Chapter 12

Pushing the 486

Chapter 12

It's Not Just a Bigger 386

So this traveling salesman is walking down a road, and he sees a group of men digging a ditch with their bare hands. "Whoa, there!" he says. "What you guys need is a Model 8088 ditch digger!" And he whips out a trowel and sells it to them.

A few days later, he stops back around. They're happy with the trowel, but he sells them the latest ditch-digging technology, the Model 80286 spade. That keeps them content until he stops by again with a Model 80386 shovel (a full 32 inches wide, with a narrow point to emulate the trowel), and *that* holds them until he comes back around with what they really need: a Model 80486 bulldozer.

Having reached the top of the line, the salesman doesn't pay them a call for a while. When he does, not only are they none too friendly, but they're digging with the 80386 shovel; the bulldozer is sitting off to one side. "Why on earth are you using that shovel?" the salesman asks. "Why aren't you digging with the bulldozer?"

"Well, Lord knows we tried," says the foreman, "but it was all we could do just to lift the damn thing!"

Substitute "processor" for the various digging implements, and you get an idea of just how different the optimization rules for the 486 are from what you're used to. Okay, it's not quite *that* bad—but upon encountering a processor where string instructions are often to be avoided and memory-to-register **MOVs** are frequently as fast as register-to-register **MOVs**, Dorothy was heard to exclaim (before she sank out

of sight in a swirl of hopelessly mixed metaphors), "I don't think we're in Kansas anymore, Toto."

Enter the 486

No chip that is a direct, fully compatible descendant of the 8088, 286, and 386 could ever be called a RISC chip, but the 486 certainly contains RISC elements, and it's those elements that are most responsible for making 486 optimization unique. Simple, common instructions are executed in a single cycle by a RISC-like core processor, but other instructions are executed pretty much as they were on the 386, where every instruction takes at least 2 cycles. For example, **MOV AL, [TestChar]** takes only 1 cycle on the 486, assuming both instruction and data are in the cache—3 cycles faster than the 386—but **STOSB** takes 5 cycles, 1 cycle *slower* than on the 386. The floating-point execution unit inside the 486 is also much faster than the 387 math coprocessor, largely because, being in the same silicon as the CPU (the 486 has a math coprocessor built in), it is more tightly coupled. The results are sometimes startling: **FMUL** (floating point multiply) is usually faster on the 486 than **IMUL** (integer multiply)!

An encyclopedic approach to 486 optimization would take a book all by itself, so in this chapter I'm only going to hit the highlights of 486 optimization, touching on several optimization rules, some documented, some not. You might also want to check out the following sources of 486 information: *i486 Microprocessor Programmer's Reference Manual,* from Intel; "8086 Optimization: Aim Down the Middle and Pray," in the March, 1991 *Dr. Dobb's Journal*; and "Peak Performance: On to the 486," in the November, 1990 *Programmer's Journal.*

Rules to Optimize By

In Appendix G of the *i486 Microprocessor Programmer's Reference Manual,* Intel lists a number of optimization techniques for the 486. While neither exhaustive (we'll look at two undocumented optimizations shortly) nor entirely accurate (we'll correct two of the rules here), Intel's list is certainly a good starting point. In particular, the list conveys the extent to which 486 optimization differs from optimization for earlier x86 processors. Generally, I'll be discussing optimization for real mode (it being the most widely used mode at the moment), although many of the rules should apply to protected mode as well.

486 optimization is generally more precise and less frustrating than optimization for other x86 processors because every 486 has an identical internal cache. Whenever both the instructions being executed and the data the instructions access are in the cache, those instructions will run in a consistent and calculatable number of cycles on all 486s, with little chance of interference from the prefetch queue and without regard to the speed of external memory.

