Chapter 13

Aiming the 486

Chapter 13

Pipelines and Other Hazards of the High End

It's a sad but true fact that 84 percent of American schoolchildren are ignorant of 92 percent of American history. Not my daughter, though. We recently visited historical Revolutionary-War-vintage Fort Ticonderoga, and she's now 97 percent aware of a key element of our national heritage: that the basic uniform for soldiers in those days was what appears to be underwear, plus a hat so that no one could complain that they were undermining family values. Ha! Just kidding! Actually, what she learned was that in those days, it was pure coincidence if a cannonball actually hit anything it was aimed at, which isn't surprising considering the lack of rifling, precision parts, and ballistics. The guides at the fort shot off three cannons; the closest they came to the target was about 50 feet, and that was only because the wind helped. I think the idea in early wars was just to put so much lead in the air that some of it was bound to hit *something;* preferably, but not necessarily, the enemy.

Nowadays, of course, we have automatic weapons that allow a teenager to singlehandedly defeat the entire U.S. Army, not to mention so-called "smart" bombs, which are smart in the sense that they can seek out and empty a taxpayer's wallet without being detected by radar. There's an obvious lesson here about progress, which I leave you to deduce for yourselves.

Here's the same lesson, in another form. Ten years ago, we had a slow processor, the 8088, for which it was devilishly hard to optimize, and for which there was no good optimization documentation available. Now we have a processor, the 486, that's 50 to

100 times faster than the 8088—and for which there is no good optimization documentation available. Sure, Intel provides a few tidbits on optimization in the back of the *i486 Microprocessor Programmer's Reference Manual,* but, as I discussed in Chapter 12, that information is both incomplete and not entirely correct. Besides, most assembly language programmers don't bother to read Intel's manuals (which are extremely informative and well done, but only slightly more fun to read than the phone book), and go right on programming the 486 using outdated 8088 optimization techniques, blissfully unaware of a new and heavily mutated generation of cycle-eaters that interact with their code in ways undreamt of even on the 386.

For example, consider how Terje Mathisen doubled the speed of his word-counting program on a 486 simply by shuffling a couple of instructions.

486 Pipeline Optimization

I've mentioned Terje Mathisen in my writings before. Terje is an assembly language programmer extraordinaire, and author of the incredibly fast public-domain word-counting program WC (which comes complete with source code; well worth a look, if you want to see what *really* fast code looks like). Terje's a regular participant in the ibm.pc/fast.code topic on Bix. In a thread titled "486 Pipeline Optimization, or TANSTATFC (There Ain't No Such Thing As The Fastest Code)," he detailed the following optimization to WC, perhaps the best example of 486 pipeline optimization I've yet seen.

