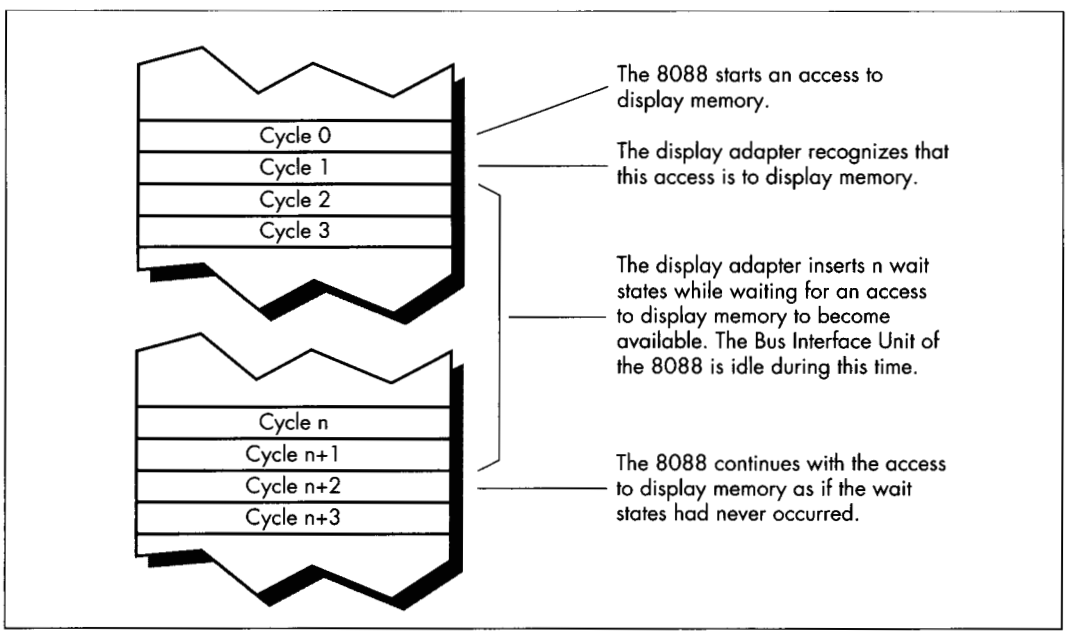
Wait states exist because the CPU must to be able to coexist with any adapter, no matter how slow (within reason). The 8088 expects to be able to complete each bus access—a memory or I/O read or write—in 4 cycles, but adapters can't always respond that quickly for a number of reasons. For example, display adapters must split access to display memory between the CPU and the circuitry that generates the video signal based on the contents of display memory, so they often can't immediately fulfill a request by the CPU for a display memory read or write. To resolve this conflict, display

adapters can tell the CPU to wait during bus accesses by inserting one or more wait states, as shown in Figure 4.6. The CPU simply sits and idles as long as wait states are inserted, then completes the access as soon as the display adapter indicates its readiness by no longer inserting wait states. The same would be true of any adapter that couldn't keep up with the CPU.

Mind you, this is all transparent to executing code. An instruction that encounters wait states runs exactly as if there were no wait states, only slower. Wait states are nothing more or less than wasted time as far as the CPU and your program are concerned.

By understanding the circumstances in which wait states can occur, you can avoid them when possible. Even when it's not possible to work around wait states, it's still to your advantage to understand how they can cause your code to run more slowly.

First, let's learn a bit more about wait states by contrast with DRAM refresh. Unlike DRAM refresh, wait states do not occur on any regularly scheduled basis, and are of no particular duration. Wait states can only occur when an instruction performs a memory or I/O read or write. Both the presence of wait states and the number of wait states inserted on any given bus access are entirely controlled by the device being accessed. When it comes to wait states, the CPU is passive, merely accepting whatever wait states the accessed device chooses to insert during the course of the access. All of this makes perfect sense given that the whole point of the wait state



Video wait states inserted by the display adapter.
Figure 4.6

mechanism is to allow a device to stretch out any access to itself for however much time it needs to perform the access.

As with DRAM refresh, wait states don't stop the 8088 completely. The Execution Unit can continue processing while wait states are inserted, so long as the EU doesn't need to perform a bus access. However, in the PC, wait states most often occur when an instruction accesses a memory operand, so in fact the Execution Unit usually is stopped by wait states. (Instruction fetches rarely wait in an 8088-based PC because system memory is zero-wait-state. AT-class memory systems routinely insert 1 or more wait states, however.)

