Chapter 20

Pentium Rules

Chapter 20

How Your Carbon-Based Optimizer Can Put the "Super" in Superscalar

At the 1983 West Coast Computer Faire, my friend Dan Illowsky, Andy Greenberg (co-author of Wizardry, at that time the best-selling computer game ever), and I had an animated discussion about starting a company in the then-budding world of microcomputer software. One hot new software category at the time was educational software, and one of the hottest new educational software companies was Spinnaker Software. Andy used Spinnaker as an example of a company that had been aimed at a good market and started up properly, and was succeeding as a result. Dan didn't buy this; his point was that Spinnaker had been given a bundle of money to get off the ground, and was growing only by spending a lot of that money in order to move its products. "Heck," said Dan, "I could get that kind of market share too if I gave away a fifty-dollar bill with each of my games."

Remember, this was a time when a program, two diskette drives (for duplicating disks), and a couple of ads were enough to start a company, and, in fact, Dan built a very successful game company out of not much more than that. (I'll never forget coming to visit one day and finding his apartment stuffed literally to the walls and ceiling with boxes of diskettes and game packages; he had left a narrow path to the computer so his wife and his mother could get in there to duplicate disks.) Back then, the field was wide open, with just about every competent programmer thinking of striking out on his or her own to try to make their fortune, and Dan and Andy

and I were no exceptions. In short, we were having a perfectly normal conversation, and Dan's comment was both appropriate, and, in retrospect, accurate.

Appropriate, save for one thing: We were having this conversation while walking through a low-rent section of Market Street in San Francisco at night. A bum sitting against a nearby building overheard Dan, and rose up, shouting in a quavering voice loud enough to wake the dead, "Fifty-dollar bill! Fifty-dollar bill! He's giving away fifty-dollar bills!" We ignored him; undaunted, he followed us for a good half mile, stopping every few feet to bellow "fifty-dollar bill!" No one else seemed to notice, and no one hassled us, but I was mighty happy to get to the sanctuary of the Fairmont Hotel and slip inside.

The point is, most actions aren't inherently good or bad; it's all a matter of context. If Dan had uttered the words "fifty-dollar bill" on the West Coast Faire's show floor, no one would have batted an eye. If he had said it in a slightly worse part of town than he did, we might have learned just how fast the three of us could run.

Similarly, there's no such thing as inherently fast code, only fast code in context. At the moment, the context is the Pentium, and the truth is that a sizable number of the x86 optimization tricks that you and I have learned over the past ten years are obsolete on the Pentium. True, the Pentium contains what amounts to about one-and-a-half 486s, but, as we'll see shortly, that doesn't mean that optimized Pentium code looks much like optimized 486 code, or that fast 486 code runs particularly well on a Pentium. (Fast Pentium code, on the other hand, does tend to run well on the 486; the only major downsides are that it's larger, and that the **FXCH** instruction, which is largely free on the Pentium, is expensive on the 486.) So discard your x86 preconceptions as we delve into superscalar optimization for this one-of-a-kind processor.

