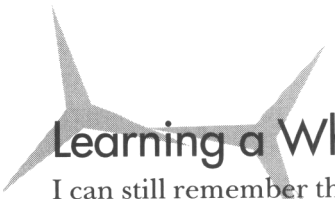
Chapter 19

Pentium: Not the Same Old Song

Chapter 19



Learning a Whole Different Set of Optimization Rules

I can still remember the day I did my first 8088 programming. I had just moved over from the distantly related Z80, so the 8088 wasn't totally alien, but it was nonetheless an incredibly exciting processor. The 8088's instruction set was vastly more powerful and varied than the Z80's, and as someone who thrives on puzzles of all sorts, from crosswords to Freecell to jigsaws to assembly language optimization, I was delighted to find that the 8088 made the optimization universe an order of magnitude more complicated—and correspondingly more interesting.

Well, the years went by and the Z80 just died, and 8088 optimization got ever more complex and intriguing as I discovered the hazards of the 8088's cycle-eaters. By the time 1989 rolled around, I had written *Zen of Assembly Language*, in which I described all that I had learned about the 8088 and concluded that 8088 optimization was a black art of infinite subtlety. Unfortunately, by that time the 286 was the standard, with the 386 coming on strong, and if the 286 was less amenable to hand optimization than the 8088 (and it surely was), then the 386 was downright unfriendly. Sure, assembly optimization could buy some performance on the 386, but only 20, 30, 40 percent or so—a far cry from the 100 to 400 percent of the 8088. At the same time, compiler technology was improving quickly, and the days of hand tuning seemed numbered.

Happily, the 486 traveled to the beat of a different drum. The 486 had some interesting internal pipeline hazards, as well as an internal cache that made cycle counting more meaningful than ever before, and careful code massaging sometimes yielded

startling results. Nonetheless, the 486 was still too simple to mark a return to the golden age of optimization.

The Return of Optimization as Art

Then the Pentium came around, and filled our code with optimization hazards, and life was good again. The Pentium has two execution pipelines and enough rules and exceptions to those rules to bring joy to the heart of the hardest-core assembly junkie. For a change, Intel documented most of the Pentium optimization rules and spread the word about them, so we don't have to go through as much spelunking of the Pentium as with its predecessors. They've done this, I suspect, largely because more than any previous x86 processor, the Pentium's performance is highly dependent on properly optimized code.

In the worst case, where the second execution pipe is dormant most of the time, the Pentium won't perform all that much better than a 486 at the same clock speed. In the best case, where the second pipe is heavily used and the Pentium's other advantages (such as branch prediction, write-back cache, 64-bit full speed external bus, and dual 8K caches) can kick in, the Pentium can be more than twice as fast as a 486. In a critical inner loop, hand optimization can double or even triple performance over 486-optimized code—and that's on top of the sorts of algorithmic and design optimizations that are routinely performed on any processor. Good compilers can make a big difference on the Pentium, too, but there are some gotchas there, to which I'll return later.

It's been a long time coming, but hard-core, big-payoff assembly language optimization is back in style, and for the rest of this book I'll be delving into the Byzantine wonders of the Pentium. In this chapter, I'll do a quick overview, then cover a variety of smaller Pentium optimization topics. In the next chapter, I'll tackle the 900-pound gorilla of Pentium optimization: superscalar (dual execution pipe) programming. Trust me, this'll be fun.

Listen, do you want to know a secret? This lead-in has been brought to you with the help of "classic rock"—another way of saying "music Baby Boomers listened to back when they cared more about music than 401Ks and regular flossing." There are so many of us Boomers that our music, even the worst of it, will never go away. When we're 90 years old, propped up in our Kraftmatic adjustable beds and surfing the 5,000-channel information superhighway from one infomercial to the next, the sound system in the retirement community will be piping in a Muzak version of "Louie, Louie," while on the holovid Country Joe McDonald and the Fish pitch Preparation H. I can hardly wait.

Gimme a "P"....

The Pentium: An Overview

Architecturally, the Pentium is vastly different in many ways from the 486, but most of those differences are transparent to programmers. After all, the whole idea behind the Pentium is that it runs the same code as previous x86 processors, but faster; otherwise, Intel could have made a faster, cheaper RISC processor. Still, knowledge of the Pentium's architecture is useful for understanding exactly how code will perform, and a few of the architectural differences are most decidedly *not* transparent to performance programmers.

