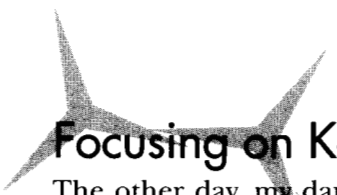
Chapter 21

Unleashing the Pentium's V-pipe

Chapter 21



Focusing on Keeping Both Pentium Pipes Full

The other day, my daughter suggested that we each draw the prettiest picture we could, then see whose was prettier. I won't comment on who won, except to note that apparently a bolt of lightning zipping toward a moose with antlers that bear an unfortunate resemblance to a propeller beanie isn't going to win me any scholarships to art school, if you catch my drift. Anyway, my drawing happened to feature the word "chartreuse" (because it rhymed with "moose" and "Zeus"—hence the lightning; more than that I am not at liberty to divulge), and she wanted to know if the moose was actually chartreuse. I had to admit that I didn't know, so we went to the dictionary, whereupon we learned that chartreuse is a pale apple-green color. Then she brought up the Windows Control Panel, pointed to the selection of predefined colors, and asked, "Which of those is chartreuse?"—and I realized that I *still* didn't know.

Some things can be described perfectly with words, but others just have to be experienced. Color is one such category, and Pentium optimization is another. I've spent the last two chapters detailing the rules for Pentium optimization, and I'll spend half of this one doing so, as well. That's good; without understanding the fundamentals, we have no chance of optimizing well. It's not enough, though. We also need to look at a real-world example of Pentium optimization in action, and we'll do that later in this chapter; after which, you should go out and do some Pentium optimization on your own. Optimization is one of those things that you can learn a lot about from reading, but ultimately it has to sink into your pores as you do it—especially Pentium

optimization because the Pentium is perhaps the most complex (and rewarding) chip to optimize for that I've ever seen.

In the last chapter, we explored the dual-execution-pipe nature of the Pentium, and learned which instructions could pair (execute simultaneously) in which pipes. Now we're ready to look at AGIs and register contention—two hazards that can prevent otherwise properly written code from taking full advantage of the Pentium's two pipes, and can thereby keep your code from pushing the Pentium to maximum performance.

Address Generation Interlocks

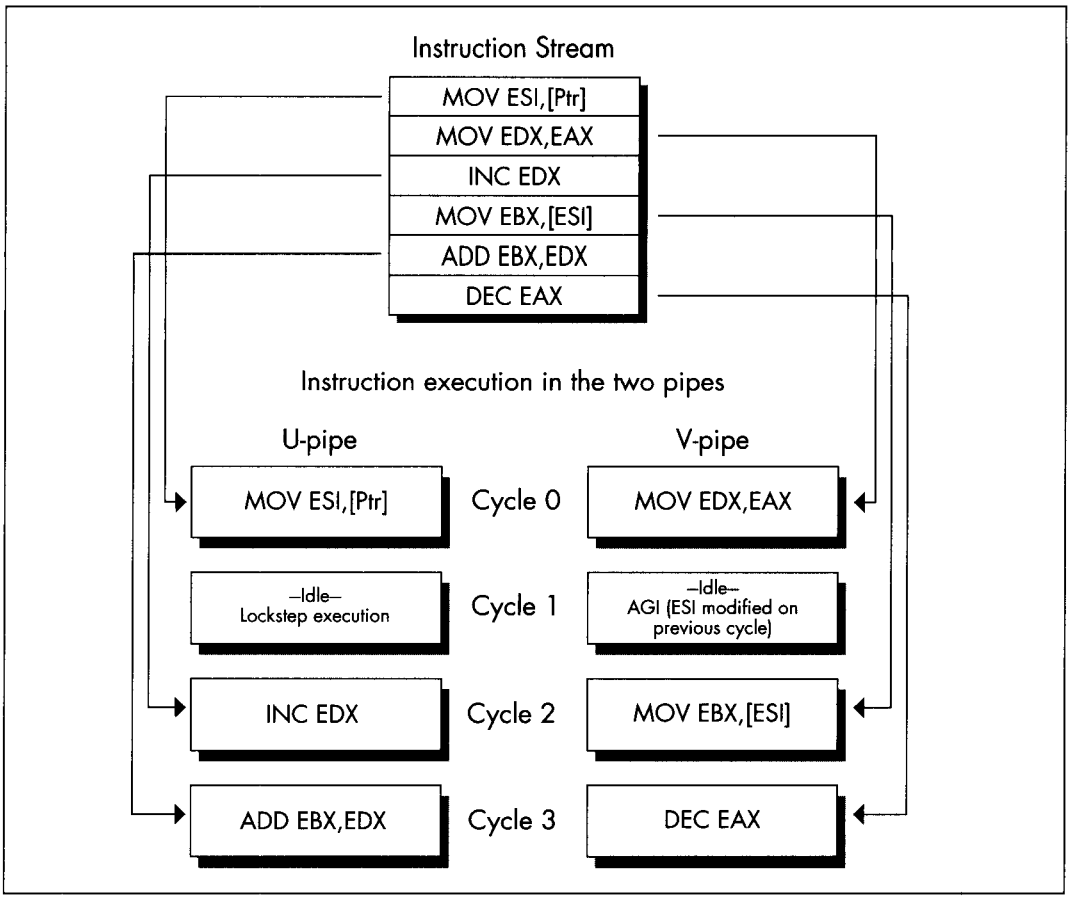
The Pentium is advertised as having a five-stage pipeline for each of its execution units. All this means is that at any given time, up to five instructions are in various stages of execution in each pipe; this overlapping of execution is done for speed, so each instruction doesn't have to wait until the previous one has finished. The only way that the Pentium's pipelining directly affects the way you program is in the areas of AGIs and register dependencies.