As it turns out, wait states pose a serious problem in just one area in the PC. While any adapter *can* insert wait states, in the PC only display adapters do so to the extent that performance is seriously affected.

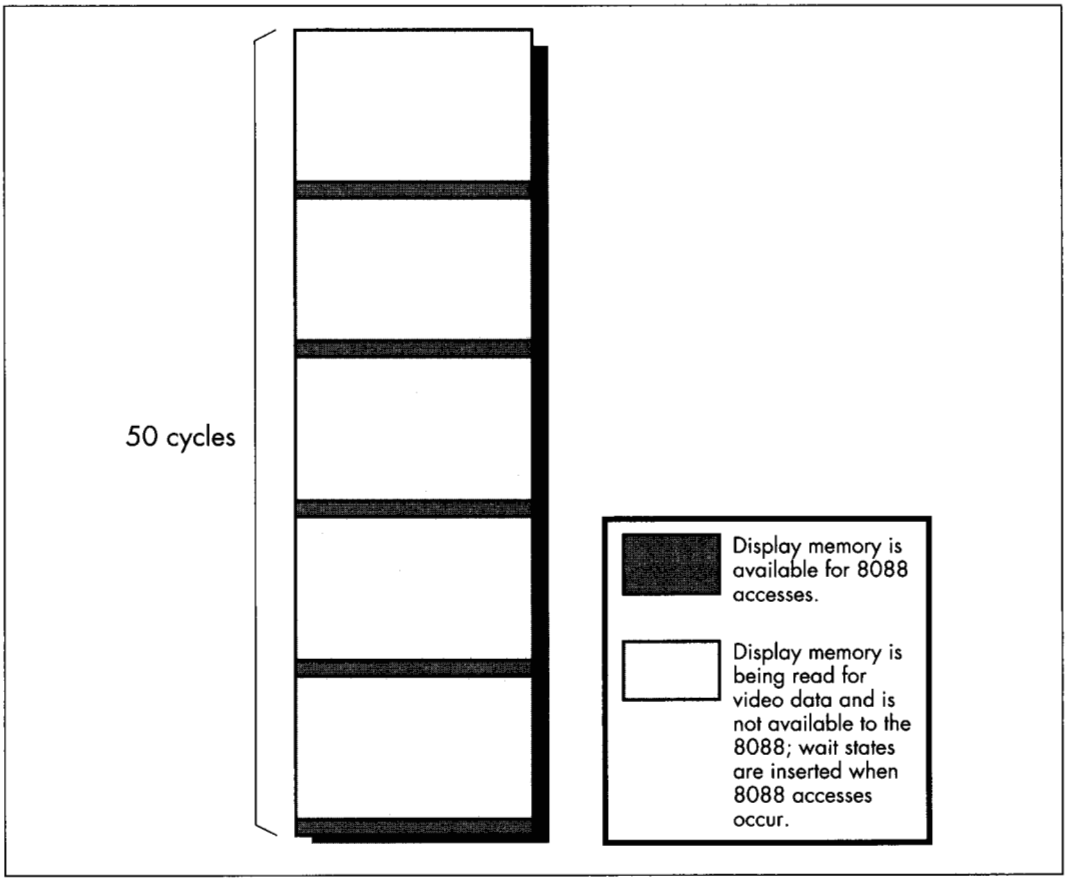
The Display Adapter Cycle-Eater

Display adapters must serve two masters, and that creates a fundamental performance problem. Master #1 is the circuitry that drives the display screen. This circuitry must constantly read display memory in order to obtain the information used to draw the characters or dots displayed on the screen. Since the screen must be redrawn between 50 and 70 times per second, and since each redraw of the screen can require as many as 36,000 reads of display memory (more in Super VGA modes), master #1 is a demanding master indeed. No matter how demanding master #1 gets, however, its needs must *always* be met—otherwise the quality of the picture on the screen would suffer.

Master #2 is the CPU, which reads from and writes to display memory in order to manipulate the bytes that the video circuitry reads to form the picture on the screen. Master #2 is less important than master #1, since the CPU affects display quality only indirectly. In other words, if the video circuitry has to wait for display memory accesses, the picture will develop holes, snow, and the like, but if the CPU has to wait for display memory accesses, the program will just run a bit slower—no big deal.

