The RISC penalty

Tom Pittman

[Tom Pittman is a microprocessor consultant specializing in compilers and emulators.—Ed.]

Everybody is in denial. Even the companies most likely to benefit economically from it act as if it were not true. The emperor has no clothes, and all his courtiers still extol the beauty of the fabric. The emperor's new garment is the putative speed of reduced-instruction-set computing and computers (RISC). But it ain't so.

What you are about to read is largely anecdotal, because there is no funding for research showing RISC is not better. All the major players in the workstation and personal computer market are committed to RISC, and they don't want to know that their corporate commitment is foolish. The whole RISC story is about not looking foolish. The only reason this foolishness has gotten so far is that the real differences between RISC and complex-instruction-set computing (CISC) are relatively small in the big performance picture. Hard work and good marketing more than compensate for the differences.

What is RISC, anyway?

Literally, it is a computer architecture with fewer, simpler instructions. That would make the PDP-8 RISC, but that is not what the definition intends. At the same time the designers simplify the instruction set, they add a whole bunch of other improvements, and the aggregate is called RISC. There are as many definitions of RISC as there are proponents, but here is the usual list of characteristics that qualify a processor as RISC:

- smaller, simpler instruction set;
- fewer instruction formats and addressing modes;
- equal-length instructions;
- single-cycle instruction execution;
- many registers;
- register-based, rather than memory-based, instructions; and
- pipelined execution.

To show how meaningless this list is, let me relate it to the so-called RISC processor with which I am most familiar: the PowerPC.

The IBM Performance Optimized with Enhanced RISC (POWER, a contrived acronym if ever there was one) architecture is the basis of the PowerPC, codeveloped by the Apple-IBM-Motorola consortium (AIM). As Apple shifts its entire production to computers based on the architecture, PowerPC becomes the most widely used RISC processor on the market. It replaces the 68000 series of processors, the market Apple also dominated, at one time selling more of these processors than all other 68000-based vendors combined.

Comparing the PowerPC to the 68000 processors it replaced, we observe the following: The entry level PowerPC 601 chip has 226 different instructions. Even discounting 28 obsolete holdovers from the IBM RS6000 workstation, this is still more than twice the 81 hardware instructions in the entry level 68000. Definitely not a reduction in the instruction set, unless you are comparing it to mainframes, as IBM surely was when they named it.

The 68000 has six addressing-mode variants and three data sizes on most arithmetic and logic operations; the 601 has three data sizes for memory access (plus string operators) and four update variants on the arithmetic and logic operations. This is a slight improvement in the RISC direction, but not by much.

continued on p. 76

0272-1732/95/$04.00 © 1995 IEEE

Micro View

continued from p. 5

The 68000 has a basic 16-bit instruction word, followed by zero or more address components, up to a maximum of 10 bytes (later processors in the family allow more complexity). In the 601 all instructions are 32 bits.

Motorola put considerable engineering (and marketing) effort into making the 68040 processor RISC-like by identifying the instruction subset used most often in software, and making those instructions execute in a single cycle. The more complex addressing modes turned out to be infrequently used, as RISC architects had already noticed. Similarly, the 601 boasts single-cycle latency on all but the loads and stores, a half-dozen or so arithmetic operators (which can stall the processor for up to 36 cycles), and all the floating-point operations. To be fair, properly scheduling the floating-point operations to space dependencies adequately results in single-cycle throughput—if your program submits to such scheduling (more on that later). This is hardly an improvement, except in floating-point operations.

The 68000 has eight 32-bit data registers and eight 32-bit address registers; the 68040 added eight 80-bit floating-point registers. The 601 has thirty-two 32-bit general registers and thirty-two 64-bit floating-point registers. Again, this is a slight improvement in the RISC direction.

The 68000 allows both register-to-register and memory-to-register operations. Except for loads and stores, the PowerPC allows only register-to-register operations.

All processors designed today are pipelined; the PowerPC is no exception. Some more serious RISC architectures refuse to stall the pipeline for such things as conditional branches. However, the trend today is toward speculative execution with recovery upon misprediction, which preserves the simpler semantic model of the older architectures.

```
+0048   MOVE.L    #$00002710,D3   |   263C 0000 2710
+004E   CMP.L     (A7),D3         |   B697
+0050   BEQ       test+$0062      |   6700 0010
+0054   ADDQ.L    #$2,$0004(A7)   |   54AF 0004
+0058   MOVE.L    (A7),D2         |   2417
+005A   ADD.L     $0004(A7),D2    |   D4AF 0004
+005E   MOVE.L    D2,(A7)         |   2E82
+0060   BRA.S     test+$0048      |   60E6
```

Figure 1. Small 68000 loop example.

