Chapter 12:   Don't Jump!

Don't jump!

Sounds crazy, doesn't it?   After all, a computer is at heart a decision-making machine that decides by branching, and any programmer worth his salt knows that jumps, calls, interrupts, and loops are integral to any program of substance.   I've led you into some mighty strange places, including unlikely string instruction applications and implausible regions of the 8088's instruction set, to say nothing of the scarcely-comprehensible cycle-eaters.   Is it possible that I've finally tipped over the edge into sheer lunacy?

No such luck--I'm merely indulging in a bit of overstatement in a good cause.   Of course you'll need to branch...but since branching is slow--make that _very_ slow--on the 8088, you'll want to branch as little as possible. If you're clever, you can often manage to eliminate virtually all branching in the most time- critical portions of your code.   Sometimes avoiding branching is merely a matter of rearranging code, and sometimes it involves a few extra bytes and some unusual code.   Either way, code that's branch-free (or nearly so) is one key to high performance.

This business of avoiding branching--a term which covers jumps, subroutine calls, subroutine returns, and interrupts--is as much a matter of the flexible mind as of pure knowledge.   You may have noticed that in recent chapters we've discussed ways to use instructions more effectively as much as we've discussed the instructions themselves. For example, much of the last chapter was about how to put the string instructions to work in unorthodox but effective ways, not about how the string instructions work per se. It's inevitable that as we've accumulated a broad base of knowledge about the 8088 and gained a better sense of how to approach high-performance coding, we've developed an itch to put that hard-won knowledge to work in developing superior code. That's the flexible mind, and we'll see plenty of it over the next three chapters. Ultimately, we're building toward Volume II, which will focus on the flexible mind and implementation.

This chapter is emphatically not going to be a comprehensive discussion of all the ways to branch on the 8088. I started this book with the assumption that you were already familiar with assembly language, and we've spent many pages since then expanding your assembler knowledge. Chapter 6 discussed the flags that are tested by the various conditional jumps, and the last chapter used branching instructions in a variety of situations. By now I trust you know that **jz** branches if the zero flag is set to 1, and that **call** pushes the address of the next instruction on the stack and branches to the specified destination. If not, get a good reference book and study the various branching instructions carefully. There's nothing Zen in their functionality--they do what they're advertised to do, and that's that.

On the other hand, there is much Zen in the way the various branching instructions perform. In Chapter 13 we'll talk about ways to branch as little as possible, and in Chapter 14 we'll talk about ways to make branches perform as well as possible when you must use them. Right now, let's find out why it is that branching as little as possible is a desirable goal.

HOW SLOW IS IT?

We want to avoid branching for one simple reason: it's slow. It's not that there's anything inherently slow about branching; branching just happens to suffer from a slow implementation on the 8088. Even the venerable Z80 branches about 50% faster than the 8088.

So how slow is branching on the 8088? Well, the answer varies from one type of branch to another, so let's pick a commonly-used jump--say, **jmp**--and see what we find. The official execution time of **jmp** is 15 cycles. Listing 12-1, which measures the performance of 1000 **jmp** instructions in a row, reports that **jmp** actually takes 3.77 us (18 cycles) to execute. (Listing 12-1 actually uses **jmp short** rather than **jmp**, since the jumps don't cover much distance. We'll discuss the distinction between the two in a little while.)

18 cycles is a long time in anybody's book...long enough to copy a byte from one memory location to another and increment both SI and DI with **movsb**, long enough to add two 32-bit values together, long enough to increment a 16-bit register at least 4 times. How could it possibly take the 8088 so long just to load a value into the Instruction Pointer? (Think about it--all a branch really consists of is setting IP, and sometimes CS as well, to point to the desired instruction.) Well, let's round up the usual suspects--the cycle eaters--and figure out what's going on. In the process, we'll surely

acquire some knowledge that we can put to good use in creating high-performance code.

BRANCHING AND CALCULATION OF THE TARGET ADDRESS

Of the 18 cycles **jmp** takes to execute in Listing 12-1, 4 cycles seem to be used to calculate the target offset. I can't state this with absolute certainty, since Intel doesn't make the inner workings of its instructions public, but it's most likely true. You see, most of the 8088's **jmp** instructions don't have the form "load the Instruction Pointer with offset xxxx," where the **jmp** instruction specifies the exact offset to branch to. (This sort of jump is known as an absolute branch, since the destination offset is specified as a fixed, or absolute offset in the code segment. Figure 12-1 shows one of the few jump instructions that does use absolute branching.) Rather, most of the 8088's **jmp** instructions have the form "add nnnn to the contents of the Instruction Pointer," where the byte or word following the **jmp** opcode specifies the distance from the current IP to the offset to branch to, as shown in Figure 12-2.

