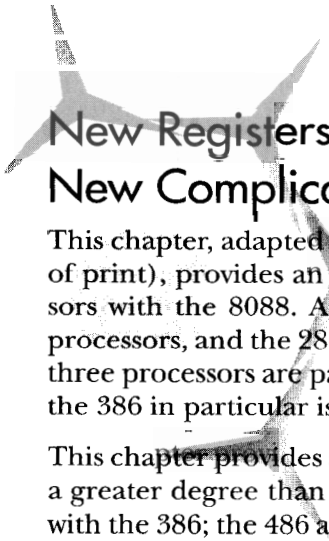
Chapter 11

Pushing the 286 and 386

Chapter 11



New Registers, New Instructions, New Timings, New Complications

This chapter, adapted from my earlier book *Zen of Assembly Language* (1989; now out of print), provides an overview of the 286 and 386, often contrasting those processors with the 8088. At the time I originally wrote this, the 8088 was the king of processors, and the 286 and 386 were the new kids on the block. Today, of course, all three processors are past their primes, but many millions of each are still in use, and the 386 in particular is still well worth considering when optimizing software.

This chapter provides an interesting look at the evolution of the x86 architecture, to a greater degree than you might expect, for the x86 family came into full maturity with the 386; the 486 and the Pentium are really nothing more than faster 386s, with very little in the way of new functionality. In contrast, the 286 added a number of instructions, respectable performance, and protected mode to the 8088's capabilities, and the 386 added more instructions and a whole new set of addressing modes, and brought the x86 family into the 32-bit world that represents the future (and, increasingly, the present) of personal computing. This chapter also provides insight into the effects on optimization of the variations in processors and memory architectures that are common in the PC world. So, although the 286 and 386 no longer represent the mainstream of computing, this chapter is a useful mix of history lesson, x86 overview, and details on two workhorse processors that are still in wide use.

Family Matters

While the x86 family is a large one, only a few members of the family—which includes the 8088, 8086, 80188, 80186, 286, 386SX, 386DX, numerous permutations of the 486, and now the Pentium—really matter.

The 8088 is now all but extinct in the PC arena. The 8086 was used fairly widely for a while, but has now all but disappeared. The 80186 and 80188 never really caught on for use in PC and don't require further discussion.

That leaves us with the high-end chips: the 286, the 386SX, the 386, the 486, and the Pentium. At this writing, the 386SX is fast going the way of the 8088; people are realizing that its relatively small cost advantage over the 386 isn't enough to offset its relatively large performance disadvantage. After all, the 386SX suffers from the same debilitating problem that looms over the 8088—a too-small bus. Internally, the 386SX is a 32-bit processor, but externally, it's a 16-bit processor, a non-optimal architecture, especially for 32-bit code.

I'm not going to discuss the 386SX in detail. If you do find yourself programming for the 386SX, follow the same general rules you should follow for the 8088: use short instructions, use the registers as heavily as possible, and don't branch. In other words, avoid memory, since the 386SX is by definition better at processing data internally than it is at accessing memory.

The 486 is a world unto itself for the purposes of optimization, and the Pentium is a *universe* unto itself. We'll treat them separately in later chapters.

This leaves us with just two processors: the 286 and the 386. Each was *the* PC standard in its day. The 286 is no longer used in new systems, but there are millions of 286-based systems still in daily use. The 386 is still being used in new systems, although it's on the downhill leg of its lifespan, and it is in even wider use than the 286. The future clearly belongs to the 486 and Pentium, but the 286 and 386 are still very much a part of the present-day landscape.

Crossing the Gulf to the 286 and the 386

Apart from vastly improved performance, the biggest difference between the 8088 and the 286 and 386 (as well as the later Intel CPUs) is that the 286 introduced protected mode, and the 386 greatly expanded the capabilities of protected mode. We're only going to talk about real-mode operation of the 286 and 386 in this book, however. Protected mode offers a whole new memory management scheme, one that isn't supported by the 8088. Only code specifically written for protected mode can run in that mode; it's an alien and hostile environment for MS-DOS programs.

In particular, segments are different creatures in protected mode. They're *selectors*—indexes into a table of segment descriptors—rather than plain old registers, and

can't be set to arbitrary values. That means that segments can't be used for temporary storage or as part of a fast indivisible 32-bit load from memory, as in

```
les   ax,dword ptr [LongVar]
mov   dx,es
```

which loads **LongVar** into DX:AX faster than this:

```
mov   ax,word ptr [LongVar]
mov   dx,word ptr [LongVar+2]
```

Protected mode uses those altered segment registers to offer access to a great deal more memory than real mode: The 286 supports 16 megabytes of memory, while the 386 supports 4 gigabytes (4K megabytes) of physical memory and 64 *terabytes* (64K gigabytes!) of virtual memory.

In protected mode, your programs generally run under an operating system (OS/2, Unix, Windows NT or the like) that exerts much more control over the computer than does MS-DOS. Protected mode operating systems can generally run multiple programs simultaneously, and the performance of any one program may depend far less on code quality than on how efficiently the program uses operating system services and how often and under what circumstances the operating system preempts the program. Protected mode programs are often mostly collections of operating system calls, and the performance of whatever code *isn't* operating-system oriented may depend primarily on how large a time slice the operating system gives that code to run in.

In short, taken as a whole, protected mode programming is a different kettle of fish altogether from what I've been describing in this book. There's certainly a knack to optimizing specifically for protected mode under a given operating system...but it's not what we've been learning, and now is not the time to pursue it further. In general, though, the optimization strategies discussed in this book still hold true in protected mode; it's just issues specific to protected mode or a particular operating system that we won't discuss.

In the Lair of the Cycle-Eaters, Part II

Under the programming interface, the 286 and 386 differ considerably from the 8088. Nonetheless, with one exception and one addition, the cycle-eaters remain much the same on computers built around the 286 and 386. Next, we'll review each of the familiar cycle-eaters I covered in Chapter 4 as they apply to the 286 and 386, and we'll look at the new member of the gang, the data alignment cycle-eater.

The one cycle-eater that vanishes on the 286 and 386 is the 8-bit bus cycle-eater. The 286 is a 16-bit processor both internally and externally, and the 386 is a 32-bit processor both internally and externally, so the Execution Unit/Bus Interface Unit size

mismatch that plagues the 8088 is eliminated. Consequently, there's no longer any need to use byte-sized memory variables in preference to word-sized variables, at least so long as word-sized variables start at even addresses, as we'll see shortly. On the other hand, access to byte-sized variables still isn't any *slower* than access to word-sized variables, so you can use whichever size suits a given task best.

You might think that the elimination of the 8-bit bus cycle-eater would mean that the prefetch queue cycle-eater would also vanish, since on the 8088 the prefetch queue cycle-eater is a side effect of the 8-bit bus. That would seem all the more likely given that both the 286 and the 386 have larger prefetch queues than the 8088 (6 bytes for the 286, 16 bytes for the 386) and can perform memory accesses, including instruction fetches, in far fewer cycles than the 8088.

However, the prefetch queue cycle-eater *doesn't* vanish on either the 286 or the 386, for several reasons. For one thing, branching instructions still empty the prefetch queue, so instruction fetching still slows things down after most branches; when the prefetch queue is empty, it doesn't much matter how big it is. (Even apart from emptying the prefetch queue, branches aren't particularly fast on the 286 or the 386, at a minimum of seven-plus cycles apiece. Avoid branching whenever possible.)

After a branch it *does* matter how fast the queue can refill, and there we come to the second reason the prefetch queue cycle-eater lives on: The 286 and 386 are so fast that sometimes the Execution Unit can execute instructions faster than they can be fetched, even though instruction fetching is *much* faster on the 286 and 386 than on the 8088.