Computer architects intended the reduction that gives RISC its name to have two effects. First, the simpler instructions execute more easily in a single cycle, possibly with fewer gate delays (so cycles can be faster). Second, because fewer and simpler instructions take fewer gates, there is more silicon left over for other speed-enhancing technologies such as registers and an on-chip cache.

Similarly, architects intended same-length instructions to provide two benefits. The first is really the same as the result of reducing the instruction set complexity: with fewer instruction sizes and formats to decode, more transistors remain for other features. More important, though, is that superscalar architectures, which decode and begin execution of multiple unrelated instructions simultaneously, are much easier to implement with equal-length instructions.

To see the advantage of uniform instruction size, we need only compare the published transistor counts for the Pentium and PowerPC. The architectures implement comparable speed enhancement technologies across the board—one for CISC (with all its many instruction sizes) and the other for RISC (the same size instruction set, but all instructions a uniform 32 bits).

The penalty

It is that same uniform instruction size that imposes what I call the RISC penalty on performance. You see, the number of transistors does not limit performance, it is memory bandwidth that limits performance. Chip designers understand the problem, implicitly if not explicitly: new processor designs incorporate ever-bigger on-chip caches. Instruction cache, data cache—the hope is that the processor will go to main memory less often if it can stash away more in its on-chip cache. It is a forlorn hope indeed.

This is where I came in.

Apple released its new computer with a very good software 68000 emulator. By my calculations, it was only one clock cycle short of optimal, a fault one could easily correct by moving the emulator code into RAM and rescheduling the decode table. But in a processor only "three to five times" faster than the 68040, a conventional emulator running seven to 12 clock cycles per emulated instruction looked pitifully slow compared to native 68040 code execution in existing high-end Macintosh computers.

But emulators and compilers are my specialty. An important part of my master's dissertation covered binary recompilation. I could do better, and I was not alone: people at Apple and elsewhere had the same idea. However, none of our work reached the market when we expected it to. We all suffered the RISC penalty.

Figure 1 shows a tiny 68000 program loop we can use to illustrate the issues. (It happens to be the inner loop of a routine that calculates the square root of 10,000 by subtracting successive odd integers from it.)

Figure 2 shows an annotated execution trace of this loop through a typical, conventional emulator. Like Apple's software, the emulator emulates the 68000 on the PowerPC. The Apple emulator performs the loop in 68 clock cycles, as compared with the 61 shown, but saves more than a half megabyte of ROM. A small loop like this, including the emulator code and tables actually used, easily fits in the processor cache.

Note that the emulated loop in Figure 2 spends half the instructions decoding the next 68000 opcode. Half of the remaining instructions preserve the semantic model of the 68000 condition codes, interrupts, and so on. Dynamic recompilation can—indeed does—eliminate much of this excess baggage.

Figure 3 shows the same program loop execution after dynamic recompilation reduced the execution time to 18 clock cycles. There is still some support for the semantic model of the 68000 emulator, because this recompiler was designed to work transparently within the context of the existing Apple emulator. (A static recompiler with better optimization cuts the loop transit time to five clock cycles. However, it doesn't preserve the exact semantic model of the 68000 emulator across emulated interrupts.)

The point is that we have working recompiler technology producing code that is the promised three to five times faster than that of the conventional emulator—but only for tiny toy programs like the one in Figure 1. When I applied the same recompiler to commercial software, the performance went through the floor. The software actually ran slower than with the conventional emulator! Why? It's the RISC penalty.

It took almost a month of performance analysis on a logic analyzer to figure out the problem. At first I assumed the time was being lost in recompiler thrashing from inadequate code cache management. The logic analyzer showed a loop that ran over one second (roughly the same time it

