Performance from Architecture: Comparing a RISC and a CISC with Similar Hardware Organization

Dileep Bhandarkar
Digital Equipment Corp.
146 Main Street (MLO5-2/G1)
Maynard, MA 01754

Douglas W. Clark*
Aiken Computation Lab
Harvard University
Cambridge, MA 02138

Abstract

Performance comparisons across different computer architectures cannot usually separate the architectural contribution from various implementation and technology contributions to performance. This paper compares an example implementation from the RISC and CISC architectural schools (a MIPS M/2000 and a Digital VAX 8700) on nine of the ten SPEC benchmarks. The organizational similarity of these machines provides an opportunity to examine the purely architectural advantages of RISC. The RISC approach offers, compared with VAX, many fewer cycles per instruction but somewhat more instructions per program. Using results from a software monitor on the MIPS machine and a hardware monitor on the VAX, this paper shows that the resulting advantage in *cycles per program* ranges from slightly under a factor of 2 to almost a factor of 4, with a geometric mean of 2.7. It also demonstrates the correlation between cycles per instruction and relative instruction count. Various reasons for this correlation, and for the consistent net advantage of RISC, are discussed.

1 Introduction

The last decade has seen the emergence and rapid success of Reduced Instruction Set Computer, or RISC, architectures. Following early work by Cray [32, 27] and Cocke [6, 7] and an implementation at IBM [25], university researchers, especially at Berkeley [23] and Stanford [16] developed design principles, built processors, and founded companies. Today the success of RISC architectures from SUN (the Berkeley-inspired SPARC design), MIPS (the Stanford-inspired MIPS design), and traditional semiconductor companies (Motorola, Intel) is evident; big computer companies like IBM, Hewlett Packard, and Digital have also embraced the concept.

The RISC approach promises many advantages over Complex Instruction Set Computer, or CISC, architectures, including superior performance, design simplicity, rapid development time, and others [19, 22]. Studying all of these factors at once is beyond the scope of this paper, which will look only at performance, and in fact only at performance from the *architectural* perspective. That is, we will try to control for all influences on performance other than architecture. We will do this by studying two machines, one from each architectural school, that are strikingly similar in hardware organization, albeit quite different in technology and cost. We will show that these differences are not due to architecture.

Our fundamental frame of reference will be the now-familiar expression of performance as a product of the number of instructions executed, the average number of machine cycles needed to execute one instruction, and the cycle time:

$$\frac{time}{program} = \frac{instructions}{program} \times \frac{cycles}{instruction} \times \frac{time}{cycle}$$

We (along with many others) have found this formulation to be a powerful tool for analyzing, comparing, and projecting processor performance.

The three terms are functions of various aspects of a system design. The number of instructions executed is a function (for a fixed algorithm and source program) of the compiler and the target architecture, and is usually independent of the detailed hardware implementation and the technology. The machine's basic cycle time, however, is a function most strongly of the underlying technology (gate speed, RAM speed, and so on), and also of the hardware structure or microarchitecture of the machine, particularly the degree of pipelining. The cycle time may also be affected by the instruction-set architecture.

The middle term—average number of cycles per executed instruction, or CPI—has the most complex determinants. The instruction-set architecture is a primary one: in a complex architecture like the VAX, there are individual instructions (such as character-string-moves) whose execution requires hundreds of cycles; a RISC would accomplish the same function with (say) hundreds of instructions each taking only one or two cycles. Another important determinant is the hardware organization, especially the degree of pipelining and the structure of the cache-memory subsystem. Finally, the compiler can affect this factor too, through its choice of certain instruction sequences over others, through the general quality of its code optimization, and (for some architectures) through its ability to schedule instructions to avoid stalls.

The essence of the RISC performance objective is this: compared with the CISC approach exemplified by VAX, instruction-set architectures should facilitate implementations that achieve a gross reduction in cycles per instruction

*on leave from Digital Equipment Corp., 1990-91.

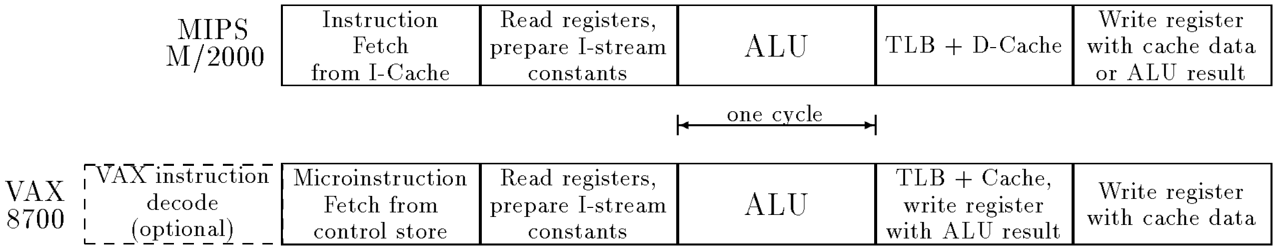

Figure 1: Simplified illustration of the two instruction pipelines. A new instruction (microinstruction on the VAX) can start every cycle. The VAX decode cycle is omitted when a microroutine is more than one microinstruction long [4].

and possibly some improvement in cycle time while allowing an increase in the number of instructions executed. The goal is a substantial net improvement in execution time.

The qualitative evidence that this goal has been achieved is by now nearly overwhelming. What is lacking, though, is a careful architectural analysis, and that is what we intend to provide in this paper. We will need to make two assumptions for our study: first, that the compilers are of equivalent quality; and second, that cycle time is not a function of architecture. We are not entirely happy with the compiler assumption, particularly since it is quite imprecise and difficult to measure, but we did use the best available compilers for each machine.

Our cycle time assumption is valid for technologies and design approaches in which the cycle time is determined by such architecture-neutral things as the time to get through an integer ALU and the time to read the first-level cache. If, on the other hand, some VAX-specific function, such as instruction decoding or control-store sequencing, limits cycle time, then the necessary adjustment to our results is a simple multiplication. In any event, we are not addressing the cycle time question here. In essence we are looking at this architectural question: what performance advantage does a RISC have over a VAX with the same cycle time and similar hardware organization, given good compilers for each machine?

The next section of this paper discusses in more detail our two machines, the measurement methods used for each, and the benchmarks that were run. Section 3 presents the basic results from our measurements, including instruction counts and average cycles per instruction for each machine. Section 4 is a discussion of these results and of several types of explanatory factors. Section 5 then briefly considers variations on implementation styles for both architectures and summarizes our basic results, concluding the paper.