AGIs are *Address Generation Interlocks,* a fancy way of saying that if a register is used to address memory, as is EBX in this instruction

```
mov [ebx],eax
```

and the value of the register is not set far enough ahead for the Pentium to perform the addressing calculations before the instruction needs the address, then the Pentium will stall the pipe in which the instruction is executing until the value becomes available and the addressing calculations have been performed. Remember, also, that instructions execute in lockstep on the Pentium, so if one pipe stalls for a cycle, making its instruction take one cycle longer, that extends by one cycle the time until the other pipe can begin its next instruction, as well.

The rule for AGIs is simple: If you modify any part of a register during a cycle, you cannot use that register to address memory during either that cycle or the next cycle. If you try to do this, the Pentium will simply stall the instruction that tries to use that register to address memory until two cycles after the register was modified. This was true on the 486 as well, but the Pentium's new twist is that since more than one instruction can execute in a single cycle, an AGI can stall an instruction that's as many as three instructions away from the changing of the addressing register, as shown in Figure 21.1, and an AGI can also cause a stall that costs as many as three instructions, as shown in Figure 21.2. This means that AGIs are both much easier to cause and potentially more expensive than on the 486, and you must keep a sharp eye out for them. It also means that it's often worth calculating a memory pointer several instructions ahead of its actual use. Unfortunately, this tends to extend the lifetimes of pointer registers to span a greater number of instructions, making the Pentium's relatively small register set seem even smaller.