Address	Cyc	Operation		Result	Memory address/comments
682263C0	1	lhau	r27,4(24)	=FFFFB697	@015D718C
682263C4	1	lwz	r4,-4(24)	=00002710	@015D7188
682263C8	0	b	68020DA8	=68020DA8	
68020DA8	1	rlwimi	r29,r27,#4,12,27	=682B6970	continuation code
68020DAC	1	mtspr	r29,lr	=682B6970	
68020DB0	1	lhau	r27,2(24)	=00006700	@015D718E
68020DB4	1	addco.	R11,r4,r0	=00002710	copy to D3 and set cc
68020DB8	1	bflr+	b8	=682B6970	decode table jump
682B6970	1	lwz	r4,0(1)	=00000004	@015DE52A
682B6974	1	rlwimi	r29,r27,#4,12,27	=68267000	
682B6978	0	b	68023EE8	=68023EE8	
68023EE8	1	mtspr	r29,lr	=68267000	
68023EEC	1	subfco.	R6,r4,r11	=0000270C	do the compare
68023EF0	1	lhau	r27,2(24)	=00000010	@015D7190
68023EF4	1	addc	r4,r4,r6	=00002710	invert the carry flag
68023EF8	0	bflr+	b8	=68267000	also test for interrupt
68267000	1	lha	r6,2(24)	=000054AF	@015D7192
68267004	1	b	68028024	=68028024	
68028024	2	beq	b2, 68027C60	=4010FFFF	test for equal
68028028	1	rlwimi	r29,r6,#4,12,27	=68254AF0	
6802802C	1	mtspr	r29,lr	=68254AF0	
68028030	1	lhau	r27,4(24)	=00000004	@015D7194
68028034	2	bflr+	b8	=68254AF0	
68254AF0	1	add	r3,r1,r27	=015DE52E	calc operand address
68254AF4	1	lhau	r27,2(24)	=00002417	@015D7196
68254AF8	0	b	68026A24	=68026A24	
68026A24	1	lwz	r4,0(3)	=00000003	@015DE52E
68026A28	1	rlwinm	r6,r29,#19,29,31	=00000002	extract the constant
68026A2C	1	rlwimi	r29,r27,#4,12,27	=68224170	
68026A30	1	addco.	R4,r6,r4	=00000005	do the add
68026A34	0	b	680269F4	=680269F4	
680269F4	1	mtspr	r29,lr	=68224170	
680269F8	1	lhau	r27,2(24)	=FFFFD4AF	@015D7198
680269FC	1	mfspr	r26,XER	=00000000	update X bit of cc
68026A00	1	stw	r4,0(3)	=00000005	@015DE52E
68026A04	0	bflr-	b8	=68224170	
68224170	1	lwz	r4,0(1)	=00000004	@015DE52A
68224174	1	rlwimi	r29,r27,#4,12,27	=682D4AF0	
68224178	0	b	68020D2C	=68020D2C	
68020D2C	1	mtspr	r29,lr	=682D4AF0	
68020D30	1	lhau	r27,2(24)	=00000004	@015D719A
68020D34	1	addco.	R10,r4,r0	=00000004	add to D2, update cc
68020D38	1	bflr+	b8	=682D4AF0	
682D4AF0	1	lwzx	r4,(1,27)	=00000005	@015DE52E
682D4AF4	1	lhau	r27,2(24)	=00002E82	@015D719C
682D4AF8	0	b	68023E84	=68023E84	
68023E84	1	addco.	R10,r4,r10	=00000009	
68023E88	1	rlwimi	r29,r27,#4,12,27	=6822E820	
68023E8C	1	mtspr	r29,lr	=6822E820	
68023E90	1	lhau	r27,2(24)	=000060E6	@015D719E
68023E94	1	mfspr	r26,XER	=00000000	update X bit of cc
68023E98	1	bflr+	b8	=6822E820	
6822E820	1	addco.	R4,r10,r0	=00000009	copy D2 to temp, set cc
6822E824	1	rlwimi	r29,r27,#4,12,27	=68260E60	
6822E828	0	b	680211EC	=680211EC	
680211EC	1	mtspr	r29,lr	=68260E60	
680211F0	1	lhau	r27,2(24)	=000060C8	@015D71A0
680211F4	1	stw	r4,0(1)	=00000009	@015DE52A
680211F8	1	bflr+	b8	=68260E60	
68260E60	1	lhau	r27,-26(24)	=0000263C	@015D7186
68260E64	2	rlwimi	r29,r27,#4,12,27	=682263C0	
68260E68	0	b	68026088	=68026088	
68026088	1	mtspr	r29,lr	=682263C0	
6802608C	1	lhau	r27,2(24)	=00000000	@015D7188
68026090	2	bflr+	b8	=682263C0	return to front of loop
	61				total loop transit time

Figure 2. Small 68000 loop emulated.