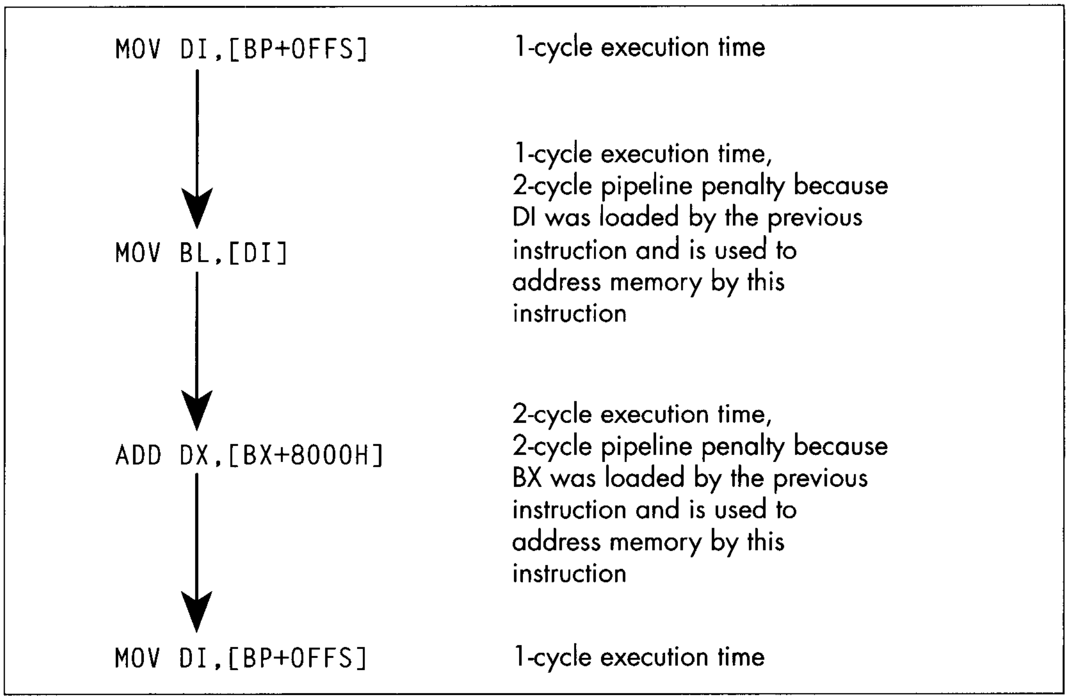
Terje's inner loop originally looked something like the code in Listing 13.1. (I've taken a few liberties for illustrative purposes.) Of course, Terje unrolls this loop a few times (128 times, to be exact). By the way, in Listing 13.1 you'll notice that Terje counts not only words but also lines, at a rate of three instructions for every two characters!

LISTING 13.1 L13-1.ASM

```
mov di,[bp+OFFS]     ;get the next pair of characters
mov bl,[di]          ;get the state value for the pair
add dx,[bx+8000h]    ;increment word and line count
                     ; appropriately for the pair
```

Listing 13.1 looks as tight as it could be, with just two one-cycle instructions, one two-cycle instruction, and no branches. It *is* tight, but those three instructions actually take a minimum of 8 cycles to execute, as shown in Figure 13.1. The problem is that DI is loaded just before being used to address memory, and that costs 2 cycles because it interrupts the 486's internal instruction pipeline. Likewise, BX is loaded just before being used to address memory, costing another two cycles. Thus, this loop takes twice as long as cycle counts would seem to indicate, simply because two registers are loaded immediately before being used, disrupting the 486's pipeline.

Listing 13.2 shows Terje's immediate response to these pipelining problems; he simply swapped the instructions that load DI and BL. This one change cut execution time per character pair from eight cycles to five cycles! The load of BL is now separated by


MOV DI,[BP+OFFS]
1-cycle execution time
1-cycle execution time, 2-cycle pipeline penalty because DI was loaded by the previous instruction and is used to address memory by this instruction
MOV BL,[DI]
2-cycle execution time, 2-cycle pipeline penalty because BX was loaded by the previous instruction and is used to address memory by this instruction
ADD DX,[BX+8000H]
MOV DI,[BP+OFFS]
1-cycle execution time

Cycle-eaters in the original WC.
Figure 13.1

one instruction from the use of BX to address memory, so the pipeline penalty is reduced from two cycles to one cycle. The load of DI is also separated by one instruction from the use of DI to address memory (remember, the loop is unrolled, so the last instruction is followed by the first instruction), but because the intervening instruction takes two cycles, there's no penalty at all.

Remember, pipeline penalties diminish with increasing number of cycles, not instructions, between the pipeline disrupter and the potentially affected instruction.

LISTING 13.2 L13-2.ASM

```
mov bl,[di]          ;get the state value for the pair
mov di,[bp+OFFS]     ;get the next pair of characters
add dx,[bx+8000h]    ;increment word and line count
                     ; appropriately for the pair
```

At this point, Terje had nearly doubled the performance of this code simply by moving one instruction. (Note that swapping the instructions also made it necessary to preload DI at the start of the loop; Listing 13.2 is not exactly equivalent to Listing 13.1.) I'll let Terje describe his next optimization in his own words:

"When I looked closely as this, I realized that the two cycles for the final **ADD** is just the sum of 1 cycle to load the data from memory, and 1 cycle to add it to DX, so the code could just as well have been written as shown in Listing 13.3. The final breakthrough came when I realized that by initializing AX to zero outside the loop, I could rearrange it as shown in Listing 13.4 and do the final **ADD DX,AX** after the loop. This way there are two single-cycle instructions between the first and the fourth line, avoiding all pipeline stalls, for a total throughput of two cycles/char."