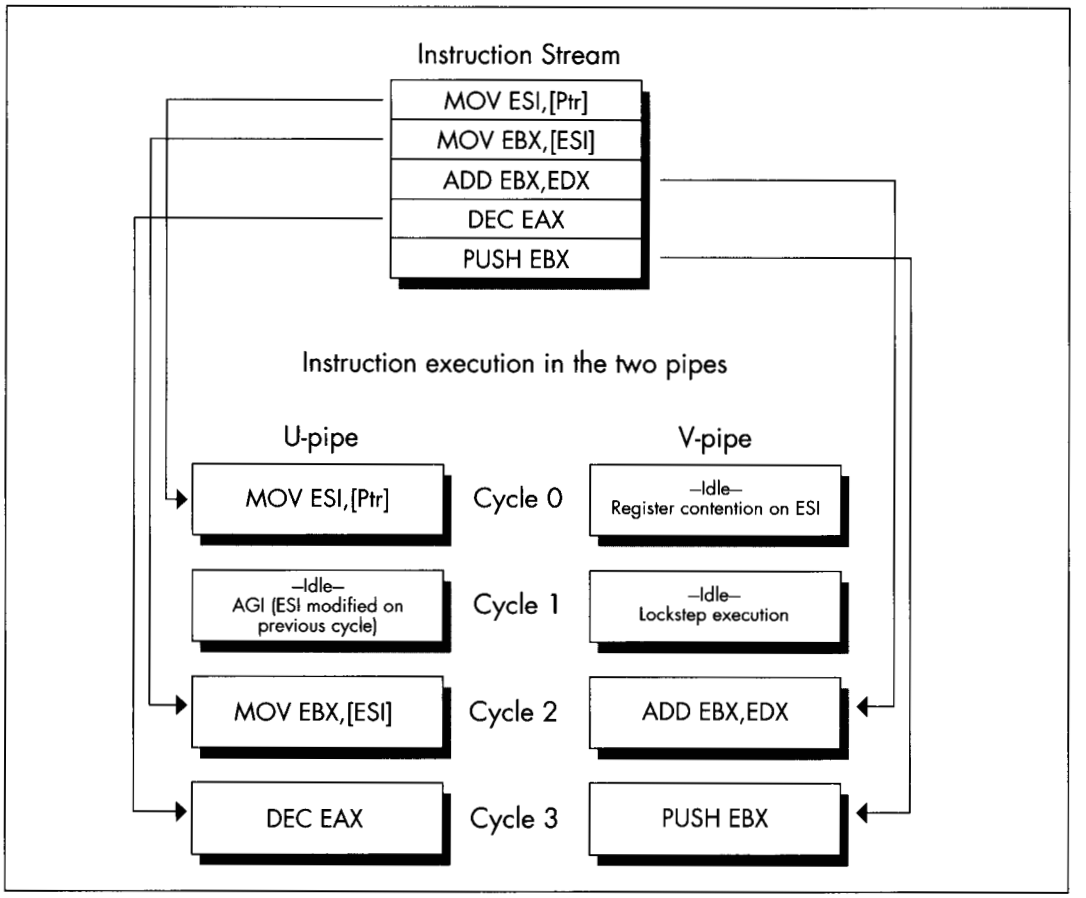


An AGI can stall up to three instructions later.
Figure 21.1

As an example of a sort of AGI that's new to the Pentium, consider the following test for a NULL pointer, followed by the use of the pointer if it's not NULL:

```
push ebx                ;U-pipe cycle 1
mov  ebx,[Ptr]          ;V-pipe cycle 1
and  ebx,ebx            ;U-pipe cycle 2
jz   short IsNull       ;V-pipe cycle 2
mov  eax,[ebx]          ;U-pipe cycle 3 AGI stall
mov  edx,[ebp-8]        ;V-pipe cycle 3 lockstep idle
                        ;U-pipe cycle 4 mov eax,[ebx]
                        ;V-pipe cycle 4 mov edx,[ebp-8]
```

This commonplace code loses a U-pipe cycle to the AGI caused by **AND EBX,EBX**, followed by the attempt two instructions later to use EBX to point to memory. The code loses a V-pipe cycle as well, because lockstep execution won't let the next V-pipe



An AGI can cost as many as 3 cycles.
Figure 21.2

instruction execute until the paired U-pipe instruction that suffered the AGI finishes. The solution is to use **TEST EBX,EBX** instead of **AND**; **TEST** can't modify EBX, so no AGI occurs. Sure, **AND EBX,EBX** doesn't modify EBX either, but the Pentium doesn't know that, so it has to insert the AGI.

As on the 486, you should keep a careful eye out for AGIs involving the stack pointer. Implicit modifiers of ESP, such as **PUSH** and **POP**, are special-cased so you don't have to worry about AGIs. However, if you explicitly modify ESP with this instruction

```
sub esp,100h
```

for example, or with the popular

```
mov esp,ebp
```

you can then get AGIs if you attempt to use ESP to address memory, either explicitly with instructions like this one

```
mov   eax,[esp+20h]
```

or via **PUSH**, **POP**, or other instructions that implicitly use ESP as an addressing register.