(All other things being equal, too-slow instruction fetching is more of a problem on the 286 than on the 386, since the 386 fetches 4 instruction bytes at a time versus the 2 instruction bytes fetched per memory access by the 286. However, the 386 also typically runs at least twice as fast as the 286, meaning that the 386 can easily execute instructions faster than they can be fetched unless very high-speed memory is used.)

The most significant reason that the prefetch queue cycle-eater not only survives but prospers on the 286 and 386, however, lies in the various memory architectures used in computers built around the 286 and 386. Due to the memory architectures, the 8-bit bus cycle-eater is replaced by a new form of the wait state cycle-eater: wait states on accesses to normal system memory.

System Wait States

The 286 and 386 were designed to lose relatively little performance to the prefetch queue cycle-eater... *when used with zero-wait-state memory:* memory that can complete memory accesses so rapidly that no wait states are needed. However, true zero-wait-state memory is almost never used with those processors. Why? Because memory that can keep up with a 286 is fairly expensive, and memory that can keep up with a 386 is *very* expensive. Instead, computer designers use alternative memory architectures

that offer more performance for the dollar—but less performance overall—than zero-wait-state memory. (It *is* possible to build zero-wait-state systems for the 286 and 386; it's just so expensive that it's rarely done.)

The IBM AT and true compatibles use one-wait-state memory (some AT clones use zero-wait-state memory, but such clones are less common than one-wait-state AT clones). The 386 systems use a wide variety of memory systems—including high-speed caches, interleaved memory, and static-column RAM—that insert anywhere from 0 to about 5 wait states (and many more if 8 or 16-bit memory expansion cards are used); the exact number of wait states inserted at any given time depends on the interaction between the code being executed and the memory system it's running on.

The performance of most 386 memory systems can vary greatly from one memory access to another, depending on factors such as what data happens to be in the cache and which interleaved bank and/or RAM column was accessed last.

The many memory systems in use make it impossible for us to optimize for 286/386 computers with the precision that's possible on the 8088. Instead, we must write code that runs reasonably well under the varying conditions found in the 286/386 arena.

The wait states that occur on most accesses to system memory in 286 and 386 computers mean that nearly every access to system memory—memory in the DOS's normal 640K memory area—is slowed down. (Accesses in computers with high-speed caches may be wait-state-free if the desired data is already in the cache, but will certainly encounter wait states if the data isn't cached; this phenomenon produces highly variable instruction execution times.) While this is our first encounter with system memory wait states, we have run into a wait-state cycle-eater before: the display adapter cycle-eater, which we discussed along with the other 8088 cycle-eaters way back in Chapter 4. System memory generally has fewer wait states per access than display memory. However, system memory is also accessed far more often than display memory, so system memory wait states hurt plenty—and the place they hurt most is instruction fetching.

Consider this: The 286 can store an immediate value to memory, as in **MOV [WordVar],0**, in just 3 cycles. However, that instruction is 6 bytes long. The 286 is capable of fetching 1 word every 2 cycles; however, the one-wait-state architecture of the AT stretches that to 3 cycles. Consequently, nine cycles are needed to fetch the six instruction bytes. On top of that, 3 cycles are needed to write to memory, bringing the total memory access time to 12 cycles. On balance, memory access time—especially instruction prefetching—greatly exceeds execution time, to the extent that this particular instruction can take up to four times as long to run as it does to execute in the Execution Unit.

And that, my friend, is unmistakably the prefetch queue cycle-eater. I might add that the prefetch queue cycle-eater is in rare good form in the above example: A 4-to-1

ratio of instruction fetch time to execution time is in a class with the best (or worst!) that's found on the 8088.

Let's check out the prefetch queue cycle-eater in action. Listing 11.1 times **MOV [WordVar],0**. The Zen timer reports that on a one-wait-state 10 MHz 286-based AT clone (the computer used for all tests in this chapter), Listing 11.1 runs in 1.27 µs per instruction. That's 12.7 cycles per instruction, just as we calculated. (That extra seven-tenths of a cycle comes from DRAM refresh, which we'll get to shortly.)

LISTING 11.1 L11-1.ASM

```
;
; *** Listing 11.1 ***
;
; Measures the performance of an immediate move to
; memory, in order to demonstrate that the prefetch
; queue cycle-eater is alive and well on the AT.
;
        jmp     Skip
;
        even            ;always make sure word-sized memory
                        ; variables are word-aligned!
WordVar dw      0
;
Skip:
        call    ZTimerOn
        rept    1000
        mov     [WordVar],0
        endm
        call    ZTimerOff
```

What does this mean? It means that, practically speaking, the 286 as used in the AT doesn't have a 16-bit bus. From a performance perspective, the 286 in an AT has two-thirds of a 16-bit bus (a 10.7-bit bus?), since every bus access on an AT takes 50 percent longer than it should. A 286 running at 10 MHz *should* be able to access memory at a maximum rate of 1 word every 200 ns; in a 10 MHz AT, however, that rate is reduced to 1 word every 300 ns by the one-wait-state memory.

In short, a close relative of our old friend the 8-bit bus cycle-eater—the system memory wait state cycle-eater—haunts us still on all but zero-wait-state 286 and 386 computers, and that means that the prefetch queue cycle-eater is alive and well. (The system memory wait state cycle-eater isn't really a new cycle-eater, but rather a variant of the general wait state cycle-eater, of which the display adapter cycle-eater is yet another variant.) While the 286 in the AT can fetch instructions much faster than can the 8088 in the PC, it can execute those instructions faster still.

The picture is less clear in the 386 world since there are so many different memory architectures, but similar problems can occur in any computer built around a 286 or 386. The prefetch queue cycle-eater is even a factor—albeit a lesser one—on zero-wait-state machines, both because branching empties the queue and because some instructions can outrun even zero-wait-state instruction fetching. (Listing 11.1 would

take at least 8 cycles per instruction on a zero-wait-state AT—5 cycles longer than the official execution time.)

To summarize:

- Memory-accessing instructions don't run at their official speeds on non-zero-wait-state 286/386 computers.
- The prefetch queue cycle-eater reduces performance on 286/386 computers, particularly when non-zero-wait-state memory is used.
- Branches often execute at less than their rated speeds on the 286 and 386 since the prefetch queue is emptied.
- The extent to which the prefetch queue and wait states affect performance varies from one 286/386 computer to another, making precise optimization impossible.

What's to be learned from all this? Several things:

- Keep your instructions short.
- Keep it in the registers; avoid memory, since memory generally can't keep up with the processor.
- Don't jump.

Of course, those are exactly the rules that apply to 8088 optimization as well. Isn't it convenient that the same general rules apply across the board?

