Chapter 3: Context

One of my favorite stories--and I am not making this up--concerns a C programmer who wrote a function to clear the screen. His function consisted of just two statements: a call to another function that printed a space character, and a **for** statement that repeated that function call 2000 times. While this fellow's function cleared the screen perfectly well, it didn't do it particularly quickly or attractively; in fact, the whole process was perfectly visible to the naked eye, with the cursor racing from the top to the bottom of the screen. Nonetheless, the programmer was incensed when someone commented that the function seemed rather slow. How could it possibly be any faster, he wondered, when it was already the irreducible minimum of two statements long?

Of course, the function wasn't two statements long in any meaningful sense; its true length would have to be measured in terms of all the machine-language instructions generated by those two C statements, as well as all the instructions executed by the function that printed the space character. By comparison with a single **rep stosw** instruction, which is the preferred way to clear the screen, this fellow's screen clear function was undoubtedly very long indeed.

The programmer's mistake was one of context. While his solution seemed optimal by the standards of the C environment he was programming in, it was considerably less ideal when applied to the PC, the environment in which the code actually had to run. While human-oriented abstractions such as high-level languages and system software have their virtues--most notably the ability to mask the complexities of processors and hardware--speed is not necessarily among those virtues.

We certainly don't want to make the same mistake, so we'll begin our search for knowledge by establishing a context for assembler programming, a usable framework within which to work for the remainder of this book. This is more challenging than it might at first glance seem, for the PC looks quite different to an assembler programmer--especially an assembler programmer interested in performance--than it does to a high-level language programmer. The difference is that a good assembler programmer sees the PC as it really is--hardware, software, warts and all--a perspective all too few programmers ever have the opportunity to enjoy.

FROM THE BOTTOM UP

In this volume, we're going to explore the knowledge needed for top-notch assembler programming. We'll start at the bottom, with the hardware of the PC, and we'll work our way up through the 8088's registers, memory addressing capabilities, and instruction set. In Volume II of The Zen of Assembly Language, we'll move on to putting that knowledge to work in the context of higher-level optimization, algorithm implementation, program design, and the like. We're not going to spend time on topics, such as BIOS and DOS calls, that are well documented elsewhere, for we've a great deal of new ground to cover.

The next three chapters, which discuss the ways in which the hardware of the PC affects performance, are the foundation for everything that follows...and they also cover the most difficult material in The Zen of Assembly Language. Don't worry if you don't understand everything you read in the upcoming chapters; the same topics will come up again and again, from a variety of perspectives, throughout The Zen of Assembly Language. Read through Chapters 3 through 5 once now, absorbing as much as you can. After you've finished Volume I, come back to these chapters and read them again.

You'll be amazed at how much sense they make--and at how much you've learned.

Let's begin our explorations.

THE TRADITIONAL MODEL

Figure 3-1 shows the traditional assembler programming model of the PC. In this model, the assembler program is separated from the hardware by layers of system software, such as DOS, the BIOS, and device drivers. Although this model recognizes that it is possible for assembler programs to make end runs around the layers to access any level of system software or the hardware directly, programs are supposed to request services from the highest level that can fulfill a given request (preferably DOS), thereby gaining hardware independence, which brings with it portability to other systems with different hardware but the same system software.

This model has many admirable qualities, and should be followed whenever possible. For example, because the DOS file system masks incompatibilities between the dozens of disk and disk controller models on the market, there's generally nothing to be gained and much to be lost by programming a disk controller directly. Similarly, the BIOS sometimes hides differences between makes of keyboards, so keystrokes should not be taken directly from the hardware unless that's absolutely necessary. Every assembler programmer should be thoroughly aware of the services provided by DOS and the BIOS, and should use them whenever they're good enough for a given purpose.

A moment's thought will show, however, that it's not always desirable to follow the model of Figure 3-1. Disk-backup software programs the disk controller directly and sells handsomely, while keyboard macro programs and many pop-up programs read the keyboard directly. Part of your job as a programmer is knowing when to break the rules embodied by Figure 3-1, and breaking the rules is tempting because this model has major failings when it comes to performance.