2 Apparatus and Methods

2.1 The Machines

We measured Digital's VAX 8700 (a single processor version of the 8800) [4, 11, 30] against MIPS Computer Systems' MIPS M/2000 [19, 26]. We concede at the outset that these two machines are very different in technology, size, and cost: the VAX processor is nine boards full of ECL gate arrays; the MIPS processor is one board with two custom CMOS chips. *However, there is another VAX, the model 4000/300, whose processor is organizationally similar to the 8700's and technologically similar to the MIPS M/2000's.* The existence of this VAX demonstrates that the technology difference between our two measured machines is not a consequence of architecture.

But why not compare the two CMOS machines directly? The main reason is that only the 8700 had the hardware instrumentation demanded by our measurements [5]. And in fact, as we will see, the CMOS VAX's resemblance to the MIPS engine is somewhat less than the 8700's.

There are strong organizational similarities between the VAX 8700 and the MIPS M/2000. Figure 1 is a simplified representation of the main CPU pipelines in the two machines. Both illustrations have abstracted away some half-cycle boundaries that appear in the actual hardware, but neither misrepresents the fundamental operation of the pipes. Both machines can issue a new instruction (microinstruction on the VAX) every cycle.

Figure 1 shows that the pipelines match up quite closely, with the obvious exception of the VAX instruction decode stage. But note that we are matching the MIPS *instruction fetch* stage with the VAX *microinstruction fetch* stage. Indeed, the 8700 micro-engine shares with the MIPS implementation the following features:

- a large set of general purpose registers;
- single-cycle three-register instructions;
- bypassing of ALU results around the register file and to the ALU inputs so that a register can be read in the instruction immediately after the one in which it is written;
- single-cycle load and store instructions that make an address by adding a displacement to a register;
- bypassing of cache read data around the register file and to the ALU input;
- a one-cycle delay slot following a load that can be filled by any instruction not using the loaded register; and
- delayed branches (but the VAX delay is longer—see Section 4.3 below).

This strong similarity between MIPS instructions and VAX 8700 microinstructions means that the comparative performance challenge for this VAX might be viewed as the problem of mapping VAX instructions into microinstructions efficiently. As we will discuss in Section 4, efficient mapping is sometimes easy but more often quite difficult.

Both implementations represent reasonable state-of-the-art "mid-range" technology. Even though the VAX 8700

Table 1: Machine Implementation Parameters

	VAX 4000/300	MIPS M/2000	VAX 8700
Chip First Silicon	1989	1988	n/a
System Ship	1990	1989	1986
CPU	REX520	R3000	n/a
Technology	Custom CMOS	Custom CMOS	ECL gate array
Component counts CPU FPU	 140K transistors, 180Kbits mem 134K transistors	 115K transistors 105K transistors	 approx. 100 gate arrays, 1200 gates each (included above)
Feature size	1.5 micron	1.2 micron	
Die size CPU FPU	 12x12 mm^2 12.7x11 mm^2	 7.6x8.7 mm^2 12.6x12.6 mm^2	n/a
Cycle time	28 ns.	40 ns.	45 ns.
On-chip cache	2 KB	none	n/a
Board cache	128 KB I+D	64 KB I, 64 KB D	64 KB I+D
TLB	64 entries	64 entries	1024 entries
Page size	512 bytes	4 Kbytes	512 bytes
Memory access time	13 cycles	12 cycles	16 cycles
FP multiply	15 cycles	5 cycles	15 cycles
FP Add	14 cycles	2 cycles	11 cycles
List price	$100K	$80K	$492K
Performance Overall SPECmark Integer SPECmark FP SPECmark	 7.9 7.7 8.1	 17.6 19.7 16.3	 5.6 5.0 6.0

uses ECL gate-array technology, an adaptation of its microarchitecture has been implemented in a VLSI CMOS chip [2, 13] that appears in VAX 6000 Model 400 and VAX 4000 Model 300 systems.

Table 1 summarizes the salient implementation characteristics of our two machines together with the VAX 4000/300. The MIPS M/2000 and the VAX 4000/300 are both implemented in custom CMOS technology, both having a one-chip CPU connected to a one-chip FPU. A direct comparison of these machines would be complicated by the fact that one has an on-chip cache and the other does not. The VAX chips use somewhat more transistors, and the CPU uses additional bits of memory for its on-chip cache and microcode. Hence the cost of the VAX chips would be greater than the cost of the MIPS chips, if they used the same fabrication process. We believe that the lower chip costs would be a small part of the overall cost of a system; system *prices*, of course, are determined by market factors and business considerations. Digital's prices for its 1990 workstations employing the VAX and MIPS chips are close: $12K for the VAXstation 3100/76 (6.6 SPECmarks) and $15K for the DECstation 5000/200 (18.5 SPECmarks).

The VAX 8700 and the MIPS M/2000 have distractingly similar cycle times. This similarity we regard merely as a coincidence; it is the machines' organizational similarity that we rely on to justify our side-by-side comparison, not their cycle times.

The MIPS machine has a few advantages: it has a separate instruction cache, slightly faster main memory, and considerably faster floating point. We will argue in Section 4 that the difference in floating-point performance has an architectural basis. The M/2000 allows some overlap of floating-point instructions [19], whereas the VAXes have very minimal overlap. These factors should all contribute to a slightly wider difference in cycles per instruction between the MIPS system and the VAX system.

2.2 The Benchmarks

We use the SPEC Release 1 benchmarks for our analysis [31]. SPEC is a non-profit corporation whose members include major workstation and computer companies such as Digital, HP, IBM, MIPS, Silicon Graphics, Sun, and others. SPEC was founded to develop a standard set of benchmarks that are application based. The first release has been available since October 1989. Ten benchmarks were selected from a large number of prospective candidates. Each represents a real application or a significant kernel extracted from an application, runs for an extended length of time, and puts a reasonable load on most modern systems. These benchmarks are much more meaningful measures of CPU performance than "toy" benchmarks (Towers of Hanoi, Puzzle, Dhrystone, Whetstone, etc.) that have sometimes been used. All SPEC benchmarks are portable, and the only program changes allowed are SPEC-approved changes for portability. They all produce substantially the same answers on all systems tested. Results are expressed in terms of the SPECratio or performance relative to the VAX-11/780 for each benchmark. The geometric mean of all ten ratios is called SPECmark.