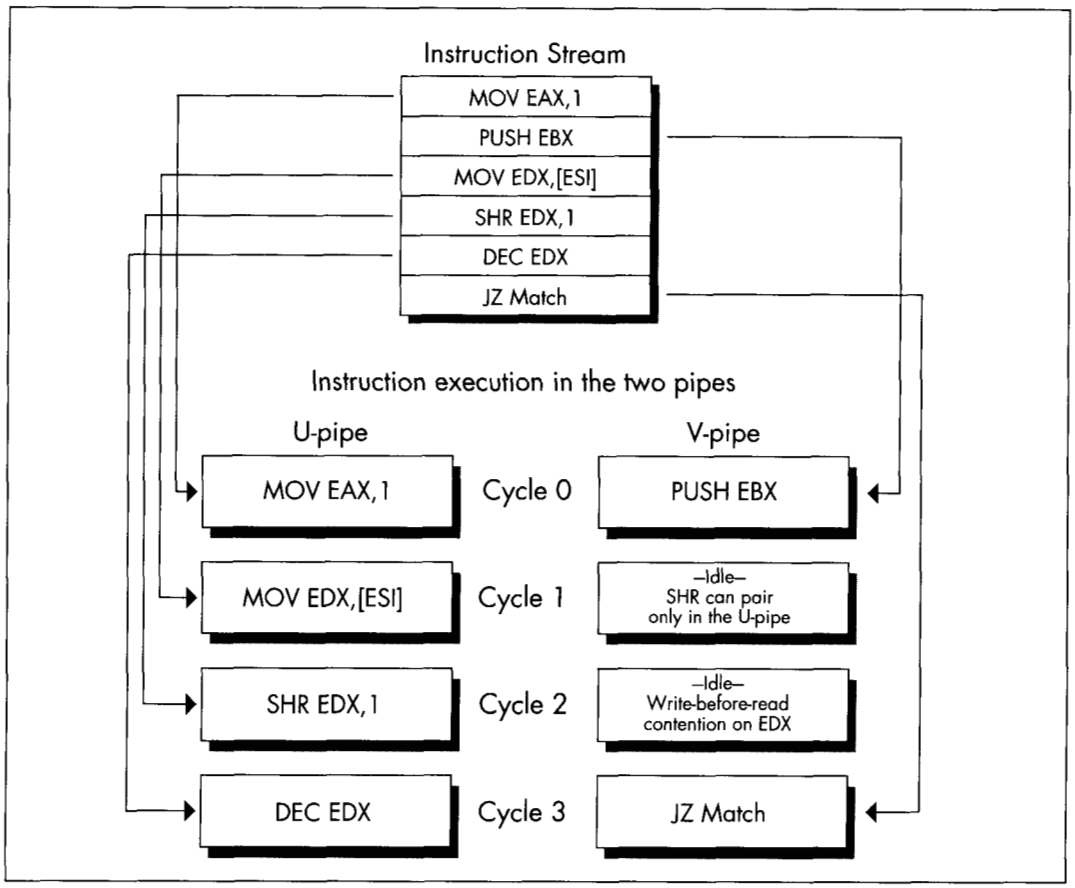
An Instruction in Every Pipe

In the last chapter, we took a quick tour of the Pentium's architecture, and started to look into the Pentium's optimization rules. Now we're ready to get to the key rules, those having to do with the Pentium's most unique and powerful feature, the ability to execute more than one instruction per cycle. This is known as *superscalar execution*, and has heretofore been the sole province of fast RISC CPUs. The Pentium has two integer execution units, called the *U-pipe* and the *V-pipe*, which can execute two separate instructions simultaneously, potentially doubling performance—but only under the proper conditions. (There is also a separate floating-point execution unit that I won't have the space to cover in this book.) Your job, as a performance programmer, is to understand the conditions needed for superscalar performance and make sure they're met, and that's what this and the next chapters are all about.

The two pipes are not independent processors housed in a single chip; that is, the Pentium is not like having two 486s in a single computer. Rather, the two pipes are integral, parallel parts of the same processor. They operate on the same instruction stream, with the V-pipe simply executing the next instruction that the U-pipe would

have handled, as shown in Figure 20.1. What the Pentium does, pure and simple, is execute a single instruction stream and, whenever possible, take the next two waiting instructions and execute both at once, rather than one after the other.

The U-pipe is the more capable of the two pipes, able to execute any instruction in the Pentium's instruction set. (A number of instructions actually use both pipes at once. Logically, though, you can think of such instructions as U-pipe instructions, and of the Pentium optimization model as one in which the U-pipe is able to execute all instructions and is always active, with the objective being to keep the V-pipe also working as much of the time as possible.) The U-pipe is generally similar to a full 486 in terms of both capabilities and instruction cycle counts. The V-pipe is a 486 subset, able to execute simple instructions such as **MOV** and **ADD**, but unable to handle **MUL**, **DIV**, string instructions, any sort of rotation or shift, or even **ADC** or **SBB**.



The Pentium's two pipes.
Figure 20.1

Getting two instructions executing simultaneously in the two pipes is trickier than it sounds, not only because the V-pipe can handle only a relatively small subset of the Pentium's instruction set, but also because those instructions that the V-pipe can handle are able to pair only with certain U-pipe instructions. For example, **MOVSD** uses both pipes, so no instruction can be executed in parallel with **MOVSD**.

*The use of both pipes does make **MOVSD** nearly twice as fast on the Pentium as on the 486, but it's nonetheless slower than using equivalent simpler instructions that allow for superscalar execution. Stick to the Pentium's RISC-like instructions—the pairable instructions I'll discuss next—when you're seeking maximum performance, with just a few exceptions such as **REP MOVS** and **REP STOS**.*

Trickier yet, register contention can shut down the V-pipe on any given cycle, and Address Generation Interlocks (AGIs) can stall either pipe at any time, as we'll see in the next chapter.

The key to Pentium optimization is to view execution as a stream of instructions going through the U- and V-pipes, and to eliminate, as much as possible, instruction mixes that take the V-pipe out of action. In practice, this is not too difficult. The only hard part is keeping in mind the long list of rules governing instruction pairing. The place to begin is with the set of instructions that can go through the V-pipe.

V-Pipe-Capable Instructions

Any instruction can go through the U-pipe, and, for practical purposes, the U-pipe is always executing instructions. (The exceptions are when the U-pipe execution unit is waiting for instruction or data bytes after a cache miss, and when a U-pipe instruction finishes before a paired V-pipe instruction, as I'll discuss below.) Only the instructions shown in Table 20.1 can go through the V-pipe. In addition, the V-pipe can execute a separate instruction only when one of the instructions listed in Table 20.2 is executing in the U-pipe; superscalar execution is not possible while any instruction not listed in Table 20.2 is executing in the U-pipe. So, for example, if you use **SHR EDX,CL**, which takes 4 cycles to execute, no other instructions can execute during those 4 cycles; if, on the other hand, you use **SHR EDX,10**, it will take 1 cycle to execute in the U-pipe, and another instruction can potentially execute concurrently in the V-pipe. (As you can see, similar instruction sequences can have vastly different performance characteristics on the Pentium.)