LISTING 13.3 L13-3.ASM

```
mov bl,[di]         ;get the state value for the pair
mov di,[bp+OFFS]    ;get the next pair of characters
mov ax,[bx+8000h]   ;increment word and line count
add dx,ax           ; appropriately for the pair
```

LISTING 13.4 L13-4.ASM

```
mov bl,[di]         ;get the state value for the pair
mov di,[bp+OFFS]    ;get the next pair of characters
add dx,ax           ;increment word and line count
                    ; appropriately for the pair
mov ax,[bx+8000h]   ;get increments for next time
```

I'd like to point out two fairly remarkable things. First, the single cycle that Terje saved in Listing 13.4 sped up his entire word-counting engine by 25 percent or more; Listing 13.4 is fully twice as fast as Listing 13.1—all the result of nothing more than shifting an instruction and splitting another into two operations. Second, Terje's word-counting engine can process more than 16 million characters *per second* on a 486/33.

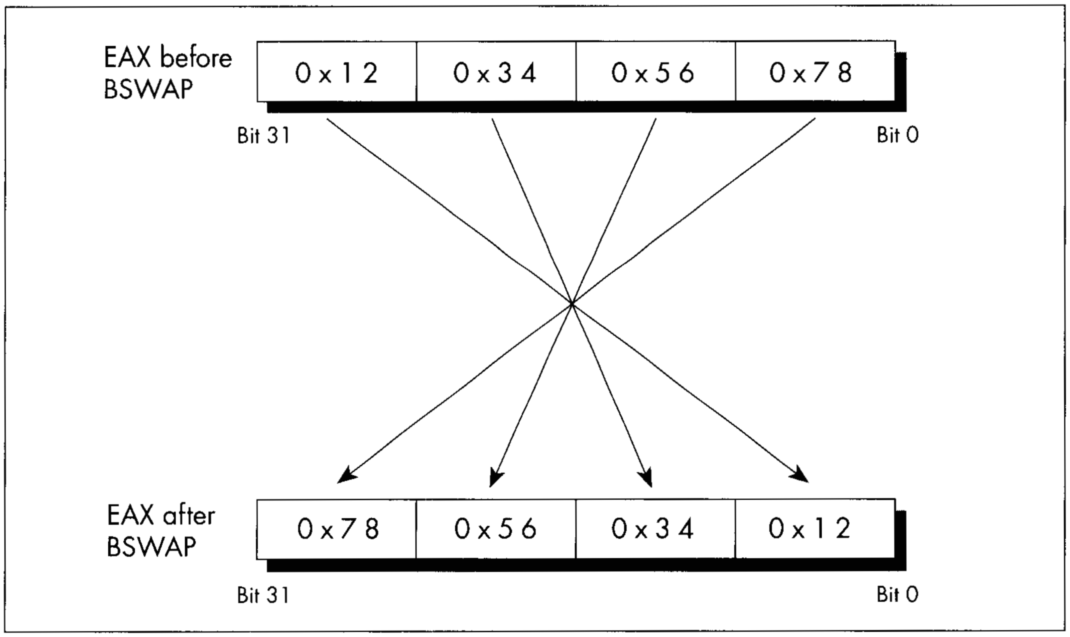
Clever 486 optimization can pay off big. QED.