Jumps that use displacements are known as relative branches, since the destination offset is specified relative to the offset of the current instruction. Relative branches are actually performed by adding a displacement to the value in the Instruction Pointer, and there's a bit of a trick there.

By the time a relative branching instruction actually gets around to branching, the IP points to the byte after the last byte of the instruction, since the IP has already been used to read in all the bytes of the branching instruction and has advanced to point to the next instruction. As shown in Figure 12-2, relative branches work by adding a displacement to the IP after it has advanced to point to the byte after the branching instruction, _not_ by adding a displacement to the offset of the branching instruction itself.

So, to sum up, most **jmp** instructions contain a field which specifies a displacement from the current IP to the target address, rather than a field which specifies the target address directly. (Jumps that _don't_ use relative branching include **jmp _reg16_**, **jmp _mem16_**, and all far jumps. All conditional jumps use relative branching.)

There are definite advantages to the use of relative rather than absolute branches. First, code that uses relative branching will work properly no matter where in memory it is loaded, since relative branch destinations aren't tied to specific memory offsets. If a block of code is moved to another area of memory, the relative displacements between the instructions remain the same, and so relative branching instructions will still work properly. This property makes relative branches useful in any code that must be moved about in memory, although by and large such code isn't needed very often.

Second (and more important), when relative branches are used, any branch whose target is within -128 to +127 bytes of the byte after the end of the branching instruction can be specified in a more compact form, with a 1-byte rather than 1-word displacement, as shown in Figure 12-3. The key, of course, is that -128 to +127 decimal is equivalent to 0FF80h to

007Fh hexadecimal, which is the range of values that can be specified with a single signed byte. The short jumps to which I referred earlier are such 1-byte-displacement short branches, in contrast to normal jumps, which use full 2-byte displacements. The smaller displacement allows short jump instructions to squeeze into 2 bytes, 1 byte less than a normal jump.

By definition, then, short branches take 1 less instruction byte than normal relative branches. The tradeoff is that short jumps can only reach offsets within the aforementioned 256- address range, while the 1-word displacement of normal branches allows them to reach any offset in the current code segment.

Since most branches are in fact to nearby addresses, the availability of short (1 displacement byte) branches can produce significant savings in code size. In fact, the 8088's conditional jumps can _only_ use 1-byte displacements, and while that's sometimes a nuisance when long conditional jumps need to be made, it does indeed help to keep code size down. There's also a definite disadvantage to the use of relative branches, and it's the usual drawback: speed, or rather the lack thereof. Adding a jump displacement to the Instruction Pointer is similar to adding a constant value to a register, a task which takes the 8088 4 cycles. By all appearances, it takes the 8088 about the same 4 cycles to add a jump displacement to the Instruction Pointer. Indeed, although there's no way to be sure exactly what's going on inside the 8088 during a **jmp**, it does make sense that the 8088 would use the same internal logic to add a constant to a register no matter whether the instruction causing the addition is a **jmp** or an **add**.

What's the evidence that the 8088 takes about 4 cycles to add a displacement to IP? Item 1: **jmp** **reg16**, an instruction which branches directly to the offset (not displacement) stored in a register, executes in just 11 cycles, 4 cycles faster than a normal **jmp**. Item 2: **jmp** **segment:offset**, the 8088's far jump that loads both CS and IP at once, executes at the same 15- cycles-per-execution speed as **jmp**. While a far jump requires that CS be loaded, it doesn't involve any displacement arithmetic. The addition of the displacement to IP pretty clearly takes longer than simply loading an offset into IP; otherwise it seems that a near jump would have to be faster than a far jump, by virtue of not having to load CS.

By the way, in this one instance it's acceptable to speculate on the basis on official execution times rather than on the basis of times reported by the Zen timer. Why? Because we're theorizing as to what's going on inside the 8088, and that's most accurately reflected by the official execution times, which ignore external data bus activity. Actual execution times include instruction fetching time, and since far jumps are 2 to 3 bytes longer than near jumps, the prefetch queue cycle-eater would obscure the comparison between the internal operations of near versus far jumps that we're trying to make. However, when it comes to evaluating real code performance, as opposed to speculating about the 8088's internal operations, you should always measure with the Zen timer.