The Pentium is essentially one full 486 execution unit (EU), plus a second stripped-down 486 EU, on a single chip. The first EU is referred to as the U execution pipe, or *U-pipe;* the second, more limited one is called the *V-pipe.* The two pipes are capable of executing instructions simultaneously, have separate write buffers, and can even access the data cache simultaneously (although with certain limitations that I'll discuss in the next chapter), so on the Pentium it is possible to execute two instructions, even instructions that access memory, in a single clock. The cycle times for instruction execution in a given pipe (both pipes process instructions at the same speed) are comparable to those for the 486, although some instructions—notably **MUL**, the repeated string instructions, and some of the shifts and rotates—have gotten faster.

My first thought upon hearing of the Pentium's dual pipes was to wonder how often the prefetch queue stalls for lack of instruction bytes, given that the demand for instruction bytes can be twice that of the 486. The answer is: rarely indeed, and then only because the code is not in the internal cache. The 486 has a single 8K cache that stores both code and data, and prefetching can stall if data fetching doesn't allow time for prefetching to occur (although this rarely happens in practice).

The Pentium, on the other hand, has two separate 8K caches, one for code and one for data, so code prefetches can never collide with data fetches; the prefetch queue can stall only when the code being fetched isn't in the internal code cache.

(And yes, self-modifying code still works; as with all Pentium changes, the dual caches introduce no incompatibilities with 386/486 code.) Also, because the code and data caches are separate, code can't be driven out of the cache in a tight loop that accesses a lot of data, unlike the 486. In addition, the Pentium expands the 486's 32-byte prefetch queue to 128 bytes. In conjunction with the branch prediction feature (described next), which allows the Pentium to prefetch properly at most branches, this larger prefetch queue means that the Pentium's two pipes should be better fed than those of any previous x86 processor.

Crossing Cache Lines

There are three other characteristics of the Pentium that make for a healthy supply of instruction bytes. One is that the Pentium can prefetch instructions across cache

lines. Unlike the 486, where there is a 3-cycle penalty for branching to an instruction that spans a cache line, there's no such penalty on the Pentium. The second is that the cache line size (the number of bytes fetched from the external cache or main memory on a cache miss) on the Pentium is 32 bytes, twice the size of the 486's cache line, so a cache miss causes a longer run of instructions to be placed in the cache than on the 486. The third is that the Pentium's external bus is twice as wide as the 486's, at 64 bits, and runs twice as fast, at 66 MHz, so the Pentium can fetch both instruction and data bytes from the external cache four times as fast as the 486.

Even when the Pentium is running flat-out with both pipes in use, it can generally consume only about twice as many bytes as the 486; so the ratio of external memory bandwidth to processing power is much improved, although real-world performance is heavily dependent on the size and speed of the external cache.

The upshot of all this is that at the same clock speed, with code and data that are mostly in the internal caches, the Pentium maxes out somewhere around twice as fast as a 486. (When the caches are missed a lot, the Pentium can get as much as three to four times faster, due to the superior external bus and bigger caches.) Most of this won't affect how you program, but it is useful to know that you don't have to worry about instruction fetching. It's also useful to know the sizes of the caches because a high cache hit rate is crucial to Pentium performance. Cache misses are vastly slower than cache hits (anywhere from two to 50 or more times as slow, depending on the speed of the external cache and whether the external cache misses as well), and the Pentium can't use the V-pipe on code that hasn't already been executed out of the cache at least once. This means that it is *very* important to get the working sets of critical loops to fit in the internal caches.

One change in the Pentium that you definitely do have to worry about is superscalar execution. Utilization of the V-pipe can range from near zero percent to 100 percent, depending on the code being executed, and careful rearrangement of code can have amazing effects. Maxing out V-pipe use is not a trivial task; I'll spend all of the next chapter discussing it so as to have time to cover it properly. In the meantime, two good references for superscalar programming and other Pentium information are Intel's *Pentium Processor User's Manual: Volume 3: Architecture and Programming Manual* (ISBN 1-55512-195-0; Intel order number 241430-001), and the article "Optimizing Pentium Code" by Mike Schmidt, in *Dr. Dobb's Journal* for January 1994.

