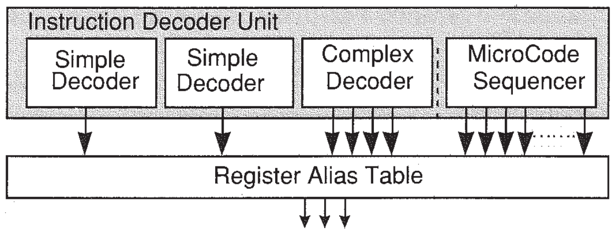


FIGURE 22-4 Instruction decoder.

22.5.2 Performance Considerations

The only thing that you have to worry about, as far as the decoders are concerned, is to apply the 4:1:1 template as often as you can. With the 4:1:1 template, when you schedule your instructions, you need to arrange them such that the first instruction breaks down to four or less micro-ops, and the following two instructions break down to one micro-op each. By repeating this template, you guarantee maximum decoder efficiency. You can easily apply the template with the help of VTune's static analyzer described in the previous chapter. Ideally, the Pentium II processor can decode three instructions every clock cycle. However, in reality, you never sustain this throughput because you cannot always apply the 4:1:1 template or because the decoder stalls from branch misprediction or RAT stalls.

Use the event counters to measure the efficiency of the instruction decoder as follows:

Instructions Decoder per clock = Inst_Decoded / Clock Cycles

22.6 Register Alias Table Unit

22.6.1 Operational Overview

Internally, the Pentium II processor has forty virtual registers, which are used to hold the intermediate calculation results. When a new micro-op is decoded, the Register Alias Table (RAT) unit renames the IA register (*eax*, *ebx*, and so forth) to one of the virtual registers. At any given instance an IA register could be mapped to one or more virtual registers.

How does it work? Consider the following sequence of instructions and their related micro-ops. (Notice that the listed micro-ops are just mnemonics that we made up to illustrate the point.)

The RAT aliases each of the IA registers, *eax* and *edx*, to one of forty internal virtual register *vr1*, *vr2*, and so forth. Notice that the RAT assigns a new virtual

are con-
ɫ:1:1
e them
s, and
ɔy
y. You
ɜr
can
ı never
emplate
ılls.

decoder

h are
o-op is
r (*eax*,
ɔe an IA

and
nemon-

ɔernal
ʳ virtual

TABLE 22-5 IA Instructions and Their Related Micro-ops

IA Instruction	IA Instruction Decoded to Micro-op
mov eax, Mem	uLoad vr0:eax, Mem
add edx, eax	uAdd vr1:edx, vr0:eax
mov eax, 12	uLoad vr2:eax, 12
add eax, ecx	uAdd vr2:eax, vr3:ecx
add ecx, edx	uAdd vr3:ecx, vr1:edx

register for the same IA register *only* when the IA instruction is loaded with a new value. If the register is only read from, the last virtual register is used. In our example, the *eax* register is assigned a new virtual register in both instructions 1 and 3 since both instructions load a new value into *eax*. But in instruction 5 the RAT uses the same virtual *vr1:edx* register since the instruction does not load a new value into *edx*; it is a source operand.

Now, let's see what happens to the micro-ops from Table 22-5 once they're handed to the execution unit:

- In clock 1 the execution unit executes micro-ops 1 and 3—in two different execution ports. Even though both micro-ops write to the same IA register, *eax*, the processor executes the opcodes at the same time since they write to two different virtual registers.
- In clock 2 the execution unit stalls on micro-ops 2 because of the dependency on the *vr0:eax* register from micro-op 1. But micro-op 4 is ready to execute, so it does—assuming that *vr3:ecx* is ready.
- Since micro-op 5 depends on the result of micro-op 2, it can only execute after micro-op 2 executes. Micro-op 2 executes whenever the value of *vr0:eax* gets its value from memory. Meanwhile, the execution unit processes other micro-ops that are ready and waiting in the ROB.

So why is register renaming useful? Consider the third micro-op `uLOad vr2:eax, 12`. Without register renaming, the micro-op has to wait for the first two micro-ops to execute before it can execute; of course, micro-op 4 has to wait as well. With register renaming, micro-ops 3 and 4 were able to execute while the processor was loading data from memory.

22.6.2 Performance Considerations

The RAT is affected by one of the major performance bottlenecks in the Pentium II processor—*partial register stalls*. You'll typically notice such stalls when you run Pentium optimized code on the Pentium II processor. Eliminating partial register stalls is one of the *most obvious* and *most rewarding optimizations* you can achieve on the Pentium II processor.

Partial stalls occur when an instruction that writes to an 8- or 16-bit register (*al, ah, ax*) is followed by an instruction that reads a larger set of that same register (*eax*). For example, the Pentium Pro will suffer a partial stall if you write to the *al* or *ah* register and then read the *ax* or *eax* register.

Notice that partial stalls can still occur even if the second instruction does not immediately follow the first instruction. Since partial register stalls could last for more than 7 cycles, on average, you can avoid partial stalls if you separate the two instructions in question by a minimum of 7 cycles. Or you can fix them.

The Pentium II processor implements special cases to eliminate partial stalls in order to simplify the blending of code across processors. In order to eliminate partial stalls, you must insert the SUB or XOR instructions *in front of* the original instruction and clear out the larger register. Figure 22-5 shows all the possibile partial register stalls and which flavor of the XOR or SUB instructions you can use to eliminate such stalls.

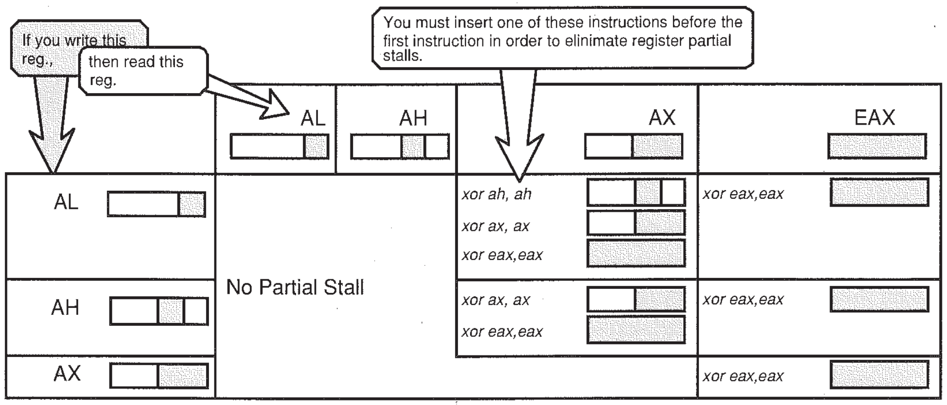


FIGURE 22-5 How to eliminate partial register stalls in the Pentium II processor.

In the three examples we've added the XOR or SUB instructions in front of the original code in order to eliminate partial register stalls.

```
xor ah, ah              sub ax, ax             xor eax, eax
mov al, mem8            mov al, mem8           mov ax, mem16
read ax                 read ax                read eax
```

You can use VTune's static analyzer to easily detect partial register stalls in your code. You can also use the Partial_Rat_Stalls event counter to measure the amount of cycles wasted by register partial stalls.

22.7 Reorder Buffer and Execution Units

22.7.1 Operational Overview

The Reorder Buffer (ROB, a.k.a. Reservation Station) is at the heart of the out-of-order execution of the Pentium II processor. The ROB can receive up to three micro-ops from the RAT and can retire up to three micro-ops in one clock cycle. It can hold a maximum of forty micro-ops at any given time. (See Figure 22-6.)

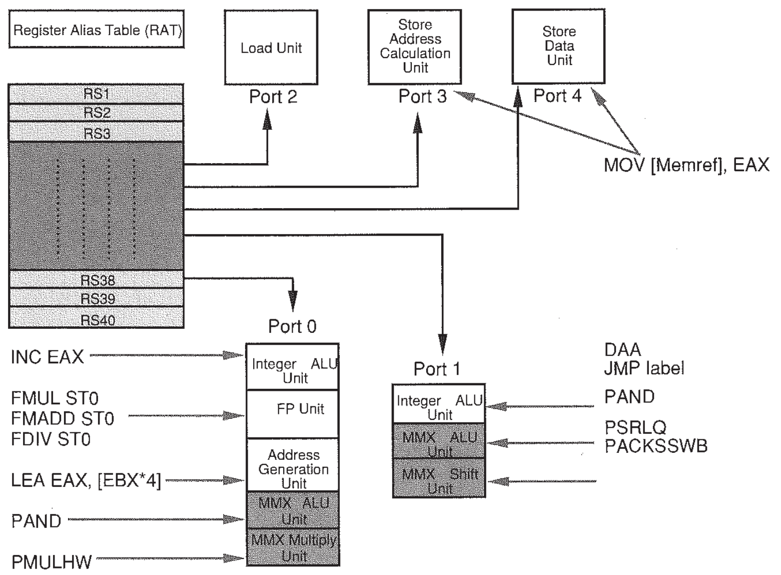


FIGURE 22-6 Pentium II processor Reorder Buffer and the execution unit (port 0–4).