The SPEC Release 1 suite consists of four integer benchmarks (gcc, espresso, eqntott, and li) written in C, and six floating-point benchmarks (spice, doduc, nasa7, matrix300, fpppp, and tomcatv) written in Fortran. For details on these programs, see [31]. Even though spice was meant to be a floating-point benchmark, the circuit being simulated results in a fairly low use of floating-point operations, and should therefore be viewed as a mixed integer and floating-point

Table 2: RISC factors

benchmark	instruc. ratio	CPI MIPS	CPI VAX	CPI ratio	RISC factor
spice2g6	2.48	1.80	8.02	4.44	1.79
matrix300	2.37	3.06	13.81	4.51	1.90
nasa7	2.10	3.01	14.95	4.97	2.37
fpppp	3.88	1.45	15.16	10.45	2.70
tomcatv	2.86	2.13	17.45	8.18	2.86
doduc	2.65	1.67	13.16	7.85	2.96
espresso	1.70	1.06	5.40	5.09	2.99
eqntott	1.08	1.25	4.38	3.51	3.25
li	1.62	1.10	6.53	5.97	3.69
geo. mean	2.17	1.71	9.87	5.77	2.66

benchmark [28].

We were not able to measure gcc on our instrumented VAX 8700, so all of our results are for the nine other benchmarks only. Also, our run of espresso used just one of the four input circuits (bca). We used the most up-to-date versions of compilers that were available to us in mid-1990 on both architectures: VAX Fortran V5.0-1 and VAX C V3.1; MIPS F77 v2.0 (v2.10 for matrix300) and CC v2.0. While we have seen some small differences in later versions of the compilers, only in the case of matrix300 on MIPS did the difference warrant repeating our measurement.

2.3 The Monitors

A hardware monitor designed specially for the VAX 8700 was used to measure the SPEC benchmarks in detail. This monitor, described in [5], uses the micro-PC histogram technique introduced in [14]: a real-time count is kept for each microinstruction, and in every cycle the microinstruction then in execution in the ALU has its count incremented. A microcoded machine such as the VAX 8700 can reveal a great deal of its detailed behavior in this way; classification of the microaddresses into appropriate groups allows many things to be measured. Since the monitor provides counts of all cycles and of all instructions, CPI can be calculated directly.

Two tools were used on MIPS M/2000 system to produce execution profiles of the SPEC benchmarks: Pixie and Pixstats [21]. Pixie reads an executable program, partitions it into basic blocks, and writes an equivalent program containing additional code that counts the execution of each basic block. When this Pixie-generated program is run, it generates a file containing the basic block counts. Then Pixstats analyzes the program execution and produces a report on opcode frequencies and various other things. CPI is calculated by dividing the CPU time in cycles from an *un*instrumented run by Pixstats' report of the instruction count (which excludes NOPs).

3 Results

3.1 Instructions and CPI

Table 2 shows that the chief architecturally-directed performance goal of the RISC approach has been achieved for the MIPS design (given the compilers). It shows that for all of the SPEC benchmarks, average CPI on the MIPS M/2000 is much less than on the VAX 8700. The number of instructions, on the other hand, has increased, but not nearly as much. The *instruction ratio* in the table is just the ratio of MIPS instruction executions to VAX instruction executions, and is always greater than 1, ranging from a little over 1 to nearly 4, with a geometric mean of 2.17. The *CPI ratio* is average VAX CPI divided by average MIPS CPI (we define it this way to make both ratios be greater than 1). It is never lower than 3, goes as high as 10.45, and has a geometric mean over the nine programs of 5.77. The combined effect of the two ratios—the net effect on performance—is what we call the *RISC factor*: it is the ratio of the number of *cycles per program* on the VAX to the corresponding number on the MIPS. It is also obviously just the CPI ratio divided by the instruction ratio. This factor ranges from just under 2 to just under 4, with a geometric mean of 2.66. In Table 2 and subsequent tables we rank the benchmarks in order of increasing RISC factor.

Let us look first at CPI. Both architectures display a wide spread of values, spanning a range of about 3:1 for MIPS and 4:1 for VAX. The heavy floating-point benchmarks have quite large CPI on the VAX, due to the floating-point hardware (Table 1); spice stands out because it actually does very little floating point (Table 3, below). It would be quite misleading to use the geometric means of CPI as "typical" figures without reference to the specific nine programs they represent.

Despite the wide variance of instruction and CPI ratios, the RISC factor spans a range of just under 2:1. The three highest RISC factors are attached to the three integer benchmarks, which have the three lowest instruction ratios but only mean-valued CPI ratios. The three lowest RISC factors, on the other hand, come from benchmarks that have three of the four lowest CPI ratios but mean-valued instruction ratios. In the middle of Table 2 lie the three benchmarks with the highest values of both ratios. The correlation between instruction and CPI ratios is a central result of this paper, and will be discussed further below.

3.2 Operation counts

Table 3 shows the execution frequency of floating-point instructions on the two architectures. The MIPS frequency is always lower than VAX because the MIPS architecture requires load and store instructions where the VAX uses operand specifiers, whose execution is charged to the floating-point instructions in which they appear. The RISC factor is clearly not a function of the floating-point percentage on either machine. Except for doduc, the raw *number* of floating-point instructions is essentially the same between the two architectures, as indeed it ought to be if the two Fortran compilers do an equally good job. The extra MIPS instructions in doduc suggest that the MIPS compiler missed some optimization that the VAX compiler found.

Table 3 also reports the number of loads and stores per instruction together with the raw count of each operation on MIPS relative to VAX.

