VIEWPOINT

CISCs are Not RISCs, and Not Converging Either

Fallacies + F.U.D. = Foolish Rewrite of Computer History

By John R. Mashey, MIPS Computer Systems

Over the past few years, we've addressed the issues of RISC and CISC differences many times, and we hoped that the issue had been settled. If you can't stand to hear another word about it, now is the time to skip to page 26. We believe, however, that this article presents a particularly clear view of the differences between RISC and CISC architectures. While the author, as a Vice President of MIPS, is hardly impartial, his views are backed up by data—something not found in most such discussions.

Consider the following statements, all of which have recently been made by prominent industry executives. Which are truths, and which are fallacies?

"To distinguish RISCs from CISCs, count instructions: RISCs have less, CISCs have more, but newer RISCs now have as many as CISCs, so there is no difference."

"Some RISCs support character strings or floating point, so they cannot be RISCs."

"RISCs and CISCs are converging. The more complex RISCs and more streamlined CISCs have all converged onto something new called CRISP (Complex Reduced Instruction Set Processor)" (Whoever invented this use for "CRISP" was apparently unaware of the well-documented AT&T CRISP RISC CPU.)

"If a CISC has pipelining and caches, it is the same as a RISC."

"CISCs can use any technique RISCs do, so there is no difference."

All of these are fallacies, either accidental or deliberate. Computer design holds enough arguable issues without having to deal with purposeful obfuscation, so I'll try to dispel some of these fallacies by analyzing data from numerous computer architectures and implementations.

Compared to the previous fallacies, the following is a much better approximation of engineering reality:

Current and proposed microprocessors employ performance-improving *implementation* techniques such as caching, pre-fetching, pipelining, superpipelining, superscalar pipelining, multiple functional units, dynamic scheduling, etc.

No one familiar with computing history could believe that microprocessors invented these techniques, many of which appeared in mainframes years ago, as far back as the 1960s [BEL71, PRA89]. Many lessons can be learned from studying the CDC 6600, which many consider the first RISC, and IBM's 360/91, an exceptionally aggressive CISC implementation.

RISC and CISC microprocessor design teams are all racing to build more aggressive implementations, often following in the tracks of mainframe/supercomputer designs, as modified by the rather different tradeoffs and requirements of low-chip-count designs. The fact that both RISCs and CISCs might use caches is no indication of convergence, but of progress along the track of high-performance designs.

However, architecture strongly affects the results of these implementation techniques:

- Cost and time of design, verification, test
- Silicon requirements
- Feasibility with a given technology or on a given date
- Performance improvement actually obtained
- Synergy with compiler technology
- Cycle-time effects (see below)

Some people believe that with enough transistors, architectural differences become irrelevant. Although there is one element of truth here, it is mostly nonsense.

The element of truth is that sometimes a simpler architecture can just barely be implemented within a useful implementation size (a board, a module, a single chip), and a more complex one cannot. This difference indeed disappears as the number of transistors increases. However, for single-chip CMOS or BiCMOS microprocessors, we are still several generations away from having all the transistors anyone would like on a chip. In other words, complexity still eats silicon that may be used for other performance-enhancing or system-cost-reducing functions. In particular, floating-point hardware can consume huge amounts of silicon.

But suppose we have near-infinite transistors, and everybody can implement incredibly aggressive microarchitectural approaches? Transistors might become almost free, but unfortunately transistors must be connected, and connections are *not* free. For example, consider "fan-out delay," where one gate's output must be input to several others, whose number increases with architectural and implementation complexity. As the number increases, sooner or later the first gate is un-