In other words, for cached code (which time-critical code almost always is), performance is predictable and can be calculated with good precision, and those calculations will apply on any 486. However, "predictable" doesn't mean "trivial"; the cycle times printed for the various instructions are not the whole story. You must be aware of all the rules, documented and undocumented, that go into calculating actual execution times—and uncovering some of those rules is exactly what this chapter is about.

The Hazards of Indexed Addressing

Rule #1: Avoid indexed addressing (that is, try not to use either two registers or scaled addressing to point to memory).

Intel cautions against using indexing to address memory because there's a one-cycle penalty for indexed addressing. True enough—but "indexed addressing" might not mean what you expect.

Traditionally, SI and DI are considered the index registers of the x86 CPUs. That is not the sense in which "indexed addressing" is meant here, however. In real mode, indexed addressing means that two registers, rather than one or none, are used to point to memory. (In this context, the use of one register to address memory is "base addressing," no matter what register is used.) **MOV AX, [BX+DI]** and **MOV CL, [BP+SI+10]** perform indexed addressing; **MOV AX,[BX]** and **MOV DL, [SI+1]** do not.

Therefore, in real mode, the rule is to avoid using two registers to point to memory whenever possible. Often, this simply means adding the two registers together outside a loop before memory is actually addressed.

As an example, you might adhere to this rule by replacing the code

```
LoopTop:
     add   ax,[bx+si]
     add   si,2
     dec   cx
     jnz   LoopTop
```

with this

```
     add   si,bx
LoopTop:
     add   ax,[si]
     add   si,2
     dec   cx
     jnz   LoopTop
     sub   si,bx
```

which calculates the same sum and leaves the registers in the same state as the first example, but avoids indexed addressing.

In protected mode, the definition of indexed addressing is a tad more complex. The use of two registers to address memory, as in **MOV EAX, [EDX+EDI]**, still qualifies

for the one-cycle penalty. In addition, the use of 386/486 scaled addressing, as in **MOV [ECX*2],EAX**, also constitutes indexed addressing, even if only one register is used to point to memory.

All this fuss over one cycle! You might well wonder how much difference one cycle could make. After all, on the 8088, effective address calculations take a *minimum* of 5 cycles. On the 486, however, 1 cycle is a big deal because many instructions, including most register-only instructions (**MOV**, **ADD**, **CMP**, and so on) execute in just 1 cycle. In particular, **MOV**s to and from memory execute in 1 cycle—if they're not hampered by something like indexed addressing, in which case they slow to half speed (or worse, as we will see shortly).

For example, consider the summing example shown earlier. The version that uses base+index ([BX+SI]) addressing executes in eight cycles per loop. As expected, the version that uses base ([SI]) addressing runs one cycle faster, at seven cycles per loop. However, the loop code executes so fast on the 486 that the single cycle saved by using base addressing makes the *whole loop* more than 14 percent faster.

In a key loop on the 486, 1 cycle can indeed matter.

Calculate Memory Pointers Ahead of Time

Rule #2: Don't use a register as a memory pointer during the next two cycles after loading it.

Intel states that if the destination of one instruction is used as the base addressing component of the next instruction, then a one-cycle penalty is imposed. This rule, unlike anything ever before seen in the x86 family, reflects the heavily pipelined nature of the 486. Apparently, the 486 starts each effective address calculation before the start of the instruction that will need it, as shown in Figure 12.1; this effectively makes the address calculation time vanish, because it happens while the preceding instruction executes.

Of course, the 486 *can't* perform an effective address calculation for a target instruction ahead of time if one of the address components isn't known until the instruction starts, and that's exactly the case when the preceding instruction modifies one of the target instruction's addressing registers. For example, in the code

```
MOV   BX,OFFSET MemVar
MOV   AX,[BX]
```

there's no way that the 486 can calculate the address referenced by **MOV AX,[BX]** until **MOV BX,OFFSET MemVar** finishes, so pipelining that calculation ahead of time is not possible. A good workaround is rearranging your code so that at least one instruction lies between the loading of the memory pointer and its use. For example, postdecrementing, as in the following

```
LoopTop:
      add   ax,[si]
      add   si,2
      dec   cx
      jnz   LoopTop
```

is faster than preincrementing, as in:

```
LoopTop:
      add   si,2
      add   ax,[SI]
      dec   cx
      jnz   LoopTop
```

Now that we understand what Intel means by this rule, let me make a very important comment: My observations indicate that for real-mode code, the documentation understates the extent of the penalty for interrupting the address calculation pipeline by loading a memory pointer just before it's used.