Near subroutine calls (except **call** **reg16**) also use displacements, and, like near jumps, near calls seem to spend several cycles performing displacement arithmetic.   On the other hand, return instructions, which pop into IP offsets previously pushed on the stack by calls, do not perform displacement arithmetic, nor do far calls.   Interrupts don't perform displacement arithmetic either; as we will see, however, interrupts have their own performance problems.

Displacement arithmetic accounts for about 4 of the 18 cycles **jmp** takes to execute.   That leaves 14 cycles, still an awfully long time. What else is **jmp** doing to keep itself busy?

BRANCHING AND THE PREFETCH QUEUE

Since the actual execution time of **jmp** in Listing 12-1 is 3 cycles longer than its official execution time, one or more of the cycle-eaters must be taking those cycles.   If past experience is any guide, it's a pretty good bet that the prefetch queue cycle-eater is rearing its ugly head once again. The DRAM refresh cycle-eater may also be taking some cycles (it usually does), but the 20% discrepancy between the official and actual execution times is far too large to be explained by DRAM refresh alone.   In any case, let's measure the execution time of **jmp** with **imul** instructions interspersed so that the prefetch queue is full when it comes time for each **jmp** to execute.

First, let's figure out the execution time of **imul** when used to calculate the 32-bit product of two 16-bit zero factors. Later, that will allow us to determine how much of the combined execution time of **imul** and **jmp** is due to **imul** alone. (By the way, we're using **imul** rather than **mul** because when I tried **mul** and **jmp** together, overall execution synchronized with DRAM refresh, distorting the results. Each **mul/jmp** pair executed in exactly 144 cycles, with DRAM refresh adding 6 of those cycles by holding up instruction fetching right after the jump. Here we have yet another example of why you should always time code in context--be careful about generalizing from artificial tests like Listing 12-2!) The Zen timer reports that the 1000 **imul** instructions in Listing 12-2 execute in 26.82 ms, or 26.82 us (128 cycles) per **imul**.

Given that, we can determine how long **jmp** takes to execute when started with the prefetch queue full. Listing 12-3, which measures the execution time of alternating **imul** and **jmp** instructions, runs in 31.18 ms. That's 31.18 us (148.8 cycles) per **imul**/**jmp** pair, or 20.8 cycles per **jmp**.

Wait one minute! **jmp** takes more than 2 cycles longer when started with the prefetch queue full in Listing 12-3 than it did in Listing 12-1. Instructions don't slow down when the prefetch queue is allowed to fill before they start--if anything, they speed up. Yet a slowdown is just what we've found.

What the heck is going on?

THE PREFETCH QUEUE EMPTIES WHEN YOU BRANCH

It's true that the prefetch queue is full when it comes time for each **jmp** to start in Listing 12-3...but it's also true that the prefetch queue is empty when **jmp** ends. To understand why that is and what the implications are, we must consider the nature of the prefetch queue.

We learned way back in Chapter 3 that the Bus Interface Unit of the 8088 reads the bytes immediately following the current instruction into the prefetch queue whenever the external data bus isn't otherwise in use. This is done in an attempt to anticipate the next few instruction-byte requests that the Execution Unit will issue. Every time that the EU requests an instruction byte and the BIU has guessed right by prefetching that byte, 4 cycles are saved that would otherwise have to be expended on fetching the requested byte while the EU waited, as shown in Figure 12-4.

What happens if the BIU guesses wrong? Nothing disastrous: since the prefetched bytes are no help in fulfilling the EU's request, the requested instruction byte must be fetched from memory at a cost of 4 cycles, just as if prefetching had never occurred.

That leaves us with an obvious question. _When_ does the BIU guess wrong? In one case and one case only:

Whenever a branch occurs.

Think of it this way. The BIU prefetches bytes sequentially, starting with the byte after the instruction being executed. So long as no branches occur, those prefetched bytes _must_ be the bytes the EU will want next, since the Instruction Pointer simply marches along from low addresses to high addresses.

When a branch occurs, however, the bytes immediately following the instruction bytes for the branch instruction are no longer necessarily the next bytes the EU will want, as shown in Figure 12-5. If they aren't, the BIU has no choice but to throw away those bytes and start fetching bytes again at the location branched to. In other words, if the BIU gambles that the EU will request instruction bytes sequentially and loses that gamble because of a branch, all pending prefetches of the instruction bytes following the branch instruction in memory are wasted.