Address	Cyc	Operation		Result	Memory address/comments
68431800	1	ori	r11,r0,#10000	=00002710	
68431804	1	lwz	r4,0(1)	=00000009	@015DE52A
68431808	2	subfco.	R4,r4,r11	=00002707	do the compare
6843180C	0	b	68448480	=68448480	
68448480	1	subfe	r6,r0,r0	=00000000	invert the carry bit
68448484	1	addc	r6,r6,r6	=00000000	
68448488	1	bne	b2, 68448498	=40101842	test for (un)equal
68448498	1	lwz	r7,4(1)	=00000005	@015DE52E
6844849C	1	addi	r4,0,#2	=00000002	get the constant
684484A0	1	addco.	R7,r4,r7	=00000007	do the add
684484A4	1	stw	r7,4(1)	=00000007	@015DE52E
684484A8	1	lwz	r10,0(1)	=00000009	@015DE52A
684484AC	1	lwz	r4,4(1)	=00000007	@015DE52E
684484B0	2	addco.	R10,r4,r10	=00000010	add to D2, update cc
684484B4	1	stw	r10,0(1)	=00000010	@015DE52A
684484B8	1	mfspr	r26,XER	=00000000	update X bit of cc
684484BC	1	addco.	R7,r10,r0	=00000010	update other cc bits
684484C0	0	bt	b8, 68450000	=40101842	test for interrupt
684484C4	0	b	68431800	=68431800	
	18				total loop transit time

Figure 3. Small 68000 loop after dynamic recompilation.

took on the Apple emulator) spent less than 15 milliseconds in recompilation. It spent the other 98 percent of the time executing already recompiled and cached native code, and in code cache management itself. I radically reworked the code cache and cut its contribution to the execution time in half, for a savings of about 10 percent. Performance at this point was less than 10 percent better than with the Apple emulator—nowhere near the expected three to five times faster.

The logic analyzer trace showed a great deal of bus activity, so I directed my performance measurement toward that aspect of execution time. Bingo! A program loop whose code and data lie entirely within the processor cache runs many times faster than one that misses the cache at every opportunity.

The Power Macintosh I used for testing runs its memory bus at half the speed of the internal processor clock. If each time through the loop accesses a new memory location known not to be cached, it adds 28 clock cycles to the loop transit time. If the memory accessed fits within the secondary (L2) cache, the penalty goes down to 13 clock cycles. (These penalties are in comparison to execution of the same loop adjusted to access data constrained to a fully cached 4-Kbyte region, which incurs no penalty.)

Memory access bursts 32 bytes (eight instructions) into the processor cache at once, and instruction prefetch begins when the instruction queue is half empty. Thus, the calculated loss is (13 clock cycles)/(4 instructions), a penalty of about three per instruction, which just about cancels the three-times speed improvement the recompiler should grant. This three-times penalty exactly matches the observed bus performance on the logic analyzer. (Curiously, the higher cost of accesses to main memory as compared to the L2 cache did not significantly contribute to a larger execution time for instructions. I do not have an explanation for that. Data access from main memory that missed the L2 cache did slow the execution further.)

One dynamic recompiler implementor told me that he had to get a raw performance improvement of five times to show any improvement to the user. Although he was at that time still convinced that his losses were due to recompiler thrashing, it is clear from subsequently published figures that he was experiencing exactly the same effect I did: 5 times/3 times = 1.7 times (compare 1.5 to 2 times in the press).

Another implementor made the same foolish mistake I did. After getting a three-times improvement in small programs, he announced a ship date for a month later. Two months after missing that date, one of the online services quoted him as saying, "We're working on it." Still another recompiler implementation has apparently been delayed until Apple ships faster hardware.

Microsoft was soundly reviled in the press for allegedly scheming to sabotage the PowerPC with its latest generation of Word and Excel. These programs ran faster on Pentium than on the PowerPC—when everything else had done better on the PowerPC. Microsoft executives publicly (and I now believe, sincerely) denied any sabotage. The explanation is simple: Word and Excel are large programs that do not fit in the processor cache. Pentium is a CISC, and CISC instructions can squeeze through the memory bottleneck faster than RISC can. That's the RISC penalty.

The problem is that we have been assuming that all programs spend most of their time in small loops like that in Figure 1. Some programs do spend the bulk of their time in small loops—the filter functions in image-processing programs such as Photoshop, for example. Other programs, like Word and Excel, dance around a very large code loop most of the time. Photoshop does well on RISC machines; Word and Excel do poorly.

Programmers are moving away from tightly coded assembly and Pascal and toward less-easily optimized C and C++. With the tremendous code burden brought on by prodigal method overriding, more and more programs will become like Word and Excel, spread out over a large expanse of code space. They will pay the RISC penalty.