BSWAP: More Useful Than You Might Think

There are only 3 non-system instructions unique to the 486. None is earthshaking, but they have their uses. Consider **BSWAP**. **BSWAP** does just what its name implies, swapping the bytes (not bits) of a 32-bit register from one end of the register to the other, as shown in Figure 13.2. (**BSWAP** can only work with 32-bit registers; memory locations and 16-bit registers are not valid operands.) The obvious use of **BSWAP** is to convert data from Intel format (least significant byte first in memory, also called *little endian*) to Motorola format (most significant byte first in memory, or *big endian*), like so:

```
lodsd
bswap
stosd
```

BSWAP can also be useful for reversing the order of pixel bits from a bitmap so that they can be rotated 32 bits at a time with an instruction such as **ROR EAX,1**. Intel's byte ordering for multiword values (least-significant byte first) loads pixels in the wrong order, so far as word rotation is concerned, but **BSWAP** can take care of that.



BSWAP in operation.
Figure 13.2

As it turns out, though, **BSWAP** is also useful in an unexpected way, having to do with making efficient use of the upper half of 32-bit registers. As any assembly language programmer knows, the x86 register set is too small; or, to phrase that another way, it sure would be nice if the register set were bigger. As any 386/486 assembly language programmer knows, there are many cases in which 16 bits is plenty. For example, a 16-bit scan-line counter generally does the trick nicely in a video driver, because there are *very* few video devices with more than 65,535 addressable scan lines. Combining these two observations yields the obvious conclusion that it would be great if there were some way to use the upper and lower 16 bits of selected 386 registers as separate 16-bit registers, effectively increasing the available register space.

Unfortunately, the x86 instruction set doesn't provide any way to work directly with only the upper half of a 32-bit register. The next best solution is to rotate the register to give you access in the lower 16 bits to the half you need at any particular time, with code along the lines of that in Listing 13.5. Having to rotate the 16-bit fields into position certainly isn't as good as having direct access to the upper half, but surely it's better than having to get the values out of memory, isn't it?

LISTING 13.5 L13-5.ASM

```
        mov     cx,[initialskip]
        shl     ecx,16          ;put skip value in upper half of ECX
        mov     cx,100          ;put loop count in CX
```

```
looptop:
     :
     ror   ecx,16      ;make skip value word accessible in CX
     add   bx,cx       ;skip BX ahead
     inc   cx          ;set next skip value
     ror   ecx,16      ;put loop count in CX
     dec   cx          ;count down loop
     jnz   looptop
```

Not necessarily. Shifts and rotates are among the worst performing instructions of the 486, taking 2 to 3 cycles to execute. Thus, it takes 2 cycles to rotate the skip value into CX in Listing 13.5, and 2 more cycles to rotate it back to the upper half of ECX. I'd say four cycles is a pretty steep price to pay, especially considering that a **MOV** to or from memory takes only one cycle. Basically, using **ROR** to access a 16-bit value in the upper half of a 16-bit register is a pretty marginal technique, unless for some reason you can't access memory at all (for example, if you're using BP as a working register, temporarily making the stack frame inaccessible).

On the 386, **ROR** was the only way to split a 32-bit register into two 16-bit registers. On the 486, however, **BSWAP** can not only do the job, but can do it better, because **BSWAP** executes in just one cycle. **BSWAP** has the added benefit of not affecting any flags, unlike **ROR**. With **BSWAP**-based code like that in Listing 13.6, the upper 16 bits of a register can be accessed with only 2 cycles of overhead and without altering any flags, making the technique of packing two 16-bit registers into one 32-bit register much more useful.

LISTING 13.6 L13-6.ASM

```
     mov   cx,[initialskip]
     bswap ecx         ;put skip value in upper half of ECX
     mov   cx,100      ;put loop count in CX
looptop:
     :
     bswap ecx         ;make skip value word accessible in CX
     add   bx,cx       ;skip BX ahead
     inc   cx          ;set next skip value
     bswap ecx         ;put loop count in CX
     dec   cx          ;count down loop
     jnz   looptop
```

Pushing and Popping Memory

Pushing or popping a memory location, as in **PUSH WORD PTR [BX]** or **POP [MemVar]**, is a compact, easy way to get a value onto or off of the stack, especially when pushing parameters for calling a C-compatible function. However, on a 486, these are unattractive instructions from a performance perspective. Pushing a memory location takes four cycles; by contrast, loading a memory location into a register takes only one cycle, and pushing a register takes just 1 more cycle, for a total of two cycles. Therefore,

```
mov   ax,[bx]
push  ax
```

is twice as fast as

```
push  word ptr [bx]
```

and the only cost is that the previous contents of AX are destroyed.