That doesn't make prefetching undesirable. The BIU prefetches only during idle times, so prefetching--even wasted prefetching--doesn't slow execution down. (At worst, prefetching might slow things down a bit by postponing memory accesses by a cycle or two--but whether and how often that happens, only Intel knows, since it's a function of the internal logic of the 8088. At any rate, wasted prefetching shouldn't greatly affect performance.) All that's lost when you branch is the performance bonus obtained when the 8088 manages to coprocess by prefetching and executing at the same time.

The 8088 could have been designed so that whenever a branch occurs, any bytes in the prefetch queue that are still usable are kept, while other, now-useless bytes are discarded. That would speed processing of code like:

```
		jz	Skip
		jmp	DistantLabel
Skip:
```

in the case where **jz** jumps, since the instruction byte at the label **Skip** might well be in the prefetch queue when the branch occurs. The 8088 could also have been designed to prefetch from both possible "next" instructions at a branch, so that the prefetch queue wouldn't be empty no matter which way the branch went.

The 8088 could have been designed to do all that and more-- but it wasn't. The BIU simply prefetches sequentially forward from the current instruction byte. Whenever a branch occurs, the prefetch queue is always emptied--even if the branched-to instruction is in the queue--and instruction fetching is started again at the new location, as illustrated by Figure 12-5. While that sounds innocent enough, it has far-reaching implications. After all, what does an empty prefetch queue mean? Right you are...

Branching always--and I do mean always--awakens the prefetch queue cycle-eater.

BRANCHING INSTRUCTIONS DO PREFETCH

Things aren't quite as bad as they might seem, however. As you'll recall, we decided back in Chapters 4 and 5 that the true execution time of an instruction is the interval from the time when the first byte of the instruction reaches the Execution Unit until the time when the first byte of the next instruction reaches the EU. Since branches always empty the prefetch queue, there obviously must be a 4-cycle delay from the time the branch is completed until the time when the first byte of the branched- to instruction reaches the EU, since that instruction byte must always be fetched from memory. In fact, the 8088 passes the first instruction byte fetched after a branch straight through to the EU as quickly as possible, since there's no question but what the EU is ready and waiting to execute that byte.

The designers of the 8088 seem to have agreed with our definition of "true" execution time. I've previously pointed out that Intel's official execution time for a given instruction doesn't include the time required to fetch the bytes of that instruction. That's not because Intel is hiding anything, but rather because the fetch time for a given instruction can vary considerably depending on the code preceding the instruction, as we've seen time and again. That's not quite the case with branching, however. Whenever a branch occurs, we can be quite certain that the prefetch queue will be emptied, and that at least one prefetch will occur before anything else happens.

What that means is that the 4 cycles required to fetch the first byte of the branched-to instruction can reliably be counted as part of the execution time of a branch, and that's exactly what Intel does. Although I've never seen documentation that explicitly states as much, official execution times that involve branches clearly include an extra 4 cycles for the fetching of the first byte of the branched-to instruction.

What evidence is there for this phenomenon? Well, Listing 12-1 is solid evidence. Listing 12-1 shows that a branching instruction (**jmp**) with an official execution time of 15 cycles actually executes in 18 cycles. If the official execution time didn't include the fetch time for the first byte of the branched- to instruction, repeated **jmp** instructions would take a minimum of 19 cycles to execute, consisting of 15 cycles of EU execution time followed by 4 cycles of BIU fetch time for the first byte of the next **jz**. In other words, the 18-cycle time that we actually measured could not happen if the 15-cycle execution time didn't include the 4 cycles required to fetch the first instruction byte at the branched-to location.

Ironically, branching instructions would superficially appear to be excellent candidates to <u>improve</u> the state of the prefetch queue. After all, **jmp** takes 15 cycles to execute, but accesses memory just once, to fetch the first byte of the branched-to instruction. Normally, such an instruction would allow 2 or 3 bytes to be prefetched, and, in fact, it's quite possible that 2 or 3 bytes <u>are</u> prefetched while **jmp** executes...but if that's true, then those prefetches are wasted. Any bytes that are prefetched during a **jmp** are thrown away at the end of the instruction, when the prefetch queue is emptied and the first byte of the instruction at the branched-to address is fetched.