PART VI

The Pentium II processor implements a data flow machine, which leads to the out-of-order execution. In a data flow machine, the order of execution of micro-ops is determined solely by the readiness of their data, not by the order in which it entered the ROB. Let's see how this model works.

	🕒
1. load R4, [R1]	4
2. shiftL R4, 2	1
3. move R2, R3	1
4. shiftL R2, 2	1
5. add R2, R3	1

Consider the coined pseudo-code fragment to the left. Assume that only one instruction can execute, and it takes the number of cycles to the right to execute. In a sequential (in-order) processor, it takes the code fragment 8 clocks to execute.

Now, consider a data flow machine where instructions execute based on the availability of their data not on the order in which they appeared. Let's examine what happens every clock cycle:

1. The first instruction starts to execute immediately.
2. The second instruction stalls for the next 3 clocks in the ROB because it needs the value of R4 to execute. Instead, instruction 3 executes (no data dependency).
3. Instruction 4 executes.
4. Instruction 5 executes. Also, R4 value becomes valid.
5. Instruction 2 is now ready to execute, so it does.

As you can see, with out-of-order execution, it only takes 5 clocks to execute compared to 8 clocks for the sequential execution model. Even though the micro-ops were executed out-of-order, the final results are exactly the same because they are written out in the order they came in.

22.7.2 Performance Considerations

As a programmer, you do not have direct control over the operation of the ROB and the execution unit. But you can affect its behavior indirectly based on your understanding of the internal architecture. Here are a few guidelines that could help you maximize the number of executed micro-ops every clock cycle.

- **Blend your instruction types.** The execution unit has five execution ports that can execute up to five micro-ops in 1 clock cycle. To maximize this number, you should use a mix of instructions as much as possible. Avoid clumping the same kind of operations together (back-to-back loads, stores, ALUs).
- **Minimize mispredicted branches and partial stalls.** Both of these are detrimental to the performance of the ROB and the execution units.

- **Keep your data in the L1 cache.** This allows the load port (2) to bring in the data as fast as possible and in turn avoids data dependency stalls among micro-ops.

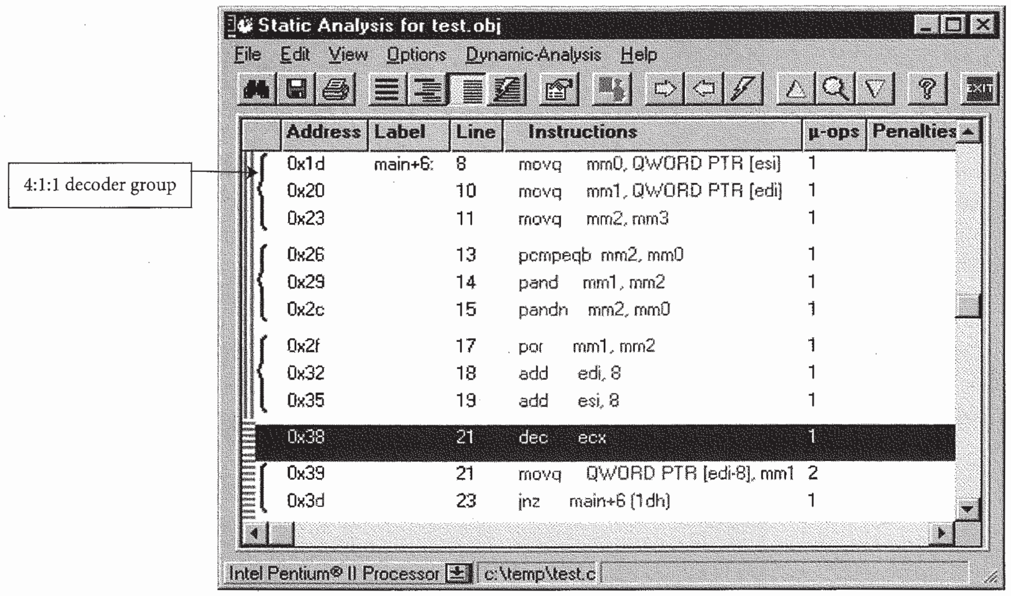
22.8 Retirement Unit

The retirement unit accepts up to three micro-ops in 1 clock cycle. It commits the final results to the IA registers or to memory. The retirement unit guarantees that the micro-ops are retired in the order in which they came into the ROB. There is almost nothing that you can do to affect the performance of the retirement unit.

22.9 Rendering Our Sprite on the Pentium II

Now that we know what's important to the Pentium II processor, let's see if our favorite sprite has any problems when it runs on it. This time, however, we'll use VTune to do the analysis.

Figure 22-7 shows the MMX sprite code analyzed for the Pentium II processor using VTune. Notice that, rather than showing the U/V pairing


4:1:1 decoder group
Static Analysis for test.obj
File Edit View Options Dynamic-Analysis Help
Intel Pentium® II Processor
c:\temp\test.c

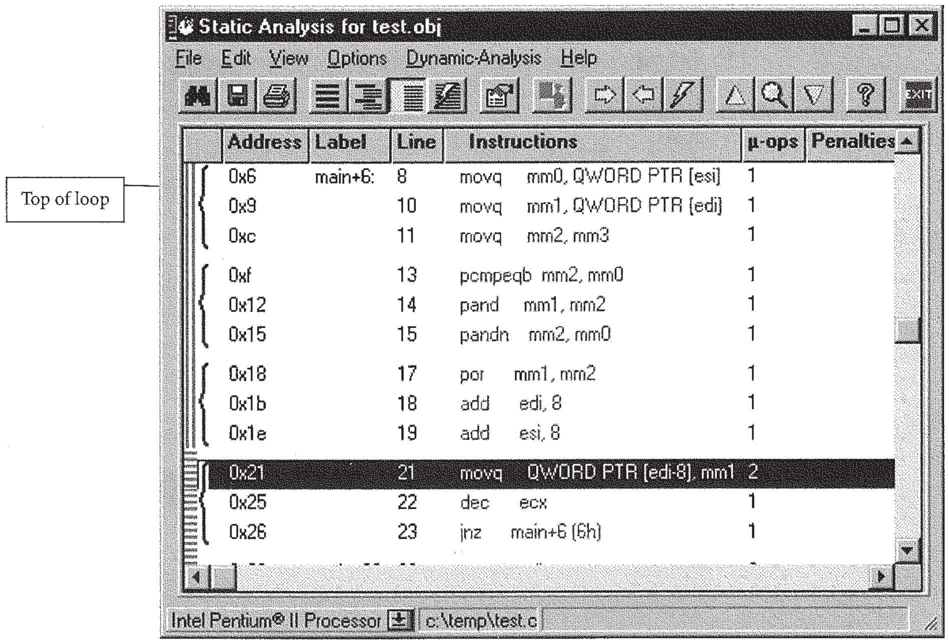
Address	Label	Line	Instructions	µ-ops	Penalties
0x1d	main+6:	8	movq mm0, QWORD PTR [esi]	1	
0x20		10	movq mm1, QWORD PTR [edi]	1	
0x23		11	movq mm2, mm3	1	
0x26		13	pcmpeqb mm2, mm0	1	
0x29		14	pand mm1, mm2	1	
0x2c		15	pandn mm2, mm0	1	
0x2f		17	por mm1, mm2	1	
0x32		18	add edi, 8	1	
0x35		19	add esi, 8	1	
0x38		21	dec ecx	1	
0x39		21	movq QWORD PTR [edi-8], mm1	2	
0x3d		23	jnz main+6 (1dh)	1	

FIGURE 22-7 MMX sprite analysis for the Pentium II processor.

columns, VTune shows a "decoder group" column and a micro-op count column. The decoder group column, indicated by the curly bracket "{," indicates when two or three instructions are decoded simultaneously because they adhere to the 4:1:1 decoder template (refer to section 22.5 for more details). In the "µ-ops" column, VTune shows the number of micro-ops that are generated when the instruction is decoded.

In the figure, notice that the highlighted instruction `dec ecx` was decoded by itself because the instruction sequence does not adhere to the 4:1:1 decoder template. The problem is caused because the `movq [edi-8],mm1` consists of two micro-ops and, thus, has to be the first instruction in a decoder group sequence.