Data Alignment

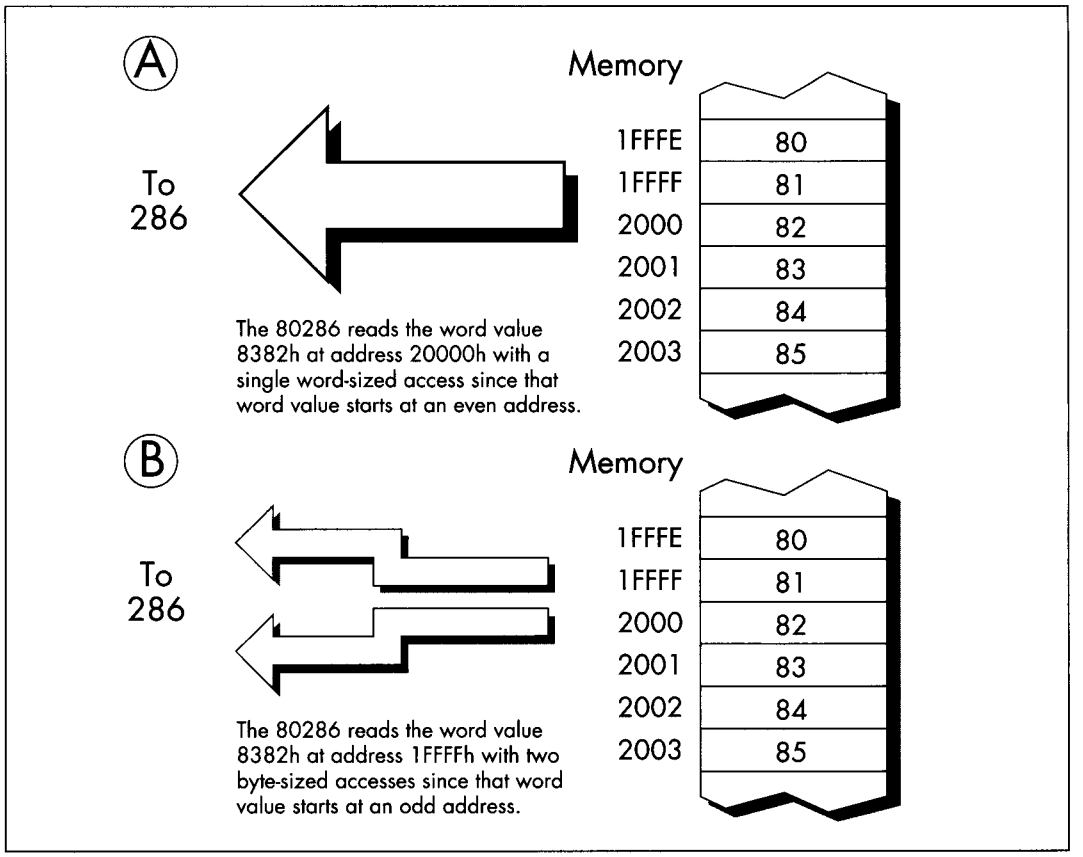
Thanks to its 16-bit bus, the 286 can access word-sized memory variables just as fast as byte-sized variables. There's a catch, however: That's only true for word-sized variables that start at even addresses. When the 286 is asked to perform a word-sized access starting at an odd address, it actually performs two separate accesses, each of which fetches 1 byte, just as the 8088 does for all word-sized accesses.

Figure 11.1 illustrates this phenomenon. The conversion of word-sized accesses to odd addresses into double byte-sized accesses is transparent to memory-accessing instructions; all any instruction knows is that the requested word has been accessed, no matter whether 1 word-sized access or 2 byte-sized accesses were required to accomplish it.

The penalty for performing a word-sized access starting at an odd address is easy to calculate: Two accesses take twice as long as one access.

In other words, the effective capacity of the 286's external data bus is halved when a word-sized access to an odd address is performed.

That, in a nutshell, is the data alignment cycle-eater, the one new cycle-eater of the 286 and 386. (The data alignment cycle-eater is a close relative of the 8088's 8-bit bus cycle-eater, but since it behaves differently—occurring only at odd addresses—and is avoided with a different workaround, we'll consider it to be a new cycle-eater.)



The data alignment cycle-eater.
Figure 11.1

The way to deal with the data alignment cycle-eater is straightforward: *Don't perform word-sized accesses to odd addresses on the 286 if you can help it.* The easiest way to avoid the data alignment cycle-eater is to place the directive **EVEN** before each of your word-sized variables. **EVEN** forces the offset of the next byte assembled to be even by inserting a **NOP** if the current offset is odd; consequently, you can ensure that any word-sized variable can be accessed efficiently by the 286 simply by preceding it with **EVEN**.

Listing 11.2, which accesses memory a word at a time with each word starting at an odd address, runs on a 10 MHz AT clone in 1.27 µs per repetition of **MOVSW**, or 0.64 µs per word-sized memory access. That's 6-plus cycles per word-sized access, which breaks down to two separate memory accesses—3 cycles to access the high byte of each word and 3 cycles to access the low byte of each word, the inevitable result of non-word-aligned word-sized memory accesses—plus a bit extra for DRAM refresh.

LISTING 11.2 L11-2.ASM

```
;
; *** Listing 11.2 ***
;
; Measures the performance of accesses to word-sized
; variables that start at odd addresses (are not
; word-aligned).
;
Skip:
        push    ds
        pop     es
        mov     si,1    ;source and destination are the same
        mov     di,si   ; and both are not word-aligned
        mov     cx,1000 ;move 1000 words
        cld
        call    ZTimerOn
        rep     movsw
        call    ZTimerOff
```

On the other hand, Listing 11.3, which is exactly the same as Listing 11.2 save that the memory accesses are word-aligned (start at even addresses), runs in 0.64 µs per repetition of **MOVSW**, or 0.32 µs per word-sized memory access. That's 3 cycles per word-sized access—exactly twice as fast as the non-word-aligned accesses of Listing 11.2, just as we predicted.

LISTING 11.3 L11-3.ASM

```
;
; *** Listing 11.3 ***
;
; Measures the performance of accesses to word-sized
; variables that start at even addresses (are word-aligned).
;
Skip:
        push    ds
        pop     es
        sub     si,si   ;source and destination are the same
        mov     di,si   ; and both are word-aligned
        mov     cx,1000 ;move 1000 words
        cld
        call    ZTimerOn
        rep     movsw
        call    ZTimerOff
```

The data alignment cycle-eater has intriguing implications for speeding up 286/386 code. The expenditure of a little care and a few bytes to make sure that word-sized variables and memory blocks are word-aligned can literally double the performance of certain code running on the 286. Even if it doesn't double performance, word alignment usually helps and never hurts.

Code Alignment

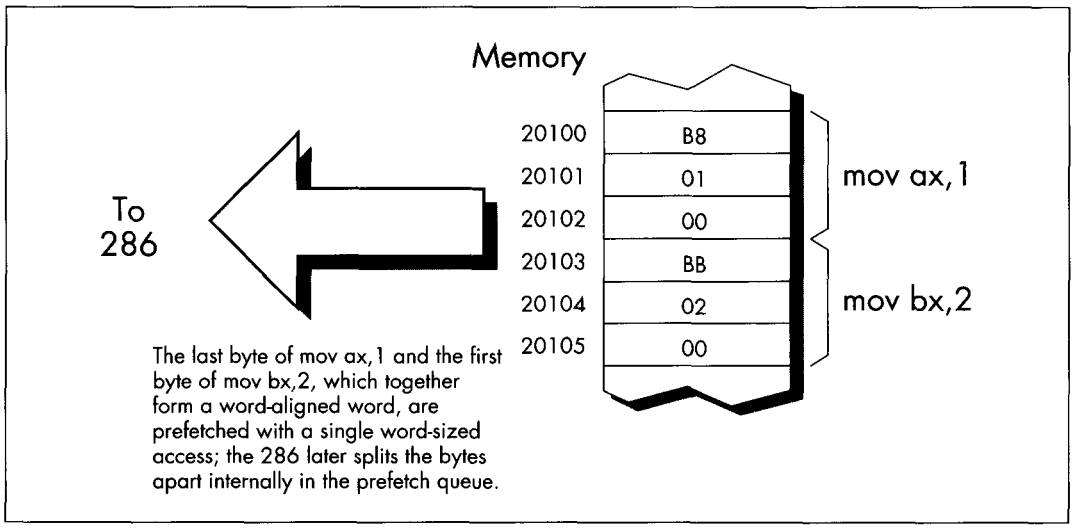
Lack of word alignment can also interfere with instruction fetching on the 286, although not to the extent that it interferes with access to word-sized memory variables.

The 286 prefetches instructions a word at a time; even if a given instruction doesn't begin at an even address, the 286 simply fetches the first byte of that instruction at the same time that it fetches the last byte of the previous instruction, as shown in Figure 11.2, then separates the bytes internally. That means that in most cases, instructions run just as fast whether they're word-aligned or not.