Basically, after the current instruction or pair of instructions is finished (that is, once neither the U- nor V-pipe is executing anything), the Pentium sends the next instruction through the U-pipe. If the instruction after the one in the U-pipe is an instruction the V-pipe can handle, if the instruction in the U-pipe is pairable, and if register contention doesn't occur, then the V-pipe starts executing that instruction, as shown in Figure 20.2. Otherwise, the second instruction waits until the first instruction is

```
MOV         reg,reg             (1 cycle)
            mem,reg             (1 cycle)
            reg,mem             (1 cycle)
            reg,immediate       (1 cycle)
            mem,immediate       (1 cycle)†

AND/OR/XOR/ADD/SUB   reg,reg          (1 cycle)
                     mem,reg          (3 cycles)
                     reg,mem          (2 cycles)
                     reg,immediate    (1 cycle)
                     mem,immediate    (3 cycles)†

INC/DEC  reg      (1 cycle)
         mem      (3 cycles)

CMP         reg,reg          (1 cycle)
            mem,reg          (2 cycles)
            reg,mem          (2 cycles)
            reg,immediate    (1 cycle)
            mem,immediate    (2 cycles)†

TEST        reg,reg          (1 cycle)
            EAX,immediate    (1 cycle)

PUSH/POP    reg              (1 cycle)
            immediate        (1 cycle)

LEA         reg,mem          (1 cycle)

JCC      near      (1 cycle if predicted correctly;
                    5 cycles otherwise in V-pipe,
                    4 cycles otherwise in U-pipe)

JMP/CALL near      (1 cycle if predicted correctly;
                    3 cycles otherwise)
```

† Can't execute in V-pipe if address contains a displacement

Table 20.1 Instructions that can execute in the V-pipe.

done, then executes in the U-pipe, possibly pairing with the next instruction in line if all pairing conditions are met.

The list of instructions the V-pipe can handle is not very long, and the list of U-pipe pairable instructions is not much longer, but these actually constitute the bulk of the instructions used in PC software. As a result, a fair amount of pairing happens even in normal, non-Pentium-optimized code. This fact, plus the 64-bit 66 MHz bus, branch prediction, dual 8K internal caches, and other Pentium features, together mean that a Pentium is considerably faster than a 486 at the same clock speed, even without Pentium-specific optimization, contrary to some reports.

Besides, almost all operations can be performed by combinations of pairable instructions. For example, **PUSH** [*mem*] is not on either list, but both **MOV** *reg*,[*mem*] and **PUSH** *reg* are, and those two instructions can be used to push a value stored in

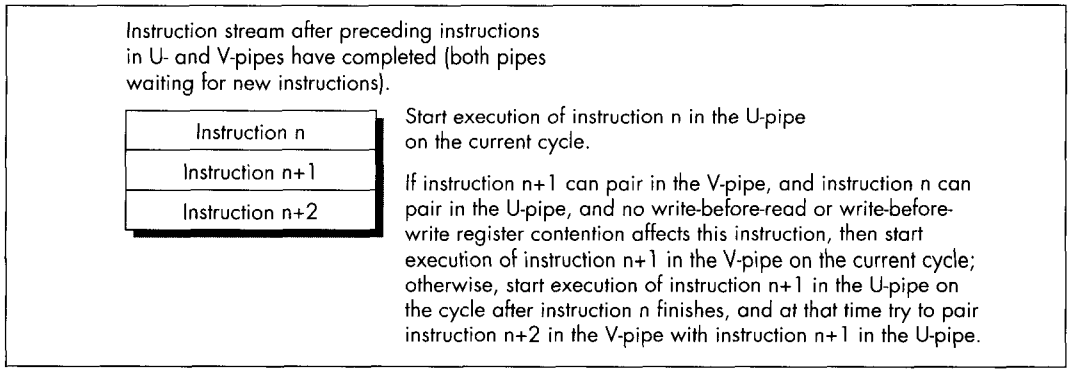
```
MOV       reg,reg           (1 cycle)
          mem,reg           (1 cycle)
          reg,mem           (1 cycle)
          reg,immediate     (1 cycle)
          mem,immediate     (1 cycle)†

AND/OR/XOR/ADD/SUB/ADC/SBB  reg,reg         (1 cycle)
                            mem,reg         (3 cycles)
                            reg,mem         (2 cycles)
                            reg,immediate   (1 cycle)
                            mem,immediate   (3 cycles)'

INC/DEC   reg     (1 cycle)
          mem     (3 cycles)

CMP       reg,reg         (1 cycle)
          mem,reg         (2 cycles)
          reg,mem         (2 cycles)
          reg,immediate   (1 cycle)
          mem,immediate   (2 cycles)†

TEST      reg,reg         (1 cycle)
          EAX,immediate   (1 cycle)

PUSH/POP  reg             (1 cycle)
          immediate       (1 cycle)

LEA       reg,mem         (1 cycle)

SHL/SHR/SAL/SAR  reg,immediate   (1 cycle)''

ROL/ROR/RCL/RCR  reg,1           (1 cycle)
```

† Can't pair if address contains a displacement
†† Includes shift-by-1 forms of instructions

Table 20.2 Instructions that, when executed in the U-pipe, allow V-pipe-executable instructions to execute simultaneously (pair) in the V-pipe.

memory. In fact, given the proper instruction stream, the discrete instructions can perform this operation effectively in just 1 cycle (taking one-half of each of 2 cycles, for 2*0.5 = 1 cycle total execution time), as shown in Figure 20.3—a full cycle *faster* than **PUSH [*mem*]**, which takes 2 cycles.