You can easily optimize the code for the Pentium II processor by switching the two instructions. In this case, the `movq [edi-8], mm1` will be decoded by the complex decoder, and the following two instructions are decoded by the two simple decoders. Figure 22-8 shows the results of optimizing our sprite. Note the differences in Line 21 of the number of micro-ops and the improvement gained.


Static Analysis for test.obj
File Edit View Options Dynamic-Analysis Help
Top of loop
Intel Pentium® II Processor
c:\temp\test.c

Address	Label	Line	Instructions	µ-ops	Penalties
0x6	main+6:	8	movq mm0, QWORD PTR [esi]	1	
0x9		10	movq mm1, QWORD PTR [edi]	1	
0xc		11	movq mm2, mm3	1	
0xf		13	pcmpeqb mm2, mm0	1	
0x12		14	pand mm1, mm2	1	
0x15		15	pandn mm2, mm0	1	
0x18		17	por mm1, mm2	1	
0x1b		18	add edi, 8	1	
0x1e		19	add esi, 8	1	
0x21		21	movq QWORD PTR [edi-8], mm1	2	
0x25		22	dec ecx	1	
0x26		23	jnz main+6 (6h)	1	

FIGURE 22-8 MMX Sprite optimized for the Pentium II processor.

VTune also warns you about partial register stalls, which are very useful to remove. Typically, you can remove partial register stalls with little or no impact on performance on the Pentium processor.

Note

In the fetch unit section, we recommended that you align loops on a 16-byte boundary. Notice, however, in Figure 22-8, we did not bother to apply our own recommendation: the top of our loop, "main+6:," is not aligned on a 16-byte boundary. Why not? The purpose of that rule was to assure that the decoder would have three instructions to decode when it jumps to the top of the loop; with luck, the three instructions follow the 4:1:1 rule. If you examine the first three instructions in the loop, you'll notice that they fit within a 16-byte block 0x00 to 0x0F. And since the fetch unit forwards 16 bytes at a time to the decoder, the decoder will have three instructions to decode in these 16 bytes.

22.10 Speed Up Graphics Writes with Write Combining

22.10.1 Operational Overview

By the time the Pentium II processor is in the mainstream market, software-only 3D games and high-resolution MPEG2 video will be widely available. Unfortunately, one of the greatest bottlenecks for these applications is the access speed to graphics memory. A typical software only MPEG2 player consumes up to 30 percent of the CPU writing to video memory.

The Pentium II processor implements the Write Combining (WC) memory type[5] in order to accelerate CPU writes to the video frame buffer. The 32-byte buffer *delays* writes on their way to a WC memory region, so applications can write 32 bytes of data to the WC buffer before it bursts them to their final destination. The 32-byte burst writes are faster than individual byte or DWORD writes, and they consume less bandwidth from the system bus.

Typically, the video driver or the BIOS sets up the frame buffer to be WC (similar to the way it is set up now as uncached memory). As usual, you can use DirectDraw to retrieve the address of the frame buffer. Therefore, there is no change required from an application point of view (well, you might want to read on).

5. Memory type: These include cached, uncached, WC, and other memory types.

Let's have a closer look at WC and determine how it enhances graphics application performance.

Assume that you are writing a 320 × 240 image to a WC frame buffer as shown in Figure 22-9. Typically, you would write the pixels from left to right, sequentially, one pixel at a time. For the sake of simplicity, also assume that the address of the frame buffer is aligned on a 32-byte boundary.

When you write the first 32 bytes of line 1 to the frame buffer, those 32 bytes actually end up in the WC buffer rather than in video memory. Once you write byte 33 to the frame buffer, the WC buffer bursts its contents (the first 32 bytes) to video memory and captures the thirty-third byte instead. Similarly, the next 31 bytes are held in the WC buffer until the sixty-fifth byte is written out. The same process repeats for every package of 32 bytes of data aligned on a 32 byte boundary.

So what about the last 32 bytes in the image. How are they flushed out? They are eventually flushed out when you write somewhere else in the video buffer (for example, when your write out the next frame) or when a task switch occurs. Actually, there are plenty of circumstances that cause the WC buffer to be flushed out:

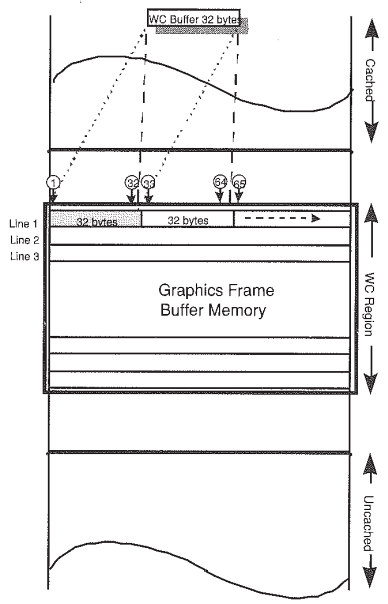


FIGURE 22-9 WC frame buffer.

- Any L1 uncached memory loads or stores (L1 cached loads and stores do not flush the WC buffer).
- Any WC memory loads or WC stores to an address that does not map into the current WC buffer.
- I/O reads or writes.
- Context switches, interrupts, IRET, CPUID, Locked instructions and WBINVD instructions.

Notice that the Pentium II processor generates a 32-byte burst write only if the WC buffer is completely full. Otherwise, it performs multiple smaller writes to the WC region. These multiple writes are still faster than writing to an uncached frame buffer.

22.10.2 Performance Considerations

In short, your WC could enhance your graphics performance if you write your data sequentially to the frame buffer. We have listed the following guidelines to remind you of what you should consider when you optimize for a WC frame buffer.

- Always write sequentially to the frame buffer in order to gain performance from 32-byte WC bursts.
- Avoid writing to the frame buffer vertically. For example, if you write to the first pixel in line 1 then line 2, since the second write does not map to the current WC buffer, the WC buffer (holding only 1 byte) will be flushed out. The same thing happens when you write to line 3, 4, and so forth.

What Have You Learned?

Now you know about the internal units of the Pentium II processor. More importantly, you know what matters to these units so you can get the best performance for your application. As a last reminder:

- Maximize your code execution from the L1 cache,
- Use the new instructions to minimize branches and mispredicted branches.
- Avoid partial stalls. They are deadly.
- Use VTune to analyze performance.
- Use a mix of instructions (loads, stores, ALUs, MMX, and so forth) and apply the 4:1:1 decoder template.
- Use Write Combining to blast your video images to the screen.
- Read the next chapter to familiarize yourself with memory optimization issues.

CHAPTER 23

Memory Optimization: Know Your Data

WHY READ THIS CHAPTER?

Throughout this section, we've stressed again and again that you should "know your data," know where it is coming from and know where it is going. We've also stressed that the optimizations for the internal components of the processor are mostly useful if the code or data is already in the L1 cache. It's a nice premise, but that's not always the case.

In this chapter we'll talk about

- how the data behaves away from home: in the L2 cache or main memory;
- how the data moves between the L1, L2, and main memory and what affects the movement of data;
- how to bring the data into the L1 cache and keep it there as long as it's needed; and
- as an added bonus, accesses to video memory, so you can understand how to write effectively to video memory.

As you know, multimedia applications deal with a huge amount of data that changes continuously from one second to the next. For example, a typical MPEG2[1] clip has 30 fps with a frame size of 704×480 pixels per frame at an average of 12 bits per pixel. Moreover, since MPEG2 uses bidirectional frame prediction, the size of the working data set[2] is typically three to four

1. *MPEG2* is a High Resolution Motion Video Compression Algorithm.
2. The working data set refers to the maximum size of data that is used by the application at any given moment.

times the size of one frame. Taking all of this into account, you can calculate the size of the working data set for an MPEG2 decoder as follows:

$$\text{Data Set Size} = \frac{4 \text{ frames} * (704 * 408 \text{ pixels}) * 12 \text{ bits/pixel}}{8 \text{ bits/byte}} = 1.9 \text{ MB}$$

All of these bits definitely do not fit in the L1 cache or even in the L2 cache—the L1 cache is 8 or 16K, and the L2 cache ranges between 256 and 512K. Therefore, at any given moment, the majority of the data resides in main memory rather than in the caches.

The main purpose of this chapter is to emphasize that memory access can be very costly, in terms of clock cycles, and to highlight certain access patterns that are more efficient than others. We'll also point out the differences between the various flavors of the Pentium and Pentium Pro processors with regards to cache and memory behavior. We'll top the chapter off with a brief discussion about accessing video memory.

23.1 Overview of the Memory Subsystem

23.1.1 Architectural Overview

Figure 23-1 shows a simplistic diagram of the memory subsystem for computers with the Pentium II processor. Notice that the L1 code and data caches are internal to the processor and run at the same speed as the core engine. The L2 cache resides on a dedicated L2 bus, external to the processor, and runs at one half to one third the speed of the processor.[3] The memory subsystem is connected to the PCI chip set, which connects the processor to main memory, PCI bus, and other peripheral devices.