VAX almost always does more memory references; the exception is stores in fpppp. One explanation for the extra references is the smaller number of general registers and the lack of floating-point registers on the VAX, a point we will discuss in Section 4. As a rule the floating-point benchmarks do more loads and stores than the integer ones—both machines have 32-bit data paths and so need two memory references for a double-precision operand. There is a wide range of loads and stores per instruction, and nothing in the table is correlated with RISC factor. Only in li is a

Table 3: Floating-point operations and 32-bit loads/stores

	floating-point operations			32-bit loads			32-bit stores			
	per instruction		MIPS count	per instruction		MIPS count	per instruction		MIPS count	RISC
benchmark	MIPS	VAX	(VAX=1)	MIPS	VAX	(VAX=1)	MIPS	VAX	(VAX=1)	factor
spice2g6	.034	.083	1.02	.09	0.94	.25	.04	0.14	.65	1.79
matrix300	.156	.370	1.00	.31	1.44	.52	.16	0.40	.93	1.90
nasa7	.216	.440	1.03	.34	1.59	.45	.13	0.52	.53	2.37
fpppp	.228	.879	1.01	.43	2.04	.81	.11	0.36	1.24	2.70
tomcatv	.267	.724	1.05	.40	1.82	.63	.12	0.62	.56	2.86
doduc	.240	.525	1.21	.28	1.03	.72	.09	0.37	.64	2.96
espresso	.000	.000	0.00	.18	0.52	.58	.02	0.14	.24	2.99
eqntott	.000	.000	0.00	.16	0.32	.55	.01	0.07	.13	3.25
li	.000	.000	0.00	.22	0.85	.42	.12	0.51	.38	3.69

Table 4: Cache behavior

	D-stream cache read misses					I-stream cache misses			
	miss ratio (%)		per instruction		MIPS count	per instruction		MIPS count	RISC
benchmark	MIPS	VAX	MIPS	VAX	(VAX=1)	MIPS	VAX	(VAX=1)	factor
spice2g6	26.9	9.1	.0250	.0856	.72	.0001	.0089	.03	1.79
matrix300	12.7	10.8	.0400	.1550	.61	.0000	.0055	.00	1.90
nasa7	12.3	8.7	.0424	.1390	.64	.0000	.0035	.00	2.37
fpppp	0.2	2.4	.0007	.0496	.06	.0024	.0588	.16	2.70
tomcatv	5.7	5.4	.0228	.0982	.66	.0000	.0040	.00	2.86
doduc	0.9	2.7	.0026	.0275	.25	.0031	.0336	.24	2.96
espresso	0.7	4.0	.0012	.0208	.10	.0002	.0026	.13	2.99
eqntott	3.3	4.0	.0055	.0128	.46	.0000	.0021	.00	3.25
li	0.6	1.8	.0013	.0158	.13	.0002	.0103	.03	3.69

significant percentage of the VAX references attributable to register saving and restoring in the procedure linkage instructions (48 percent of all loads and stores).

3.3 Cache behavior

Table 4 reports the cache behavior of the nine benchmarks on the two machines. The VAX results come from the hardware monitor, which is attached not only to the micro-PC but also to the memory bus [5]; the MIPS results come from cache simulations [20]. All three of the caches (mixed Instructions and Data on the VAX, separate on MIPS) are 64 KBytes, direct-mapped, and write-through (except the MIPS I-cache), with 64-byte blocks. SPEC benchmark cache performance in other configurations has been investigated by Pnevmatikatos and Hill [24].

There is a relationship between the RISC factor and the D-stream miss ratio, particularly on the VAX: roughly speaking, higher miss ratios are attached to lower RISC factors. In particular, the three benchmarks with the highest miss ratios on both machines also have the three lowest RISC factors. These three also have three of the four highest relative *counts* of misses on MIPS. The compelling explanation for the low RISC factors is of course the fact that a cache miss of fixed delay degrades the performance of a low-CPI RISC machine more than it does a high-CPI VAX.

The I-stream is much less important than the D-stream for almost all benchmarks on both machines. Particularly in the MIPS M/2000, with its separate I-cache, the I-stream cache behavior is excellent. The VAX implementation, with its shared cache, experiences many more I-stream misses, but the effect is still small: the program with the highest percentage of cycles lost to I-stream stall is li, which loses only 4 percent. The next-highest I-stream stall figure is 1.8 percent.

One might assume that the MIPS D-stream miss ratio would be consistently lower than the VAX one, given the separate D-cache. But Table 4 shows that the MIPS number is actually worse on four benchmarks and close on a fifth. While detailed cache behavior is often quite inscrutable, there is an intuitive explanation for this. VAX loads outnumber MIPS loads in large part because VAX has fewer registers, and so the "extra" loads are often references to data that the MIPS compiler keeps in registers. If the compiler's judgement is good, then these loads ought to be more likely to hit in the VAX cache than the rest of the loads. This effect would be strongest when the VAX I-stream is not a major factor in cache performance, and in fact Table 4 shows that for the most part small VAX I-stream miss rates come from programs in which the MIPS D-stream miss ratio is above or close to the VAX's (the exception is espresso).

4 Discussion

Figure 2 illustrates the relationship between the instruction ratio and the CPI ratio. As we pointed out earlier, the RISC factor itself has lower variance than either of its constituents; this is illustrated in the figure by the tendency of the points to cluster around a single line of constant relative performance, namely the line $MIPS = 2.66 \times VAX$. The correlation has a simple and natural explanation: given reasonable compilers, higher VAX CPI should correspond to a higher relative instruction count on MIPS. In this section we will explore the correlation of the ratios, consider why RISC has a significant net advantage, and look at explanations for what variance there is in RISC factor.

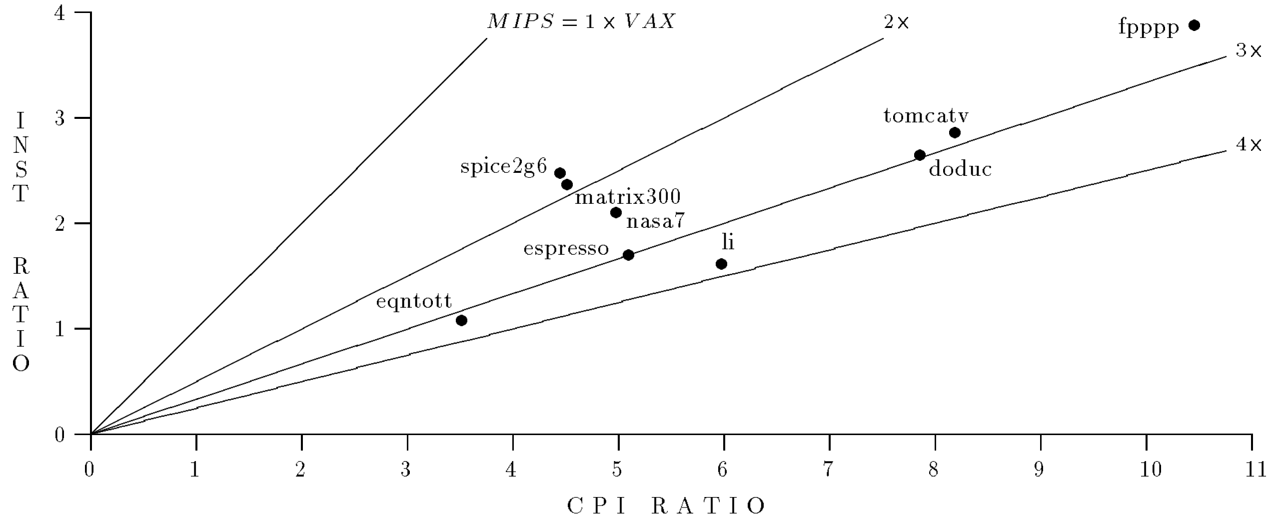

Figure 2: Instruction ratio versus CPI ratio. Lines of constant RISC factor are shown.