The truth of the matter appears to be that if a register is the destination of one instruction and is then used by the next instruction to address memory in real mode, not one but two cycles are lost!

In 32-bit protected mode, however, the penalty is, in fact, the 1 cycle that Intel documents.

Considering that **MOV** normally takes only one cycle total, that's quite a loss. For example, the postdecrement loop shown above is 2 full cycles faster than the preincrement loop, resulting in a 29 percent improvement in the performance of the entire loop. But wait, there's more. If a register is loaded 2 cycles (which generally means 2 instructions, but, because some 486 instructions take more than 1 cycle,

Cycle #	Instruction being executed	Address being calculated (arrow points to cycle during which address is used)
n	NOV AX,BX	[BX]
n+1	MOV [BX],1	[SI+1]
n+2	MOV AL,[SI+1]	
n+3	MOV CX,DX	

One-cycle-ahead address pipelining.
Figure 12.1

the 2 are not always equivalent) before it's used to point to memory, 1 cycle is lost. Therefore, whereas this code

```
mov  bx,offset MemVar
mov  ax,[bx]
inc  dx
dec  cx
jnz  LoopTop
```

loses two cycles from interrupting the address calculation pipeline, this code

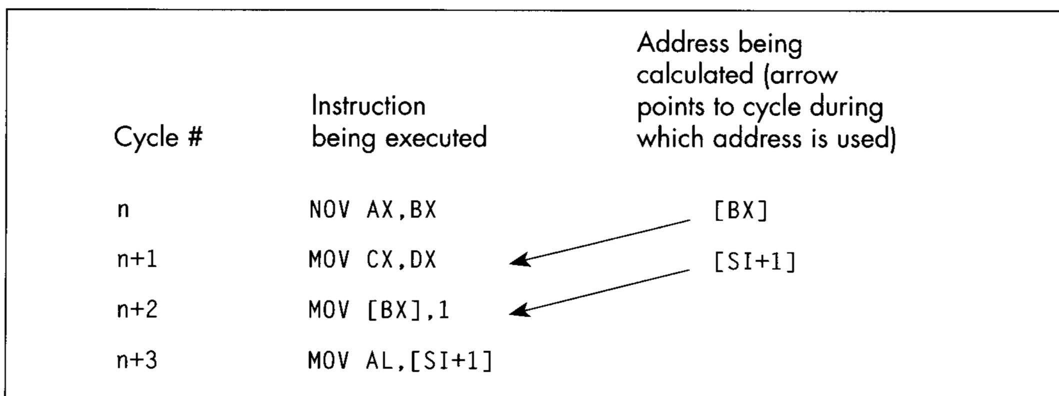
```
mov  bx,offset MemVar
inc  dx
mov  ax,[bx]
dec  cx
jnz  LoopTop
```

loses only one cycle, and this code

```
mov  bx,offset MemVar
inc  dx
dec  cx
mov  ax,[bx]
jnz  LoopTop
```

loses no cycles at all. Apparently, the 486's addressing calculation pipeline actually starts 2 cycles ahead, as shown in Figure 12.2. (In truth, my best guess at the moment is that the addressing pipeline really does start only 1 cycle ahead; the additional cycle crops up when the addressing pipeline has to wait for a register to be written into the register file before it can read it out for use in addressing calculations. However, I'm guessing here, and the 2-cycle-ahead model in Figure 12.2 will do just fine for optimization purposes.)