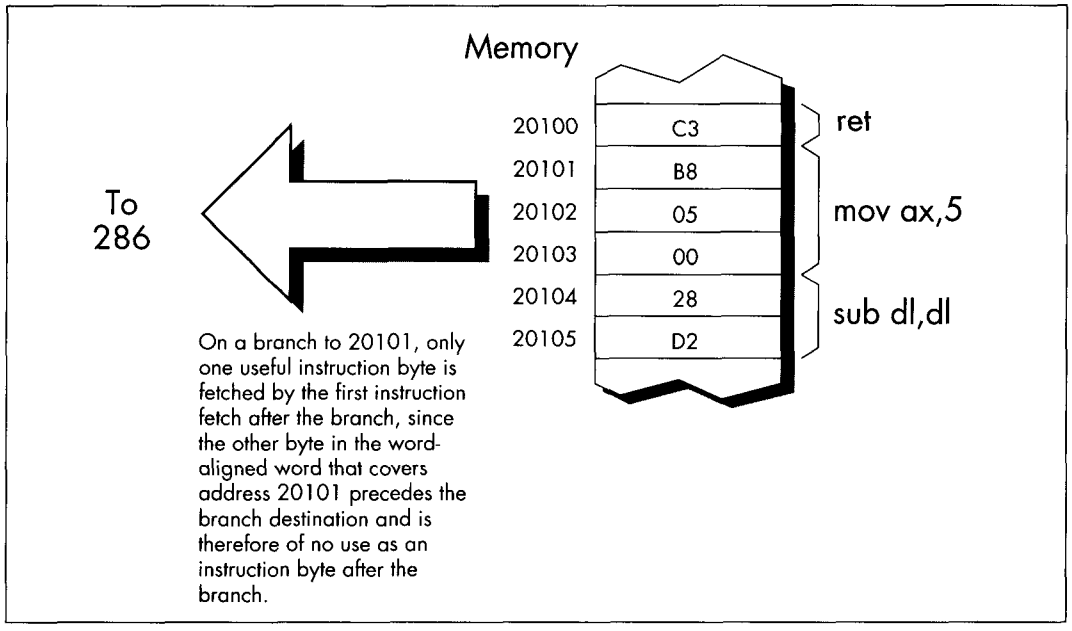
There is, however, a non-word-alignment penalty on *branches* to odd addresses. On a branch to an odd address, the 286 is only able to fetch 1 useful byte with the first instruction fetch following the branch, as shown in Figure 11.3. In other words, lack of word alignment of the target instruction for any branch effectively cuts the instruction-fetching power of the 286 in half for the first instruction fetch after that branch. While that may not sound like much, you'd be surprised at what it can do to tight loops; in fact, a brief story is in order.

When I was developing the Zen timer, I used my trusty 10 MHz 286-based AT clone to verify the basic functionality of the timer by measuring the performance of simple instruction sequences. I was cruising along with no problems until I timed the following code:

```
      mov   cx,1000
      call  ZTimerOn
LoopTop:
      loop  LoopTop
      call  ZTimerOff
```



Word-aligned prefetching on the 286.
Figure 11.2



How instruction bytes are fetched after a branch.
Figure 11.3

Now, this code *should* run in, say, about 12 cycles per loop at most. Instead, it took over 14 cycles per loop, an execution time that I could not explain in any way. After rolling it around in my head for a while, I took a look at the code under a debugger...and the answer leaped out at me. *The loop began at an odd address!* That meant that two instruction fetches were required each time through the loop; one to get the opcode byte of the **LOOP** instruction, which resided at the end of one word-aligned word, and another to get the displacement byte, which resided at the start of the next word-aligned word.

One simple change brought the execution time down to a reasonable 12.5 cycles per loop:

```
     mov   cx,1000
     call  ZTimerOn
     even
LoopTop:
     loop  LoopTop
     call  ZTimerOff
```

While word-aligning branch destinations can improve branching performance, it's a nuisance and can increase code size a good deal, so it's not worth doing in most code. Besides, **EVEN** inserts a **NOP** instruction if necessary, and the time required to

execute a **NOP** can sometimes cancel the performance advantage of having a word-aligned branch destination.

Consequently, it's best to word-align only those branch destinations that can be reached solely by branching.

I recommend that you only go out of your way to word-align the start offsets of your subroutines, as in:

```
          even
FindChar  proc  near
          :
```

In my experience, this simple practice is the one form of code alignment that consistently provides a reasonable return for bytes and effort expended, although sometimes it also pays to word-align tight time-critical loops.

Alignment and the 386

So far we've only discussed alignment as it pertains to the 286. What, you may well ask, of the 386?

The 386 adds the issue of *doubleword* alignment (that is, alignment to addresses that are multiples of four.) The rule for the 386 is: Word-sized memory accesses should be word-aligned (it's impossible for word-aligned word-sized accesses to cross doubleword boundaries), and doubleword-sized memory accesses should be doubleword-aligned. However, in real (as opposed to 32-bit protected) mode, doubleword-sized memory accesses are rare, so the simple word-alignment rule we've developed for the 286 serves for the 386 in real mode as well.

As for code alignment...the subroutine-start word-alignment rule of the 286 serves reasonably well there too since it avoids the worst case, where just 1 byte is fetched on entry to a subroutine. While optimum performance would dictate doubleword alignment of subroutines, that takes 3 bytes, a high price to pay for an optimization that improves performance *only* on the post 286 processors.

Alignment and the Stack

One side-effect of the data alignment cycle-eater of the 286 and 386 is that you should *never* allow the stack pointer to become odd. (You can make the stack pointer odd by adding an odd value to it or subtracting an odd value from it, or by loading it with an odd value.) An odd stack pointer on the 286 or 386 (or a non-doubleword-aligned stack in 32-bit protected mode on the 386, 486, or Pentium) will significantly reduce the performance of **PUSH**, **POP**, **CALL**, and **RET**, as well as **INT** and **IRET**, which are executed to invoke DOS and BIOS functions, handle keystrokes and incoming serial characters, and manage the mouse. I know of a Forth programmer who vastly

improved the performance of a complex application on the AT simply by forcing the Forth interpreter to maintain an even stack pointer at all times.

An interesting corollary to this rule is that you shouldn't **INC SP** twice to add 2, even though that takes fewer bytes than **ADD SP,2**. The stack pointer is odd between the first and second **INC**, so any interrupt occurring between the two instructions will be serviced more slowly than it normally would. The same goes for decrementing twice; use **SUB SP,2** instead.

Keep the stack pointer aligned at all times.

The DRAM Refresh Cycle-Eater: Still an Act of God

The DRAM refresh cycle-eater is the cycle-eater that's least changed from its 8088 form on the 286 and 386. In the AT, DRAM refresh uses a little over five percent of all available memory accesses, slightly less than it uses in the PC, but in the same ballpark. While the DRAM refresh penalty varies somewhat on various AT clones and 386 computers (in fact, a few computers are built around static RAM, which requires no refresh at all; likewise, caches are made of static RAM so cached systems generally suffer less from DRAM refresh), the 5 percent figure is a good rule of thumb.

Basically, the effect of the DRAM refresh cycle-eater is pretty much the same throughout the PC-compatible world: fairly small, so it doesn't greatly affect performance; unavoidable, so there's no point in worrying about it anyway; and a nuisance since it results in fractional cycle counts when using the Zen timer. Just as with the PC, a given code sequence on the AT can execute at varying speeds at different times as a result of the interaction between the code and DRAM refresh.

There's nothing much new with DRAM refresh on 286/386 computers, then. Be aware of it, but don't overly concern yourself—DRAM refresh is still an act of God, and there's not a blessed thing you can do about it. Happily, the internal caches of the 486 and Pentium make DRAM refresh largely a performance non-issue on those processors.

The Display Adapter Cycle-Eater

Finally we come to the last of the cycle-eaters, the display adapter cycle-eater. There are two ways of looking at this cycle-eater on 286/386 computers: (1) It's much worse than it was on the PC, or (2) it's just about the same as it was on the PC.