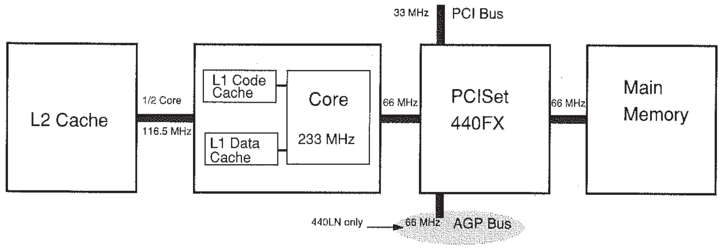


FIGURE 23-1 Memory architecture of a system with the Pentium II processor.

3. The fraction of the bus speed depends on the type of L2 cache used and the speed of the processor.

The PCI chip set is the glue logic between the processor, memory, DMA, and the PCI and AGP[4] buses. It manages and controls the traffic between the processor and all of these devices. A dedicated bus connects the system memory to the PCISet. The PCI bus connects the PCISet to I/O adapters, such as graphics, sound, and network cards. The AGP bus is a specialized graphics bus that was designed with 3D acceleration in mind; notice that the 440LX PCISet is the first chip set with the AGP bus.

23.1.2 Memory Pages and Memory Access Patterns

We've mentioned, throughout this section, that the L1 and L2 caches are divided into 32-byte cache lines, which represent the least amount of data that can be transferred between the L1 cache and main memory. For the curious only: you can find out more about the internal architecture of the caches from the Intel manuals (things like two-way and four-way set associate, and so forth).

Internally, the system memory is divided into smaller units called *memory pages.* Memory pages are typically 2K in size and are aligned on a 2K boundary. The only reason we're talking about memory pages here is that because of the design of DRAM chips, certain memory access patterns are more efficient than others. In the discussion that follows, you need to come out with one thing: *consecutive accesses within the same memory page are more efficient than consecutive accesses that cross multiple memory pages.*

In this discussion, we're assuming that the processor missed both the L1 and L2 caches and that it is now fetching data from main memory. As we mentioned earlier, the processor fetches an entire cache at a time from main memory and writes it out to the cache. Since the processor has a 64-bit data bus, it can fetch an entire cache line with *four* bus transactions.

Now, when the processor requests data from main memory, the memory page where the data exists is first "opened"—this is done in the hardware—and then the data is retrieved. Once the page is open, it takes less time to read or write other data to the same page. Typically, the data sheet for the memory chip specifies how long it takes to open the page and perform the first read, and how long it takes to perform subsequent reads once the page is open.

4. The Accelerated Graphics Port (AGP) is a specialized graphics bus designed with 3D rendering in mind.

For example, the data sheet of an Enhanced Data Out (EDO) memory chip specifies the sequence {10-2-2-2}{3-2-2-2} where the numbers represent clock cycles. Each curly bracket indicates four bus cycles of 64 bits each—that's one cache line. The first sequence, {10-2-2-2}, specifies the timing if the page is first opened and accessed four times. The second sequence, {3-2-2-2}, specifies the timing if the page was already open and accessed four additional times—that means you did not access any other memory page in between. The last sequence repeats as long as you access memory within the same page. One last thing: only one memory page can be open at any given moment.

The data sheet we have been discussing relates to a memory bus running at 66 MHz. Now, if we look at another processing speed, say a 233-MHz processor, the timing becomes {35-7-7-7}{11-7-7-7} in processor clocks.

Whenever your application jumps to another memory page, the current open page is first closed before opening the new page. As a result, it takes an additional 24 processor clocks to switch between memory pages—that's a lot of processor clocks to waste. So what can you do about it? Maybe nothing! Maybe a lot! The whole point is that you should try to organize your memory footprint in such a way that you bring the data from main memory to the L1 cache in the most efficient manner. For example, if you know that most of your data resides in main memory, for example, MPEG2, you might try to arrange the data in a smarter fashion such that you can burst it to the L1 cache faster.

In MPEG2's motion compensation,[5] for example, you typically access three reference frame buffers and write the output to a fourth buffer or directly to the screen. Typically, when the buffers are allocated, they are allocated in a contiguous fashion, separately, as shown in Figure 23-2.With the allocation scheme shown in Figure 23-2a, when you access the three frames, you'll definitely cross memory page boundaries and thus reduce the overall application performance. Now, if you interleave the frames on a line-by-line boundary, as shown in Figure 23-2b, you'll have a better chance of accessing the three frames from the same memory page, and thus increasing memory access efficiency.

5. Motion Compensation is used when inter-frame decoding is used.

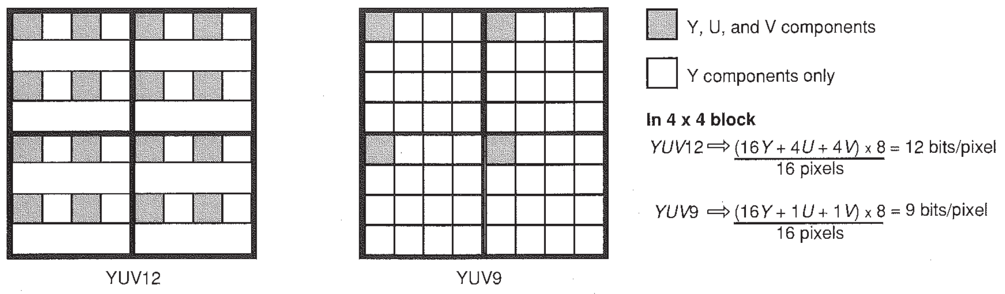

YUV12
YUV9
Y, U, and V components
Y components only
In 4 x 4 block
YUV12 ⟹ (16Y + 4U + 4V) x 8 / 16 pixels = 12 bits/pixel
YUV9 ⟹ (16Y + 1U + 1V) x 8 / 16 pixels = 9 bits/pixel

FIGURE 23-2 MPEG2 frame buffer allocation strategy.

23.1.3 Memory Timing

To complete the picture, let's look at a comparison of the L1 and L2 caches and system memory.

TABLE 23-1 Memory Architecture and Timing for a System Using the Pentium II Processor and EDO Memory

Bus	Bus Clocks	CPU Clocks at 233 MHz	Total CPU Clocks (Bandwidth)
L1 cache	{1-1-1-1}	{1-1-1-1}	4 (1864 MB/Second)
L2 cache	{5-1-1-1}	{10-2-2-2}	16 (466 MB/Second)

EDO memory	{10-2-2-2}
	{3-2-2-2}
		{35-7-7-7}
		{11-7-7-7}
			56 (133 MB/Second)
			32 (233 MB/Second)

SDRAM	{11-1-1-1}
	{2-1-1-1}
		{39-4-4-4}
		{7-4-4-4}
			51 (146 MB/Second)
			19 (392 MB/Second)

Access timing for main memory depends on the type of PCISet and memory used in the system (available types include EDO, FPRAM, SDRAM[6]). SDRAM offers the best access timing because it has a lower repetition[7] rate {11-1-1-1}{2-1-1-1} relative to EDO {10-2-2-2}{3-2-2-2}. But SDRAMs are only supported on systems with the PCISet 440/LX chip set or later.

6. EDO: Enhanced Data Out; FPRAM: FastPage RAM; SDRAM: Synchronous DRAM.
7. Repetition rate: the timing for fetching the last 3 quad words in a cache line.

PART VI

From the CPU point of view, notice that the total number of clocks spent accessing main memory depends on the speed of the processor. Faster processors actually wait more clocks for memory than do slower processors. For example, if a memory chip takes one nanosecond to respond, a processor running at 233 MHz waits 233 clocks before it receives the data, and a 200 MHz processor waits 200 clocks before it gets the same data. Even though both processors waited the same physical time, 1 nanosecond, the faster processor ticked more clocks in that time—and thus it is losing more clocks that could be spent doing something more useful.

23.1.4 Performance Considerations

The Pentium II processor includes event counters that can help you understand the memory footprints of your application. Notice that even though some of these counters are not 100 percent accurate, they can give you a good indication of your application cache and memory behavior.