A fundamental rule of Pentium optimization is that it pays to break complex instructions into equivalent simple instructions, then shuffle the simple instructions for maximum use of the V-pipe. This is true partly because most of the pairable instructions are simple instructions, and partly because breaking instructions into pieces allows more freedom to rearrange code to avoid the AGIs and register contention I'll discuss in the next chapter.



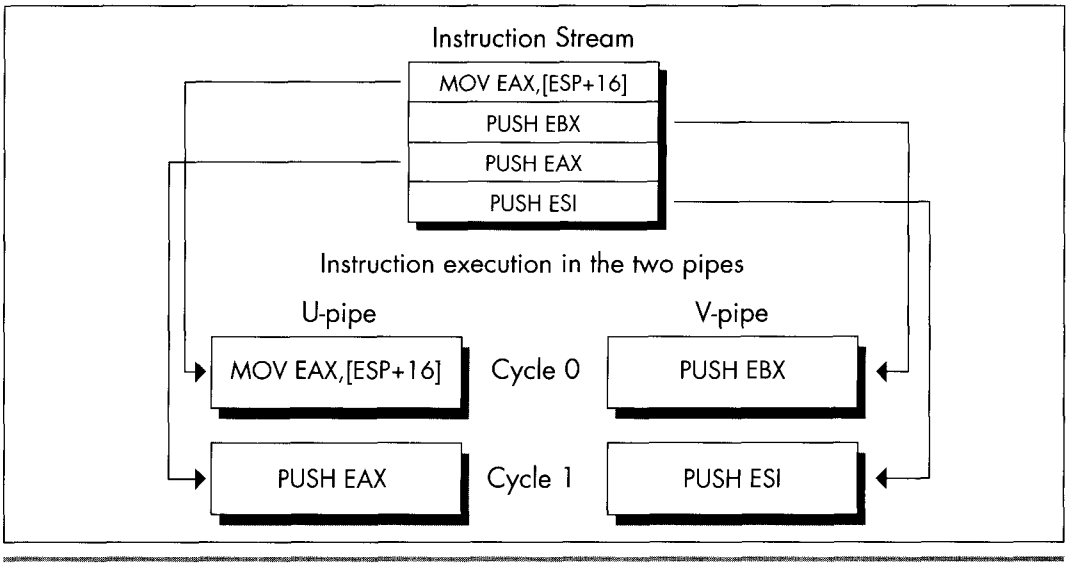
Instruction flow through the two pipes.
Figure 20.2

One downside of this "RISCification" (turning complex instructions into simple, RISC-like ones) of Pentium-optimized code is that it makes for substantially larger code. For example,

```
push dword ptr [esi]
```

is one byte smaller than this sequence:

```
mov eax,[esi]
push eax
```



Pushing a value from memory effectively in one cycle.
Figure 20.3

A more telling example is the following

```
add [MemVar],eax
```

versus the equivalent:

```
mov   edx,[MemVar]
add   edx,eax
mov   [MemVar],edx
```

The single complex instruction takes 3 cycles and is 6 bytes long; with proper sequencing, interleaving the simple instructions with other instructions that don't use EDX or **MemVar**, the three-instruction sequence can be reduced to 1.5 cycles, but it is *14* bytes long.

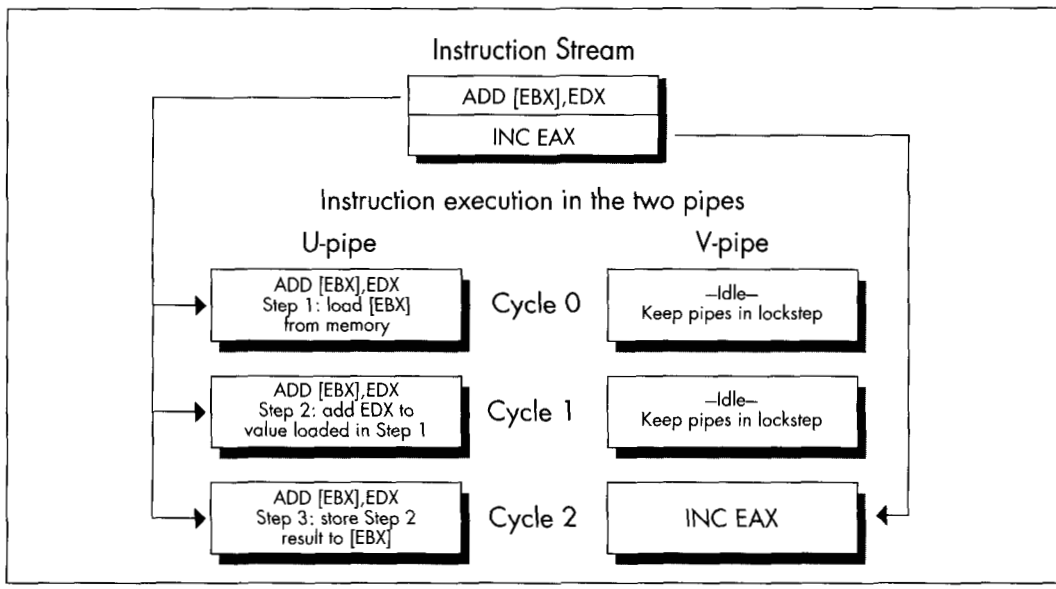
It's not unusual for Pentium optimization to approximately double both performance and code size at the same time. In an important loop, go for performance and ignore the size, but on a program-wide basis, the size bears watching.