On the 486, any instruction that had both a constant value and an addressing displacement, such as

```
mov dword ptr [ebp+16],1
```

suffered a 1-cycle penalty, taking a total of 2 cycles. Such instructions take only one cycle on the Pentium, but they cannot pair, so they're still the most expensive sort of **MOV**. Knowing this can speed up something as simple as zeroing two memory variables, as in

```
sub eax,eax           ;U-pipe 1
                      ;any V-pipe pairable
                      ; instruction can go here,
                      ; or SUB could be in V-pipe
mov [MemVar1],eax     ;U-pipe 2
mov [MemVar2],eax     ;V-pipe 2
```

which should never be slower and should potentially be 0.5 cycles faster, and six bytes smaller than this sequence:

```
mov [MemVar1],0   ;U-pipe 1
mov [MemVar2],0   ;U-pipe 2
```

Note, however, that my experiments thus far indicate that the two writes in the first case don't actually pair (possibly because the memory variables have never been read into the internal cache), so you might want to insert an instruction between the two **MOV**s—and, of course, this is yet another reason why you should always measure your code's actual performance.

Register Contention

Finally, we come to the last major component of superscalar optimization: register contention. The basic premise here is simple: You can't use the same register in two inherently sequential ways in a single cycle. For example, you can't execute

```
inc eax     ;U-pipe cycle 1
            ;V-pipe idle cycle 1
            ; due to dependency
and ebx,eax ;U-pipe cycle 2
```

in a single cycle; **AND EBX,EAX** can't execute until the value in EAX is known, and that can't happen until **INC EAX** is done. Consequently, the V-pipe idles while **INC**

EAX executes in the U-pipe. We saw this in the last chapter when we discussed splitting instructions into simple instructions, and it is by far the most common sort of register contention, known as read-after-write register contention. Read-after-write register contention is the primary reason we have to interleave independent operations in order to get maximum V-pipe usage.

The other sort of register contention is known as write-after-write. Write-after-write register contention happens when two instructions try to write to the same register on the same cycle. While that may not seem like a particularly useful operation in general, it can happen when subregisters are being set, as in the following

```
sub eax,eax      ;U-pipe cycle 1
                 ;V-pipe idle cycle 1
                 ; due to register contention
mov al,[Var]     ;U-pipe cycle 2
```

where an attempt is made to set both EAX and its AL subregister on the same cycle. Write-after-write contention implies that the two instructions comprising the above substitute for **MOVZX** should have at least one unrelated instruction between them when **SUB EAX,EAX** executes in the V-pipe.

Exceptions to Register Contention

Intel has special-cased some very useful exceptions to register contention. Happily, write-after-read operations do *not* cause contention. Such operations, as in

```
mov eax,edx ;U-pipe cycle 1
sub edx,edxX ;V-pipe cycle 1
```

are free of charge.

Also, stack-related instructions that modify ESP only implicitly (without ESP as part of any explicit operand) do not cause AGIs, and neither do they cause register contention with other instructions that use ESP only implicitly; such instructions include **PUSH *reg/immed***, **POP *reg***, and **CALL**. (However, these instructions do cause register contention on ESP—but not AGIs—with instructions that use ESP explicitly, such as **MOV EAX,[ESP+4]**.) Without this special case, the following sequence would hardly use the V-pipe at all:

```
mov  eax,[MemVar] ;U-pipe cycle 1
push esi          ;V-pipe cycle 1
push eax          ;U-pipe cycle 2
push edi          ;V-pipe cycle 2
push ebx          ;U-pipe cycle 3
call FooTilde     ;V-pipe cycle 3
```

But in fact, all the instructions pair, even though ESP is modified five times in the space of six instructions.

The final register-contention special case is both remarkable and remarkably important. There is exactly one sort of instruction that can pair only in the V-pipe: branches.

Any near call or conditional or unconditional near jump can execute in the V-pipe paired with any pairable U-pipe instruction, as illustrated by this sequence:

```
LoopTop:
   mov [esi],eax ;U-pipe cycle 1
   add esi,4     ;V-pipe cycle 1
   dec ecx       ;U-pipe cycle 2
   jnz LoopTop   ;V-pipe cycle 2
```

Branches can't pair in the U-pipe; a branch that executes in the U-pipe runs alone, with the V-pipe idle. If a call or jump is correctly predicted by the Pentium's branch prediction circuitry (as discussed in the last chapter), it executes in a single cycle, pairing if it runs in the V-pipe; if mispredicted, conditional jumps take 4 cycles in the U-pipe and 5 cycles in the V-pipe, and mispredicted calls and unconditional jumps take 3 cycles in either pipe. Note that **RET** can't pair.