TABLE 23-2 Pentium II Processor Cache and Bus Performance Event Counters

Event Counter	Usage
DATA_MemRef	All memory accesses including reads and writes to any memory type
L2_LD, L2_ST	Number of data load/store that miss in the L1 data cache and are issued to the L2 cache
L2_LD_Ifetch	All instruction and data load requests that miss the L1 cache and are issued to the L2 cache
L2_Rqsts	All L2 requests including data loads/stores, instruction fetches, and locked accesses
BUS_TranAny	Number of all transactions on the bus
BUS_Tran_BRD	Number of data cache line reads from the bus
BUS_Trans_WB	Number of cache lines evicted from the L2 cache because of conflict with another cache line
BUS_BrdyClocks	Number of clocks when the bus is not idle

Assuming that you can quantify the amount of data that you read and write in a portion of your application, you can derive the following formulas:

$$\text{L1 Data Miss Ratio} = \frac{\text{L2\_LD} + \text{L2\_ST}}{\text{Total Mem Ref}}$$

Since L1 cache misses generate L2 cache accesses, we are using the L2 event counters to quantify the L1 data miss ratio rather than using the DCU (L1) event counters.

$$\text{L2 Data Read Miss Ratio} = \frac{\text{BUS\_TranBRD} - \text{BUS\_TranIFetch}}{\text{Total Mem Ref}}$$

$$\%\ \text{L2 Data Requests} = \frac{\text{L2\_LD} + \text{L2\_ST}}{\text{L2\_Rqsts}}$$

The L2 data read miss ratio represents the number of cache line reads or writes that missed the L2 cache and caused a line to be brought in from memory. L2 holds both instruction and data. The *%L2* data requests represent the percentage of data accesses only from L2.

$$\text{Bus Utilization} = \frac{\text{BUS\_BrdyClocks}}{\text{Total Clocks}}$$

$$\%\ \text{Bus Data Reads} = \frac{\text{BUS\_TranBRD} - \text{BUS\_TranIFetch}}{\text{BUS\_TranAny}}$$

The Bus Utilization indicates how often the bus is busy moving data around (not idle). This includes all bus transactions whether it's from the CPU or from another bus master, DMA, or another processor.

The *%Bus Data Reads* represents the percentage of the bus used for data reads.

23.2 Architectural Differences among the Pentium and Pentium Pro Processors

To optimize your application for multiple IA processors, you need to pay attention to some of the architectural differences between the Pentium and Pentium Pro processors. For example, there are differences in the behavior of the cache subsystem and the organization of the Write buffers. These architectural differences affect the way you should proceed in optimizing your memory.

23.2.1 Architectural Cache Differences

On Pentium processors, when you write to an address in memory that does not exist in the L1 cache, the data is written directly to the L2 cache without touching the L1 cache. If the data does not exist in the L2 cache, the data is written directly to system memory without touching the L2 cache. This is known as a *Read Allocate Cache.*

Watch out if

- Only small portion of cache line is touched or
- Write stride is greater than 32 bytes cache line.

On Pentium Pro processors, if the processor encounters a cache write miss, it first bursts the entire cache line to the L1 cache from main memory or the L2 cache, and then writes the data to the L1. This is known as a *Write Allocate on a Write Cache Miss.* This behavior is typically advantageous since sequential stores in the same cache line are faster because they hit the L1 cache—unlike the Pentium processor where they'll be written through. In addition, when the stores are committed to main memory or the L2 cache, they are committed in one 32-byte burst write, which is faster than individual memory writes—thus reducing overall bus utilization.

The Pentium Pro processor implements a nonblocking cache compared to the Pentium processor, which implements a blocking cache. When the Pentium processor encountered a read miss, first the processor has to satisfy the read before it continues execution at the next instruction. When the Pentium Pro processor encounters a read miss in the L1 cache, it blocks the execution of that specific micro-op and all future micro-ops depending on its results; but it allows other micro-ops to execute and even access data off the L1 cache.

Processors with MMX technology double the size of the L1 cache relative to their non-MMX counterparts. The Pentium and Pentium pro processors include two independent instruction and data L1 caches of 8K each. Processors with MMX technology include two independent instruction and data L1 caches of 16K each.

23.2.2 Write Buffer Differences

Write buffers allow the processor to go on to the next instruction while it is writing data to uncached memory, writing through memory, or when the write misses the L1 cache. Instead of waiting for the write to go all the way to memory, the processor places the data in one of the Write buffers and goes to the next instruction. The Write buffers are flushed out to memory when the data bus is available or on the next write to a full Write buffer.

As we mentioned in the Pentium processor chapter, the Pentium processor has two dedicated 32-bit Write buffers: one for the U pipe and the other for the V pipe. The Pentium processor with MMX technology has four independent 32-bit Write buffers, all of which can be accessed from either pipe.

```
Sequence 1
1.  mov [esi], eax      <- U
2.     inc esi          <- v
3.  mov [edi], ebx      <- U
4.     inc edi          <- v

Sequence 2
1.  mov [esi], eax      <- U
2.     mov [edi], ebx   <- v
3.  inc esi             <- U
4.     inc edi          <- v
```

For higher write performance on the Pentium processor, you should arrange your memory writes through both pipelines, rather than through just one. Consider the first code sequence to the right where instructions 1 and 3 are both issued in the U pipe. When instruction 1 executes, it writes its data into the dedicated U pipe Write buffer, allowing the processor to execute the next instruction. But when instruction 3 executes in the U pipe, the processor stalls until the contents of the U pipe Write buffer are flushed out to memory. Now, if you rearrange the code as shown in Sequence 2 the second write will be issued in the V pipe and will end up in the V pipe's dedicated Write buffer—and the processor can go on to the next instruction in both pipes.

On the Pentium processor with MMX technology, both sequences execute the same since both pipelines can write to any of the four Write buffers.

The Pentium Pro and Pentium II processors implement four independent *32-byte* Write buffers. The Write buffers temporarily hold memory writes until the bus is available. They combine multiple data writes into larger memory writes—up to 32 bytes each—which can be burst to main memory. Typically, you don't have to worry about scheduling instructions for the Write buffers since you cannot easily affect their behavior.

23.2.3 Data Controlled Unit Splits on the Pentium Pro Processor

DCU splits happen on Pentium Pro processors *without* MMX technology, when an unaligned access crosses a cache line boundary. On average, the processor takes 9–12 cycles to recover from a DCU split—that is a huge amount of time compared to the 1 cycle that it takes for aligned access.

In addition, Pentium Pro processors *without* MMX technology encounter a similar problem when an unaligned cache access crosses an 8-byte boundary. Such a split imposes a 5–7 clock penalty on the processor.

You can minimize DCU splits by minimizing misaligned memory accesses. You can use the Misalign_MemRef event counter to quantify the amount of DCU splits in your application. Notice that this counter only counts the number of misaligned data memory references that cross an 8-byte boundary rather than all misaligned accesses. Since the other misaligned accesses, DWORDs, for example, do not affect performance, there is no need to count them.

23.2.4 Partial Memory Stalls

The Pentium Pro and Pentium II processors stall when a memory store is followed by a memory load of a different data size or alignment. Notice that this problem is different from but similar to the *partial register stall* problem. When a partial memory stall occurs, the micro-op that wants to load memory has to wait until the micro-op that stored the data retires—and that could take a long time depending on the state of the machine. You can easily avoid such stalls by rewriting the code to avoid the penalty. Even though you might end up with more instructions to execute, the extra instructions can reduce stall time considerably.

In Figure 23-3, you see a list of all the situations in which a partial memory stall can crop up. The highlighted text is a modified sequence of code that will accomplish the same exact thing as the original code, only without the partial memory stall.

👎	`mov [esi], cx`	16 bit store	`mov [esi], cx`	16 bit store
	`mov eax, [esi]`	32 bit load	`mov eax, [esi+1]`	32 bit load
	`mov eax, [esi]`	*No Partial Memory Stall*	`mov eax, [esi+1]`	*No Partial Memory Stall*
👍	`mov [esi], cx`		`mov [esi], cx`	
	`mov ax, cx`		`mov al, ch`	
👎	`mov [esi], ecx`	32 bit store	`mov [esi],eax`	32 bit store
	`mov eax, [esi+2]`	32 bit load	`movq mm0,[esi]`	64 bit load
	`mov eax, [esi+2]`	*No Partial Memory Stall*	`mov [esi], eax`	*No Partial Memory Stall*
👍	`mov [esi], ecx`		`movd mm0, [esi+4]`	
	`shr ecx, 16`		`movd mm1, eax`	
	`mov ax, cx`		`psllq mm0, 32`	
			`por mm0, mm1`	
👍	`mov [esi], eax`	32 bit store	`movq [esi],mm0`	64 bit store
	`add ebx,[esi]`	32 bit load	`pand mm1, [esi]`	64 bit load

FIGURE 23-4 Restarting your code to avoid partial stalls.