	Architecture Characteristics											
Processor	A	B	C	D	E	F	G	H	I	J	K	# Odd
P1	4	1	4	1	0	0	1	0	1	8	3	1
P2	5	1	4	1	0	0	1	0	1	5	4	0
P3	2	1	4	2	0	0	1	0	1	5	4	0
P4	2	1	4	3	0	0	1	0	1	5	0	1
P5	5	1	4	10	0	0	1	0	1	5	4	1
P6	5	2	4	1	0	0	1	0	1	4	3	3
P7	1	1	4	4	0	0	1	1	1	5	5	0
P8	2	1	4	4	0	0	1	0	1	5	4	0
P9	26	4	8	2	0	1	2	2	4	4	2	2
P10	12	12	12	15	0	1	2	2	4	3	3	1
P11	10	21	21	23	1	1	2	2	4	3	3	0
P12	11	11	22	44	1	1	2	2	8	4	3	0
P13	13	56	56	22	1	1	6	2	24	4	0	0
Rule	<6	=1	=4	<5	=0	=0	=1	<2	=1	>4	>3	
# Odd	0	1	0	2	2	0	0	0	0	1	3	

Table 1. Key characteristics for 13 different processor architectures, showing RISC and CISC groupings.

able to drive the others at speed, requiring the first gate to drive two more, each of which drives half of the final gates. This adds another level of gate delay, which may lengthen the cycle time if it happens to be in a critical path, of which there are many in well-tuned high-performance CPUs. As the number of final gates increases, the tree of gates expands, and this fact can hardly be unknown to anyone who designs microprocessors, since "optimal fan-out design" is commonly taught in EE courses.

Another problem may be worse: wires don't shrink as fast as transistors. The delay of on-chip wires (of which 32, 64, or more are needed for current buses) is roughly proportional to R (resistance)×C (capacitance). R × C increases as the wire's cross-section decreases or the wire's length increases. If the space between wires decreases too much, they interfere with each other. As cycle times decrease, and transistors shrink, the wires may consume more of the space and cycle time, and complexity only makes this worse. For the big, fast chips that we'll see over the next few years, complexity can still create cycle-time problems that do *not* go away. Just because implementations of two architectures use the same technique does not mean that the cost and performance effects are the same.

Unlike implementation techniques, RISC and CISC architectures possess definite, objective characteristics that distinguish one from the other. RISC was supposed to mean "Reduced Complexity..," not "Reduced Number...," so forget about instruction-counting, which in fact, is quite difficult to do in any truly consistent way. I think my article [MAS86] was reasonably consistent with the views of RISC architects of the time, and if you read that, you'll find it never once says anything about counting instructions.

As RISC and CISC architectures have evolved, they've shown little sign of convergence. For upward compatibility, few architectures evolve by subtracting features, so RISCs have added features, just as CISCs have. However, they've seldom added features that would cause them to be confused for CISCs. For example, they've kept single instruction sizes, simple addressing modes, load/store architectures, and more registers. They've *never* added indirect addressing, general memory-to-memory operations, multiple instruction prefixes, or truly CISCy operations like Edit-and-Mark. The whole point of RISC *architecture* was to enable aggressive *implementations*, most of whose techniques predate any microprocessors.

Before doing the detailed analysis, let's try a quick reality check: if there is no difference between RISC and CISC, why have DEC, IBM, Intel, and Motorola (all companies with huge investments in CISC architectures) all chosen to build RISCs as well, despite the potential market confusion? Are these people all crazy or foolish? I doubt it.

Analysis By the Numbers

I claim that there exist simple rules that clearly distinguish RISC and CISC architectures, for most popular CPUs. Table 1 shows the key architectural characteristics for a variety of architectures. Each of the rows labeled P1–P13 shows data for an implementation of a CPU architecture. Each of the columns labeled A–K illustrates an architectural characteristic, most of which can strongly influence the complexity, cost, or performance of implementations of that architecture. For now, I'll treat the table as raw data to be

Processor Code	Processor Name
P1	AMD 29000
P2	MIPS R3000
P3	Sun SPARC
P4	Motorola 88100
P5	HP PA-RISC
P6	IBM RT
P7	IBM POWER
P8	Intel i860
P9	IBM 3090
P10	Intel 486
P11	National 32016
P12	Motorola 68040
P13	DEC CVAX

Table 2. Processor codes for Table 1.

analyzed for differences, but will later decode the labels. Given this table of data, I think there is a clear line between P8 and P9 dividing the table into two groups. The top group is very consistent within itself, and differs strongly from the bottom group, which also shows more intra-group variation.