Either way, the display adapter cycle-eater is extremely bad news on 286/386 computers and on 486s and Pentiums as well. In fact, this cycle-eater on those systems is largely responsible for the popularity of VESA local bus (VLB).

The two ways of looking at the display adapter cycle-eater on 286/386 computers are actually the same. As you'll recall from my earlier discussion of the matter in Chapter 4, display adapters offer only a limited number of accesses to display memory

during any given period of time. The 8088 is capable of making use of most but not all of those slots with **REP MOVSW**, so the number of memory accesses allowed by a display adapter such as a standard VGA is reasonably well-matched to an 8088's memory access speed. Granted, access to a VGA slows the 8088 down considerably—but, as we're about to find out, "considerably" is a relative term. What a VGA does to PC performance is nothing compared to what it does to faster computers.

Under ideal conditions, a 286 can access memory much, much faster than an 8088. A 10 MHz 286 is capable of accessing a word of system memory every 0.20 µs with **REP MOVSW**, dwarfing the 1 byte every 1.31 µs that the 8088 in a PC can manage. However, access to display memory is anything but ideal for a 286. For one thing, most display adapters are 8-bit devices, although newer adapters are 16-bit in nature. One consequence of that is that only 1 byte can be read or written per access to display memory; word-sized accesses to 8-bit devices are automatically split into 2 separate byte-sized accesses by the AT's bus. Another consequence is that accesses are simply slower; the AT's bus inserts additional wait states on accesses to 8-bit devices since it must assume that such devices were designed for PCs and may not run reliably at AT speeds.

However, the 8-bit size of most display adapters is but one of the two factors that reduce the speed with which the 286 can access display memory. Far more cycles are eaten by the inherent memory-access limitations of display adapters—that is, the limited number of display memory accesses that display adapters make available to the 286. Look at it this way: If **REP MOVSW** on a PC can use more than half of all available accesses to display memory, then how much faster can code running on a 286 or 386 possibly run when accessing display memory?

That's right—less than twice as fast.

In other words, instructions that access display memory won't run a whole lot faster on ATs and faster computers than they do on PCs. That explains one of the two viewpoints expressed at the beginning of this section: The display adapter cycle-eater is just about the same on high-end computers as it is on the PC, in the sense that it allows instructions that access display memory to run at just about the same speed on all computers.

Of course, the picture is quite a bit different when you compare the performance of instructions that access display memory to the *maximum* performance of those instructions. Instructions that access display memory receive many more wait states when running on a 286 than they do on an 8088. Why? While the 286 is capable of accessing memory much more often than the 8088, we've seen that the frequency of access to display memory is determined not by processor speed but by the display adapter itself. As a result, both processors are actually allowed just about the same maximum number of accesses to display memory in any given time. By definition, then, the 286 must spend many more cycles waiting than does the 8088.

And that explains the second viewpoint expressed above regarding the display adapter cycle-eater vis-a-vis the 286 and 386. The display adapter cycle-eater, as measured in cycles lost to wait states, is indeed much worse on AT-class computers than it is on the PC, and it's worse still on more powerful computers.

How bad is the display adapter cycle-eater on an AT? It's this bad: Based on my (not inconsiderable) experience in timing display adapter access, I've found that the display adapter cycle-eater can slow an AT—or even a 386 computer—to near-PC speeds when display memory is accessed.

I know that's hard to believe, but the display adapter cycle-eater gives out just so many display memory accesses in a given time, and no more, no matter how fast the processor is. In fact, the faster the processor, the more the display adapter cycle-eater hurts the performance of instructions that access display memory. The display adapter cycle-eater is not only still present in 286/386 computers, it's worse than ever.

What can we do about this new, more virulent form of the display adapter cycle-eater? The workaround is the same as it was on the PC: Access display memory as little as you possibly can.

New Instructions and Features: The 286

The 286 and 386 offer a number of new instructions. The 286 has a relatively small number of instructions that the 8088 lacks, while the 386 has those instructions and quite a few more, along with new addressing modes and data sizes. We'll discuss the 286 and the 386 separately in this regard.

The 286 has a number of instructions designed for protected-mode operations. As I've said, we're not going to discuss protected mode in this book; in any case, protected-mode instructions are generally used only by operating systems. (I should mention that the 286's protected mode brings with it the ability to address 16 MB of memory, a considerable improvement over the 8088's 1 MB. In real mode, however, programs are still limited to 1 MB of addressable memory on the 286. In either mode, each segment is still limited to 64K.)

There are also a handful of 286-specific real-mode instructions, and they can be quite useful. **BOUND** checks array bounds. **ENTER** and **LEAVE** support compact and speedy stack frame construction and removal, ideal for interfacing to high-level languages such as C and Pascal (although these instructions are actually relatively slow on the 386 and its successors, and should be used with caution when performance matters). **INS** and **OUTS** are new string instructions that support efficient data transfer between memory and I/O ports. Finally, **PUSHA** and **POPA** push and pop all eight general-purpose registers.

A couple of old instructions gain new features on the 286. For one, the 286 version of **PUSH** is capable of pushing a constant on the stack. For another, the 286 allows all shifts and rotates to be performed for not just 1 bit or the number of bits specified by CL, but for *any* constant number of bits.

New Instructions and Features: The 386

The 386 is somewhat more complex than the 286 regarding new features. Once again, we won't discuss protected mode, which on the 386 comes with the ability to address up to 4 gigabytes per segment and 64 terabytes in all. In real mode (and in virtual-86 mode, which allows the 386 to multitask MS-DOS applications, and which is identical to real mode so far as MS-DOS programs are concerned), programs running on the 386 are still limited to 1 MB of addressable memory and 64K per segment.

The 386 has many new instructions, as well as new registers, addressing modes and data sizes that have trickled down from protected mode. Let's take a quick look at these new real-mode features.

Even in real mode, it's possible to access many of the 386's new and extended registers. Most of these registers are simply 32-bit extensions of the 16-bit registers of the 8088. For example, EAX is a 32-bit register containing AX as its lower 16 bits, EBX is a 32-bit register containing BX as its lower 16 bits, and so on. There are also two new segment registers: FS and GS.

The 386 also comes with a slew of new real-mode instructions beyond those supported by the 8088 and 286. These instructions can scan data on a bit-by-bit basis, set the Carry flag to the value of a specified bit, sign-extend or zero-extend data as it's moved, set a register or memory variable to 1 or 0 on the basis of any of the conditions that can be tested with conditional jumps, and more. (Again, beware: Many of these complex 386-specific instructions are slower than equivalent sequences of simple instructions on the 486 and especially on the Pentium.) What's more, both old and new instructions support 32-bit operations on the 386. For example, it's relatively simple to copy data in chunks of 4 bytes on a 386, even in real mode, by using the **MOVSD** ("move string double") instruction, or to negate a 32-bit value with **NEG EAX**.

Finally, it's possible in real mode to use the 386's new addressing modes, in which *any* 32-bit general-purpose register or pair of registers can be used to address memory. What's more, multiplication of memory-addressing registers by 2, 4, or 8 for look-ups in word, doubleword, or quadword tables can be built right into the memory addressing mode. (The 32-bit addressing modes are discussed further in later chapters.) In protected mode, these new addressing modes allow you to address a full 4 gigabytes per segment, but in real mode you're still limited to 64K, even with 32-bit registers and the new addressing modes, unless you play some unorthodox tricks with the segment registers.

Note well: Those tricks don't necessarily work with system software such as Windows, so I'd recommend against using them. If you want 4-gigabyte segments, use a 32-bit environment such as Win32.

Optimization Rules: The More Things Change...

Let's see what we've learned about 286/386 optimization. Mostly what we've learned is that our familiar PC cycle-eaters still apply, although in somewhat different forms, and that the major optimization rules for the PC hold true on ATs and 386-based computers. You won't go wrong on any of these computers if you keep your instructions short, use the registers heavily and avoid memory, don't branch, and avoid accessing display memory like the plague.