Clearly, there's considerable optimization potential in careful rearrangement of 486 code.



Two-cycle-ahead address pipelining.
Figure 12.2

Caveat Programmor

A caution: I'm quite certain that the 2-cycle-ahead addressing pipeline interruption penalty I've described exists in the two 486s I've tested. However, there's no guarantee that Intel won't change this aspect of the 486 in the future, especially given that the documentation indicates otherwise. Perhaps the 2-cycle penalty is the result of a bug in the initial steps of the 486, and will revert to the documented 1-cycle penalty someday; likewise for the undocumented optimizations I'll describe below. Nonetheless, none of the optimizations I suggest would hurt performance even if the undocumented performance characteristics of the 486 were to vanish, and they certainly will help performance on at least some 486s right now, so I feel they're well worth using.

There is, of course, no guarantee that I'm entirely correct about the optimizations discussed in this chapter. Without knowing the internals of the 486, all I can do is time code and make inferences from the results; I invite you to deduce your own rules and cross-check them against mine. Also, most likely there are other optimizations that I'm unaware of. If you have further information on these or any other undocumented optimizations, please write and let me know. And, of course, if anyone from Intel is reading this and wants to give us the gospel truth, please do!

Stack Addressing and Address Pipelining

Rule #2A: Rule #2 sometimes, but not always, applies to the stack pointer when it is implicitly used to point to memory.

Intel states that the stack pointer is an implied destination register for **CALL**, **ENTER**, **LEAVE**, **RET**, **PUSH**, and **POP** (which alter (E)SP), and that it is the implied base addressing register for **PUSH**, **POP**, and **RET** (which use (E)SP to address memory). Intel then implies that the aforementioned addressing pipeline penalty is incurred whenever the stack pointer is used as a destination by one of the first set of instructions and is then immediately used to address memory by one of the second set. This raises the specter of unpleasant programming contortions such as intermixing **PUSH**es and **POP**s with other instructions to avoid interrupting the addressing pipeline. Fortunately, matters are actually not so grim as Intel's documentation would indicate; my tests indicate that the addressing pipeline penalty pops up only spottily when the stack pointer is involved.

For example, you'd certainly expect a sequence such as

```
:
pop   ax
ret
pop   ax
ret
:
```

to exhibit the addressing pipeline interruption phenomenon (SP is both destination and addressing register for both instructions, according to Intel), but this code runs in six cycles per **POP/RET** pair, matching the official execution times exactly. Likewise, a sequence like

```
pop  dx
pop  cx
pop  bx
pop  ax
```

runs in one cycle per instruction, just as it should.

On the other hand, performing arithmetic directly on SP as an *explicit* destination—for example, to deallocate local variables—and then using **PUSH**, **POP**, or **RET**, definitely can interrupt the addressing pipeline. For example

```
add  sp,10h
ret
```

loses two cycles because SP is the explicit destination of one instruction and then the implied addressing register for the next, and the sequence

```
add  sp,10h
pop  ax
```

loses two cycles for the same reason.

I certainly haven't tried all possible combinations, but the results so far indicate that the stack pointer incurs the addressing pipeline penalty only if (E)SP is the *explicit* destination of one instruction and is then used by one of the two following instructions to address memory. So, for instance, SP isn't the explicit operand of **POP AX**—AX is—and no cycles are lost if **POP AX** is followed by **POP** or **RET**. Happily, then, we need not worry about the sequence in which we use **PUSH** and **POP**. However, adding to, moving to, or subtracting from the stack pointer should ideally be done at least two cycles before **PUSH**, **POP**, **RET**, or any other instruction that uses the stack pointer to address memory.

Problems with Byte Registers

There are two ways to lose cycles by using byte registers, and neither of them is documented by Intel, so far as I know. Let's start with the lesser and simpler of the two.