The row labeled "Rule" gives, for each column, a numeric rule used tentatively to separate the entries in that column into two groups, by drawing a line under the lowest entry in each column that obeys the rule. In every case except columns D and E, the rule splits rows P8 and P9. Then, any entry is boxed where the numeric rule is inconsistent with the P8/P9 line. For example, P6-B's value of 2 is on the "wrong" side of the line, as is P10-E's value of 0. Under "# Odd" at the right is shown the count of exceptions per CPU. Of the 13 CPUs, 7 show no exceptions; 4 show 1 exception. Only one has three exceptions, and that CPU is no longer being sold! At the bottom is shown the number of odd cases per column, and there are likewise relatively few inconsistencies.

Put another way: if you know the rule, and a CPU's value for a given characteristic, you have a good chance of choosing the group to which it belongs. I claim that there is a clear dividing line, but readers should study this table and see if the reasoning makes sense.

I've omitted various architectural characteristics that might also be important, sometimes because I couldn't figure out any objective way to produce numbers that made sense. For example, I think architecture P10 has more irregularities and complexities not found in P9, P11, P12, or even P13, but I couldn't find any simple metric to describe this. Also, although not strictly a property of an architecture, an interesting property might have been "allows dynamic self-modification of the next following instruction." I've avoided adding up some set of numbers to get an "index of RISC-iness" or some such thing. Finally, I've omitted two architectures (Intel's i960 and Intergraph's Clipper) that are truly on the RISC/CISC border, and, in fact, are often described that way.

Architecture Characteristics

These are the meanings of the columns in Table 1.

"A" gives the approximate age of the architecture in years. Of course, if it was not already obvious, this makes it clear that the top group are RISCs, and the bottom, CISCs. Thus, the more of the rules obeyed by a CPU, the more likely it would be called a RISC.

"B" gives the number of different instruction sizes. RISCs usually have one size.

"C" gives the size of the largest instruction, in bytes, uniformly 4 for RISCs.

"D" gives the number of distinct memory addressing modes, which is sometimes a little tricky to count consistently.

"E" shows the absence (0) or presence (1) of indirect addressing. No RISC I've ever seen has this; a few CISCs were lucky to escape having it.

"F" shows the absence (0) or presence (1) of operations that combine loads/stores with arithmetic, i.e., (0) means "load/store architecture"

"G" shows the number of distinct memory addresses that can be generated by a single instruction. For RISCs this number is generally 1.

"H" shows the handling of unaligned data, where 0 means that any memory reference is required to be aligned on the "natural" boundary, 2 means that unaligned references are allowed almost anywhere, and 1 means that unaligned references are permitted only in certain (easy) cases in hardware, and that other cases are trapped and fixed by software.

"I" gives the maximum number of memory management translations that might be required for data for a single instruction, including effects of unaligned operands, indirect addressing, and number of operands allowed.

"J" gives the number of bits available to address integer registers, i.e., 5 implies up to 32 integer registers.

"K" gives the number of bits available to address floating point registers distinct from integer registers. Here, 0 means that the integer and floating registers are combined.

The CPUs

Table 2 shows the names of the processors listed in Table 1. People who disagree with my interpretations should get a copy of [MAS91], which has considerably more detail and backup data. It also has more data on the evolution patterns within CPU families, as multiple cases are given for 6 of the families. There is insufficient

space here, but the data shows that quite often CISCs have evolved in the direction of increased CISCiness; that is, they had attributes that would have fallen in the RISC group, and then changed them in such a way as to fall within the CISC group. A perfect example is the addition of numerous new addressing modes (including indirect addressing) in Motorola's 68020. On the other hand, RISCs may have added instructions, but seldom, if ever, those features that would have moved them into the CISC category.

Finally, let me point out a few anomalies (or potential ones) in Table 1.

P5-D (10 addressing modes for HP-PA) seems odd. This is an artifact of the way I counted, where every distinct addressing mode or encoding was counted separately. This makes PA look much more complex here than it really is, in that there are really only a small number of distinct mechanisms, still encoded straightforwardly within a 32-bit instruction. See the analysis of addressing modes on the final page of [MAS91]. Many popular CISC addressing modes just don't occur in RISCs.