So, the time required to fetch the branched-to instruction accounts for 4 cycles of the unusually long time the 8088 requires to branch. Once again, we've fingered the prefetch queue cycle-eater as a prime contributor to poor performance. You might think that for once the 8-bit bus isn't a factor; after all, the same emptying of the prefetch queue during each branch would occur on an 8086, wouldn't it?

The prefetch queue would indeed be emptied on an 8086--but it would refill much more rapidly. Remember, instructions are fetched a word at a time on the 16-bit 8086. In particular, one- half of the time the 4 cycles expended on the critical first fetch after a branch would fetch not 1 but 2 bytes on an 8086 (1 byte if the address branched to is odd, 2 bytes if it is even, since the 8086 can only read words that start at even addresses). By contrast, the 8088 can only fetch 1 byte during the final 4 cycles of a branch, and therein lies the answer to our mystery of how code could possibly slow down when started with the prefetch queue full.

BRANCHING AND THE SECOND BYTE OF THE BRANCHED-TO INSTRUCTION

Although the execution time of each branch includes the 4 cycles required to fetch the first byte of the branched-to instruction, that's not the end of the impact of branching on instruction fetching. When a branch instruction ends, the EU is just starting to execute the first byte of the branched-to instruction, the BIU is just starting to fetch the following instruction byte...and the prefetch queue is empty. In other words, the single instruction fetch built into the execution time of each branch doesn't fully account for the prefetch queue cycle-eater consequences of branching, but merely defers them for one byte. No matter how you look at it, the prefetch queue is flat-out empty after every branch.

Now, sometimes the prefetch queue doesn't eat a single additional cycle after a branching instruction fetches the first byte of the branched-to instruction. That happens when the 8088 doesn't need a second instruction byte for at least 4 cycles after the branch finishes, thereby giving the BIU enough time to fetch the second instruction byte. For example, consider Listing 12-4, which shows **jmp** (actually, **jmp short**, but we'll just use "**jmp**" for simplicity) instructions alternating with **push ax** instructions.

What's interesting about **push ax** is that it's a 1-byte instruction that takes 15 cycles to execute but only accesses memory twice, using just 8 cycles in the process. That means that after each branch, in the time during which **push ax** executes, there are 7 cycles free for prefetching the instruction bytes of the next **jmp**. That's long enough to fetch the opcode byte for **jmp**, and most of the displacement byte as well, and when **jmp** starts to execute, the BIU can likely finish fetching the displacement byte before it's needed. In Listing 12-4, in other words, the prefetch queue should never be empty either before or after **jmp** is executed, and that should make for faster execution.

Incidentally, **push** is a good instruction to start a subroutine with, in light of the beneficial prefetch queue effects described above. Why? Because **push** allows the 8088 to recover partially from the emptying of the prefetch queue caused by subroutine calls. By happy chance, pushing registers in order to preserve them is a common way to start a subroutine.

At any rate, let's try out our theories in the real world. Listing 12-4 runs in 6704 ms, or 32 cycles per **push ax**/**jmp** pair. **push ax** officially runs in 15 cycles, and since it's a "prefetch- positive" instruction--the prefetch queue tends to be more full when **push ax** finishes than when **push ax** starts--15 cycles should prove to be the actual execution time as well. Listing 12-5 confirms this, running in 3142 microseconds, or exactly 15 cycles per **push ax**.

A quick subtraction reveals that each **jmp** in Listing 12-4 takes 17 cycles. That's 1 cycle better than the execution time of **jmp** in Listing 12-1, and more than 3 cycles better than the execution time of **jmp** in Listing 12-3, confirming our speculations about post-branch prefetching. It seems that we have indeed found the answer to the mystery of how **jmp** can run slower when the prefetch queue is allowed to fill before **jmp** is started: because the prefetch queue is emptied after a branch, one or more instructions following a branch can suffer from reduced performance at the hands of the prefetch queue cycle- eater. The fetch time for the first instruction byte after the branch is built into the branch, but not the fetch time for the second byte, or the bytes after that.

So exactly what happens when Listing 12-3 runs to slow performance by 3-plus cycles relative to Listing 12-4? I can only speculate, but it seems likely that when the first byte of an **imul** instruction is fetched, the EU is ready for the second byte of the **imul**--the mod-reg-rm byte--after just 1 cycle, as shown in Figure 12-6. After all, the EU can't do much processing of a multiplication until the source and destination are known, so it makes sense that the mod-reg-rm byte would be needed right away. Unfortunately, the branch preceding each **imul** in Listing 12-3 empties the prefetch queue, so the EU must wait for several cycles while the mod-reg-rm byte is fetched from memory.