It matters a great deal which master is more important, for while both the CPU and the video circuitry must gain access to display memory, only one of the two masters can read or write display memory at any one time. Potential conflicts are resolved by flat-out guaranteeing the video circuitry however many accesses to display memory it needs, with the CPU waiting for whatever display memory accesses are left over.

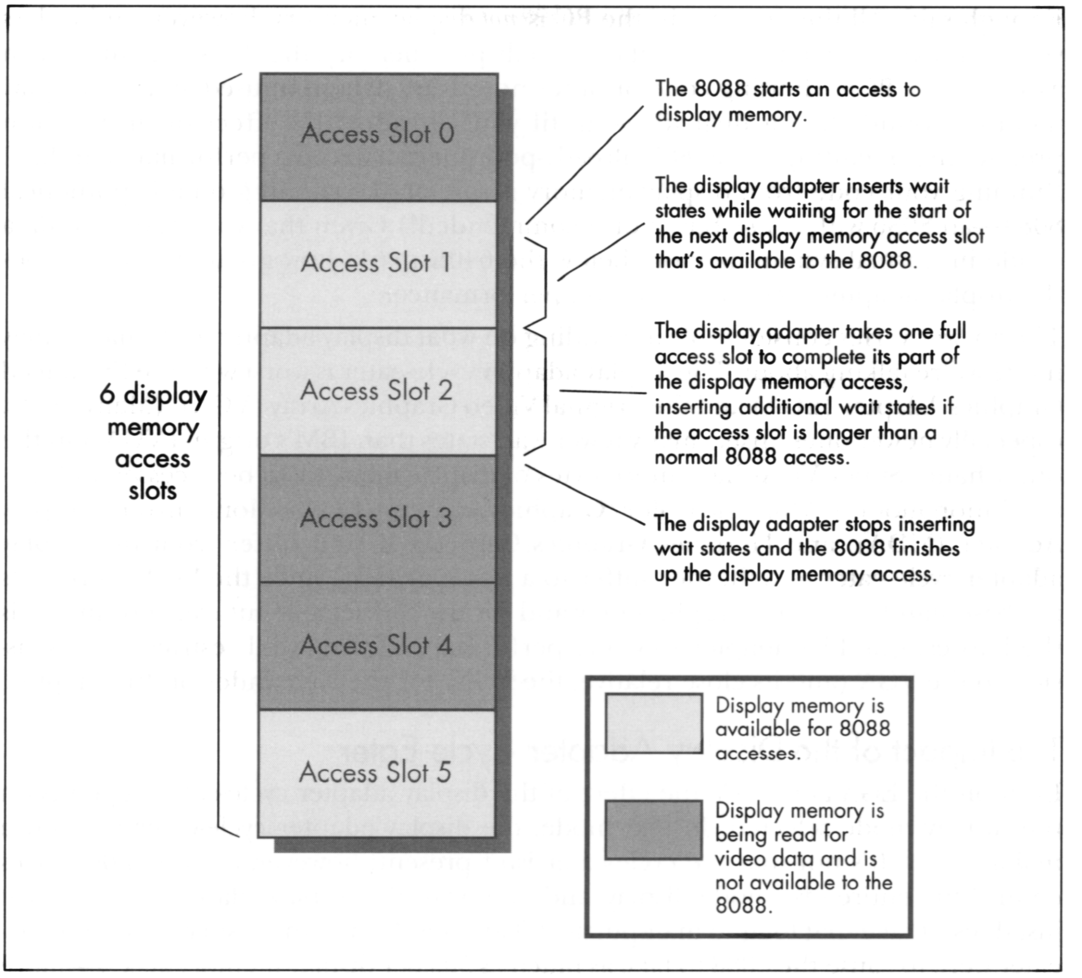
It turns out that the 8088 CPU has to do a lot of waiting, for three reasons. First, the video circuitry can take as much as about 90 percent of the available display memory access time, as shown in Figure 4.7, leaving as little as about 10 percent of all display memory accesses for the 8088. (These percentages vary considerably among the many EGA and VGA clones.)