Although we haven't touched on them, repeated string instructions are still desirable on the 286 and 386 since they provide a great deal of functionality per instruction byte and eliminate both the prefetch queue cycle-eater and branching. However, string instructions are not quite so spectacularly superior on the 286 and 386 as they are on the 8088 since non-string memory-accessing instructions have been speeded up considerably on the newer processors.

There's one cycle-eater with new implications on the 286 and 386, and that's the data alignment cycle-eater. From the data alignment cycle-eater we get a new rule: Word-align your word-sized variables, and start your subroutines at even addresses.

Detailed Optimization

While the major 8088 optimization rules hold true on computers built around the 286 and 386, many of the instruction-specific optimizations no longer hold, for the execution times of most instructions are quite different on the 286 and 386 than on the 8088. We have already seen one such example of the sometimes vast difference between 8088 and 286/386 instruction execution times: **MOV [WordVar],0**, which has an Execution Unit execution time of 20 cycles on the 8088, has an EU execution time of just 3 cycles on the 286 and 2 cycles on the 386.

In fact, the performance of virtually all memory-accessing instructions has been improved enormously on the 286 and 386. The key to this improvement is the near elimination of effective address (EA) calculation time. Where an 8088 takes from 5 to 12 cycles to calculate an EA, a 286 or 386 usually takes no time whatsoever to perform the calculation. If a base+index+displacement addressing mode, such as **MOV AX,[WordArray+BX+SI]**, is used on a 286 or 386, 1 cycle is taken to perform the EA calculation, but that's both the worst case and the only case in which there's any EA overhead at all.

The elimination of EA calculation time means that the EU execution time of memory-addressing instructions is much closer to the EU execution time of register-only instructions. For instance, on the 8088 **ADD [WordVar],100H** is a 31-cycle instruction, while **ADD DX,100H** is a 4-cycle instruction—a ratio of nearly 8 to 1. By contrast,

on the 286 **ADD [WordVar],100H** is a 7-cycle instruction, while **ADD DX,100H** is a 3-cycle instruction—a ratio of just 2.3 to 1.

It would seem, then, that it's less necessary to use the registers on the 286 than it was on the 8088, but that's simply not the case, for reasons we've already seen. The key is this: The 286 can execute memory-addressing instructions so fast that there's no spare instruction prefetching time during those instructions, so the prefetch queue runs dry, especially on the AT, with its one-wait-state memory. On the AT, the 6-byte instruction **ADD [WordVar],100H** is effectively at least a 15-cycle instruction, because 3 cycles are needed to fetch each of the three instruction words and 6 more cycles are needed to read **WordVar** and write the result back to memory.

Granted, the register-only instruction **ADD DX,100H** also slows down—to 6 cycles—because of instruction prefetching, leaving a ratio of 2.5 to 1. Now, however, let's look at the performance of the same code on an 8088. The register-only code would run in 16 cycles (4 instruction bytes at 4 cycles per byte), while the memory-accessing code would run in 40 cycles (6 instruction bytes at 4 cycles per byte, plus 2 word-sized memory accesses at 8 cycles per word). That's a ratio of 2.5 to 1, *exactly the same as on the 286.*

This is all theoretical. We put our trust not in theory but in actual performance, so let's run this code through the Zen timer. On a PC, Listing 11.4, which performs register-only addition, runs in 3.62 ms, while Listing 11.5, which performs addition to a memory variable, runs in 10.05 ms. On a 10 MHz AT clone, Listing 11.4 runs in 0.64 ms, while Listing 11.5 runs in 1.80 ms. Obviously, the AT is much faster...but the ratio of Listing 11.5 to Listing 11.4 is virtually identical on both computers, at 2.78 for the PC and 2.81 for the AT. If anything, the register-only form of **ADD** has a slightly *larger* advantage on the AT than it does on the PC in this case.

Theory confirmed.

LISTING 11.4 L11-4.ASM

```
;
; *** Listing 11.4 ***
;
; Measures the performance of adding an immediate value
; to a register, for comparison with Listing 11.5, which
; adds an immediate value to a memory variable.
;
        call    ZTimerOn
        rept    1000
        add     dx,100h
        endm
        call    ZTimerOff
```

LISTING 11.5 L11-5.ASM

```
;
; *** Listing 11.5 ***
;
; Measures the performance of adding an immediate value
; to a memory variable, for comparison with Listing 11.4,
; which adds an immediate value to a register.
```

```
;
        jmp     Skip
;
        even            ;always make sure word-sized memory
                        ; variables are word-aligned!
WordVar dw      0
;
Skip:
        call    ZTimerOn
        rept    1000
        add     [WordVar]100h
        endm
        call    ZTimerOff
```

What's going on? Simply this: Instruction fetching is controlling overall execution time on *both* processors. Both the 8088 in a PC and the 286 in an AT can execute the bytes of the instructions in Listings 11.4 and 11.5 faster than they can be fetched. Since the instructions are exactly the same lengths on both processors, it stands to reason that the ratio of the overall execution times of the instructions should be the same on both processors as well. Instruction length controls execution time, and the instruction lengths are the same—therefore the ratios of the execution times are the same. The 286 can both fetch and execute instruction bytes faster than the 8088 can, so code executes much faster on the 286; nonetheless, because the 286 can also execute those instruction bytes much faster than it can fetch them, overall performance is still largely determined by the size of the instructions.

Is this always the case? No. When the prefetch queue is full, memory-accessing instructions on the 286 and 386 are much faster (relative to register-only instructions) than they are on the 8088. Given the system wait states prevalent on 286 and 386 computers, however, the prefetch queue is likely to be empty quite a bit, especially when code consisting of instructions with short EU execution times is executed. Of course, that's just the sort of code we're likely to write when we're optimizing, so the performance of high-speed code is more likely to be controlled by instruction size than by EU execution time on most 286 and 386 computers, just as it is on the PC.

All of which is just a way of saying that faster memory access and EA calculation notwithstanding, it's just as desirable to keep instructions short and memory accesses to a minimum on the 286 and 386 as it is on the 8088. And the way to do that is to use the registers as heavily as possible, use string instructions, use short forms of instructions, and the like.

The more things change, the more they remain the same....

POPF and the 286

We've one final 286-related item to discuss: the hardware malfunction of **POPF** under certain circumstances on the 286.

The problem is this: Sometimes **POPF** permits interrupts to occur when interrupts are initially off and the setting popped into the Interrupt flag from the stack keeps

interrupts off. In other words, an interrupt can happen even though the Interrupt flag is never set to 1. Now, I don't want to blow this particular bug out of proportion. It only causes problems in code that cannot tolerate interrupts under any circumstances, and that's a rare sort of code, especially in user programs. However, some code really does need to have interrupts absolutely disabled, with no chance of an interrupt sneaking through. For example, a critical portion of a disk BIOS might need to retrieve data from the disk controller the instant it becomes available; even a few hundred microseconds of delay could result in a sector's worth of data misread. In this case, one misplaced interrupt during a **POPF** could result in a trashed hard disk if that interrupt occurs while the disk BIOS is reading a sector of the File Allocation Table.