Who's in First?

One of the trickiest things about superscalar optimization is that a given instruction stream can execute at a different speed depending on the pipe where it starts execution, because which instruction goes through the U-pipe first determines which of the following instructions will be able to pair. If we take the last example and add one more instruction, the other instructions will go through different pipes than previously, and cause the loop as a whole to take 50 percent longer, even though we only added 25 percent more cycles:

```
LoopTop:
   inc edx             ;U-pipe cycle 1
   mov [esi],eax       ;V-pipe cycle 1
   add esi,4           ;U-pipe cycle 2
   dec ecx             ;V-pipe cycle 2
   jnz LoopTop         ;U-pipe cycle 3
                       ;V-pipe idle cycle 3
                       ; because JNZ can't
                       ; pair in the U-pipe
```

It's actually not hard to figure out which instructions go through which pipes; just back up until you find an instruction that can't pair or can only go through the U-pipe, and work forward from there, given the knowledge that that instruction executes in the U-pipe. The easiest thing to look for is branches. All branch target instructions execute in the U-pipe, as do all instructions after conditional branches that fall through. Instructions with prefix bytes are generally good U-pipe markers, although they're expensive instructions that should be avoided whenever possible, and have at least one aberration with regard to pipe usage, as discussed below. Shifts, rotates, **ADC**, **SBB**, and all other instructions not listed in Table 20.1 in the last chapter are likewise U-pipe markers.

Pentium Optimization in Action

Now, let's take a look at one of the simplest, tightest pieces of code imaginable, and see what our new Pentium perspective reveals. Listing 21.1 shows a loop implementing the TCP/IP checksum, a 16-bit checksum that wraps carries around to the low bit so that the result is endian-independent. This makes it easy to perform checksums on blocks of data regardless of the endian characteristics of the machines on which those blocks are generated and received. (Thanks to fellow performance enthusiast Terje Mathisen for suggesting this checksum as fertile ground for Pentium optimization, in the ibm.pc/fast.code forum on Bix.) The loop in Listing 21.1 consists of exactly five instructions; it's hard to imagine that there's a lot of performance to be wrung from this snippet, right?

LISTING 21.1 L21-1.ASM

```
; Calculates TCP/IP (16-bit carry-wrapping) checksum for buffer
;  starting at ESI, of length ECX words.
; Returns checksum in AX.
; ECX and ESI destroyed.
; All cycle counts assume 32-bit protected mode.
; Assumes buffer length > 0.
; Note that timing indicates that the pipe sequence and
;  cycle counts shown (based on documented execution rules)
;  differ from the actual execution sequence and cycle counts;
;  this loop has been measured to execute in 5 cycles; apparently,
;  the 1st half of ADD somehow pairs with the prefix byte, or the
;  refix byte gets executed ahead of time.

        sub     ax,ax           ;initialize the checksum

ckloop:
        add     ax,[esi]        ;cycle 1 U-pipe prefix byte
                                ;cycle 1 V-pipe idle (no pairing w/prefix)
                                ;cycle 2 U-pipe 1st half of ADD
                                ;cycle 2 V-pipe idle (register contention)
                                ;cycle 3 U-pipe 2nd half of ADD
                                ;cycle 3 V-pipe idle (register contention)
        adc     ax,0            ;cycle 4 U-pipe prefix byte
                                ;cycle 4 V-pipe idle (no pairing w/prefix)
                                ;cycle 5 U-pipe ADC AX,0
        add     esi,2           ;cycle 5 V-pipe
        dec     ecx             ;cycle 6 U-pipe
        jnz     ckloop          ;cycle 6 V-pipe
```

Wrong, wrong, wrong! As detailed in Listing 21.1, this loop should take 6 cycles per checksummed word in 32-bit protected mode, a ridiculously high number for the Pentium. (You'll see why I say "should take," not "takes," shortly.) We should lose 2 cycles in each pipe to the two size prefixes (because the **ADD**s are 16-bit operations in a 32-bit segment), and another 2 cycles because of register contention that arises when **ADC AX,0** has to wait for the result of **ADD AX,[ESI]**. Then, too, even though **DEC** and **JNZ** can pair and the branch prediction for **JNZ** is presumably correct virtually all the time, they do take a full cycle, and maybe we can do something about that as well.