4.1 Exploring the extremes

A number of factors are at work in Figure 2. Some help explain the tradeoff between MIPS instructions and VAX CPI, while others help explain the consistent net advantage of MIPS. There are even a few factors that favor VAX. Before considering carefully these various factors we will take a closer look at two of the SPEC benchmarks, fpppp and eqntott. Benchmark fpppp has the highest CPI ratio *and* the highest instruction ratio of any of the benchmarks; eqntott is just the reverse. Neither one, however, has the highest or lowest CPI on either machine, and neither has the highest or lowest RISC factor.

Benchmark fpppp has extraordinary instruction and CPI ratios: the MIPS instruction count is nearly 4 times the VAX count, and the 8700's average CPI is over 10 times the M/2000's ! Our measurements show that this program has the highest number of operand specifiers per VAX instruction of any benchmark (Figure 3, below), the highest number of loads per instruction in both architectures (Table 3), and the highest frequency of floating-point operations on VAX (Table 3). Because the number of loads is similar, and the number of floating-point operations nearly identical on the two architectures, it is reasonable to imagine a correspondence between a VAX floating-point instruction and a sequence of MIPS instructions.

Suppose the operands are in memory on both architectures. Then the VAX will load its double-precision operands with operand-specifier microcode, whose cycles of execution are charged to the floating-point instruction. The MIPS machine will instead do two single-cycle 32-bit load instructions per double-precision operand, followed by a floating-point instruction that operates on registers. Any necessary address calculations that can be done in operand specifiers would further increase the MIPS instruction count and the VAX CPI. If the result needs to go to memory, the VAX will again use a multiple-cycle operand specifier microroutine and charge the cycles to the floating-point instruction. The MIPS machine will do two single-cycle stores, possibly surrounded by address arithmetic instructions. Compared with the other benchmarks, fpppp will see these effects more strongly because of its large number of loads and high density of (double-precision) operand specifiers. The result is an unusually high instruction ratio and CPI ratio.

If this were the entire story, fpppp's RISC factor would be 1.0 and not 2.7. The main explanation is the relative performance of the floating-point hardware. The MIPS implementation is much faster (see Table 1), and also allows some instruction overlap. The VAX spends more than half its CPI in floating-point instruction execution (not counting operand specifiers) and has only trivial instruction overlap. Other possible contributing factors are the effect of the larger number of registers and the faster MIPS branches (see below); but since the number of loads is close between the two machines, and the number of branches quite small, we believe these effects are much smaller than the effect of the floating-point hardware.

Benchmark eqntott is very different from fpppp. It has the lowest CPI ratio and the lowest instruction ratio of all nine benchmarks. In fact the two machines execute almost the same number of instructions (Table 2). For this program, then, we need to find explanations that raise VAX CPI *without* simultaneously raising the MIPS instruction count. VAX operand specifiers once again explain a good deal. Eqntott has the lowest number of operand specifiers per VAX instruction of any of the benchmarks (Figure 3), and also has a very small number of loads and stores (Table 3). So to the extent that we can fairly imagine a VAX instruction mapping into some sequence of MIPS instructions, what is happening here is exactly the opposite of what happened in fpppp. That is, operand specifier processing does not correspond to extra MIPS instructions very often. Also, the frequent use of registers increases VAX CPI without increasing MIPS instruction count because the 8700 usually uses a separate cycle for each register operand. A two-register integer add, for example, takes one instruction on both architectures, but three cycles on the VAX 8700 versus MIPS' one. Although the number of loads is small, VAX does almost twice as many as MIPS, raising the VAX CPI and providing evidence that this benchmark benefits from the larger number of registers in MIPS. Finally, both

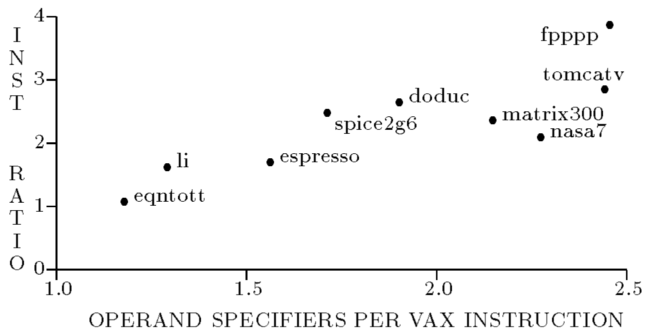

Figure 3: The correlation between the number of operand specifiers per VAX instruction and the instruction ratio

implementations of eqntott branch very frequently, which is relatively bad for VAX CPI, since simple branches take more cycles on the VAX 8700.

4.2 Architectural factors with compensating influence

We will now consider more closely the factors we have encountered in looking at fpppp and eqntott, beginning with those that have compensating influence on the two architectures: about the same cost in MIPS instructions and VAX CPI.

VAX operand specifiers: loads and stores. Most VAX memory references and loads of immediate data are done by operand specifier microcode. Some of these are quite simple, loading a single I-stream constant, say, or using the contents of a register to address memory. The MIPS architecture would need to use separate load and store instructions to accomplish the same function. Some specifiers do various kinds of address calculation (indexing, auto-increment, and so on) that take multiple VAX cycles and would correspond to multiple MIPS instructions. And finally, double-precision operands are loaded and stored by single (two-cycle) operand specifiers on VAX, where they would (in the simple case) take two instructions on MIPS. The average number of operand specifiers per VAX instruction is in fact correlated with instruction ratio, as shown in Figure 3.

Fancy VAX instructions: necessary functionality. Some VAX instructions perform functions more sophisticated than MIPS can accomplish in a single instruction. Loop control instructions, for example, increment the loop index, test it against a limit, and do a conditional branch. When the same or a similar function is required on MIPS, it will use multiple instructions. If the VAX microcode and the MIPS sequence use the same algorithm, we have, again, compensating effects on instruction ratio and CPI ratio.