Allocation of display memory access.
Figure 4.7

Second, because the displayed dots (or *pixels*, short for "picture elements") must be drawn on the screen at a constant speed, many display adapters provide memory accesses only at fixed intervals. As a result, time can be lost while the 8088 synchronizes with the start of the next display adapter memory access, even if the video circuitry isn't accessing display memory at that time, as shown in Figure 4.8.

Finally, the time it takes a display adapter to complete a memory access is related to the speed of the clock which generates pixels on the screen rather than to the memory access speed of the 8088. Consequently, the time taken for display memory to complete an 8088 read or write access is often longer than the time taken for system memory to complete an access, even if the 8088 lucks into hitting a free display memory access just as it becomes available, again as shown in Figure 4.8. Any or all of



Display memory access slots.
Figure 4.8

the three factors I've described can result in wait states, slowing the 8088 and creating the display adapter cycle-eater.

If some of this is Greek to you, don't worry. The important point is that display memory is not very fast compared to normal system memory. How slow is it? *Incredibly* slow. Remember how slow IBM's ill-fated PC*jr* was? In case you've forgotten, I'll refresh your memory: The PC*jr* was at best only half as fast as the PC. The PC*jr* had an 8088 running at 4.77 MHz, just like the PC—why do you suppose it was so much slower? I'll tell you why: *All the memory in the PCjr was display memory.*

Enough said. All the memory in the PC is *not* display memory, however, and unless you're thickheaded enough to put code in display memory, the PC isn't going to run as slowly as a PC*jr*. (Putting code or other non-video data in unused areas of display memory sounds like a neat idea—until you consider the effect on instruction prefetching of cutting the 8088's already-poor memory access performance in half. Running your code from display memory is sort of like running on a hypothetical 8084—an 8086 with a *4-bit* bus. Not recommended!) Given that your code and data reside in normal system memory below the 640K mark, how great an impact does the display adapter cycle-eater have on performance?

The answer varies considerably depending on what display adapter and what display mode we're talking about. The display adapter cycle-eater is worst with the Enhanced Graphics Adapter (EGA) and the original Video Graphics Array (VGA). (Many VGAs, especially newer ones, insert many fewer wait states than IBM's original VGA. On the other hand, Super VGAs have more bytes of display memory to be accessed in high-resolution mode.) While the Color/Graphics Adapter (CGA), Monochrome Display Adapter (MDA), and Hercules Graphics Card (HGC) all suffer from the display adapter cycle-eater as well, they suffer to a lesser degree. Since the VGA represents the base standard for PC graphics now and for the foreseeable future, and since it is the hardest graphics adapter to wring performance from, we'll restrict our discussion to the VGA (and its close relative, the EGA) for the remainder of this chapter.

The Impact of the Display Adapter Cycle-Eater

Even on the EGA and VGA, the effect of the display adapter cycle-eater depends on the display mode selected. In text mode, the display adapter cycle-eater is rarely a major factor. It's not that the cycle-eater isn't present; however, a mere 4,000 bytes control the entire text mode display, and even with the display adapter cycle-eater it just doesn't take that long to manipulate 4,000 bytes. Even if the display adapter cycle-eater were to cause the 8088 to take as much as 5µs per display memory access—more than five times normal—it would still take only 4,000× 2× 5µs, or 40 ms, to read and write every byte of display memory. That's a lot of time as measured in 8088 cycles, but it's less than the blink of an eye in human time, and video performance only matters in human time. After all, the whole point of drawing graphics is to convey visual information, and if that information can be presented faster than the eye can see, that is by definition fast enough.

That's not to say that the display adapter cycle-eater *can't* matter in text mode. In Chapter 3, I recounted the story of a debate among letter-writers to a magazine about exactly how quickly characters could be written to display memory without causing snow. The writers carefully added up Intel's instruction cycle times to see how many writes to display memory they could squeeze into a single horizontal retrace interval. (On a CGA, it's only during the short horizontal retrace interval and the longer vertical retrace interval that display memory can be accessed in 80-column text mode without causing snow.) Of

course, now we know that their cardinal sin was to ignore the prefetch queue; even if there were no wait states, their calculations would have been overly optimistic. There *are* display memory wait states as well, however, so the calculations were not just optimistic but wildly optimistic.