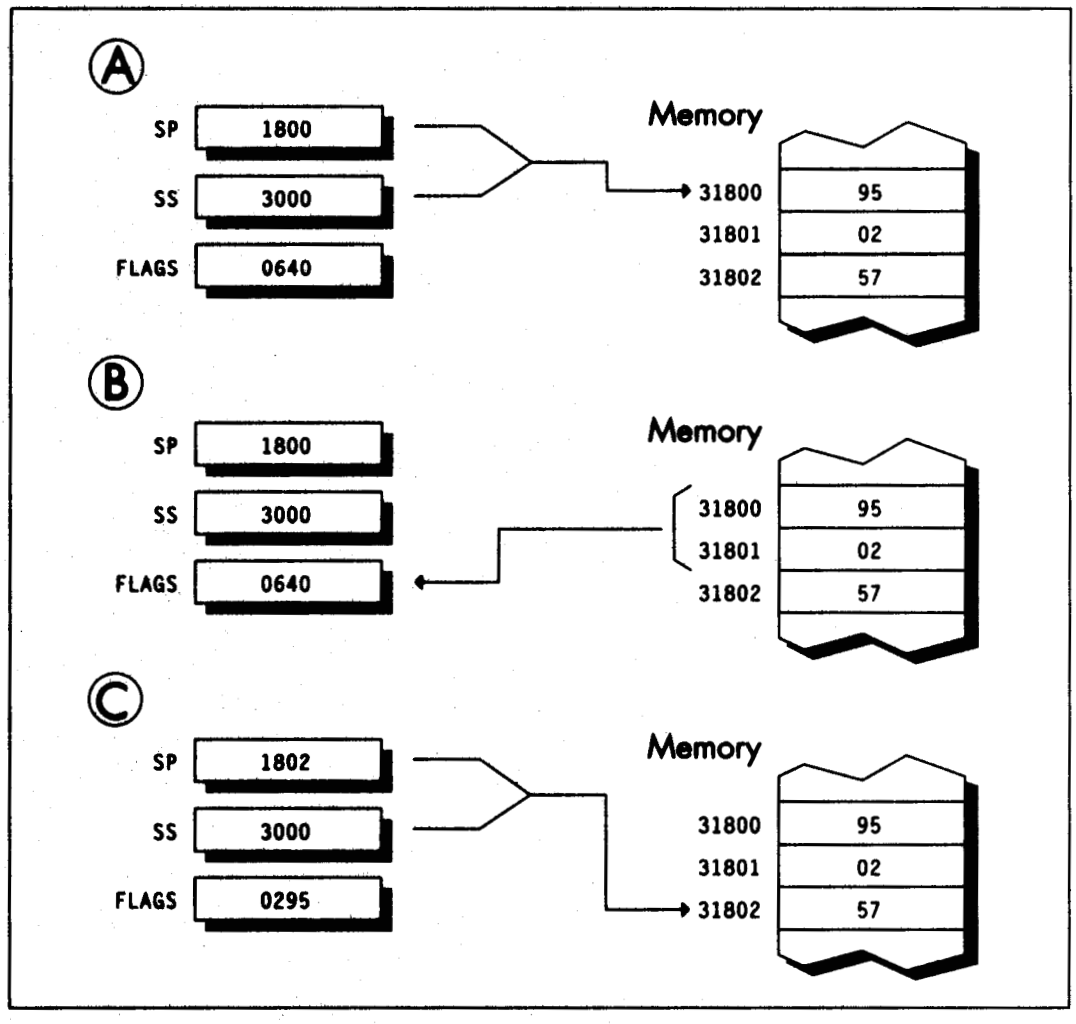
There is a workaround for the **POPF** bug. While the workaround is easy to use, it's considerably slower than **POPF**, and costs a few bytes as well, so you won't want to use it in code that can tolerate interrupts. On the other hand, in code that truly cannot be interrupted, you should view those extra cycles and bytes as cheap insurance against mysterious and erratic program crashes.

One obvious reason to discuss the **POPF** workaround is that it's useful. Another reason is that the workaround is an excellent example of Zen-level assembly coding, in that there's a well-defined goal to be achieved but no obvious way to do so. The goal is to reproduce the functionality of the **POPF** instruction without using **POPF**, and the place to start is by asking exactly what **POPF** does.

All **POPF** does is pop the word on top of the stack into the FLAGS register, as shown in Figure 11.4. How can we do that without **POPF**? Of course, the 286's designers intended us to use **POPF** for this purpose, and didn't intentionally provide any alternative approach, so we'll have to devise an alternative approach of our own. To do that, we'll have to search for instructions that contain some of the same functionality as **POPF**, in the hope that one of those instructions can be used in some way to replace **POPF**.

Well, there's only one instruction other than **POPF** that loads the FLAGS register directly from the stack, and that's **IRET**, which loads the FLAGS register from the stack as it branches, as shown in Figure 11.5. **IRET** has no known bugs of the sort that plague **POPF**, so it's certainly a candidate to replace **POPF** in non-interruptible applications. Unfortunately, **IRET** loads the FLAGS register with the *third* word down on the stack, not the word on top of the stack, as is the case with **POPF**; the far return address that **IRET** pops into CS:IP lies between the top of the stack and the word popped into the FLAGS register.

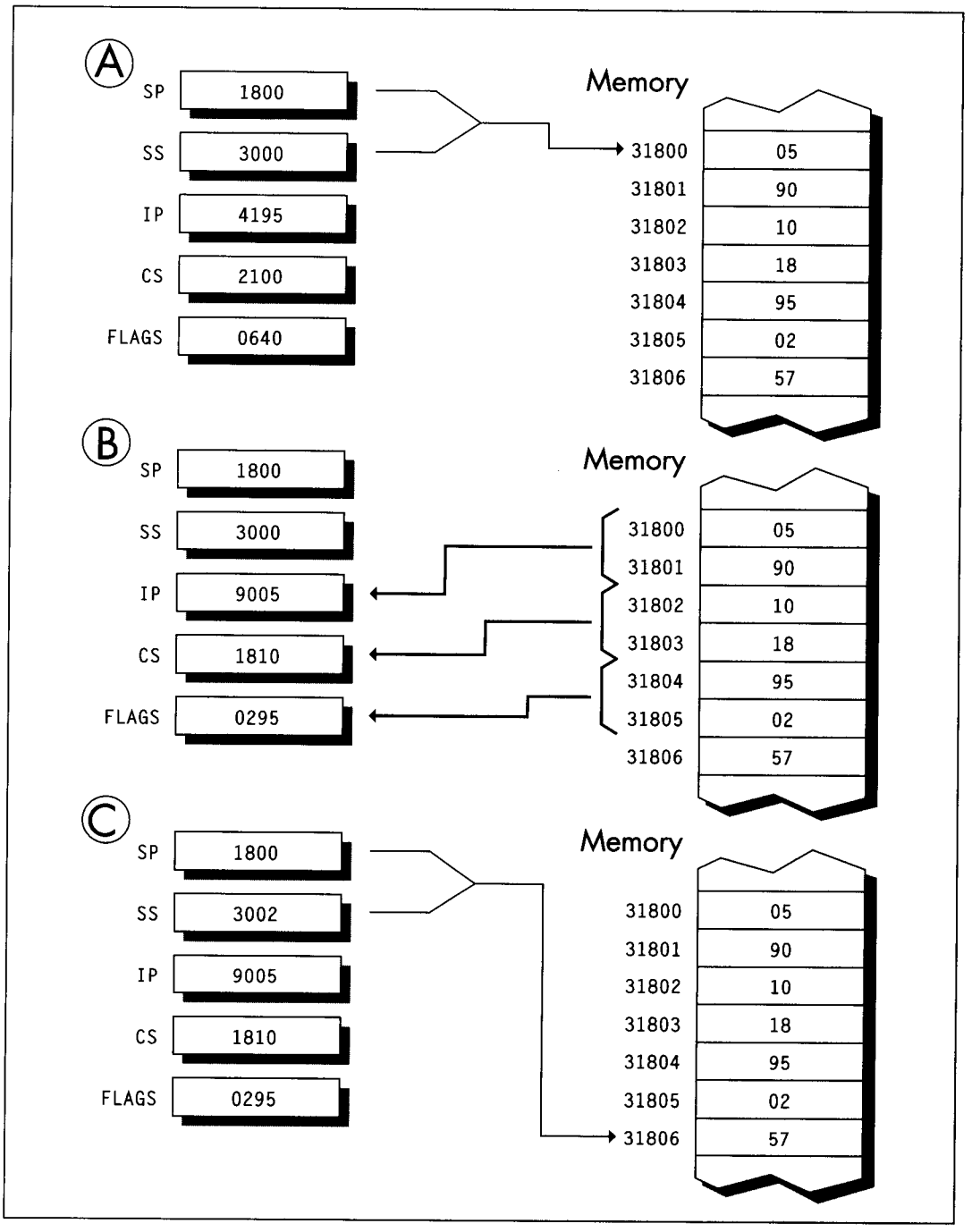
Obviously, the segment:offset that **IRET** expects to find on the stack above the pushed flags isn't present when the stack is set up for **POPF**, so we'll have to adjust the stack a bit before we can substitute **IRET** for **POPF**. What we'll have to do is push the segment:offset of the instruction after our workaround code onto the stack right above the pushed flags. **IRET** will then branch to that address and pop the flags,