4.3 Architectural factors favoring MIPS

A number of factors contribute to the consistent net advantage of the M/2000. Most result in increased VAX CPI, and two (number of registers and branch displacement size) can also inflate the VAX instruction count.

Operand specifier decoding. The VAX 8700 (and most other models) usually takes at least one cycle to process each operand specifier. When the specifier references memory, there is a compensating influence on MIPS instruction count. But for register and literal specifiers, this simply means more VAX CPI without a matching effect on MIPS. A three-register integer add, for example, takes four cycles on the 8700 but just one on the MIPS machine.

Number of registers. The MIPS architecture has 32 (32-bit wide) general registers and 16 (64-bit wide) floating-point registers; VAX has 15 (32-bit wide) general registers that can be used for both integer and floating-point data. This can obviously lead to more memory references on the VAX (done either with operand specifiers, or with instructions, if an operand is to be loaded into a register and re-used, or a result saved), while having no compensating effect on MIPS. These extra memory references take cycles to execute and may cost still more cycles if they miss in the cache or stall for some other reason.

Floating-point hardware and instruction overlap. The use of a large and separate set of floating-point registers helps MIPS, especially in late-1980s CMOS, where the floating-point unit is not in the same chip as the CPU. Floating point operations can be performed without requiring data to be moved between chips. In a VAX microprocessor implementation such as the 6000/400 or the 4000/300, several cycles are required to move both source operands from the CPU into the FPU, and read the results back into the CPU. For example, in these VAXes the actual floating-point multiply takes only five cycles inside the FPU compared to the fifteen cycles required for the entire multiply instruction. Since VAX uses the same registers for integer and floating point, significant overlapping of instructions would require complex register scoreboarding. Thus the configuration of registers is an architectural difference with significant performance consequences when the FPU is not integrated within the CPU. Having only register destinations for floating-point instructions is another such difference; because of this it is much easier to overlap execution of multi-cycle instructions on MIPS.

Simple jumps and branches. The time for the simplest taken branch (or unconditional jump) on the VAX 8700 is five cycles. On MIPS, which has a delayed branch, it is one cycle if the delay slot is filled, and two otherwise. This difference is due in large part to the VAX *condition codes*, which are set in a late pipeline stage and influence the earliest pipe stage (instruction decode) when a conditional branch is done, thereby creating a pipeline bubble. This bubble cannot be filled by other non-branch instructions because the condition codes are set by almost every VAX instruction [9]. This in turn means that adding an instruction cache would not pay unless branch prediction hardware were added too. The rarer unconditional jumps could profit from an I-cache, but this was not reason enough to justify one in the 8700, and so these jumps also take five cycles. MIPS conditional branches instead use the condition of a register, which is read in an early pipe stage, and which, of course, is not changed by instructions inserted between the write of the register and the branch. Independent of the use of the MIPS branch-delay slot, which we regard as a separate effect (see below), the slower branches cost VAX CPI. Different VAXes may have different implementations of the branch instructions, of course, but it is difficult to see how any VAX could achieve the MIPS speed without lots of extra hardware (e.g., branch prediction).

Fancy VAX instructions: unnecessary overhead and wasted generality. Some complex VAX instructions implement functionality that is simply not needed, or is too general, or both. Perhaps in some of these cases the VAX compilers could use simpler instructions, but where they do not, we have an effect on VAX CPI with no increase in MIPS instruction count. The classic example of this is the VAX procedure call and return instructions. Sometimes the extra overhead includes memory references, as in the procedure instructions, where registers are sometimes saved and restored needlessly.

Instruction scheduling: filled delay slots. The MIPS architecture allows instructions to be inserted in code positions that might otherwise be lost to pipeline delays. The instruction after a conditional branch is always executed and the instruction after a load can do anything except reference the loaded register [19]. This ability is not present in the VAX architecture (although the 8700 *microcode* uses both delay slots when it can). Sometimes the branch-delay or load-delay slot cannot be used, and must be filled by a NOP. But when the delay slot is filled by a useful instruction, the effect is a relative decrease in MIPS CPI.

Translation buffers. The MIPS architecture has a much larger page size, which means, among other things, that a MIPS TLB can map much more memory than a VAX TLB with the same number of entries. (The M/2000's small TLB maps one-half the memory of the 8700's much larger TLB.) Also, MIPS TLB entries are tagged with a process ID, which means that the TLB need not be flushed on a process context switch. The usual arrangement on VAXes is to flush the process half of the TLB on a context switch.

Branch displacement size. Simple conditional branch instructions have 8-bit PC displacements in the VAX architecture, and effectively 18-bit ones in MIPS. When 8 bits is too few and 18 is enough, a VAX program will use an extra instruction.

4.4 Architectural factors favoring VAX

There are in fact two architectural features that favor VAX in this comparison. Neither appears to have a significant effect in the SPEC benchmarks.

Big I-stream constants. The VAX architecture includes address displacements and absolute addresses of 32 bits, and immediate data of whatever size the opcode demands. It is possible to implement the delivery of a 32-bit I-stream constant so that it is as fast as the delivery of a 16-bit or 8-bit constant. When a big displacement, absolute address, or large data constant is needed by the program but not available in a register, the MIPS architecture would use two instructions in the simple case, versus the VAX's one-cycle operand specifier.

Not-taken branches. VAX implementations can easily make not-taken conditional branches execute in one cycle (the 8700 does this). The MIPS architecture requires the execution of one instruction after a conditional branch, and when that instruction is a NOP, the effective cost of the branch is two cycles.

4.5 Variance of the RISC factor

No single phenomenon explains the variance of the programs' RISC factors around the mean of 2.66, but there are a couple of suggestive effects. The floating-point benchmarks do relatively better on the VAX, the integer ones on MIPS. However, the *percentage* of floating point (Table 3) seems not to be relevant: the lowest RISC factor, for example, is attached to the program (spice) with the smallest amount of floating point (leaving out the integer benchmarks); the biggest floating-point percentages go with programs with mean RISC factor (fpppp and tomcatv). All of the floating-point benchmarks are written in Fortran and all the integer ones in C, so in fact we can't disentangle the contribution of the compiler difference from the contribution of floating point, but it seems likely to us that the effect of the compiler by itself is small.

Both machines' cache behavior seems loosely correlated with RISC factor, as is shown in Table 4. The D-stream cache miss ratio, especially in the VAX, falls as the RISC factor rises, with a few exceptions in each architecture.