Likewise, popping a memory location takes six cycles, but popping a register and writing it to memory takes only two cycles combined. The *i486 Microprocessor Programmer's Reference Manual* lists a 4-cycle execution time for popping a register, but pay that no mind; popping a register takes only 1 cycle.

Why is it that such a convenient operation as pushing or popping memory is so slow? The rule on the 486 is that simple operations, which can be executed in a single cycle by the 486's RISC core, are fast; whereas complex operations, which must be carried out in microcode just as they were on the 386, are almost all relatively slow. Slow, complex operations include all the string instructions except **REP MOVS**, as well as **XLAT**, **LOOP**, and, of course, **PUSH *mem*** and **POP *mem***.

*Whenever possible, try to use the 486's 1-cycle instructions, including **MOV**, **ADD**, **SUB**, **CMP**, **ADC**, **SBB**, **XOR**, **AND**, **OR**, **TEST**, **LEA**, and **PUSH reg** and **POP reg**. These instructions have an added benefit in that it's often possible to rearrange them for maximum pipeline efficiency, as is the case with Terje's optimization described earlier in this chapter.*

Optimal 1-Bit Shifts and Rotates

On a 486, the n-bit forms of the shift and rotate instructions—as in **ROR AX,2** and **SHL BX,9**—are 2-cycle instructions, but the 1-bit forms—as in **ROR AX,1** and **SHL BX,1**—are *3-cycle* instructions. Go figure.

Assemblers default to the 1-bit instruction for 1-bit shifts and rotates. That's not unreasonable since the 1-bit form is a byte shorter and is just as fast as the n-bit forms on a 386 and faster on a 286, and the n-bit form doesn't even exist on an 8088. In a really critical loop, however, it might be worth hand-assembling the n-bit form of a single-bit shift or rotate in order to save that cycle. The easiest way to do this is to assemble a 2-bit form of the desired instruction, as in **SHL AX,2**, then look at the hex codes that the assembler generates and use **DB** to insert them in your program code, with the value two replaced with the value one. For example, you could determine that **SHL AX,2** assembles to the bytes 0C1H 0E0H 002H, either by looking at the disassembly in a debugger or by having the assembler generate a listing file. You could then insert the n-bit version of **SHL AX,1** in your code as follows:

```
mov   ax,1
db    0c1h, 0e0h, 001h
mov   dx,ax
```

At the end of this sequence, DX will contain 2, and the fast n-bit version of **SHL AX,1** will have executed. If you use this approach, I'd recommend using a macro, rather than sticking DBs in the middle of your code.

Again, this technique is advantageous *only* on a 486. It also doesn't apply to **RCL** and **RCR**, where you definitely want to use the 1-bit versions whenever you can, because the n-bit versions are horrendously slow. But if you're optimizing for the 486, these tidbits can save a few critical cycles—and Lord knows that if you're optimizing for the 486—that is, if you need even more performance than you get from unoptimized code on a 486—you almost certainly need all the speed you can get.

32-Bit Addressing Modes

The 386 and 486 both support 32-bit addressing modes, in which any register may serve as the base memory addressing register, and almost any register may serve as the potentially scaled index register. For example,

```
mov al,BaseTable[ecx+edx*4]
```

uses a perfectly valid 32-bit address, with the byte accessed being the one at the offset in DS pointed to by the sum of EDX times 4 plus the offset of **BaseTable** plus ECX. This is a very powerful memory addressing scheme, far superior to 8088-style 16-bit addressing, but it's not without its quirks and costs, so let's take a quick look at 32-bit addressing. (By the way, 32-bit addressing is not limited to protected mode; 32-bit instructions may be used in real mode, although each instruction that uses 32-bit addressing must have an address-size prefix byte, and the presence of a prefix byte costs a cycle on a 486.)