People might argue that the i860 has a maximum instruction size of 8 bytes (P8-C). I believe that this case is rather different from the typical CISC case, in which one must fetch the instruction and decode it right then, to determine how long it is. Rather, if a specific mode bit is set, the i860 fetches pairs of instructions with strong issue restrictions, but without needing dynamic decoding.

Regarding the IBM 3090, people familiar with the S/360 architecture know quite well that one large subset of instructions (RR, most RX, most RS, most SI) are actually rather RISC-like. Also, instruction decoding is somewhat different from the rest of the CISCs, in that the instruction length is easily determined without sequential decoding often required by others. It is interesting to note that years ago, the fastest S/360 (/91) and best cost/performance S/360 (/44) essentially implemented the RISC-like subset of the instructions in hardware, and trapped the others for software emulation [BEL71]. For people trying to disentangle the RISC-CISC claims and counter-claims, I strongly recommend a good study of the history of the S/360, using for example [PRA89], as it illustrates the problems and solutions of building fast implementations of complex architectures.

Why the Characteristics Might Matter

Architectural characteristics are important for a variety of reasons, and a complete exposition on this topic would require an article in itself. A few examples serve to illustrate the key issues, however.

Consider the effect of dynamic instruction modification, something I'm sure most computer designers wish they'd never allowed. In some CISCs (such as S/360 and x86, but not VAX, for example), one instruction can modify the next, and software that uses this capability does exist. (I know, in a former life I did this kind of thing!) In a simple sequential design, this is easy. Consider the effects on an aggressively parallel design, including separate I- and D-caches. The CPU can never commit the results of any instruction until it is sure that the result of any proceeding store or move does not affect that instruction, despite the fact that this happens only occasionally. This is certainly possible (Amdahl CPUs have done it for years), but it gets more and more expensive as the implementation becomes more aggressive.

As another example, consider memory addressing, which I think is much more relevant to RISC-vs.-CISC than any instruction-counting. (It is unfortunate that it is also more difficult to explain!) In a cached, virtual-memory design (which includes all current general-purpose CPUs) a memory reference requires a host of actions. The CPU must access the cache, and, in many designs, translate the virtual address to a physical address. Any MMU access may cause a trap. Any cache miss may cause a complex ripple of effects, especially when dealing with two-level write-back caches in a multiprocessor, cache-coherent environment. All of this is bad enough for RISCs, whose load instructions each make exactly one aligned memory reference, or at least do not cross cache-line boundaries. Each uses the MMU exactly once, and thus causes at most one trap.

At the other extreme, consider the VAX, which supports instructions with up to 6 memory operands, each of which might use indirect addressing, and each of which (both indirect pointer and final operand) might be unaligned across page boundaries, giving $6 \times 2 \times 2 = 24$ potential MMU traps. Although you'd sometimes like to simply start doing memory-operations, sanity sometimes requires checking that all parts of all operands are accessible before starting. Alternatively, you might include elaborate hardware for undoing the changes made during the instruction, or else provide a complex state-dumping upon exception (as in the 68020) that is very implementation-dependent. For example, each VAX memory operand may include an auto-increment, which must be undone if the last byte of the last memory operand causes a page fault that requires the instruction to be restarted from the beginning.

Personally, I find it amazing and impressive that S/360 and especially VAX implementors make their machines as fast as they do.

To characterize the difficulties, I'd observe that some of the popular CISC architectures were created before people understood the interactions of caches, virtual memory, microarchitectural parallelism, and software interactions as well as they are now known. As a

result, sometimes an architectural feature that was innocuous in a simple sequential implementation requires gates and gate delays out of all proportion when present in a compatible, but more aggressive, implementation. What seems to irritate some CISC designers the most is needing significant implementation complexity just to watch out for some truly rare event. (In bar conversations with such designers, this irritation often surfaces in comments of the form "You RISC wimps! You don't know how lucky you are not to have to worry about *this*, or *that*, or *those* nonobvious implications of architecture XYZ.")

Conclusions

Fast computers often use similar implementation techniques, and few of those techniques are truly new. However, architecture strongly influences the ability to use various implementation techniques, the cost of doing so, and the resulting performance gained.