There are some peculiarities of the two programs with extreme RISC factors. Li has the highest RISC factor, and stands out from the other benchmarks in several ways:

- it has the lowest (VAX) and second-lowest(MIPS) D-stream cache miss ratios;
- it spends the greatest percentage of its VAX cycles in the procedure call and return instructions (28 percent of all cycles, compared with the second-highest value, 7.5 percent, for espresso); and
- it has by far the highest proportion of address-unaligned memory references in the VAX, which are handled by costly microcode traps in the 8700.

Spice has the lowest RISC factor, and its own pecularities. Saavedra-Barrera has observed that the particular input circuit used in the SPEC version causes spice to spend an unusually large amount of time in one small integer routine [28]. We have already seen that spice has the highest D-stream miss ratios, with the MIPS value being quite high (26.9 percent). It also has the lowest number of loads per instruction on MIPS, so that the high miss ratio hurts less than it otherwise might.

5 Conclusion

We now speculate briefly on future implementation directions for each architecture, and then summarize the paper.

5.1 Futures

CPI is a function of a computer's architecture, but also of its hardware implementation, of course. The VAX 8700 designers strove to minimize cycle time at the possible expense of cycles per instruction, using a straightforward pipelined microengine. It is possible to reduce VAX CPI further by adding gates (and complexity and cost). High-end VAX implementations such as the models 8600 [8, 10] and 9000 [15, 12] attempt to do just that. The model 9000, in particular, uses a large amount of logic (roughly one million gate-array gates for the system [1]) to achieve the lowest CPI of any VAX, as shown in Figure 4.

The CPI improvement is the highest for the floating-point benchmarks. The VAX 9000 attempts to issue simple (but multi-specifier) VAX instructions at the rate of one per cycle, and includes the necessary register scoreboarding and other hardware features to allow substantial floating-point instruction overlap. The result is a demonstration that a large number of gates can yield a VAX implementation with

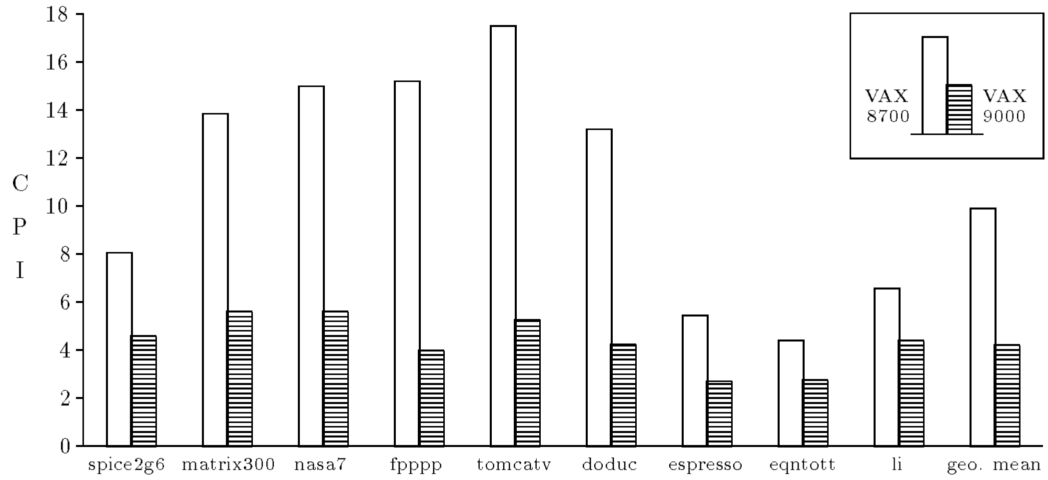

Figure 4: CPI on two VAX implementations

cycles per program comparable to a simple RISC implementation. It seems likely, however, that such an implementation would not be able to achieve the cycle time that a RISC design could, in the same technology. As gate densities increase, it is conceivable that some future single-chip CMOS VAX implementation might achieve CPI numbers that are close to the VAX 9000's.

Just as VAX CPI can be improved by the gate-intensive approach of the model 9000 design, so RISC CPI can be improved by superscalar or superpipelined designs [18, 29]. The IBM RISC System/6000 [17], for example, has a peak issue rate of *four* instructions per cycle.

So while VAX may "catch up" to *current* single-instruction-issue RISC performance, RISC designs will push on with earlier adoption of advanced implementation techniques, achieving still higher performance. The VAX architectural disadvantage might thus be viewed as a time lag of some number of years.

5.2 Summary and caveats

In this paper we have attempted to isolate architecture from implementation in our examination of organizationally similar RISC and CISC engines. The RISC, the MIPS M/2000, has significantly higher architecturally-determined performance than the CISC, the Digital VAX 8700, on the SPEC benchmarks. We observed a wide variability in both instruction ratio and CPI ratio, but found that these two ratios are correlated. As the following table shows, the span of the net performance advantage—what we called the *RISC factor*—is significantly narrower than the span of either ratio:

	min	geo. mean	max
VAX CPI	5.4	9.9	17.4
MIPS CPI	1.1	1.7	3.1
CPI ratio (VAX/MIPS)	3.5	5.8	10.4
Inst. ratio (MIPS/VAX)	1.1	2.2	3.9
RISC factor	1.8	2.7	3.7

We examined a number of architectural factors that help explain the variance of the ratios, and the overall advantage of MIPS.

Three caveats go along with our results. First, we cannot easily disentangle the influence of the compiler from the influence of the architecture. Thus, strictly speaking, our results do not compare the VAX and MIPS architectures *per se*, but rather the combination of architecture with compiler. We have assumed that the compiler quality (in terms of generated code speed) is the "same" for both, while at the same time demonstrating occasional instances of quality differences. So contrary to our assumption, it may very well be that compiler differences, not architecture, are responsible for some of the performance differences we measured.

Second, we measured a rather small number of programs. Measurements that attempt to characterize machines broadly should be based on much more data. It would be desirable, too, to have a wider variety of programming languages and applications represented in the set.

Finally, we have looked in this paper at application-level processor performance only. At the system level, other architectural factors may affect relative performance. Anderson *et al.* [3] have recently studied some operating system primitives and found that the performance of these primitives on RISCs, as compared to VAX, has not scaled with application program performance. And of course the I/O system will determine the performance of some programs, quite independent of processor architecture.

But while our quantitative results may change somewhat as compilers evolve, as more programs are measured, and as operating-system effects are included, we believe that the fundamental finding will stand up: from the architectural point of view (that is, neglecting cycle time), RISC as exemplified by MIPS offers a significant processor performance advantage over a VAX of comparable hardware organization.