Rule #3: Do not load a byte portion of a register during one instruction, then use that register in its entirety as a source register during the next instruction.

So, for example, it would be a bad idea to do this

```
mov  ah,o
        :
mov  cx,[MemVar1]
mov  al,[MemVar2]
add  cx,ax
```

because AL is loaded by one instruction, then AX is used as the source register for the next instruction. A cycle can be saved simply by rearranging the instructions so that the byte register load isn't immediately followed by the word register usage, like so:

```
mov   ah,0
      :
mov   al,[MemVar2]
mov   cx,[MemVar1]
add   cx,ax
```

Strange as it may seem, this rule is neither arbitrary nor nonsensical. Basically, when a byte destination register is part of a word source register for the next instruction, the 486 is unable to directly use the result from the first instruction as the source for the second instruction, because only part of the register required by the second instruction is contained in the first instruction's result. The full, updated register value must be read from the register file, and that value can't be read out until the result from the first instruction has been written *into* the register file, a process that takes an extra cycle. I'm not going to explain this in great detail because it's not important that you understand why this rule exists (only that it *does* in fact exist), but it is an interesting window on the way the 486 works.

In case you're curious, there's no such penalty for the typical **XLAT** sequence like

```
mov   bx,offset MemTable
      :
mov   al,[si]
xlat
```

even though AL must be converted to a word by **XLAT** before it can be added to BX and used to address memory. In fact, none of the penalties mentioned in this chapter apply to **XLAT**, apparently because **XLAT** is so slow—4 cycles—that it gives the 486 time to perform addressing calculations during the course of the instruction.

*While it's nice that **XLAT** doesn't suffer from the various 486 addressing penalties, the reason for that is basically that **XLAT** is slow, so there's still no compelling reason to use **XLAT** on the 486.*

In general, penalties for interrupting the 486's pipeline apply primarily to the fast core instructions of the 486, most notably register-only instructions and **MOV**, although arithmetic and logical operations that access memory are also often affected. I don't know all the performance dependencies, and I don't plan to; figuring all of them out would be a big, boring job of little value. Basically, on the 486 you should concentrate on using those fast core instructions when performance matters, and all the rules I'll discuss do indeed apply to those instructions.

You don't need to understand every corner of the 486 universe unless you're a diehard ASMhead who does this stuff for fun. Just learn enough to be able to speed up

the key portions of your programs, and spend the rest of your time on a fast design and overall implementation.

More Fun with Byte Registers

Rule #4: Don't load *any* byte register exactly 2 cycles before using *any* register to address memory.

This, the last of this chapter's rules, is the strangest of the lot. If any byte register is loaded, and then two cycles later any register is used to point to memory, one cycle is lost. So, for example, this code

```
mov   al,bl
mov   cx,dx
mov   si,[di]
```

takes four rather than the expected three cycles to execute. Note that it is *not* required that the byte register be part of the register used to address memory; any byte register will do the trick.

Worse still, loading byte registers both one and two cycles before a register is used to address memory costs two cycles, as in

```
mov   bl,al
mov   cl,3
mov   bx,[si]
```

which takes five rather than three cycles to run. However, there is *no* penalty if a byte register is loaded one cycle but not two cycles before a register is used to address memory. Therefore,

```
mov   cx,3
mov   dl,al
mov   si,[bx]
```

runs in the expected three cycles.

In truth, I do not know why this happens. Clearly, it has something to do with interrupting the start of the addressing pipeline, and I have my theories about how this works, but at this point they're pure speculation. Whatever the reason for this rule, ignorance of it—and of its interaction with the other rules—could lead to considerable performance loss in seemingly air-tight code. For instance, a casual observer would expect the following code to run in 3 cycles:

```
mov   bx,offset MemVar
mov   cl,al
mov   ax,[bx]
```