23.3 Maximizing Aligned Data and MMX Stack Accesses

You recall, from Chapter 20, that MMX instructions that perform unaligned accesses to video memory execute more slowly than do instructions that perform aligned accesses. Actually, the same concept applies to all types of memory accesses including integer, floating point, and MMX. On an unaligned memory access, the Pentium and Pentium Pro processors split the unaligned memory accesses into 2 bus cycles, causing a slowdown by more than 50 percent.

The Pentium processor takes 3 cycles to execute an unaligned cache access. The Pentium Pro processor wastes 5–7 cycles on unaligned cache accesses that cross a 64-bit boundary and 9–12 cycles on unaligned cache accesses that cross a cache-line boundary (DCU splits).

Unaligned accesses to *uncached* memory are split into two accesses, and the result is degradation of application performance. It's bad enough that uncached memory accesses take a long time to execute; unaligned memory accesses to uncached memory could take double the time to execute and can drastically degrade application performance.

23.3.1 The Pitfalls of Unaligned MMX Stack Access

MMX = __int64: Compilers align local and global variables according to their types.
Declare MMX variables with the __INT64 TYPE.

One of the common pitfalls in MMX programming is accepting the default compiler alignment for function parameters' variables. When a function is called, the compiler ensures that the function parameters are aligned on a 4-byte boundary, which is not ideal for MMX instruction performance. To remedy this problem, copy any MMX function parameters to local variables and use the local variables instead, as follows:

```
void MMXFunction (
    int iWidth,
    __int64 iColor)                         ◊ Parameter __int64 aligned on 4 byte.
{
    __int64 iColorCopy = iColor;            ◊ Local __int64 aligned on 8 byte.
    -> Use iColorCopy in function
}
```

23.4 Accessing Cached Memory

So what's the moral of the story? Well, there are two: (1) maximize your "good" accesses from the L1 cache; and (2) bring in the data to the L1 cache as fast as possible.

You've already seen what a good cache access can accomplish in the above discussions about aligned accesses, DCU splits, and so forth. You can reap the best benefits of such accesses if you maximize your L1 data accesses.

What do we mean? Let's assume that you want to access a 32-K buffer multiple times within a loop, and you have obeyed all the good access rules above (assuring proper alignment, avoiding DCU splits, and so forth). First, notice that the buffer size is larger than the L1 cache. In this case, if you access the entire buffer on every pass of the loop, when you access the second half of the buffer, the first half will be evicted from the L1 cache. As you restart at the top of the buffer, the first half of the buffer will be brought into the L1 cache, again, and the second half will be evicted. Now, depending on your application, you might be able to avoid thrashing in the L1 cache by breaking the processing of your loop into multiple parts and accessing half of the data at a time.

What about the issue of bursting data from main memory to the L1 cache on the Pentium processor family? As we mentioned in the Pentium processor chapter, it is advantageous to pre-allocate the data specially if you expect back-to-back L1 cache misses or if you will be performing multiple writes to uncached memory (refer to Chapter 19 for more details). But keep in mind that preallocation is useful only if (1) the size of the data set does not fit in the L2 cache (if it does, pre-allocation might actually take more cycles); or (2) you use the majority of the data that you pre-allocate into the L1 cache.

23.5 Writing to Video Memory

23.5.1 Using Aligned Accesses to Video Memory

In Chapter 20, you've seen that unaligned writes to video memory take much longer to execute than do aligned writes. We've repeated the table from that chapter below for your convenience (see Table 23-3).

TABLE 23-3 Measured Cycle Timing of Both Nonoptimized and Optimized MMX Technology Sprite Loops

	Nonoptimized		Optimized	
Alignment	Clocks/ Sprite	Clocks/ 8 pixels	Clocks/ Sprite	Clocks/ 8 pixels
0	**110407**	**159**	**109732**	**158**
1	180585	260	179676	259
2	180425	260	179558	259
3	180546	260	179487	259
4	150358	217	149725	216
5	185099	267	184392	266
6	185399	267	184364	266
7	185398	267	184277	266

Here are the rules: processors with MMX technology achieve the best write bandwidth to video memory if they perform *aligned quad word* write. Processors without MMX technology achieve their best write bandwidth to video memory if they perform *aligned double word* writes. In either case, unaligned memory writes to video memory have a detrimental effect on the bandwidth of writes to video memory.

With the sprite example, we had a choice between making an unaligned access to read the original sprite from system memory or making an unaligned access to write the final result to video memory. Since unaligned accesses to video memory are more costly than unaligned accesses to system memory, we decided to go with the first alternative—ensure that all accesses to video memory are aligned on an 8-byte boundary. With this implementation, we achieved an average time of 160 clocks per quad word, regardless of the location or alignment of the sprite on the screen.

23.5.2 Spacing Out Writes to Video Memory with Write Buffers

The Pentium processor has two Write buffers and the Pentium processor with MMX technology has four Write buffers. Write buffers queue uncached memory writes on their way to memory and allow the processor to continue execution at the next instruction. For more details about these Write buffers, refer to section 23.2.2.

Since there is a limited number of Write buffers, you can easily fill up these buffers if you perform back-to-back writes to video memory, in a bitmap copy, for example. Once the Write buffers are full, the processor stalls on the next video memory write until one of the Write buffers is flushed out. The series of stalls will be repeated for the entire bitmap. As a result, valuable processor cycles are wasted between video memory writes.

Notice that the processor stalls only if you access uncached memory (read or write) or if you encounter an L1 cache miss (read or write). If you can guarantee that all accesses are in the L1 cache or a register, however, you can spare those dead cycles and perform some useful operations in between writes to video memory.

Consider a situation where you manipulate an image in system memory and then copy the result to video memory—for example, a color space conversion routine.[8] In this case, the back-to-back copy of the final image will

8. Color space converters are used in video decoders where they convert from the YUV color space preferred by video compression algorithms to the RGB color space.

stall the processor once the Write buffers are full. You can spare those dead cycles if you rearrange the code in such a way that you would perform color conversion in between writes to video memory. From our experience, you actually get the color conversion for free.

Upon a closer analysis of our MMX sprite sample, we found that we are getting the calculations for merging the sprite with the background for free. Moreover, we actually have a few more dead cycles in the loop that we could use to do more, so we did. We decided to add a new effect to the sprite—a bias would be added to the visible pixels of the sprite every time the sprite is updated on the screen.

Notice in the following code that since an MMX register can hold up to 8 packed pixels, we needed to duplicate the bias value in each of the 8 bytes—for example, to add 7 to each pixel, we need to use the value 0×0707070707070707. Even though it is not necessary, we decided to build this packed bias using a few shift and OR operations inside the inner loop rather than using a lookup table, for example. Once the packed bias is ready, we would add it to the sprite before we merge it with the background, as shown in the highlighted code below.

```
DoQWord:
    // build the packed bias...   Assume it is 0x07
    movq     mm5, qwBias          // 0x00000000 00000007

    movq     mm6, mm5
    Psllq    mm5, 8               // 0x00000000 00000700
    por      mm5, mm6             // 0x00000000 00000707

    Movq     mm6, mm5
    Psllq    mm5, 16              // 0x00000000 07070000
    Por      mm5, mm6             // 0x00000000 07070707

    Movq     mm6, mm5
    Psllq    mm5, 32              // 0x07070707 00000000
    Por      mm5, mm6             // 0x07070707 07070707

    movq     mm0, [esi]
    paddb    mm0, mm5             // add it to the sprite

    movq     mm2, mm3
    movq     mm1, [edi]
    pcmpeqb  mm2, mm0
    pand     mm1, mm2
    pandn    mm2, mm0
    por      mm1, mm2
    add      edi, 8
    add esi, 8
    dec ecx

    movq     [edi-8], mm1
    jnz      DoQWord
```

When we measured the performance of the code with the new calculations, we got little or no difference in the time it would take to execute this loop.

WHAT HAVE YOU LEARNED?

In this chapter, we examined the issues surrounding the system components, other than the processor, that affect the overall performance of your application. At this stage you should

- have a good understanding of the architecture of the memory subsystem on the PC;
- understand the timing and the internal structure of memory;
- have an idea of the architectural differences between the Pentium and Pentium Pro processor families;
- know how to access both cached and uncached memory types; and
- be able to figure out how to write data to video memory in the most efficient way.

ations,
loop.

ther than
tage you

the PC;

tium Pro

t way.

EPILOGUE

The Finale

We've reached the end of the book. We've covered several multimedia architectures including DirectDraw, Direct3D, DirectSound, DirectShow, RDX, RSX, and RealMedia. We've also talked about some of the most recent Intel Architecture processors for the PC. But this is far from the end. Welcome to the treadmill.

In these closing pages, we'd like to touch upon some upcoming areas of development, such as

- the spiral continues: faster processors, tighter multimedia architectures;
- multimedia amidst the Internet explosion;
- cheaper, faster, better 3D;
- multimedia in the home;
- and multimedia conferencing.