One shortcoming of the model of Figure 3-1 is that DOS and the BIOS provide inadequate services in some areas, and no services at all in others. For instance, the half-hearted support DOS and the BIOS provide for serial communications is an insult to the potential of the PC's communications hardware. Likewise, the graphics primitives offered by the BIOS are so slow and limited as to be virtually useless. While device drivers can extend DOS's capabilities in some areas, many of the drivers are themselves embarrassingly slow and limited. As an example, the ANSI.SYS driver, which provides extended screen control in text mode, is so sluggish that a single screen update can take a second and more--quite a contrast with the instant screen updates most text editors and word processors offer.

When you use a system service, you're accepting someone else's solution to a problem; while it may be a good solution, you don't know that unless you check. After all, you may well be a better programmer than the author of the system software, and you're bound to be better attuned to your particular needs than he was. In short, you should know the system services well and use them fully, but you should also learn when it pays to replace them with your own code.

CYCLE-EATERS

The second shortcoming of the model shown in Figure 3-1 is that it makes the hardware seem to be just another system resource, and a rather remote and uninteresting resource, at that. Nothing could be further from the truth! After all, in order to be useful programs must ultimately perform input from and output to the real world, and all input and output requires interaction with the hardware. True, DOS and the BIOS may handle much of your I/O, but DOS and the BIOS themselves are nothing more than assembly-language programs.

Also, programs access memory almost continuously, and memory is of course part of the PC's hardware. It's hard to write a code sequence of more than a few dozen instructions in which memory isn't accessed at least once as either a stack operand or as a direct instruction operand. I/O ports are also accessed heavily in some applications. Every single memory and I/O access of any kind must interact with the hardware via the PC's data bus.

It's easy to think of the PC's hardware and bus as being transparent to programs; hardware appears to be available on demand, while the bus seems to be nothing more than a path for data to take on the way to and from the hardware. Not so. While the PC bus is in fact generally transparent to programs, the many demands on the bus and the relatively low rate at which the bus, the 8088, and the PC's memory together can support data transfers can have a significant effect on performance, as we'll see shortly. Moreover, there are a number of memory and I/O devices for the PC that can't access data fast enough to keep up with the PC bus; to compensate, they make the 8088 wait, sometimes for several cycles, while they catch up. Inevitably, program performance suffers from these characteristics of the hardware and bus.

For the remainder of this book, I'm going to refer to PC bus- and hardware-resident gremlins that affect code performance as "cycle-eaters." There are cycle-eaters of many sorts, of which the prefetch queue and display adapter cycle-eaters are perhaps the best-known; 8-bit cards in ATs and dynamic RAM refresh are other examples. Cycle-eaters are undeniably difficult to pin down. Once you've identified and understood them, though, you'll be among the elite few who can deal with the most powerful--and least understood--aspect of assembler programming.

Just how important are cycle-eaters? Well, thanks to the display adapter cycle-eater, the code in Listing 3-1, which accesses memory on an Enhanced Graphics Adapter (EGA), runs in 26.06 ms. That's more than twice as long as the 11.24 ms of Listing 3-2, which is identical to Listing 3-1 except that it accesses normal system memory rather than display memory. That's a difference in performance as great as that between an 8-MHz AT and a 16-MHz 80386 machine! Clearly, cycle-eaters cannot be ignored, and in the chapters to come we'll spend considerable time tracking them down and devising ways to work around them.

Given cycle-eaters and our understanding of layered system software as simply another sort of code, the programming model shown in Figure 3-2 is more appropriate than that of Figure 3-1. All system and application software, whether generated from high- level or assembler source code, ultimately becomes a series of machine-language instructions for the 8088. The 8088 executes each of those instructions in turn, accessing memory and devices as needed by way of the PC bus. In this three-level structure, the 8088 provides software with a programming interface, and in turn rests on the PC's hardware. Thanks to cycle-eaters, the PC's hardware and bus emerge as important factors in performance.

The primary virtue of Figure 3-2 is that it moves us away from the comfortable, human-oriented perspective of Figure 3-1 and forces us to view program execution at a level closer to the true nature of the beast, as consisting of nothing more than the performance of a sequence of instructions that command the 8088 to perform actions; in some cases, those actions involve accessing memory and/or devices over the PC bus.