Electrical engineering fundamentals say that complexity still costs. Transistors may get to be almost free, but wires and gate delays will not.

Contrary to the belief of some, there are some clear architectural distinctions between the CPUs commonly labeled RISC and CISC, and there is very little sign of architectural convergence.

Of course, none of this proves that RISC is automatically better than CISC, and in fact, a good CISC implementation should beat a poor RISC implementation. Perhaps we should seek more specific A-vs.-B comparisons, but it would certainly be better if people would stop trying to obfuscate well-known computer history. ♦

References

[BEL71] Bell and Newell, Computer Structures: Readings and Examples, McGraw-Hill, 1971. Many currently-popular aggressive implementation methods date from the late 1960s. For example, study CDC 6600 (1964) and IBM 360/91 (1967).

[BHA91] Bhandarkar and Clark, "Performance from Architecture: Comparing a RISC and a CISC with Similar Hardware Organization," Proceedings of ASPLOS III, ACM/IEEE, April 1991. Serious analysis of where the performance goes.

[HEN90] Hennessy and Patterson, Computer Architecture: A Quantitative Approach, Morgan Kaufman, San Mateo, CA, 1990. The classic book.

[MAS86] Mashey, "RISC, MIPS, and the Motion of Complexity," Proceedings of UniForum, 1986, Anaheim, CA., pp. 116–124. For the record, what the author was saying about RISC 5 years ago, for comparison.

[MAS91] Mashey, "CISCs are not RISCs, 1991 Edition," MIPS Computer Systems, September 1991. The presentation of which this article is a small subset. You can request a copy by sending e-mail with your paper mailing address to bdeprima@mips.com, or call 408/524-7012 and ask for a copy of the "CISCanard" presentation.

[PRA89] N.S. Prasad, IBM Mainframes: Architecture and Design, McGraw-Hill, New York, 1989.

Superscalar PA-RISC

Continued from page 17

Special I/O drivers enable the external caches to operate at 100 MHz using 9-ns SRAMs. In addition, attention to SRAM characteristics and board layout are required. One benefit of integrating the FP unit and the integer unit was a reduction of the load on the SRAM data bus, which eases the timing problems slightly. HP expects to use multi-chip module packaging to reach even higher clock rates.

Where the Snakes FP unit could issue an instruction every two cycles and had a uniform latency of three cycles, the FP unit on the 7100 can issue every cycle and operation latency is reduced to two cycles.

The 7100 also incorporates a few small improvements that will increase general performance. While a miss is being processed, the data cache does not block further loads and stores that hit in the cache. Only when the data for a load miss is actually needed does the processor stall. On Snakes, a store followed by another cache access incurs a two-cycle penalty, but the penalty is only one cycle on the 7100. To speed some important operating system functions, the block-copy hint (don't allocate and zero the block being written) for the store instruction is implemented in the 7100.

Superscalar Capabilities

Like DEC's Alpha chip but unlike the SuperSPARC and 88110, the 7100 cannot issue two integer ALU instructions at the same time. The superscalar capabilities only allow an integer and an FP instruction to be issued together. Unlike Alpha, however, the 7100 does not have a separate branch or load/store unit, which is probably a consequence of using an existing integer unit design. For this processor, FP loads and stores are considered integer instructions. To reduce the time needed to decode multiple-issue opportunities, the I-cache stores a "pre-decode" bit with each instruction that indicates whether the instruction is destined for the integer or FP data paths.

Since PA-RISC has a composite multiply-add instruction, the peak execution rate at 100 MHz is 200 MFLOPS. The combination of the short, 2-cycle latency of add, subtract, and multiply (only one stall cycle is inserted between dependent operations) with the ability to issue an FP operation and an FP load or store together should make this implementation a floating-point screamer. Many FP applications are limited by operand bandwidth, and small, on-chip first-level caches just make the problem worse. A 100-MHz system using this chip with a 2-MB external data cache is likely to out-perform a 150-MHz Alpha implementation for FP applications even if the Alpha system has a large secondary cache. ♦