The first thing to do is to time the code in Listing 21.1 to verify our analysis. When I unleashed the Zen timer on Listing 21.1, I found, to my surprise, that the code actually takes only five cycles per checksum word processed, not six. A little more experimentation revealed that adding a size prefix to the two-cycle **ADD EAX,[ESI]** instruction doesn't cost anything, certainly not the one full cycle in each pipe that a prefix is supposed to take. More experimentation showed that prefix bytes do cost the documented extra cycle when used with one -cycle instructions such as **MOV**. At this point, my preliminary conclusion is that prefixes can pair with the first cycle of at least some multiple-cycle instructions. Determining exactly why this happens will take further research on my part, but the most important conclusion is that you *must* measure your code!

The first, obvious thing we can do to Listing 21.1 is change **ADC AX,0** to **ADC EAX,0**, eliminating a prefix byte and saving a full cycle. Now we're down from five to four cycles. What next?

Listing 21.2 shows one interesting alternative that doesn't really buy us anything. Here, we've eliminated all size prefixes by doing byte-sized **MOV**s and **ADD**s, but because the size prefix on **ADD AX,[ESI]**, for whatever reason, didn't cost anything in Listing 21.1, our efforts are to no avail—Listing 21.2 still takes 4 cycles per checksummed word. What's worth noting about Listing 21.2 is the extent to which the code is broken into simple instructions and reordered so as to avoid size prefixes, register contention, AGIs, and data bank conflicts (the latter because both **[ESI]** and **[ESI+1]** are in the same cache data bank, as discussed in the last chapter).

LISTING 21.2 L21-2.ASM

```
; Calculates TCP/IP (16-bit carry-wrapping) checksum for buffer
;  starting at ESI, of length ECX words.
; Returns checksum in AX.
; High word of EAX, DX, ECX and ESI destroyed.
; All cycle counts assume 32-bit protected mode.
; Assumes buffer length > 0.

        sub     eax,eax         ;initialize the checksum
        mov     dx,[esi]        ;first word to checksum
        dec     ecx             ;we'll do 1 checksum outside the loop
        jz      short ckloopend ;only 1 checksum to do
        add     esi,2           ;point to the next word to checksum

ckloop:
        add     al,dl           ;cycle 1 U-pipe
        mov     dl,[esi]        ;cycle 1 V-pipe
        adc     ah,dh           ;cycle 2 U-pipe
        mov     dh,[esi+1]      ;cycle 2 V-pipe
        adc     eax,0           ;cycle 3 U-pipe
        add     esi,2           ;cycle 3 V-pipe
        dec     ecx             ;cycle 4 U-pipe
        jnz     ckloop          ;cycle 4 V-pipe

ckloopend:
        add     ax,dx           ;checksum the last word
        adc     eax,0
```

Listing 21.3 is a more sophisticated attempt to speed up the checksum calculation. Here we see a hallmark of Pentium optimization: two operations (the checksumming of the current and next pair of words) interleaved together to allow both pipes to run at near maximum capacity. Another hallmark that's apparent in Listing 21.3 is that Pentium-optimized code tends to use more registers and require more instructions than 486-optimized code. Again, note the careful mixing of byte-sized reads to avoid AGIs, register contention, and cache bank collisions, in particular the way in which the byte reads of memory are interspersed with the additions to avoid register contention, and the placement of **ADD ESI,4** to avoid an AGI.