Cache Organization

There are two other interesting changes in the Pentium's cache organization. First, the cache is two-way set-associative, whereas the 486 is four-way set-associative. The details of this don't matter, but simply put, this, combined with the 32-byte cache line size, means that the Pentium has somewhat coarser granularity in both space and time than the 486 in terms of packing bytes into the cache, although the total

cache space is now bigger. There's nothing you can do about this, but it may make it a little harder to get a loop's working set into the cache. Second, the internal cache can now be configured (by the BIOS or OS; you won't have to worry about it) for write-back rather than write-through operation. This means that writes to the internal data cache don't necessarily get propagated to the external bus until other demands for cache space force the data out of the cache, making repeated writes to memory variables such as loop counters cheaper on average than on the 486, although not as cheap as registers.

As a final note on Pentium architecture for this chapter, the pipeline stalls (what Intel calls AGIs, for *Address Generation Interlocks*) that I discussed earlier in this book (see Chapter 12) are still present in the Pentium. In fact, they're there in spades on the Pentium; the two pipelines mean that an AGI can now slow down execution of an instruction that's *three* instructions away from the AGI (because four instructions can execute in two cycles). So, for example, the code sequence

```
add edx,4     ;U-pipe cycle 1
mov ecx,[ebx] ;V-pipe cycle 1
add ebx,4     ;U-pipe cycle 2
mov [edx],ecx ;V-pipe cycle 3
              ; due to AGI
              ; (would have been
              ; V-pipe cycle 2)
```

takes three cycles rather than the two cycles it should take, because EDX was modified on cycle 1 and an attempt was made to use it on cycle two, before the AGI had time to clear—even though there are two instructions between the instructions that are actually involved in the AGI. Rearranging the code like

```
mov ecx,[ebx]   ;U-pipe cycle 1
add ebx,4       ;V-pipe cycle 1
mov [edx+4],ecx ;U-pipe cycle 2
add edx,4       ;V-pipe cycle 2
```

makes it functionally identical, but cuts the cycles to 2—a 50 percent improvement. Clearly, avoiding AGIs becomes a much more challenging and rewarding game in a superscalar world, one to which I'll return in the next chapter.

Faster Addressing and More

I'll spend the rest of this chapter covering a variety of Pentium optimization tips. For starters, effective address calculations (that is, the addition and scaling required to calculate a memory operand's address, as for example in **MOV EAX,[EBX+ECX*2+4]**) never take any extra cycles on the Pentium (other than possibly an AGI cycle), even for the use of base+index addressing (as in **MOV [ESI+EDI],EAX**) or scaling (*2, *4, or *8, as in **INC ARRAY[ESI*4]**). On the 486, both of the latter cases cause a 1-cycle penalty. The faster effective address calculations have the side effect of making **LEA** very attractive as an arithmetic instruction. **LEA** can add any two registers, one of

which can be multiplied by one, two, four, or eight, plus a constant value, and can store the result in any register—all in one cycle, apart from AGIs. Not only that, but as we'll see in the next chapter, **LEA** can go through either pipe, whereas **SHL** can only go through the U-pipe, so **LEA** is often a superior choice for multiplication by three, four, five, eight, or nine. (**ADD** is the best choice for multiplication by two.) If you use **LEA** for arithmetic, do remember that unlike **ADD** and **SHL**, it doesn't modify any flags.

As on the 486, memory operands should not cross any more alignment boundaries than absolutely necessary. Word operands should be word-aligned, dword operands should be dword-aligned, and qword operands (double-precision variables) should be qword-aligned. Spanning a dword boundary, as in

```
mov ebx,3
     :
mov eax,[ebx]
```

costs three cycles. On the other hand, as noted above, branch targets can now span cache lines with impunity, so on the Pentium there's no good argument for the paragraph (that is, 16-byte) alignment that Intel recommends for 486 jump targets. The 32-byte alignment might make for slightly more efficient Pentium cache usage, but would make code much bigger overall.