The operation of POPF.
Figure 11.4

ending up at the instruction after the workaround code with the flags popped. That's just the result that would have occurred had we executed **POPF**—with the bonus that no interrupts can accidentally occur when the Interrupt flag is 0 both before and after the pop.

How can we push the segment:offset of the next instruction? Well, finding the offset of the next instruction by performing a near call to that instruction is a tried-and-true trick. We can do something similar here, but in this case we need a far call, since **IRET** requires both a segment and an offset. We'll also branch backward so that the



The operation of IRET.
Figure 11.5

address pushed on the stack will point to the instruction we want to continue with. The code works out like this:

```
      jmp   short popfskip
popfiret:
      iret              ;branches to the instruction after the
                        ; call, popping the word below the address
                        ; pushed by CALL into the FLAGS register
popfskip:
      call  far ptr popfiret
                        ;pushes the segment:offset of the next
                        ; instruction on the stack just above
                        ; the flags word, setting things up so
                        ; that IRET will branch to the next
                        ; instruction and pop the flags
; When execution reaches the instruction following this comment,
; the word that was on top of the stack when JMP SHORT POPFSKIP
; was reached has been popped into the FLAGS register, just as
; if a POPF instruction had been executed.
```

The operation of this code is illustrated in Figure 11.6.

The **POPF** workaround can best be implemented as a macro; we can also emulate a far call by pushing CS and performing a near call, thereby shrinking the workaround code by 1 byte:

```
EMULATE_POPF      macro
      local popfskip, popfiret
      jmp   short popfskip
popfiret:
      iret
popfskip:
      push  cs
      call  popfiret
      endm
```

By the way, the flags can be popped much more quickly if you're willing to alter a register in the process. For example, the following macro emulates **POPF** with just one branch, but wipes out AX:

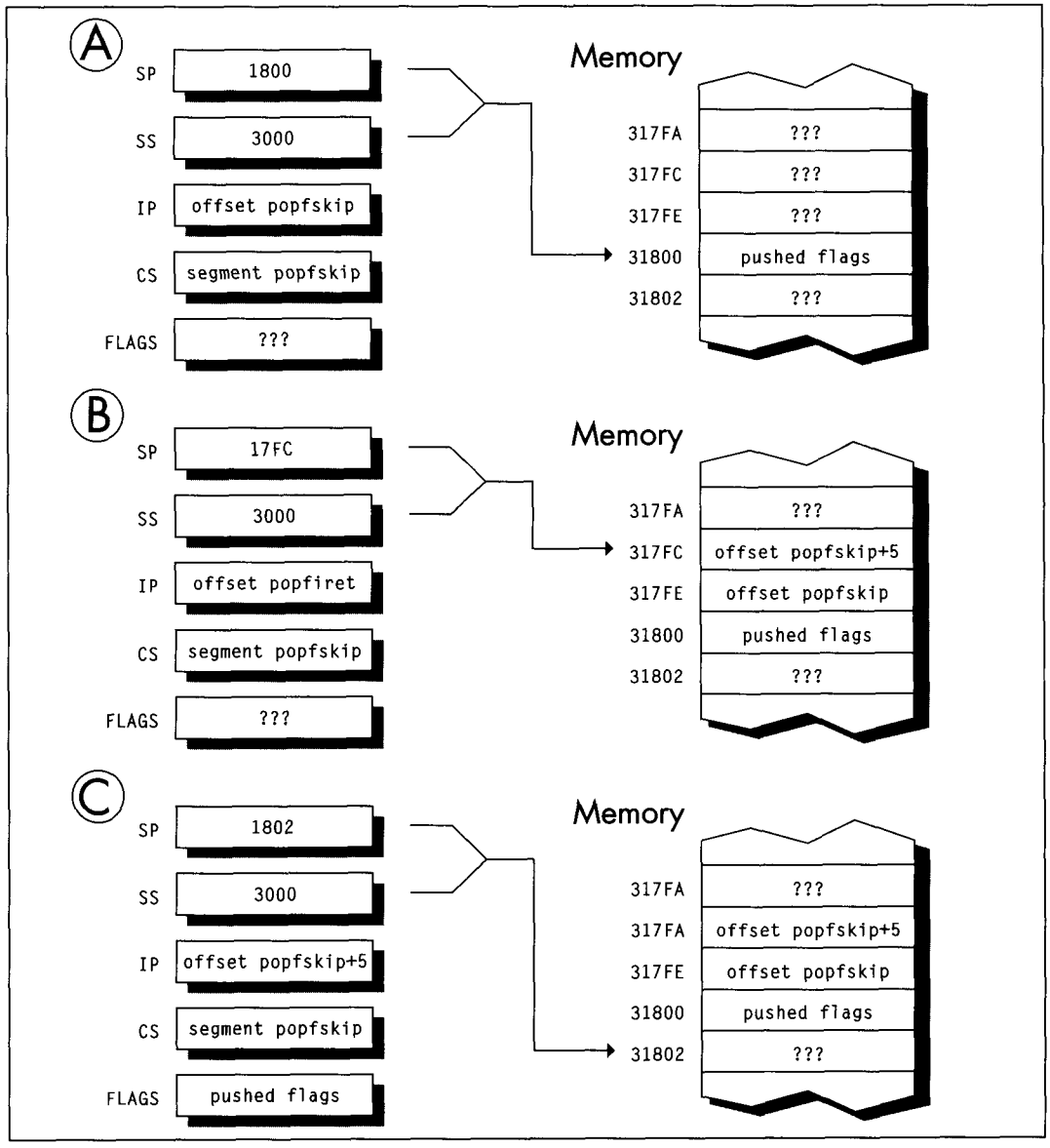
```
EMULATE_POPF_TRASH_AX  macro
      push  cs
      mov   ax,offset $+5
      push  ax
      iret
      endm
```

It's not a perfect substitute for **POPF**, since **POPF** doesn't alter any registers, but it's faster and shorter than **EMULATE_POPF** when you can spare the register. If you're using 286-specific instructions, you can use

```
      .286
            :
EMULATE_POPF      macro
      push  cs
      push  offset $+4
```

```
        iret
        endm
```

which is shorter still, alters no registers, and branches just once. (Of course, this version of **EMULATE_POPF** won't work on an 8088.)



Workaround code for the POPF bug.
Figure 11.6

The standard version of **EMULATE_POPF** is 6 bytes longer than **POPF** and much slower, as you'd expect given that it involves three branches. Anyone in his/her right mind would prefer **POPF** to a larger, slower, three-branch macro—given a choice. In non-interruptible code, however, there's no choice here; the safer—if slower—approach is the best. (Having people associate your programs with crashed computers is *not* a desirable situation, no matter how unfair the circumstances under which it occurs.)

And now you know the nature of and the workaround for the **POPF** bug. Whether you ever need the workaround or not, it's a neatly packaged example of the tremendous flexibility of the x86 instruction set.