Any register may serve as the base register component of an address. Any register except ESP may also serve as the index register, which can be scaled by 1, 2, 4, or 8. (Scaling is very handy for performing lookups in arrays and tables.) The same register may serve as both base and index register, except for ESP, which can only be the base. Incidentally, it makes sense that ESP can't be scaled; ESP presumably always points to a valid stack, and I can't think of any reason you'd want to use the stack pointer times 2, 4, or 8 in an address. ESP is, by its nature, a base rather than index pointer.

That's all there is to the functionality of 32-bit addressing; it's very simple, much simpler than 16-bit addressing, with its sharply limited memory addressing register combinations. The costs of 32-bit addressing are a bit more subtle. The only performance cost (apart from the aforementioned 1-cycle penalty for using 32-bit addressing in real mode) is a 1-cycle penalty imposed for using an index register. In this context, you use an index register when you use a register that's scaled, or when you use the sum of two registers to point to memory. **MOV BL,[EBX*2]** uses an index register and takes an extra cycle, as does **MOV CL,[EAX+EDX]**; **MOV CL,[EAX+100H]** is not indexed, however.

The other cost of 32-bit addressing is in instruction size. Old-style 16-bit addressing usually (except in a few special cases) uses one extra byte, which Intel calls the Mod-R/M byte, which is placed immediately after each instruction's opcode to describe the memory addressing mode, plus 1 or 2 optional bytes of addressing displacement—that is, a constant value to add into the address. In many cases, 32-bit addressing continues to use the Mod-R/M byte, albeit with a different interpretation; in these cases, 32-bit addressing is no larger than 16-bit addressing, except when a 32-bit displacement is involved. For example, **MOV AL, [EBX]** is a 2-byte instruction; **MOV AL, [EBX+10H]** is a 3-byte instruction; and **MOV AL, [EBX+10000H]** is a 6-byte instruction.

Note that 1 and 4-byte displacements, but not 2-byte displacements, are supported for 32-bit addressing. Code size can be greatly improved by keeping stack frame variables within 128 bytes of EBP, and variables in pointed-to structures within 127 bytes of the start of the structure, so that displacements can be 1 rather than 4 bytes.

However, because 32-bit addressing supports many more addressing combinations than 16-bit addressing, the Mod-R/M byte can't describe all the combinations. Therefore, whenever an index register (as described above) is involved, a second byte, the SIB byte, follows the Mod-R/M byte to provide additional address information. Consequently, whenever you use a scaled memory addressing register or use the sum of two registers to point to memory, you automatically add 1 cycle and 1 byte to that instruction. This is not to say that you shouldn't use index registers when they're needed, but if you find yourself using them inside key loops, you should see if it's possible to move the index calculation outside the loop as, for example, in a loop like this:

```
LoopTop:
     add   ax,DataTable[ebx*2]
     inc   ebx
     dec   cx
     jnz   LoopTop
```

You could change this to the following for greater performance:

```
     add   ebx,ebx     ;ebx*2
LoopTop:
     add   ax,DataTable[ebx]
     add   ebxX,2
     dec   cx
     jnz   LoopTop
     shr   ebx,1 ;ebx*2/2
```

I'll end this chapter with two more quirks of 32-bit addressing. First, as with 16-bit addressing, addressing that uses EBP as a base register both accesses the SS segment by default and always has a displacement of at least 1 byte. This reflects the common use of EBP to address a stack frame, but is worth keeping in mind if you should happen to use EBP to address non-stack memory.

Lastly, as I mentioned, ESP cannot be scaled. In fact, ESP cannot be an index register; it must be a base register. Ironically, however, ESP is the one register that cannot be used to address memory without the presence of an SIB byte, even if it's used without an index register. This is an outcome of the way in which the SIB byte extends the capabilities of the Mod-R/M byte, and there's nothing to be done about it, but it's at least worth noting that ESP-based, non-indexed addressing makes for instructions that are a byte larger than other non-indexed addressing (but not any slower; there's no 1-cycle penalty for using ESP as a base register) on the 486.