Lockstep Execution

You may wonder why anyone would bother breaking **ADD [MemVar],EAX** into three instructions, given that this instruction can go through either pipe with equal ease. The answer is that while the memory-accessing instructions other than **MOV**, **PUSH**, and **POP** listed in Table 20.1 (that is, **INC/DEC [*mem*]**, **ADD/SUB/XOR/AND/OR/CMP/ADC/SBB *reg*,[*mem*]**, and **ADD/SUB/XOR/AND/OR/CMP/ADC/SBB [*mem*],*reg/immed***) can be paired, they do not provide the 100 percent overlap that we seek. If you look at Tables 20.1 and 20.2, you will see that instructions taking from 1 to 3 cycles can pair. However, any pair of instructions goes through the two pipes in lockstep. This means, for example, that if **ADD [EBX],EDX** is going through the U-pipe, and **INC EAX** is going through the V-pipe, the V-pipe will be idle for 2 of the 3 cycles that the U-pipe takes to execute its instruction, as shown in Figure 20.4. Out of the theoretical 6 cycles of work that can be done during this time, we actually get only 4 cycles of work, or 67 percent utilization. Even though these instructions pair, then, this sequence fails to make maximum use of the Pentium's horsepower.

The key here is that when two instructions pair, both execution units are tied up until both instructions have finished (which means at least for the amount of time required for the longer of the two to execute, plus possibly some extra cycles for pairable instructions that can't fully overlap, as described below). The logical conclusion would seem to be that we should strive to pair instructions of the same lengths, but that is often not correct.

The actual rule is that we should strive to pair one-cycle instructions (or, at most, two-cycle instructions, but not three-cycle instructions), which in turn leads to the corollary that we should, in general, use mostly one-cycle instructions when optimizing.

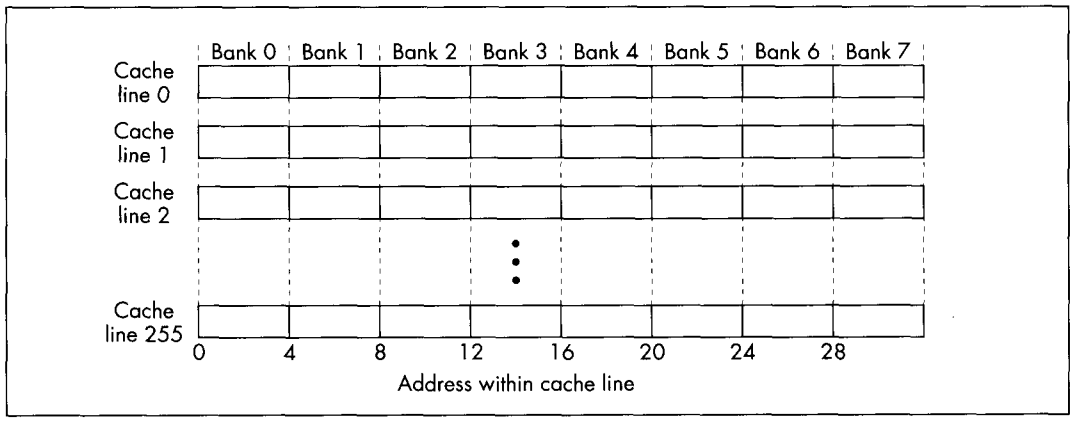


Lockstep execution and idle time in the V-pipe.
Figure 20.4

Here's why. The Pentium is fully capable of handling instructions that use memory operands in either pipe, or, if necessary, in both pipes at once. Each pipe has its own write FIFO, which buffers the last few writes and takes care of writing the data out while the Pentium continues processing. The Pentium also has a write-back internal data cache, so data that is frequently changed doesn't have to be written to external memory (which is much slower than the cache) very often. This combination means that unless you write large blocks of data at a high speed, the Pentium should be able to keep up with both pipes' memory writes without stalling execution.