Text mode situations such as the above notwithstanding, where the display adapter cycle-eater really kicks in is in graphics mode, and most especially in the high-resolution graphics modes of the EGA and VGA. The problem here is not that there are necessarily more wait states per access in high-resolution graphics modes (that varies from adapter to adapter and mode to mode). Rather, the problem is simply that are many more bytes of display memory per screen in these modes than in lower-resolution graphics modes and in text modes, so many more display memory accesses—each incurring its share of display memory wait states—are required in order to draw an image of a given size. When accessing the many thousands of bytes used in the high-resolution graphics modes, the cumulative effects of display memory wait states can seriously impact code performance, even as measured in human time.

For example, if we assume the same 5 µs per display memory access for the EGA's high-resolution graphics mode that we assumed for text mode, it would take 26,000 × 2 × 5 µs, or 260 ms, to scroll the screen once in the EGA's high-resolution graphics mode, mode 10H. That's more than one-quarter of a second—noticeable by human standards, an eternity by computer standards.

That sounds pretty serious, but we did make an unfounded assumption about memory access speed. Let's get some hard numbers. Listing 4.11 accesses display memory at the 8088's maximum speed, by way of a **REP MOVSW** with display memory as both source and destination. The code in Listing 4.11 executes in 3.18 µs per access to display memory—not as long as we had assumed, but a long time nonetheless.

LISTING 4.11 LST4-11.ASM

```
; Times speed of memory access to Enhanced Graphics
; Adapter graphics mode display memory at A000:0000.
;
      mov   ax,0010h
      int   10h         ;select hi-res EGA graphics
                        ; mode 10 hex (AH=0 selects
                        ; BIOS set mode function,
                        ; with AL=mode to select)
;
      mov   ax,0a000h
      mov   ds,ax
      mov   es,ax       ;move to & from same segment
      sub   si,si       ;move to & from same offset
      mov   di,si
      mov   cx,800h     ;move 2K words
      cld
      call  ZTimerOn
      rep   movsw       ;simply read each of the first
                        ; 2K words of the destination segment,
                        ; writing each byte immediately back
```

```
                    ; to the same address. No memory
                    ; locations are actually altered; this
                    ; is just to measure memory access
                    ; times
        call  ZTimerOff
;
        mov   ax,0003h
        int   10h           ;return to text mode
```

For comparison, let's see how long the same code takes when accessing normal system RAM instead of display memory. The code in Listing 4.12, which performs a **REP MOVSW** from the code segment to the code segment, executes in 1.39 µs per display memory access. That means that on average, 1.79 µs (more than 8 cycles!) are lost to the display adapter cycle-eater on each access. In other words, the display adapter cycle-eater can *more than double* the execution time of 8088 code!

LISTING 4.12 LST4-12.ASM

```
; Times speed of memory access to normal system
; memory.
;
        mov   ax,ds
        mov   es,ax         ;move to & from same segment
        sub   si,si         ;move to & from same offset
        mov   di,si
        mov   cx,800h       ;move 2K words
        cld
        call  ZTimerOn
        rep   movsw         ;simply read each of the first
                            ; 2K words of the destination segment,
                            ; writing each byte immediately back
                            ; to the same address. No memory
                            ; locations are actually altered; this
                            ; is just to measure memory access
                            ; times
        call  ZTimerOff
```

Bear in mind that we're talking about a worst case here; the impact of the display adapter cycle-eater is proportional to the percent of time a given code sequence spends accessing display memory.

*A line-drawing subroutine, which executes perhaps a dozen instructions for each display memory access, generally loses less performance to the display adapter cycle-eater than does a block-copy or scrolling subroutine that uses **REP MOVS** instructions. Scaled and three-dimensional graphics, which spend a great deal of time performing calculations (often using very slow floating-point arithmetic), tend to suffer less.*

In addition, code that accesses display memory infrequently tends to suffer only about half of the maximum display memory wait states, because on average such code will access display memory halfway between one available display memory access slot and

the next. As a result, code that accesses display memory less intensively than the code in Listing 4.11 will on average lose 4 or 5 rather than 8-plus cycles to the display adapter cycle-eater on each memory access.