We hope you find the years ahead as exciting as we think they will be.

E.1 The Spiral Continues

E.1.1 The Hardware Spiral

Processors have gotten faster and continue to get even faster. It seems that barely a year after the introduction of a processor, it becomes the baseline processor, and a newer, faster processor is introduced. Of late we've begun

RANDY (THE KID) KWONG, A GRESHAM HIGH SCHOOL STUDENT, SURPRISED US WITH HIS SAVOIR-FAIRE.

to see multiprocessor systems become popular as server platforms. Before we know it, we may find multiprocessor systems becoming commonplace on our desktops.

Similarly, the entire PC subsystem continues to evolve. It needs to, in order to keep up with the data transfer demands forced by speedier processors and more complex peripherals. In the near future you can expect both a whole slew of new AGP-based multimedia peripherals and other advances in memory architectures.

With new processors, evolved subsystems, and possibly multiprocessor platforms, you will once again be faced with the issues you face today, namely, more power and scalability. We hope that tools like Intel's VTune and NuMega's SoftIce will continue to support optimizing for the new system architectures.

E.1.2 The Software Spiral

Just as the hardware will evolve, so too will the software architectures. Today's architectures for 2D and 3D graphics, video, audio, and spatial audio were developed as individual entities. The DirectX SDK packages these technologies together as a single offering.

Look for future generations of DirectX to improve the integration of the individual components. Also, look for continued merging of other architectures. Take, for example, the recent announcement by Microsoft of its incorporation of Real Networks' Real Media Architecture.

With luck continued advances in these multimedia architectures will support scalability across system architectures.

E.2 Remote Multimedia (a.k.a. Internet Multimedia)

The Internet is everywhere! Everyone is talking about it! Just about everyone wants to get onboard. Yet the Internet hasn't been with us for very long. There's a lot more in store for us. For those who can remember that far back, the Internet's development is probably as exciting as the birth of the PC itself.

E.2.1 Internet Languages

Internet Web pages today are based on static description languages such as HTML or VRML. These languages respond to user interactions with a simple hypertext interface. More sophisticated languages are needed to allow richer responses. Enter Java and VRML 2.0.

Created by Sun Microsystems, the Java programming language is becoming widely accepted as the de facto Internet interactive language. VRML 2.0, based on the Moving Worlds proposal from Silicon Graphics, adds audio and video sources and time and user responses to the static 3D worlds of VRML 1.0. But the cross-platform capabilities and security features of these languages may impose significant performance overhead.

If performance becomes a bottleneck, keep an eye out for alternative Internet programming languages that are tuned to the PC platform. Microsoft's Dynamic HTML, to be released as part of Internet Explorer 4.0, is one such candidate.

The standard Java programming language does not inherently contain rich multimedia constructs. Intel, Sun, and Silicon Graphics have jointly specified Java Media Framework (JMF) for multimedia extensions to Java. Intel will deliver JMF optimized for Intel Architecture platforms; Sun and Silicon Graphics will deliver JMF versions optimized for their respective platforms. In addition, the MPEG committee is working on expanding the scope of the MPEG standard in upcoming versions (MPEG4, MPEG7) to define a multimedia programming language that can be implemented on top of Java.

E.2.2 Multimedia on the Internet

Bringing multimedia to the Internet is not a trivial problem. Bandwidth constraints on today's Internet connections do not allow for rich multimedia. So companies are inventing multimedia technologies tailored for the Internet. For example, Progressive Downloads try to maintain user interest by allowing users to preview partial multimedia data while entire files are being downloaded. Similarly, Progressive 3D Meshes and Multi-Layered video codecs allow data to be authored with many levels of detail: the higher bandwidth the connection, the richer the picture.

Delivering real-time audio and video data across the Internet requires architectures to support streaming data types, to support synchronizing the streams, and to address end-to-end delays for continuous timely delivery. RealNetworks' RealMedia Architecture and Bamba from IBM AlphaWorks are two such architectures. IPIX technology from Interactive Pictures Corporation is another Internet audio/video architecture that provides surround video capabilities. Look for upcoming Internet multimedia architectures to integrate the progressive download solutions with the streaming architectures.

E.2.3 Evolving Hardware for the Internet

Just as software architectures will evolve, so too will the hardware. Hardware providers are aggressively pursuing increased bandwidth channels. Cable, satellite, and 56K modems are technologies targeted to the home and small businesses. Other technologies such as DSL and ADSL are being tested to improve bandwidth to the home over regular phone lines.

This increasing variation of bandwidth capabilities will require Internet content providers to author scalable multimedia content. Similarly, application developers will look for scalability constructs (hardware mechanisms and software APIs) to tailor applications to available bandwidth and effective throughput.

E.2.4 Multimedia Conferencing

Today we have primitive video and audio conferencing over the Internet and over POTS[1] lines. With the better bandwidth capabilities of ISDN, companies like Intel and PictureTel have developed teleconferencing products that deliver better picture quality and a reasonably acceptable user experience. As the Internet pipes to the home get bigger, we'll see similarly improved teleconferencing quality over the Internet.

Teleconferencing applications need teleconferencing APIs, and today's products are based on in-house interfaces. Microsoft has recently introduced NetShow, a conferencing API for Windows 9x, but it is still in its early stages. Look for more comprehensive APIs to support echo cancelation, initiating and responding to calls, packaging and parsing multiple data streams, data sharing, recording a conference, and sharing documents among multiple remote sites.

E.3 Better, Faster, Cheaper 3D

We've seen the first few generations of 3D on the PC, with the initial 3D games, followed by several general-purpose libraries and most recently the first revisions of Microsoft's Direct3D. 3D on the PC has been born, and now for its growth.

1. POTS: Plain Old Telephone System.

E.3.1 3D Hardware Spiral

The birth of 3D has fueled the demand for richer, faster 3D through hardware accelerators. A whole slew of 3D hardware products has been introduced recently, including, among others, products based on the Virge family of 3D chips from S3, the 3D RAGE family of 3D chips from ATI, the Vérité family from Rendition, and the Voodoo product line from 3Dfx Interactive.

Early revisions had difficulties with Direct3D support. Look for drivers to deliver improved performance and stability with the upcoming DirectX5 release from Microsoft.

In addition, some second-generation 3D hardware products have been announced. Two announcements of particular interest are the Talisman effort from Microsoft and the Intel740 effort from Intel.

The Talisman effort, spearheaded by Microsoft, is aimed at developing high-performance, high-quality 3D with approximations tailored for the PC environment. Microsoft is developing a full-featured Talisman reference card in conjunction with Philips, Cirrus, SEI, and Fujitsu. De-featured Talisman cards at lower price points will also become available.

The Intel740 effort is a codevelopment of Intel, Lockheed Martin, and Chips and Technologies. The three companies are developing a graphics chip that combines Real 3D technology from Lockheed Martin with 2D and video technology from Chips and Technologies and AGP technology from Intel. Lockheed Martin's Real 3D is also featured in Sega Enterprise's Model 2 and Model 3 arcade platforms.

E.3.2 3D Software Spiral

Once again, just as the hardware evolves, so will the software. DirectX5 offers 3D advances such as the Draw Primitives API to simplify base 3D. Similarly, in response to customer demand, expect improvements in the performance and feature set of Direct3D's Hardware Emulation Layers.

3D APIs and objects have grown based on 3D application needs. With faster computers, the demands will grow, and we will see newer 3D concepts and APIs. For example, traditional polygonal modeling is not well suited for rendering streaky objects like hair. Developers will experiment with software modeling techniques. Techniques that win favor with the development community will probably be implemented in hardware.

E.3.3 3D Scalability

Once again, the hardware and software spiral presents us, developers, with the power-spectrum/scalability problem. In the 3D area, special features are being introduced to control scalability, such as Procedural Textures, Levels of Detail, and Progressive Meshes.

Procedural Textures define textures as parameterized images. The parameters can be varied based on the capabilities of the platform. The more powerful the computer, the richer the textured image. Representations of fire, water, and clouds are examples of some parameterized textures.

Levels of Detail and Progressive Meshes allow a 3D scene to be authored with elaborate detail. On less powerful platforms, details are dropped in order to provide real-time response, although at reduced richness.

E.3.4 Emerging Application Areas

As 3D technologies have progressed, more research is being invested in the application of 3D into emerging areas. One such emerging area is information visualization, which attempts to deal with the problem of parsing through the large quantities of information unleashed upon us by the computer age.

Spotfire, a Data Mining product from the Swedish IVEE Development AB, and various Information Visualization projects in the Civiscape project at MIT's Media Lab are examples of efforts in this field.