Acknowledgments. We would like to thank Rajesh Kothari and Simon Steely for their assistance in performing the measurements, Earl Killian for providing cache simulation results for the M/2000, and John DeRosa, Joel Emer, Bob Sproull, Bob Supnik, and an anonymous referee for their comments on an earlier draft of this paper.

References

[1] Adiletta, M.J., *et al.* Semiconductor Technology in a High-performance VAX System. *Digital Technical Journal 2*, 4 (Fall 1990), pp. 43-60.

[2] Allmon, R. *et al.* CMOS Implementation of a 32b Computer. *1989 ISSCC Technical Digest*, Feb. 1989, pp. 80-81.

[3] Anderson, T.E., Levy, H.M., Bershad, B.N., and Lazowska, E.D. The Interaction of Architecture and Operating System Design. *Proc. Fourth Int. Conf. on Architectural Support for Prog. Lang. and Op. Syst.*, ACM/IEEE, Palo Alto, CA, April 1991, to appear.

[4] Clark, D.W. Pipelining and Performance in the VAX 8800. *Proc. Second Int. Conf. on Architectural Support for Prog. Lang. and Op. Syst.*, ACM/IEEE, Palo Alto, CA, Oct. 1987, pp. 173-177.

[5] Clark, D.W., Bannon, P.J., and Keller, J.B. Measuring VAX 8800 Performance with a Histogram Hardware Monitor. *Proc. 15th Annual International Symposium on Computer Architecture*, Honolulu, May 1988, pp. 176-185.

[6] Cocke, J. The Search for Performance in Scientific Processors. *Comm. ACM 31*, 3 (March 1988), pp. 250-253.

[7] Cocke, J. and Markstein, V. The evolution of RISC Technology at IBM. *IBM J. of Research and Dev. 34*, 1 (Jan. 1990), pp. 4-11.

[8] DeRosa, J., Glackemeyer, R., and Knight, T. Design and Implementation of the VAX 8600 Pipeline. *Computer 18*, 5 (May 1985), pp. 38-48.

[9] DeRosa, J. and Levy, H.M. An Evaluation of Branch Architectures. *Proc. 14th Annual International Symposium on Computer Architecture*, Pittsburgh, PA, June 1987.

[10] Digital Equipment Corp. *Digital Technical Journal 1* (Aug. 1985), DEC, Maynard, MA. This entire issue deals with the VAX 8600.

[11] Digital Equipment Corp. *Digital Technical Journal 4* (Feb. 1987), DEC, Maynard, MA. This entire issue deals with the VAX 8800 family.

[12] Digital Equipment Corp. *Digital Technical Journal 2*, 4 (Fall 1990), DEC, Maynard, MA. This entire issue deals with the VAX 9000.

[13] Durdan, W.H. *et al.* An Overview of the VAX 6000 Model 400 Chip Set. *Digital Technical Journal 2,* 2 (Spring 1990), Digital Equipment Corp., Maynard, MA, pp. 73-83.

[14] Emer, J.S. and Clark, D.W. A Characterization of Processor Performance in the VAX-11/780. *Proc. 11th Annual International Symposium on Computer Architecture*, Ann Arbor, MI, June 1984, pp. 301-310.

[15] Fossum, T. and Fite, D. Designing a VAX for High Performance. *Compcon Spring 90*, IEEE, San Francisco, 1990, pp. 36-43.

[16] Hennessy, J.L., *et al.*, The MIPS Machine. *Proc. Compcon Spring 82*, IEEE, San Francisco, 1982.

[17] International Business Machines Corp. *Journal of Research and Development 34*, 1 (Jan. 1990). This entire issue deals with the IBM RISC System/6000.

[18] Jouppi, N.P. and Wall, D.W. Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines. *Proc. Third Int. Conf. on Architectural Support for Prog. Lang. and Op. Syst.*, ACM/IEEE, Boston, MA, April 1989, pp. 272-282.

[19] Kane, G. *MIPS R2000 RISC Architecture*. Englewood Cliffs, NJ: Prentice-Hall, 1987.

[20] Killian, E. MIPS cache simulation results. Personal communication, Nov. 1990.

[21] MIPS Computer Systems, Inc. *MIPS Language Programmer's Guide*, 1986.

[22] Patterson, D.A. Reduced Instruction Set Computers. *Comm. ACM 28*, 1 (Jan. 1985), pp. 8-21.

[23] Patterson, D.A. and Sequin, C. RISC-1: A Reduced Instruction Set VLSI Computer. *Proc. 8th Annual International Symposium on Computer Architecture*, Minneapolis, May 1981, pp. 443-457.

[24] Pnevmatikatos, D.N. and Hill, M.D. Cache Performance of the Integer SPEC Benchmarks on a RISC. *ACM Comp. Arch. News 18,* 2 (June 1990), pp. 53-68.

[25] Radin, G. The 801 Minicomputer. *Proc. Symp. on Architectural Support for Prog. Lang. and Op. Syst.*, ACM/IEEE, Palo Alto, CA, March 1982, pp. 39-47.

[26] Riordan, T., *et al.* Design Using the MIPS R3000/R3010 RISC Chipset. *Proc. Compcon Spring 89*, IEEE, San Francisco, Spring 1989.

[27] Russell, R.M. The Cray-1 Computer System. *Comm. ACM 21*, 1 (Jan. 1978), pp. 63-72.

[28] Saavedra-Barrera, R.H. The SPEC and Perfect Club Benchmarks: Promises and Limitations. *Hot Chips Symposium 2*. Santa Clara, CA, Aug. 1990.

[29] Smith, M.D., Johnson, M., and Horowitz, M.A. Limits on Multiple Instruction Issue. *Proc. Third Int. Conf. on Architectural Support for Prog. Lang. and Op. Syst.*, ACM/IEEE, Boston, MA, April 1989, pp. 290-302.

[30] Strecker, W.D. VAX-11/780—A Virtual Address Extension for the PDP-11 Family Computers. *Proc. NCC*, AFIPS Press, Montvale, NJ, 1978, pp. 967-980.

[31] Systems Performance Evaluation Cooperative. *SPEC Newsletter: Benchmark Results*, Waterside Assoc., Fremont, CA, Fall 89, Winter 90, Spring 90.

[32] Thornton, J.E. *Design of a Computer: The Control Data 6600*. Glenview, IL: Scott, Foresman, and Co., 1970.