Nonetheless, the display adapter cycle-eater always takes its toll on graphics code. Interestingly, that toll becomes much higher on ATs and 80386 machines because while those computers can execute many more instructions per microsecond than can the 8088-based PC, it takes just as long to access display memory on those computers as on the 8088-based PC. Remember, the limited speed of access to a graphics adapter is an inherent characteristic of the adapter, so the fastest computer around can't access display memory one iota faster than the adapter will allow.

What to Do about the Display Adapter Cycle-Eater?

What can we do about the display adapter cycle-eater? Well, we can minimize display memory accesses whenever possible. In particular, we can try to avoid read/modify/write display memory operations of the sort used to mask individual pixels and clip images. Why? Because read/modify/write operations require two display memory accesses (one read and one write) each time display memory is manipulated. Instead, we should try to use writes of the sort that set all the pixels in a given byte of display memory at once, since such writes don't require accompanying read accesses. The key here is that only half as many display memory accesses are required to write a byte to display memory as are required to read a byte from display memory, mask part of it off and alter the rest, and write the byte back to display memory. Half as many display memory accesses means half as many display memory wait states.

Moreover, 486s and Pentiums, as well as recent Super VGAs, employ write-caching schemes that make display memory writes considerably faster than display memory reads.

Along the same line, the display adapter cycle-eater makes the popular exclusive-OR animation technique, which requires paired reads and writes of display memory, less-than-ideal for the PC. Exclusive-OR animation should be avoided in favor of simply writing images to display memory whenever possible.

Another principle for display adapter programming on the 8088 is to perform multiple accesses to display memory very rapidly, in order to make use of as many of the scarce accesses to display memory as possible. This is especially important when many large images need to be drawn quickly, since only by using virtually every available display memory access can many bytes be written to display memory in a short period of time. Repeated string instructions are ideal for making maximum use of display memory accesses; of course, repeated string instructions can only be used on whole bytes, so this is another point in favor of modifying display memory a byte at a time. (On faster processors, however, display memory is so slow that it often pays to do several

instructions worth of work between display memory accesses, to take advantage of cycles that would otherwise be wasted on the wait states.)

It would be handy to explore the display adapter cycle-eater issue in depth, with lots of example code and execution timings, but alas, I don't have the space for that right now. For the time being, all you really need to know about the display adapter cycle-eater is that on the 8088 you can lose more than 8 cycles of execution time on each access to display memory. For intensive access to display memory, the loss really can be as high as 8-plus cycles (and up to 50, 100, or even more on 486s and Pentiums paired with slow VGAs), while for average graphics code the loss is closer to 4 cycles; in either case, the impact on performance is significant. There is only one way to discover just how significant the impact of the display adapter cycle-eater is for any particular graphics code, and that is of course to measure the performance of that code.

Cycle-Eaters: A Summary

We've covered a great deal of sophisticated material in this chapter, so don't feel bad if you haven't understood everything you've read; it will all become clear from further reading, especially once you study, time, and tune code that you have written yourself. What's really important is that you come away from this chapter understanding that on the 8088:

- The 8-bit bus cycle-eater causes each access to a word-sized operand to be 4 cycles longer than an equivalent access to a byte-sized operand.
- The prefetch queue cycle-eater can cause instruction execution times to be as much as four times longer than the officially documented cycle times.
- The DRAM refresh cycle-eater slows most PC code, with performance reductions ranging as high as 8.33 percent.
- The display adapter cycle-eater typically doubles and can more than triple the length of the standard 4-cycle access to display memory, with intensive display memory access suffering most.

This basic knowledge about cycle-eaters puts you in a good position to understand the results reported by the Zen timer, and that means that you're well on your way to writing high-performance assembler code.

What Does It All Mean?

There you have it: life under the programming interface. It's not a particularly pretty picture for the inhabitants of that strange realm where hardware and software meet are little-known cycle-eaters that sap the speed from your unsuspecting code. Still, some of those cycle-eaters can be minimized by keeping instructions short, using the registers, using byte-sized memory operands, and accessing display memory as little as possible. None of the cycle-eaters can be eliminated, and dynamic RAM refresh can scarcely be addressed at all; still, aren't you better off knowing how fast your

code *really* runs—and why—than you were reading the official execution times and guessing? And while specific cycle-eaters vary in importance on later x86-family processors, with some cycle-eaters vanishing altogether and new ones appearing, the concept that understanding these obscure gremlins is a key to performance remains unchanged, as we'll see again and again in later chapters.