The Pentium is also designed to satisfy both pipes' needs for reading memory operands with little waiting. The data cache is constructed so that both pipes can read from the cache *on the same cycle.* This feat is accomplished by organizing the data cache as eight-banked memory, as shown in Figure 20.5, with each 32-byte cache line consisting of 8 dwords, 1 in each bank. The banks are independent of one another, so as long as the desired data is in the cache and the U- and V-pipes don't try to read from the same bank on the same cycle, both pipes can read memory operands on the same cycle. (If there is a cache bank collision, the V-pipe instruction stalls for one cycle.)

Normally, you won't pay close attention to which of the eight dword banks your paired memory accesses fall in—that's just too much work—but you might want to watch out for simultaneously read addresses that have the same values for address



The Pentium's eight bank data cache.
Figure 20.5

bits 2, 3, and 4 (fall in the same bank) in tight loops, and you should also avoid sequences like

```
mov   bl,[esi]
mov   bh,[esi+1]
```

because both operands will generally be in the same bank. An alternative is to place another instruction between the two instructions that access the same bank, as in this sequence:

```
mov   bl,[esi]
mov   edi,edx
mov   bh,[esi+1]
```

By the way, the reason a code sequence that takes two instructions to load a single word is attractive in a 32-bit segment is because it takes only one cycle when the two instructions can be paired with other instructions; by contrast, the obvious way of loading BX

```
mov bx,[esi]
```

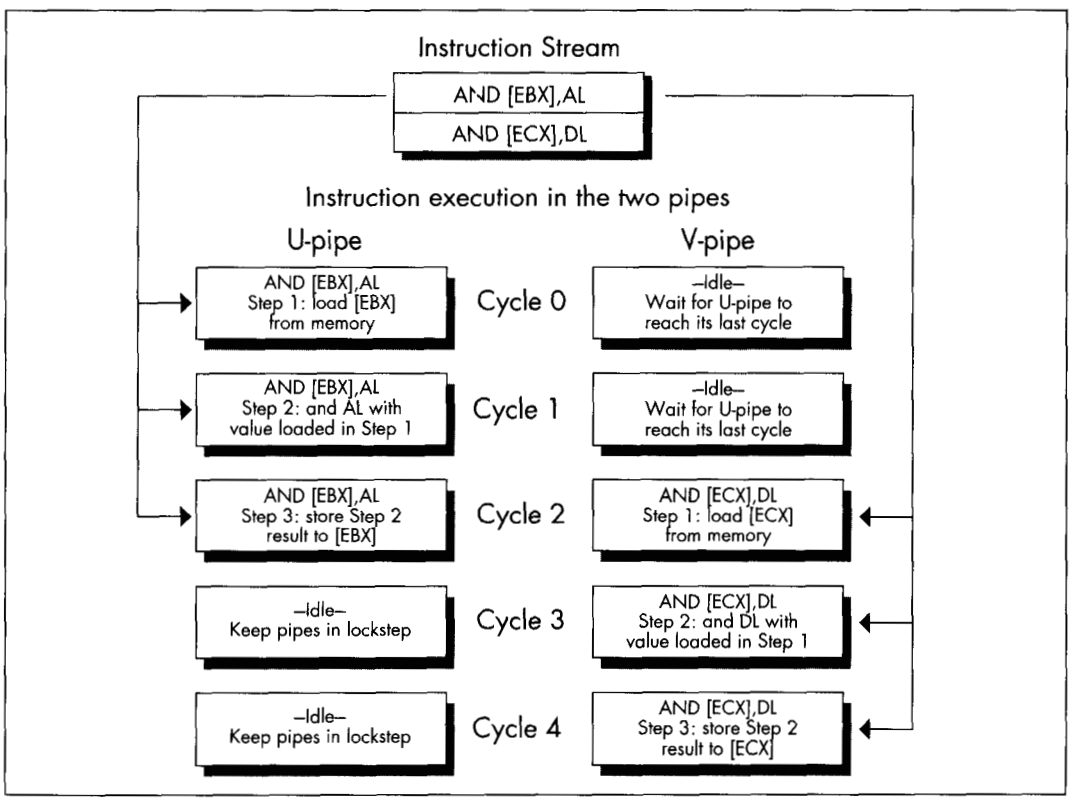
takes 1.5 to two cycles because the size prefix can't pair, as described below. This is yet another example of how different Pentium optimization can be from everything we've learned about its predecessors.