From the software side, we can now see that all code consists of machine-language instructions in the end, so the distinction between high-level languages, system software, and assembler vanishes. From the hardware side, we can see that the 8088 is not the lowest level, and we can begin to appreciate the many ways in which hardware can directly affect code performance.

We need to see still more of the beast, however, and the place we'll start is with the equivalence of code and data.

CODE IS DATA

Code is nothing more than data that the 8088 interprets as instructions. The most obvious case of this is self-modifying code, where the 8088 treats its code as data in order to modify it, then executes those same bytes as instructions. There are many other examples, though--after all, what is a compiler but a program that transforms source code data into machine-language data? Both code and data consist of byte values stored in system memory; the only thing that differentiates code from any other sort of data is that the bytes that code is made of have a special meaning to the 8088, in that when fetched as instructions they instruct the 8088 to perform a series of (presumably related) actions. In other words, the meaning of byte values as code rather than data is strictly a matter of context.

Why is this important? It's important because the 8088 is really two processors in one, and therein lies a tale.

INSIDE THE 8088

Internally, the 8088 consists of two complementary processors: the Bus Interface Unit (BIU) and the Execution Unit (EU), as shown in Figure 3-3.   The EU is what we normally think of as being a processor; it contains the flags, the general- purpose registers, and the Arithmetic Logic Unit (ALU), and executes instructions.   In fact, the EU performs just about every function you could want from a processor--except one.   The one thing the EU does not do is access memory or perform I/O.   That's the BIU's job, so whenever the EU needs a memory or I/O access performed, it sends a request to the BIU, which carries out the access, transferring the data according to the EU's specifications.   The two units are capable of operating in parallel whenever they've got independent tasks to perform; put another way, the BIU can access memory or I/O at the same time that the EU is processing an instruction, so long as neither task is dependent on the other.

Each BIU memory access transfers 1 byte, since the 8088 has an 8-bit external data bus.   The 8088 is designed so that each byte access takes a minimum of 4 cycles; given the PC's 4.77-MHz processor clock, which results in a 209.5 ns cycle time, the 8088 supports a maximum data transfer rate of 1 byte/838 ns, or about 1.2 bytes/us.   That's an important number, and we'll come back to it shortly.

The EU is capable of working with both 8- and 16-bit memory operands.   Because the 8088 can only access memory a byte at a time, however, the BIU splits each of the EU's 16-bit memory requests into a pair of 8-bit accesses. Since each 8-bit access requires a minimum of 4 cycles to execute, each 16-bit memory request takes at least 8 cycles, or 1.676 us. The instruction timings shown in Appendix A reflect the additional overhead of word memory accesses by indicating that 4 additional cycles per memory access should be added to the stated instruction execution times when word rather than byte memory operands are used.

The BIU contains all the memory-related logic of the 8088, including the segment registers and the Instruction Pointer, which points to the next instruction to be executed. Since code is just another sort of data, it makes sense that the Instruction Pointer resides in the BIU; after all, code bytes are read from memory just as data bytes are. In fact, the BIU takes on a bit of autonomy when it comes to fetching instructions. Whenever the EU isn't making any memory or I/O requests, the BIU uses the otherwise idle time to fetch the bytes at the addresses immediately following the current instruction, on the reasonable theory that those addresses are likely to contain the next instructions that the EU will want. The BIU of the 8088 can store up to 4 potential instruction bytes in an internal prefetch queue, and other 8086-family processors can store more bytes still.

Instruction prefetching isn't always advantageous. In particular, if the instruction the 8088 is currently executing results in a branch of any sort, the bytes in the instruction queue are of no value, since they are the bytes the 8088 would have executed had the branch not been performed. As a result, all the 8088 can do when a branch occurs is discard all the bytes in the prefetch queue and start fetching instructions all over again.