The emperor's new clothes

Then there is the emperor problem. RISC is politically correct in the computing world; CISC is not. RISC is

where the technology is going; CISC is what it is leaving behind. Even Intel, reaping huge profits on the very CISC-like Pentium, is reportedly busy behind the scenes working with one of the workstation vendors on a RISC chip to carry the ball when (as we all know and believe) their CISC flagship runs out of gas.

A couple of years ago, shortly after the PowerPC was announced, Motorola also announced its next-generation 68000 processor, the 68060, which reportedly would be just as fast as the PowerPC. I suspect Motorola was right. We don't hear anything about the 68060 anymore.

When Apple first went public with their intention to migrate Macintosh to RISC, it made an all-out effort to get developers on board with native RISC code. Though prerelease computers were not widely available to developers, there was plenty of instruction and advice on how to convert. But when the Power Macintosh rolled out, there were only 30 native-code applications available. A year later the number had crept slowly up to 500. Why is that?

I doubt you will get this from any programmer, but I think it's the RISC penalty. Apple tells developers that if they have been following guidelines or writing in ANSI C, it's just a matter of a few text substitutions and recompiling the code in the new compiler, and—*presto!*—three to five times speedup. So the programmer tries it, and—*presto!*—1.3 or 1.5 times speedup. He sees that and exclaims, "I'm an idiot! Everybody else is getting three to five times!" or words to that effect. Of course he's not an idiot. Programmers are very clever and inventive, so they tune and tweak and recode and optimize until finally the code is actually three to five times faster. This takes several months, but when they finally get the desired (and promised) performance, they ship quick and never breathe a word about how hard it was. To confess any difficulty would be to admit to faulty intelligence.

A leading software developer journal for Macintosh computers has been beating the drum for RISC. Every month or two it runs another article on how to hand-optimize your code so you can get great performance improvements. It never mentions that the same tricks applied to 68000 (or any other processor) code would give the same impressive improvements on that platform.

What kind of tricks are being recommended to show off the performance advantages of RISC? Here are some recent suggestions actually offered to Power Macintosh programmers.

Performance analysis. This will better focus optimization attention on program hot spots. Performance analysis was a fine idea when Donald Knuth first proposed it 20 years ago, long before there was such a thing as RISC. We all rise before its hoary head. Of course, any two different architectures, regardless of whether they are RISC or CISC, will show different hot-spot signatures.

Improved algorithms. This hoary nostrum was commonly invoked whenever C performance was compared to equivalent Pascal programs and came off rather the worse. If Pascal had received half the developmental support that C continues to get, the battering would have continued. But, as in the RISC/CISC debate, market survival has little to do with performance.

In those days, as now, many argued that improving algorithms could reap performance improvements much larger than those attributable to such minor differences as language (or processor architecture). The argument is valid, but contributes nothing to any alleged superiority of RISC, which fares no better with improved algorithms than does CISC.

Cache locality. This is the idea that frequently used data is likely to occur in clusters. Hardware designers have so thoroughly embraced this concept that now it can be turned on its head. Programmers should cluster data (and code, but that is much harder in modern languages like C++) to make it more likely to fit into the cache. This was less significant in older CISC processors, but not because they were CISCs. As newer hardware designs increase processor clock rates and cache size, the effect is more noticeable in both RISC and CISC equally.

Data alignment. Lack of alignment has always cost a cycle or two ever since it stopped crashing the machine. Today's RISC is no exception. On some RISC hardware (notably the PowerPC 601) it costs at most one clock cycle per access, except in the rare case of an access split over a virtual-memory page boundary. In that case, the hardware crashes as in the olden days, but the operating system is more polite in recovering execution through software emulation.

Newer hardware may require slow emulated recovery more often, but none of this has appeared on the market yet. Nevertheless, data alignment is often mentioned as a significant way to gain the performance advantage inherent in RISC, and by implication, even to effect improvement over CISC emulation on the same hardware! What fine, gauzy clothes the emperor is wearing today!

Strength reduction. By this process, we replace long, tedious machine instructions with semantically equivalent, but simpler, operations—for example, replacing multiplies and divides by shifts. This is as close as I've heard to a mea culpa—an admission that two years of claiming compiled C code performs better on RISC than hand-coded assembly language is simply unfounded. Now programmers are implicitly urged to program around the inefficient code produced by RISC C compilers. Of course, it has nothing at all to do with RISC: I have never seen compiled C code on any platform that relatively straightforward, handwrit-

ten assembly code could not best. Perhaps I should call it the C penalty. But this is about RISC. C is just as bad on RISC as any other platform.