LISTING 21.3 L21-3.ASM

```
; Calculates TCP/IP (16-bit carry-wrapping) checksum for buffer
;  starting at ESI, of length ECX words.
; Returns checksum in AX.
; High word of EAX, BX, EDX, ECX and ESI destroyed.
; All cycle counts assume 32-bit protected mode.
; Assumes buffer length > 0.

        sub     eax,eax             ;initialize the checksum
        sub     edx,edx             ;prepare for later ORing
        shr     ecx,1               ;we'll do two words per loop
        jnc     short ckloopsetup ;even number of words
        mov     ax,[esi]            ;do the odd word
        jz      short ckloopdone  ;no more words to checksum
        add     esi,2               ;point to the next word
ckloopsetup:
         mov     dx,[esi]           ;load most of 1st word to
         mov     bl,[esi+2]         ; checksum (last byte loaded in loop)
         dec     ecx                ;any more dwords to checksum?
         jz    short ckloopend      ;no

ckloop:
        mov     bh,[esi+3]   ;cycle 1 U-pipe
        add     esi,4        ;cycle 1 V-pipe
        shl     ebx,16       ;cycle 2 U-pipe
                             ;cycle 2 V-pipe idle
                             ; (register contention)
        or      ebx,edx      ;cycle 3 U-pipe
        mov     dl,[esi]     ;cycle 3 V-pipe
        add     eax,ebx      ;cycle 4 U-pipe
        mov     bl,[esi+2]   ;cycle 4 V-pipe
        adc     eax,0        ;cycle 5 U-pipe
        mov     dh,[esi+1]   ;cycle 5 V-pipe
        dec     ecx          ;cycle 6 U-pipe
        jnz     ckloop       ;cycle 6 V-pipe

ckloopend:
        mov     bh,[esi+3]   ;checksum the last dword
            add   ax,dx
            adc   ax,bx
            adc   ax,0

            mov     edx,eax  ;compress the 32-bit checksum
            shr     edx,16   ; into a 16-bit checksum
            add     ax,dx
            adc     eax,0
ckloopdone:
```

The checksum loop in Listing 21.3 takes longer than the loop in Listing 21.2, at 6 cycles versus 4 cycles for Listing 21.2—but Listing 21.3 does two checksum operations in those 6 cycles, so we've cut the time per checksum addition from 4 to 3 cycles. You might think that this small an improvement doesn't justify the additional complexity of Listing 21.3, but it is a one-third speedup, well worth it if this is a critical loop—and, in general, if it isn't critical, there's no point in hand-tuning it. That's why I haven't bothered to try to optimize the non-inner-loop code in Listing 21.3; it's only executed once per checksum, so it's unlikely that a cycle or two saved there would make any real-world difference.

Listing 21.3 could be made a bit faster yet with some loop unrolling, but that would make the code quite a bit more complex for relatively little return. Instead, why not make the code more complex and get a *big* return? Listing 21.4 does exactly that by loading one dword at a time to eliminate both the word prefix of Listing 21.1 and the multiple byte-sized accesses of Listing 21.3. An obvious drawback to this is the considerable complexity needed to ensure that the dword accesses are dword-aligned (remember that unaligned dword accesses cost three cycles each), and to handle buffer lengths that aren't dword multiples. I've handled these problems by requiring that the buffer be dword-aligned and a dword multiple in length, which is of course not always the case in the real world. However, the point of these listings is to illustrate Pentium optimization—dword issues, being non-inner-loop stuff, are solvable details that aren't germane to the main focus. In any case, the complexity and assumptions are well justified by the performance of this code: three cycles per loop, or 1.5 cycles per checksummed word, more than three times the speed of the original code. Again, note that the actual order in which the instructions are arranged is dictated by the various optimization hazards of the Pentium.

LISTING 21.4 L21-4.ASM

```
; Calculates TCP/IP (16-bit carry-wrapping) checksum for buffer
;  starting at ESI, of length ECX words.
; Returns checksum in AX.
; High word of EAX, ECX, EDX, and ESI destroyed.
; All cycle counts assume 32-bit protected mode.
; Assumes buffer starts on a dword boundary, is a dword multiple
; in length, and length > 0.

        sub     eax,eax         ;initialize the checksum
        shr     ecx,1           ;we'll do two words per loop
        mov     edx,[esi]       ;preload the first dword
        add     esi,4           ;point to the next dword
        dec     ecx             ;we'll do 1 checksum outside the loop
        jz      short ckloopend ;only 1 checksum to do

ckloop:
        add     eax,edx         ;cycle 1 U-pipe
        mov     edx,[esi]       ;cycle 1 V-pipe
        adc     eax,0           ;cycle 2 U-pipe
        add     esi,4           ;cycle 2 V-pipe
        dec     ecx             ;cycle 3 U-pipe
        jnz     ckloop          ;cycle 3 V-pipe
```

```
ckloopend:
        add     eax,edx         ;checksum the last dword
        adc     eax,0
        mov     edx,eax         ;compress the 32-bit checksum
        shr     edx,16          ; into a 16-bit checksum
        add     ax,dx
        adc     eax,0
```