In fact, given that most jump targets aren't in performance-critical code, it's hard to make a compelling argument for aligning branch targets even on the 486. I'd say that no alignment (except possibly where you know a branch target lies in a key loop), or at most dword alignment (for the 386) is plenty, and can shrink code size considerably.

Instruction prefixes are awfully expensive; avoid them if you can. (These include size and addressing prefixes, segment overrides, **LOCK**, and the 0FH prefixes that extend the instruction set with instructions such as **MOVSX**. The exceptions are conditional jumps, a fast special case.) At a minimum, a prefix byte generally takes an extra cycle and shuts down the V-pipe for that cycle, effectively costing as much as two normal instructions (although prefix cycles can overlap with previous multicycle instructions, or AGIs, as on the 486). This means that using 32-bit addressing or 32-bit operands in a 16-bit segment, or vice versa, makes for bigger code that's significantly slower. So, for example, you should generally avoid 16-bit variables (shorts, in C) in 32-bit code, although if using 32-bit variables where they're not needed makes your data space get a lot bigger, you may want to stick with shorts, especially since longs use the cache less efficiently than shorts. The trade-off depends on the amount of data and the number of instructions that reference that data. (eight-bit variables, such as chars, have no extra overhead and can be used freely, although they may be less desirable than longs for compilers that tend to promote variables to longs when performing calculations.) Likewise, you should if possible avoid putting data in the code segment and referring to it with a CS: prefix, or otherwise using segment overrides.

LOCK is a particularly costly instruction, especially on multiprocessor machines, because it locks the bus and requires that the hardware be brought into a synchronized state. The cost varies depending on the processor and system, but **LOCK** can make an **INC** [*mem*] instruction (which normally takes 3 cycles) 5, 10, or more cycles slower. Most programmers will never use **LOCK** on purpose—it's primarily an operating system instruction—but there's a hidden gotcha here because the **XCHG** instruction *always* locks the bus when used with a memory operand.

__XCHG__ is a tempting instruction that's often used in assembly language; for example, exchanging with video memory is a popular way to read and write VGA memory in a single instruction—but it's now a bad idea. As it happens, on the 486 and Pentium, using __MOV__s to read and write memory is faster, anyway; and even on the 486, my measurements indicate a five-cycle tax for __LOCK__ in general, and a nine-cycle execution time for __XCHG__ with memory. Avoid __XCHG__ with memory if you possibly can.

As with the 486, don't use **ENTER** or **LEAVE**, which are slower than the equivalent discrete instructions. Also, start using **TEST** ***reg,reg*** instead of **AND** ***reg,reg*** or **OR** ***reg,reg*** to test whether a register is zero. The reason, as we'll see in Chapter 21, is that **TEST**, unlike **AND** and **OR**, never modifies the target register. Although in this particular case **AND** and **OR** don't modify the target register either, the Pentium has no way of knowing that ahead of time, so if **AND** or **OR** goes through the U-pipe, the Pentium may have to shut down the V-pipe for a cycle to avoid potential dependencies on the result of the **AND** or **OR**. **TEST** suffers from no such potential dependencies.

Branch Prediction

One brand-spanking-new feature of the Pentium is *branch prediction*, whereby the Pentium tries to guess, based on past history, which way (or, for conditional jumps, whether or not), your code will jump at each branch, and prefetches along the likelier path. If the guess is correct, the branch or fall-through takes only 1 cycle—2 cycles less than a branch and the same as a fall-through on the 486; if the guess is wrong, the branch or fall-through takes 4 or 5 cycles (if it executes in the U- or V-pipe, respectively)—1 or 2 cycles more than a branch and 3 or 4 cycles more than a fall-through on the 486.

Branch prediction is unprecedented in the x86, and fundamentally alters the nature of pedal-to-the-metal optimization, for the simple reason that it renders unrolled loops largely obsolete. Rare indeed is the loop that can't afford to spare even 1 or 0 (yes, zero!) cycles per iteration for loop counting, and that's how low the cost can go for maintaining a loop on the Pentium.

Also, unrolled loops are bigger than normal loops, so there are extra (and expensive) cache misses the first time through the loop if the entire loop isn't already in

the cache; then, too, an unrolled loop will shoulder other code out of the internal and external caches. If in a critical loop you absolutely need the time taken by the loop control instructions, or if you need an extra register that can be freed by unrolling a loop, then by all means unroll the loop. Don't expect the sort of speed-up you get from this on the 486 or especially the 386, though, and watch out for the cache effects.