A more sophisticated programmer would expect to lose one cycle, because BX is loaded two cycles before being used to address memory. In fact, though, this code takes 5 cycles—2 cycles, or 67 percent, longer than normal. Why? Well, under normal conditions,

loading a byte register—CL in this case—one cycle before using a register to address memory produces no penalty; loading 2 cycles ahead is the only case that normally incurs a penalty. However, think of Rule #4 as meaning that loading a byte register disrupts the memory addressing pipeline as it starts up. Viewed that way, we can see that **MOV BX,OFFSET MemVar** interrupts the addressing pipeline, forcing it to start again, and then, presumably, **MOV CL,AL** interrupts the pipeline again because the pipeline is now on its first cycle: the one that loading a byte register can affect.

I know—it seems awfully complicated. It isn't, really. Generally, try not to use byte destinations exactly two cycles before using a register to address memory, and try not to load a register either one or two cycles before using it to address memory, and you'll be fine.

Timing Your Own 486 Code

In case you want to do some 486 performance analysis of your own, let me show you how I arrived at one of the above conclusions; at the same time, I can warn you of the timing hazards of the cache. Listings 12.1 and 12.2 show the code I ran through the Zen timer in order to establish the effects of loading a byte register before using a register to address memory. Listing 12.1 ran in 120 μs on a 33 MHz 486, or 4 cycles per repetition (120 μs/1000 repetitions = 120 ns per repetition; 120 ns per repetition/30 ns per cycle = 4 cycles per repetition); Listing 12.2 ran in 90 μs, or 3 cycles, establishing that loading a byte register costs a cycle only when it's performed exactly 2 cycles before addressing memory.

LISTING 12.1 LST12-1.ASM

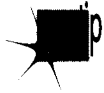
```
; Measures the effect of loading a byte register 2 cycles before
; using a register to address memory.
     mov   bp,2        ;run the test code twice to make sure
                       ; it's cached
     sub   bx,bx
CacheFillLoop:
     call  ZTimerOn ;start timing
     rept  1000
     mov   dl,cl
     nop
     mov   ax,[bx]
     endm
     call  ZTimerOff ;stop timing
     dec   bp
     jz    Done
     jmp   CacheFillLoop
Done:
```

LISTING 12.2 LST12-2.ASM

```
; Measures the effect of loading a byte register 1 cycle before
; using a register to address memory.
     mov   bp,2        ;run the test code twice to make sure
                       ; it's cached
     sub   bx,bx
```

```
CacheFillLoop:
     call  ZTimerOn ;start timing
     rept  1000
     nop
     mov   dl,cl
     mov   ax,[bx]
     endm
     call  ZTimerOff ;stop timing
     dec   bp
     jz    Done
     jmp   CacheFillLoop
Done:
```

Note that Listings 12.1 and 12.2 each repeat the timing of the code under test a second time, to make sure that the instructions are in the cache on the second pass, the one for which results are displayed. Also note that the code is less than 8K in size, so that it can all fit in the 486's 8K internal cache. If I double the **REPT** value in Listing 12.2 to 2,000, making the test code larger than 8K, the execution time more than doubles to 224 µs, or 3.7 cycles per repetition; the extra seven-tenths of a cycle comes from fetching non-cached instruction bytes.



Whenever you see non-integral timing results of this sort, it's a good bet that the test code or data isn't cached.

The Story Continues

There's certainly plenty more 486 lore to explore, including the 486's unique prefetch queue, more optimization rules, branching optimizations, performance implications of the cache, the cost of cache misses for reads, and the implications of cache write-through for writes. Nonetheless, we've covered quite a bit of ground in this chapter, and I trust you've gotten a feel for the considerable extent to which 486 optimization differs from what you're used to. Odd as 486 optimization is, though, it's well worth mastering, for the 486 is, at its best, so staggeringly fast that carefully crafted 486 code can do more than twice as much per cycle as the best 386 code—which makes it perhaps 50 times as fast as optimized code for the original PC.

Sometimes it *is* hard to believe we're still in Kansas!