Listing 21.5 improves upon Listing 21.4 by processing 2 dwords per loop, thereby bringing the time per checksummed word down to exactly 1 cycle. Listing 21.5 basically does nothing but unroll Listing 21.4's loop one time, demonstrating that the venerable optimization technique of loop unrolling still has some life left in it on the Pentium. The cost for this is, as usual, increased code size and complexity, and the use of more registers.

LISTING 21.5 L21-5.ASM

```
; Calculates TCP/IP (16-bit carry-wrapping) checksum for buffer
;  starting at ESI, of length ECX words.
; Returns checksum in AX.
; High word of EAX, EBX, ECX, EDX, and ESI destroyed.
; All cycle counts assume 32-bit protected mode.
; Assumes buffer starts on a dword boundary, is a dword multiple
;  in length, and length > 0.

        sub     eax,eax          ;initialize the checksum
        shr     ecx,2            ;we'll do two dwords per loop
        jnc     short noodddword ;is there an odd dword in buffer?
        mov     eax,[esi]        ;checksum the odd dword
        jz      short ckloopdone ;no, done
        add     esi,4            ;point to the next dword
noodddword:
        mov     edx,[esi]       ;preload the first dword
        mov     ebx,[esi+4]     ;preload the second dword
        dec     ecx             ;we'll do 1 checksum outside the loop
        jz      short ckloopend ;only 1 checksum to do
        add     esi,8           ;point to the next dword

ckloop:
        add     eax,edx         ;cycle 1 U-pipe
        mov     edx,[esi]       ;cycle 1 V-pipe
        adc     eax,ebx         ;cycle 2 U-pipe
        mov     ebx,[esi+4]     ;cycle 2 V-pipe
        adc     eax,0           ;cycle 3 U-pipe
        add     esi,8           ;cycle 3 V-pipe
        dec     ecx             ;cycle 4 U-pipe
        jnz     ckloop          ;cycle 4 V-pipe

ckloopend:
        add     eax,edx         ;checksum the last two dwords
        adc     eax,ebx
        adc     eax,0
ckloopdone:
        mov     edx,eax         ;compress the 32-bit checksum
        shr     edx,16          ; into a 16-bit checksum
        add     ax,dx
        adc     eax,0
```

Listing 21.5 is undeniably intricate code, and not the sort of thing one would choose to write as a matter of course. On the other hand, it's five times as fast as the tight, seemingly-speedy loop in Listing 21.1 (and six times as fast as Listing 21.1 would have been if the prefix byte had behaved as expected). That's an awful lot of speed to wring out of a five-instruction loop, and the TCP/IP checksum is, in fact, used by network software, an area in which a five-times speedup might make a significant difference in overall system performance.

I don't claim that Listing 21.5 is the fastest possible way to do a TCP/IP checksum on a Pentium; in fact, it isn't. Unrolling the loop one more time, together with a trick of Terje's that uses **LEA** to advance ESI (neither **LEA** nor **DEC** affects the carry flag, allowing Terje to add the carry from the previous loop iteration into the next iteration's checksum via **ADC**), produces a version that's a full 33 percent faster. Nonetheless, Listings 21.1 through 21.5 illustrate many of the techniques and considerations in Pentium optimization. Hand-optimization for the Pentium isn't simple, and requires careful measurement to check the efficacy of your optimizations, so reserve it for when you really, really need it—but when you need it, you need it *bad.*

A Quick Note on the 386 and 486

I've mentioned that Pentium-optimized code does fine on the 486, but not always so well on the 386. On a 486, Listing 21.1 runs at 9 cycles per checksummed word, and Listing 21.5 runs at 2.5 cycles per checksummed word, a healthy 3.6-times speedup. On a 386, Listing 21.1 runs at 22 cycles per word; Listing 21.5 runs at 7 cycles per word, a 3.1-times speedup. As is often the case, Pentium optimization helped the other processors, but not as much as it helped the Pentium, and less on the 386 than on the 486.