Nonetheless, the prefetching scheme often allows the BIU to have the next instruction byte waiting when the EU comes calling for it. Bear in mind that the EU and BIU can operate at the same time; it's only when the EU is waiting for the BIU to finish a memory or I/O operation for it that the EU is held up. The virtue of the 8088's internal architecture, then, is that the EU can increase its effective processing time because the BIU often coprocesses with it. Since instruction fetches occur in a constant stream--usually much more frequently than memory operand accesses--instruction prefetching is the most important sort of coprocessing the BIU performs.

It's worth noting at this point that the execution time specified by Intel for any given instruction running on the 8088 (as shown in Appendix A) assumes that the BIU has already prefetched that instruction and has it ready and waiting for the EU. <u>If the next instruction is not waiting for the EU when the EU completes the current instruction, at least some of the time required to fetch the next instruction must be added to its specified execution time in order to arrive at the actual execution time.</u>

The degree to which the EU and BIU can coprocess during instruction fetching is not uniform for all types of code; in fact, it varies considerably depending on the mix of instructions being executed. Multiplication and division instructions are ideal for coprocessing, since the BIU can prefetch until the queue is full while these very long instructions execute. Among other instructions, oddly enough, it is code that performs many memory accesses that allows the EU and BIU to coprocess most effectively, because the 8088 is relatively slow at executing instructions that access memory (as we'll see in Chapter 7). While a single memory-accessing instruction is being executed, the BIU can often prefetch 1 to 4 instruction bytes (depending on the instruction being performed) and still leave time for the memory access to occur. Execution of a memory-accessing instruction and prefetching of the next instruction can generally proceed simultaneously, so such instructions often run at close to full speed.

Ironically, code that primarily performs register-only operations and rarely accesses memory affords little opportunity for prefetching, because register-only instructions execute so rapidly that the BIU can't fetch instruction bytes nearly as rapidly as the EU can execute them. To see why this is so, recall that the 8088 can fetch 1 byte every 4 cycles, or 0.838 us. The **shr** instruction is 2 bytes long, so it takes 1.676 us to fetch each **shr** instruction. However, the EU can <u>execute</u> a **shr** in just 2 cycles, or 0.419 us, four times as rapidly as the BIU can fetch the same instruction.

The instruction queue can be depleted quickly by register- only instructions. <u>Given enough such instructions in a row, the overall time required to complete a series of register-only instructions is determined almost entirely by the time required to fetch the instructions from memory.</u> This is precisely the respect in which Figure 3-2 fails us; because of the prefetch queue, the instructions the 8088 executes must be viewed as data, stored along with other program data and accessed through the same PC bus and BIU, as shown in Figure 3-4. Seen in this light, it becomes apparent that instruction fetches are subject to the same cycle-eaters as are memory operand accesses. What's more, the BIU emerges as potentially the greatest cycle-eater of all, as code and data bytes struggle to get through the BIU fast enough to keep the EU busy, a phenomenon I'll refer to as the prefetch queue cycle-eater from now on. As we will see, designing code to work around the prefetch queue cycle-eater and keep the EU busy is a difficult but rewarding task.

STEPCHILD OF THE 8086

You might justifiably wonder why Intel would design a processor with an EU that can execute instructions faster than the BIU can possibly fetch them. The answer is that they didn't; they designed the 8086, then created the 8088 as a poor man's 8086.

The 8086 is completely software compatible with the 8088, and in fact differs from the 8088 in only one important respect, the width of the external data bus (the bus that goes off-chip to memory and peripherals); where the 8088 has an 8-bit wide external data bus, the 8086 has a 16-bit wide bus. (The 8086 also has a 6- rather than 4-byte prefetch queue, which gives it a bit of an advantage in keeping the EU busy.) Both the 8086 and 8088 have 16-bit EUs and 16-bit internal data buses, but while the 8086's BIU can fulfill most 16-bit memory requests with a single memory access, the 8088's BIU must convert 16-bit memory requests into 8-bit memory accesses. Figure 3-5, which charts internal and external data bus sizes for processors from the 8080 through the 80386, shows that the 8088 is something of an aberration in that it is the only widely-used processor in the 8086 family with mismatched internal and external data bus sizes. (The

80386SX, which may well become a successful low-cost substitute for the 80386, also has mismatched internal and external bus sizes, and as a result suffers from many of the same performance constraints as does the 8088.)