You may well wonder exactly *when* the Pentium correctly predicts branching. Alas, this is one area that Intel has declined to document, beyond saying that you should endeavor to fall through branches when you have a choice. That's good advice on every other x86 processor, anyway, so it's well worth following. Also, it's a pretty safe bet that in a tight loop, the Pentium will start guessing the right branch direction at the bottom of the loop pretty quickly, so you can treat loop branches as one-cycle instructions.

It's an equally safe bet that it's a bad move to have in a loop a conditional branch that goes both ways on a random basis; it's hard to see how the Pentium could consistently predict such branches correctly, and mispredicted branches are more expensive than they might appear to be. Not only does a mispredicted branch take 4 or 5 cycles, but the Pentium can potentially execute as many as 8 or 10 instructions in that time—3 times as many as the 486 can execute during its branch time—so correct branch prediction (or eliminating branch instructions, if possible) is very important in inner loops. Note that on the 486 you can count on a branch to take 1 cycle when it falls through, but on the Pentium you can't be sure whether it will take 1 or either 4 or 5 cycles on any given iteration.

As things currently stand, branch prediction is an annoyance for assembly language optimization because it's impossible to be certain exactly how code will perform until you measure it, and even then it's difficult to be sure exactly where the cycles went. All I can say is try to fall through branches if possible, and try to be consistent in your branching if not.

Miscellaneous Pentium Topics

The Pentium has all the instructions of the 486, plus a few new ones. One much-needed instruction that has finally made it into the instruction set is **CPUID**, which allows your code to determine what processor it's running on. **CPUID** is 15 years late, but at least it's finally here. Another new instruction is **CMPXCHG8B**, which does a compare and conditional exchange on a qword. **CMPXCHG8B** doesn't seem to me to be a particularly useful instruction, but I'm sure Intel wouldn't have added it without a reason; if you know of a use for it, please pass it along to me.

486 versus Pentium Optimization

Many Pentium optimizations help, or at least don't hurt, on the 486. Many, but not all—and many *do* hurt on the 386. As I discuss various Pentium optimizations, I will attempt to note the effects on the 486 as well, but doing this in complete detail

would double the sizes of these discussions and make them hard to follow. In general, I'd recommend reserving Pentium optimization for your most critical code, and even there, it's a good idea to have at least two code paths, one for the 386 and one for the 486/Pentium. It's also a good idea to time your code on a 486 before and after Pentium-optimizing it, to make sure you haven't hurt performance on what will be, after all, by far the most important processor over the next couple of years.

With that in mind, is optimizing for the Pentium even worthwhile today? That depends on your application and its market—but if you want absolutely the best possible performance for your DOS and Windows apps on the fastest hardware, Pentium optimization can make your code *scream.*

Going Superscalar

In the next chapter, we'll look into the single biggest element of Pentium performance, cranking up the Pentium's second execution pipe. This is the area in which compiler technology is most touted for the Pentium, the two thoughts apparently being that (1) most existing code is in C, so recompiling to use the second pipe better is an automatic win, and (2) it's so complicated to optimize Pentium code that only a compiler can do it well. The first point is a reasonable one, but it does suffer from one flaw for large programs, in that Pentium-optimized code is larger than 486- or 386-optimized code, for reasons that will become apparent in the next chapter. Larger code means more cache misses and more page faults; and while most of the code in any program is not critical to performance, compilers optimize code indiscriminately.

The result is that Pentium compiler optimization not only expands code, but can be less beneficial than expected or even slower in some cases. What makes more sense is enabling Pentium optimization *only* for key code. Better yet, you could hand-tune the most important code—and yes, you can absolutely do a better job with a small, critical loop than any PC compiler I've ever seen, or expect to see. Sure, you keep hearing how great each new compiler generation is, and compilers certainly have improved; but they play by the same rules we do, and we're more flexible and know more about what we're doing—and now we have the wonderfully complex and powerful Pentium upon which to loose our carbon-based optimizers.

A compiler that generates better code than a good assembly programmer? That'll be the day.