The problem with pairing non-single-cycle instructions arises when a pipe executes an instruction other than **MOV** that has an explicit memory operand. (I'll call these *complex memory instructions.* They're the only pairable instructions, other than branches, that take more than one cycle.) We've already seen that, because instructions go through the pipes in lockstep, if one pipe executes a complex memory instruction

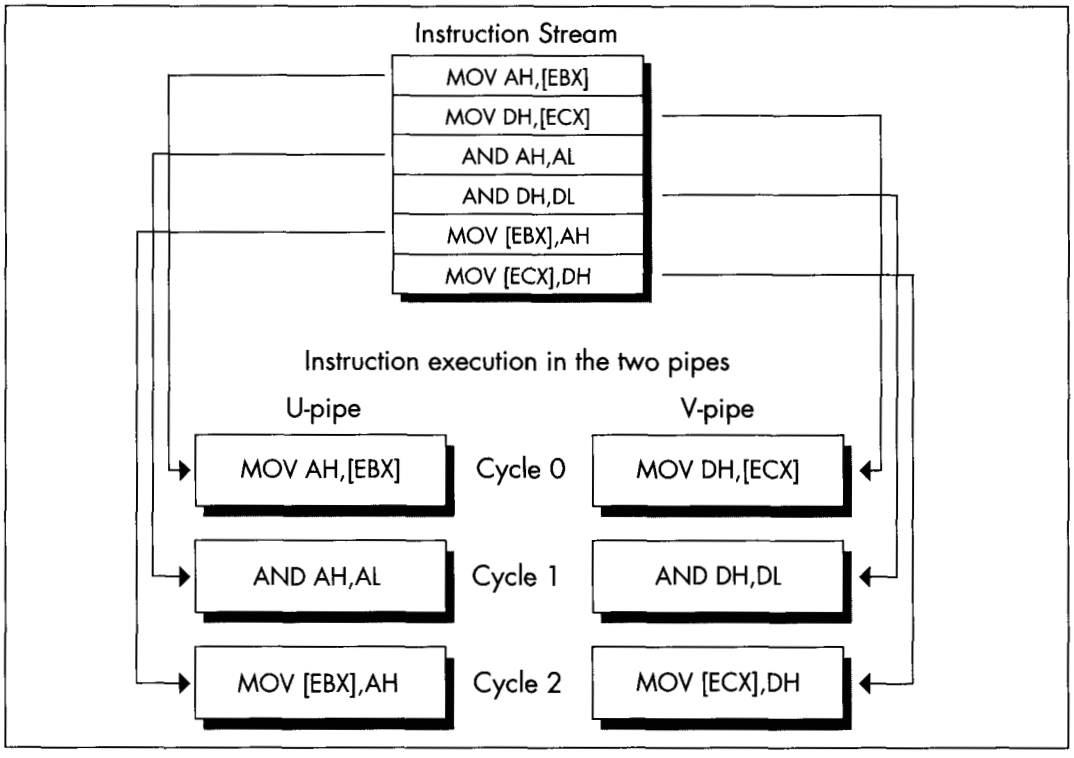
such as **ADD EAX,[EBX]** while the other pipe executes a single-cycle instruction, the pipe with the faster instruction will sit idle for part of the time, wasting cycles. You might think that if both pipes execute complex instructions of the same length, then neither would lie idle, but that turns out to not always be the case. Two two-cycle instructions (instructions with register destination operands) can indeed pair and execute in two cycles, so it's okay to pair two instructions such as these:

```
add esi,[SourceSkip]        ;U-pipe cycles 1 and 2
add edi,[DestinationSkip]   ;V-pipe cycles 1 and 2
```

However, this beneficial pairing does not extend to non-**MOV** instructions with explicit memory destination operands, such as **ADD [EBX],EAX**. The Pentium executes only one such memory instruction at a time; if two memory-destination complex instructions get paired, first the U-pipe instruction is executed, and then the V-pipe instruction, with only one cycle of overlap, as shown in Figure 20.6. I don't know for sure, but I'd guess that this is to guarantee that the two pipes will never perform out-of-order



Non-overlapped lockstep execution.
Figure 20.6



Interleaving simple instructions for maximum performance.
Figure 20.7

access to any given memory location. Thus, even though **AND [EBX],AL** pairs with **AND [ECX],DL**, the two instructions take 5 cycles in all to execute, and 4 cycles of idle time—2 in the U-pipe and 2 in the V-pipe, out of 10 cycles in all—are incurred in the process.

The solution is to break the instructions into simple instructions and interleave them, as shown in Figure 20.7, which accomplishes the same task in 3 cycles, with no idle cycles whatsoever. Figure 20.7 is a good example of what optimized Pentium code generally looks like: mostly one-cycle instructions, mixed together so that at least two operations are in progress at once. It's not the easiest code to read or write, but it's the only way to get both pipes running at capacity.

Superscalar Notes

You may well ask why it's necessary to interleave operations, as is done in Figure 20.7. It seems simpler just to turn

```
and [ebx],al
```

into

```
mov   dl,[ebx]
and   dl,al
mov   [ebx],dl
```

and be done with it. The problem here is one of dependency. Before the Pentium can execute **AND DL,AL**, it must first know what is in DL, and it can't know that until it loads DL from the address pointed to by EBX. Therefore, **AND DL,AL** can't happen until the cycle after **MOV DL,[EBX]** executes. Likewise, the result can't be stored until the cycle after **AND DL,AL** has finished. This means that these instructions, as written, can't possibly pair, so the sequence takes the same three cycles as **AND [EBX],AL**. (Now it should be clear why **AND [EBX], AL** takes 3 cycles.) Consequently, it's necessary to interleave these instructions with instructions that use other registers, so this set of operations can execute in one pipe while the other, unrelated set executes in the other pipe, as is done in Figure 20.7.