Based on visualization research at Xerox PARC, InXight, a Xerox New Enterprise Company, was launched to convert research efforts into usable products. InXight markets an SDK with advanced UI controls such as Hyperbolic Tree, Cone Tree, Table Lens, and Perspective Wall to manipulate large quantities of data.

Look for more advances in 3D user interfaces and 3D controls. From there, it won't take long until 3D creeps into commonplace business and home applications.

E.4 Multimedia in the Home

Electronic mail and browsing for information (surfing the Web) are the primary activities on the Internet today. WebTV seems to be providing a continuation of this model by making it easier to Web-surf in comfort.

Much like the Sega and Nintendo entertainment machines, WebTV uses the TV as a display device for Web-surfing computers. As digital TVs enter the marketplace, we will see more devices using the TV for display purposes. And we will herald a new class of applications to take advantage of the computing power in the home. We will also see applications being developed for electronic commerce, as soon as adequate security mechanisms and APIs are developed.

For the PC to be used as the central compute facility in the home, it will have to be powered on for use by remote devices. Answer: Instant On, a new feature in Windows98, will allow the PC to be "awoken" by peripheral devices even though the PC may seem to be off. For example, an Internet call from the outside can awaken your PC to receive a mail message, or your PC can wake up to act as an Internet answering machine.

Instant On can offer "compute-power" to any smart device. Expect, therefore, a slew of new "peripherals" to control home devices—the VCR, air-conditioning, or the sprinkler. Imagine, calling in on your vacation to turn off that iron!

Obviously all these advances will require new APIs and new communication protocols. More excitement for us programmers.

E.5 Some Web Sites for Further Reading

This Book

http://www.awl.com/cseng/titles/0-201-30944-0

Multimedia Architectures

rdx, rsx, directx, rma, apple

Upcoming 3D Graphics Hardware

http://www.microsoft.com/hwdev/devdes/talis1.HTM
http://www.research.microsoft.com/SIGGRAPH96/Talisman
http://www.intel.com/pressroom/archive/releases/lock.htm
http://www.real3d.com

Current 3D Graphics Hardware

http://www.3dfx.com
http://www.atitech.ca
http://www.diamondmm.com
http://www.s3.com

Internet Multimedia

http://www.sdsc.edu/vrml
http://www.alphaworks.ibm.com/formula/bamba
http://www.ipix.com

3D User Interfaces

http://civiscape.media.mit.edu/civiscape
http://www.inxight.com/index.shtml
http://www.ivee.com

Index

ds, 37
27
12–
–119
-103
0–111
6

tion,

CDROM License Agreement Notice

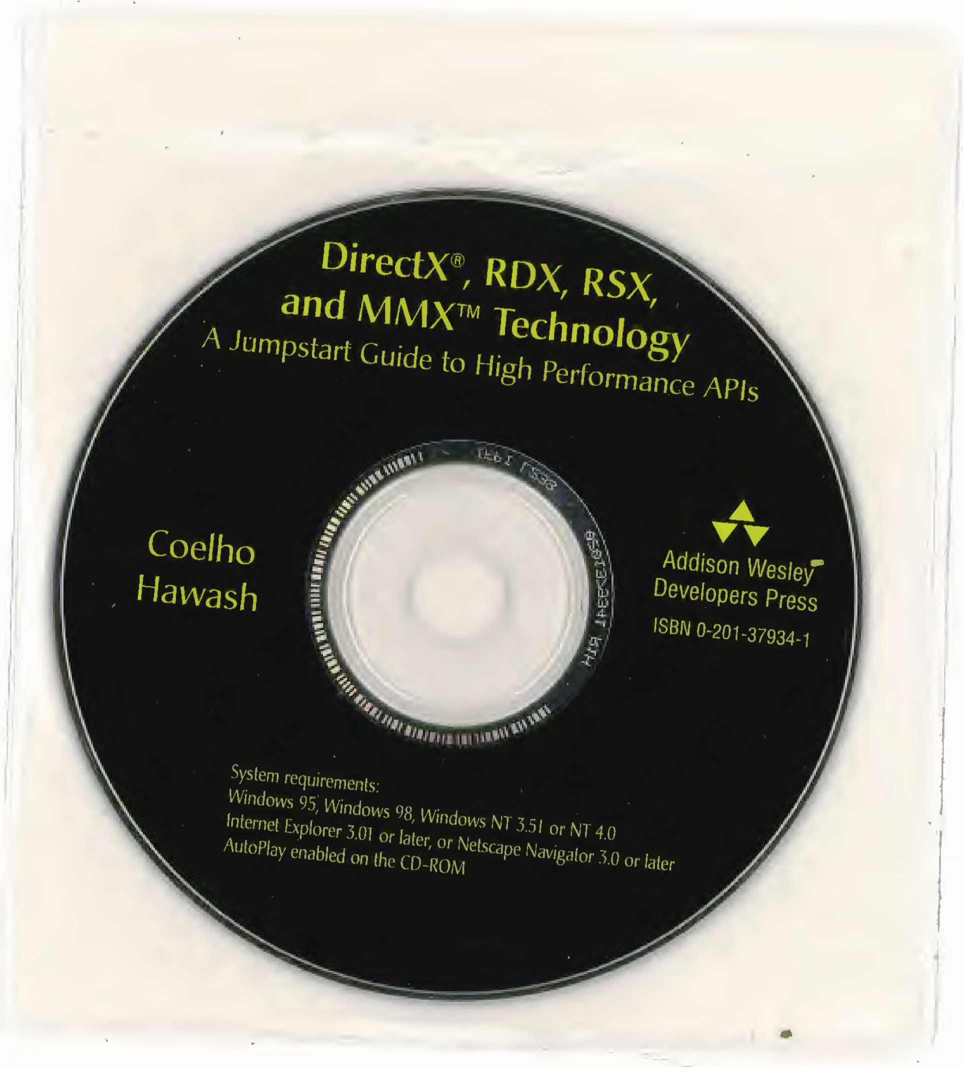
More information and updates are available at
`http://www.awl.com/cseng/titles/0-201-30944-0/`

use
first
th all

nder
riod,
pur-

order

soft-
esley
rising
tates.
may


DirectX®, RDX, RSX, and MMX™ Technology
A Jumpstart Guide to High Performance APIs
Coelho
Hawash
Addison Wesley
Developers Press
ISBN 0-201-37934-1
System requirements:
Windows 95, Windows 98, Windows NT 3.51 or NT 4.0
Internet Explorer 3.01 or later, or Netscape Navigator 3.0 or later
AutoPlay enabled on the CD-ROM

Graphics Programming/Windows Programming

DirectX®, RDX, RSX, and MMX™ Technology

Unitl now multimedia developers had to program directly to hardware in order to maximize application performance. DirectX, RDX, RSX, and MMX technology are new advancements that enable programmers to write applications that take advantage of hardware acceleration without direct hardware programming.

Written by Intel experts who are developing and applying these new technologies, *DirectX®, RDX, RSX, and MMX™ Technology: A Jumpstart Guide to High Performance APIs* takes a hands-on approach to illustrate the latest technologies from Microsoft, Intel, and RealNetworks.

This book:

- Shows programmers how to get up to speed on each API and provides key hints, tips, and advice throughout the text
- Covers DirectX (DirectDraw®, Direct3D®, DirectSound®) and DirectShow (formerly ActiveMovie) APIs from Microsoft; RDX and RSX from Intel; and RealMedia from RealNetworks
- Illustrates optimization techniques for Pentium, Pentium with MMX Technology, and Pentium II processors
- Demonstrates how to use Intel's VTune and PMonitor for processor and memory optimization

Maher Hawash has been a multimedia software developer at Intel for the past five years. He graduated with an MSEE from the University of Texas at Arlington. As a lead engineer on the MMX technology software team, he developed the MMX technology emulator and optimized MPEG decoders. As part of the Intel Architecture Labs (IAL), Maher focuses on video technologies, including VFW, ActiveMovie, Indeo, ProShare video conferencing, and the Intel Smart Video Recorder.

Rohan Coelho holds degrees in physics and computer science. For most of his eight years at Intel he has specialized in multimedia technologies at the Intel Architecture Labs (IAL). He wrote the first Indeo Video decoder, participated with Microsoft in developing VFW and ActiveMovie, co-architected DCI for Windows 3.1, worked with Sega on their SonicPC product, architected RDX, and optimized 3D rendering for MMX technology. He has published several papers and holds multiple patents.

http://www.awl.com/cseng/titles/0-201-30944-0/

Cover design by Chris Norum
♻ Text printed on recycled paper

ADDISON-WESLEY
Addison-Wesley Developers Press
is an imprint of Addison Wesley Longman, Inc.

ISBN 0-201-30944-0

$44.95 US
$62.95 CANADA