There is a significant price to be paid for the 8088's mismatched bus sizes. Why? Well, the 8086 was designed to support efficient and balanced memory access, with the external data bus in general in use as much as possible without that bus becoming a bottleneck. In other words, the 16-bit external data bus of the 8086 was designed to provide a memory access rate roughly equal to the processing rate of which the 16-bit EU is capable. While the 8088 offers the same internal 16-bit architecture as the 8086, the 8-bit external data bus of the 8088 can provide at best only half the memory access rate of the 8086, so the balance of the 8086 is lost.

The obvious effect of the 8088's mismatched bus sizes is that accesses to word-sized memory operands take 4 cycles longer on an 8088 than on an 8086, but that's actually not the most significant fallout of the 8-bit external data bus. More significant is the prefetch queue cycle-eater, which is the result of the inability of the 8088's BIU to fetch instructions and operands over the 8-bit external data bus as fast as the 16- bit EU can process them, thereby limiting the performance of the 8088's fastest instructions. By contrast, the 8086, for which the EU was originally designed, has little trouble keeping the EU supplied with instructions and data; the 8086's BIU fetches 2 instruction bytes in the same time it takes the 8088 to fetch a single byte, making the 8086 instruction fetching rate twice that of the 8088.

How significant is the performance impact of the 8088's 8- bit external data bus? While normal code is estimated to run only about one-third faster on an 8086 than on an 8088, high- performance 8086 code can--as we've already seen--run as much as four times more slowly on an 8088 once the prefetch queue empties, because code performance is limited by the rate at which the BIU can transfer data a byte at a time. In the case where both the 8088 and 8086 prefetch queues are emptied, the 8086 runs fast assembler code only twice as fast as the 8088, but the 8086 has a bigger prefetch queue than the 8088 and fetches instructions twice as fast, so the 8086 queue empties much more slowly--and in any case, twice as fast is nothing to sniff at.

In short, the 8086 is just like the 8088--except that it's somewhere between 0% and 300% faster, depending on what code happens to be executing, with a typical performance advantage of somewhere between 33% and 100% for high-performance assembler code.

Why then does the 8088 exist, and why has it become so popular? An 8-bit-bus version of the 8086 (that is, the 8088) was desirable in the late 1970s because at that time it was significantly more expensive to build a computer with a 16-bit data bus than with an 8-bit data bus. The 8088 allowed the construction of low-cost, low-performance computers that would run 8086 software, albeit more slowly. As it turned out, the cost advantage of an 8-bit memory data bus quickly became relatively insignificant, and the 8088 might have vanished into obscurity had IBM not selected it for the PC; then we might never have had the pleasure of wrestling with the prefetch queue cycle- eater. However, IBM did select the 8088 for the PC, and the rest is history.

Incidentally, an imbalance between processing speed and memory access speed remains a factor today with the 80286-based IBM AT and with many 80386-based computers. The memory in those computers often does not run at the speeds the processors are capable of, and assembler code encounters the same sorts of performance losses when running on those computers as it does on the 8088. We'll return to that topic in Chapter 15.

WHICH MODEL TO USE?

Each of the three programming models I've presented offers a useful perspective on assembler programming for the PC. However, it is the model shown in Figure 3-4 that best reflects the true nature of the 8088; consequently, that model is the most useful of the three for tapping the unique potential of assembler. While we'll use elements of all three models in The Zen of Assembly Language, we'll concentrate on the perspective of Figure 3-4 as we explore high-performance assembler programming.

Keep the following concepts in mind as you read on:

_ All code is machine language in the end: don't assume that anyone else's code, even system software, is best suited for your needs.

_ 1.2 bytes/us: at its best, the 8088's BIU can transfer data no faster than this.

_ The 8088 is not the lowest level: know how the PC's hardware and bus affect memory access speed.

_ Code is data: when the BIU and the PC's hardware and bus affect memory access speed, they affect code fetching as well as data access, since code is just another sort of data in system memory.

Short and simple as the above list may seem, in it you will find every one of the concepts that form the foundation of the Zen of assembler--and with them the key to high-performance code.