What we've just seen is the read-after-write form of the superscalar hazard known as *register contention*. I'll return to the subject of register contention in the next chapter; in the remainder of this chapter I'd like to cover a few short items about superscalar execution.

Register Starvation

The above examples should make it pretty clear that effective superscalar programming puts a lot of strain on the Pentium's relatively small register set. There are only seven general-purpose registers (I strongly suggest using EBP in critical loops), and it does not help to have to sacrifice one of those registers for temporary storage on each complex memory operation; in pre-superscalar days, we used to employ those handy CISC memory instructions to do all that stuff without using any extra registers.

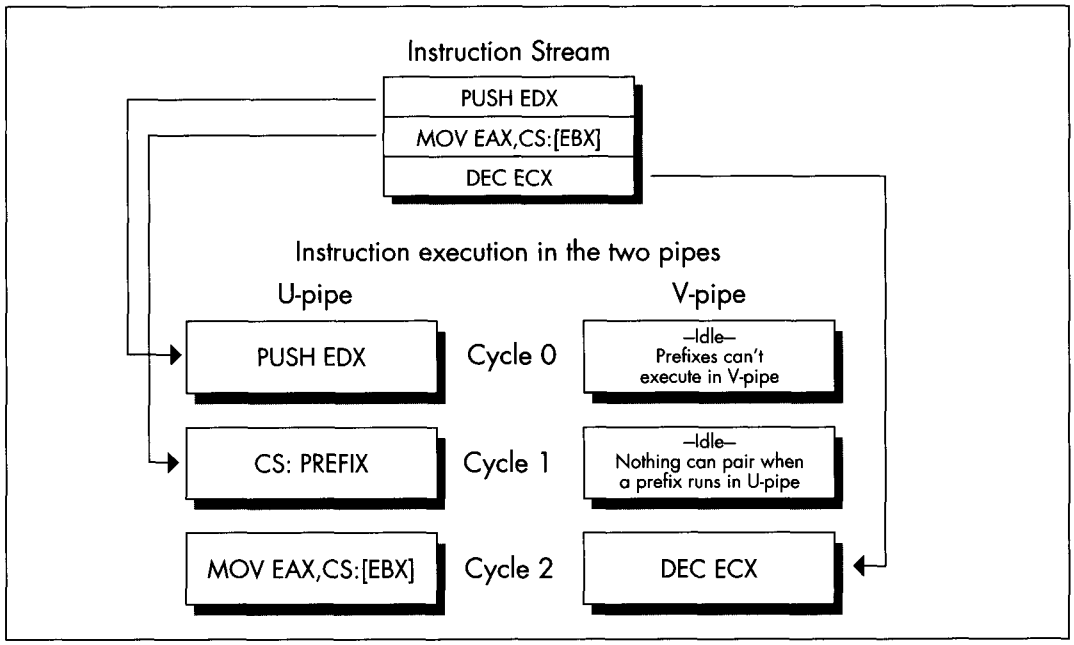
More problematic still is that for maximum pairing, you'll typically have two operations proceeding at once, one in each pipe, and trying to keep two operations in registers at once is difficult indeed. There's not much to be done about this, other than clever and Spartan register usage, but be aware that it's a major element of Pentium performance programming.

Also be aware that prefixes of every sort, with the sole exception of the 0FH prefix on non-short conditional jumps, always execute in the U-pipe, and that Intel's documentation indicates that no pairing can happen while a prefix byte executes. (As I'll discuss in the next chapter, my experiments indicate that this rule doesn't always apply to multiple-cycle instructions, but you still won't go far wrong by assuming that the above rule is correct and trying to eliminate prefix bytes.) A prefix byte takes one cycle to execute; after that cycle, the actual prefixed instruction itself will go through the U-pipe, and if it and the following instruction are mutually pairable, then they

will pair. Nonetheless, prefix bytes are very expensive, effectively taking at least as long as two normal instructions, and possibly, if a prefixed instruction could otherwise have paired in the V-pipe with the previous instruction, taking as long as three normal instructions, as shown in Figure 20.8.

Finally, bear in mind that if the instructions being executed have not already been executed at least once since they were loaded into the internal cache, they can pair only if the first (U-pipe) instruction is not only pairable but also exactly 1 byte long, a category that includes only **INC** ***reg***, **DEC** ***reg***, **PUSH** ***reg***, and **POP** ***reg***. Knowing this can help you understand why sometimes, timing reveals that your code runs slower than it seems it should, although this will generally occur only when the cache working set for the code you're timing is on the order of 8K or more—an awful lot of code to try to optimize.

It should be excruciatingly clear by this point that you *must* time your Pentium-optimized code if you're to have any hope of knowing if your optimizations are working as well as you think they are; there are just too many details involved for you to be sure your optimizations are working properly without checking. My most basic optimization rule has always been to grab the Zen timer and *measure actual performance*—and nowhere is this more true than on the Pentium. Don't believe it until you measure it!



Prefix delays.
Figure 20.8