Register variables. These were originally added to C when it was nothing but a high-level assembler for the PDP-11 (which had only eight user-programmable registers). Now they really shine on RISC, where 32 registers is a minimum entry ticket. Or do they? Programmers are told to program them carefully, because the compiler has problems determining if a register variable must be flushed to memory. Strongly typed languages like Pascal do not suffer these problems.

In fact, a few years ago I was amused to hear of a compiler for an Australian implementation of Modula-2 (a language similar to Pascal) that produced code outperforming C on the same system. This was despite the fact that the Modula-2 was compiled to C and used the same C compiler for code generation! The reason given was that the Modula-2 compiler refrained from all the sleazy, low-level programming tricks that C encourages programmers to use in the name of efficiency. Indeed, most modern C compilers completely ignore programmer designations of register variables, preferring to do their own analysis. On RISC it matters more than on CISC, but it seems that compiler technology did not rise to the need.

Floating point. On the PowerPC, floating-point operation is screaming fast. This has nothing to do with RISC and everything to do with a screaming-fast floating-point engine with every bit the power, performance, and speed of the original Cray supercomputers.

Indeed, it is in some ways clock-for-clock faster, because of the combined multiply-add instruction, but, I repeat, this is hardly a RISC-like feature. Multiply-add was first proposed in 1978 by the IEEE Std 754 floating-point standard architect, William Kahan, long before RISC existed. (Kahan's proposal is actually more efficient than the PowerPC implementation, though it hardly matters.)

The point is that PowerPC programs often gain a substantial boost in performance when integer-coded data is partially or fully recast as floating-point data. Simultaneous dispatch of integer and floating-point instructions is easier in RISC than CISC, although that disadvantage seems hardly to impede the Pentium at all.

Scheduling. By this process, a compiler arranges code to improve the chance of multiple dispatch. Scheduling enables superscalar RISC (and CISC) computers to improve performance through simultaneous parallel execution. Although scheduling is not that difficult (once you get the hang of it), scheduling algorithms make or break RISC compilers. Again, recommendations to programmers often focus on working around inefficiencies the compilers introduce. Of course, it is also here that the RISC penalty becomes most noticeable: What good is superscalar dispatch if the whole instruction queue has stalled waiting for code from main memory?

We are beginning to see glimmerings of recognition of these problems here and there in the industry. The IEEE Computer Society's Microprocessor Standards Committee recently started a new study group to look into standards for a faster processor-memory bus. They still don't know how to make the memory faster, but if and when somebody solves that problem, they will have a bus to support it.

A RISC chip company is rumored to be working on a packing scheme to reduce its 32-bit instructions to complex 16-bit code decoded inside the processor using a lookup table. I think this is moving in the right direction, but it doesn't go far enough.

Fewer than a couple dozen instructions perform over 90 percent of all computing. Apart from address constants, 5-bit code would be perfectly adequate, giving a zero-address architecture up to a six-times performance advantage over conventional 32-bit RISC. It's too late for AIM to follow this trail; it may already be too late for Intel. Perhaps a hungry young start-up with Pacific Rim funding can leapfrog into the next century—and leave all competitors in the dust, suffering the RISC penalty.

Suggested readings

R. Hess, "New 68K Emulator Gearing Up for PCI," *MacWeek,* Dec. 5, 1994, p. 1.

R. Hess, "Connectix Preps Fast Emulator," *MacWeek,* Jan. 23, 1995, p. 1.

T. Pittman and J. Peters, *The Art of Compiler Design: Theory and Practice,* Prentice Hall, Englewood Cliffs, N.J., 1992, pp. 247-267, 287-334.

T. Pittman, "The Technology of Emulation: 68K on PowerPC," *MacTech Magazine,* Sept. 1994, p. 56.

C. Seiter, "Secrets of Software Speeds," *MacWorld,* Jun. 1995, p. 41.

E. Traut, "Taking Extreme Advantage of PowerPC," *MacTech Magazine,* Apr. 1995, p. 67.

PowerPC 601 RISC Microprocessor User's Manual, MPC601UM/AD, Motorola, Houston, Tex., 1993.

Reader Interest Survey

Indicate your interest in this column by circling the appropriate number on the Reader Service Card.

Low 174 Medium 175 High 176