In Listing 12-4, on the other hand, the first byte fetched after each branch is the instruction byte for **push ax**. Since that's the only byte of the instruction, the EU can proceed right through to completion of the instruction without requiring additional instruction bytes, affording ample time for the BIU to fetch at least the first byte of the next **jmp**, as shown in Figure 12-7. As a result, the prefetch queue cycle-eater has little or no impact on the performance of this code.

Finally, the code in Listing 12-1 falls somewhere between Listings 12-3 and 12-4 as regards post-branch prefetching. Presumably, the EU has a more immediate need for the mod-reg-rm byte when executing **imul** than it does for the displacement byte when executing **jmp**.

Each **push ax**/**jmp** pair in Listing 12-4 still takes 2 cycles longer than it should according to the official execution times, so at least one cycle-eater must still be active. Perhaps the prefetch queue cycle-eater is still taking 2 cycles, or perhaps the DRAM refresh cycle-eater is taking 1 cycle and the prefetch queue cycle-eater is taking another cycle. There's really no way to tell where those 2 cycles are going without getting out hardware and watching the 8088 run--and it's not worth worrying about anyway.

In the grand scheme of things, it matters not a whit which cycle-eater is taking what portion of the cycles in Listings 12- 1, 12-3, and 12-4. Even if it did matter, there's no point to trying to understand exactly how the prefetch queue behaves after branching. The detailed behavior of the cycle-eaters is highly variable in real code, and is extremely difficult to pin down precisely. Moreover, that behavior depends on the internal logic of the 8088, which is forever hidden from our view.

What is important is that you understand that the true execution times of branching instructions are almost always longer than the official times because the prefetch queue is guaranteed to be empty after each and every branch. True, the fetch time for the first instruction byte after a branch is accounted for in official branching execution times (making those times very slow). However, the prefetch queue is still empty after that first byte is fetched and begins execution, and the time the Execution Unit usually spends waiting for subsequent bytes to arrive is not accounted for in the official execution times.

Sometimes, as in Listing 12-4, there may be no further instruction-fetch penalty following a branch, but those circumstances are few and far between, since they require that a branch be followed by an instruction byte that causes the 8088 not to require another instruction byte for at least 4 cycles. The truth of the matter is that it took me a bit of searching to find an instruction (**push ax**) that met that criterion. In real code, branching almost always incurs a delayed prefetch penalty.

It's this simple. Branches empty the prefetch queue. Many of the 8088's fastest instructions run well below their maximum speed when the prefetch queue is empty, and most instructions slow down at least a little. It stands to reason, then, that branches reduce the performance of the branched-to code, with the reduction most severe for the sort of high-performance code we're most interested in writing.

DON'T JUMP!

Slow as they seem from the official execution times, branches are actually even slower than that, since they put the PC in just the right state for the prefetch queue cycle-eater to do its worst. Every time you branch, you expend at least 11 cycles, and usually more...and then you're left with an empty prefetch queue. Is that the sort of instruction you want mucking up your time-critical code? Hardly. I'll say it again:

Don't jump!

NOW THAT WE KNOW WHY NOT TO BRANCH...

We've accounted for 11 of the 18 cycles that **jmp** takes to execute in Listing 12-1: 4 cycles to perform displacement arithmetic, 4 cycles to fetch the first byte of the next **jmp**, and 3 cycles lost to the prefetch queue cycle-eater after the branch empties the queue. (Some of that 3-cycle loss may be due to DRAM refresh as well.)

That leaves us with 7 cycles unaccounted for. One of those cycles goes to decoding the instruction, but frankly I'm not certain where the other 6 go. The 8088 has to load the IP with the target address and empty the prefetch queue, but I wouldn't expect that to take 6 cycles; more like 1 cycle, or 2 at most. Several additional cycles may go to calculating the 20-bit address at which to fetch the first byte of the branched-to instruction. In fact, that's a pretty good bet: the 8088 takes a minimum of 5 cycles to perform effective address calculations, which would neatly account for most of the remaining 6 cycles. However, I don't know for sure that that's the case, and probably never will.

No matter. We've established where the bulk of the time goes when a **jmp** occurs, and in the process we've found that branches are slow indeed--even slower than documented, thanks to the prefetch queue cycle-eater. In other words, we've learned why it's desirable not to branch in high-performance code. Now it's time to find out how to go about that unusual but essential task.