RISC vs. CISC: The Post-RISC Era
"A historical approach to the debate"

by Hannibal

(http://arstechnica.com/cpu/4q99/risc-cisc/rvc-2.html)

I. Framing the Debate

The majority of today's processors can't rightfully be called completely RISC or completely CISC. The two textbook architectures have evolved towards each other to such an extent that there's no longer a clear distinction between their respective approaches to increasing performance and efficiency. To be specific, chips that implement the x86 CISC ISA have come to look a lot like chips that implement various RISC ISA's; the instruction set architecture is the same, but under the hood it's a whole different ball game. But this hasn't been a one-way trend. Rather, the same goes for today's so-called RISC CPUs. They've added more instructions and more complexity to the point where they're every bit as complex as their CISC counterparts. Thus the "RISC vs. CISC" debate really exists only in the minds of marketing departments and platform advocates whose purpose in creating and perpetuating this fictitious conflict is to promote their pet product by means of name-calling and sloganeering.

At this point, I'd like to reference a statement made by David Ditzel, the chief architect of Sun's SPARC family and CEO of Transmeta.

> "Today [in RISC] we have large design teams and long design cycles," he said. "The performance story is also much less clear now. The die sizes are no longer small. It just doesn't seem to make as much sense." The result is the current crop of complex RISC chips. "Superscalar and out-of-order execution are the biggest problem areas that have impeded performance [leaps]," Ditzel said. "The MIPS R10,000 and HP PA-8000 seem much more complex to me than today's standard CISC architecture, which is the Pentium II. So where is the advantage of RISC, if the chips aren't as simple anymore?"

This statement is important, and it sums up the current feeling among researchers. Instead of RISC or CISC CPUs, what we have now no longer fits in the old categories. Welcome to the post-RISC era. What follows is a completely revised and re-clarified thesis which found its first expression here on Ars over a year ago, before Ditzel spoke his mind on the matter, and before I had the chance to exchange e-mail with so many thoughtful and informed readers.

In this paper, I'll argue the following points:

1. RISC was not a specific technology as much as it was a design strategy that developed in reaction to a particular school of thought in computer design. It was a rebellion against prevailing norms -- norms that no longer prevail in today's world -- norms that I'll talk about.
2. "CISC" was invented retroactively as a catch-all term for the type of thinking against which RISC was a reaction.
3. We now live in a "post-RISC" world, where the terms RISC and CISC have lost their relevance (except to marketing departments and platform advocates). In a post-RISC world, each architecture and implementation must be judged on its own merits, and not in terms of a narrow, bipolar, compartmentalized worldview that tries to cram all designs into one of two "camps."

After charting the historical development of the RISC and CISC design strategies, and situating those philosophies in their proper historical/technological context, I'll discuss the idea of a post-RISC processor, and show how such processors don't fit neatly into the RISC and CISC categories.

A. The historical approach

Perhaps the most common approach to comparing RISC and CISC is to list the features of each and place them side-by-side for comparison, discussing how each feature aids or hinders performance. This approach is fine if you're comparing two contemporary and competing pieces of technology, like OS's, video cards, specific CPUs, etc., but it fails when applied to RISC and CISC. It fails because RISC and CISC are not so much technologies as they are design strategies -- approaches to achieving a specific set of goals that were defined in relation to a particular set of problems. Or, to be a bit more abstract, we could also call them design philosophies, or ways of thinking about a set of problems and their solutions.

It's important to see these two design strategies as having developed out of a particular set of technological conditions that existed at a specific point in time. Each was an approach to designing machines that designers felt made the most efficient use of the technological resources then available. In formulating and applying these strategies, researchers took into account the limitations of the day's technology—limitations that don't necessarily exist today. Understanding what those limitations were and how computer architects worked within them is the key to understanding RISC and CISC. Thus, a true RISC vs. CISC comparison requires more than just feature lists, SPEC benchmarks and sloganeering—it requires a historical context.

In order to understand the historical and technological context out of which RISC and CISC developed, it is first necessary to understand the state of the art in VLSI, storage/memory, and compilers in the late 70's and early 80's. These three technologies defined the technological environment in which researchers worked to build the fastest machines.

B. Storage and memory

It's hard to underestimate the effects that the state of storage technology had on computer design in the 70's and 80's. In the 1970's, computers used magnetic core memory to store program code; core memory was not only expensive, it was agonizingly slow. After the introduction of RAM things got a bit better on the speed front, but this didn't address the cost part of the equation. To help you wrap your mind around the situation, consider the fact that in 1977, 1MB of DRAM cost about $5,000. By 1994, that price had dropped to under $6 (in 1977 dollars) [2]. In addition to the high price of RAM, secondary storage was expensive and slow, so paging large volumes of code into RAM from the secondary store impeded performance in a major way.

The high cost of main memory and the slowness of secondary storage conspired to make code bloat a deadly serious issue. Good code was compact code; you needed to be able to fit all of it in a small amount of memory. Because RAM counted for a significant portion of the overall cost of a system, a reduction in code-size translated directly in to a reduction in the total system cost. (In the early 90's, RAM accounted for around %36 of the total system cost, and this was after RAM had become quite a bit cheaper [4].) We'll talk a bit more about code size and system cost when we consider in detail the rationales behind CISC computing.

C. Compilers

David Patterson, in a recently published retrospective article on his original proposal paper for the RISC I project at Berkeley, writes:

> ‘Something to keep in mind while reading the paper was how lousy the compilers were of that generation. C programmers had to write the word "register" next to variables to try to get compilers to use registers. As a former Berkeley Ph.D. who started a small computer company said later, "people would accept any piece of junk you gave them, as long as the code worked." Part of the reason was simply the speed of processors and the size of memory, as programmers had limited patience on how long they were willing to wait for compilers.’ [3]

The compiler’s job was fairly simple at that point: translate statements written in a high level language (HLL), like C or PASCAL, into assembly language. The assembly language was then converted into machine code by an assembler. The compilation stage took a long time, and the output was hardly optimal. As long as the HLL to assembly translation was correct, that was about the best you could hope for. If you really wanted compact, optimized code, your only choice was to code in assembler. (In fact, some would argue that this is still the case today.)

D. VLSI

The state of the art in Very Large Scale Integration (VLSI) yielded transistor densities that were low by today’s standards. You just couldn’t fit too much functionality onto one chip. Back in 1981 when Patterson and Sequin first proposed the RISC I project (RISC I later became the foundation for Sun’s SPARC architecture), a million transistors on a single chip was a lot [1]. Because of the scarcity of available transistor resources, the CISC machines of the day, like the VAX, had their various functional units split up across multiple chips. This was a problem, because the delay-power penalty on data transfers between chips limited performance. A single-chip implementation would have been ideal, but, for reasons we’ll get into in a moment, it wasn’t feasible without a radical rethinking of current designs.

II. The CISC solution

A. The HLLCA and the software crisis

Both the sorry state of early compilers and the memory-induced constraints on code size caused some researchers in the late 60’s and early 70’s to predict a coming "software crisis." Hardware was getting cheaper, they argued, while software costs were spiraling out of control. A number of these researchers insisted that the only way to stave off impending doom was to shift the burden of complexity from the (increasingly expensive) software level to the (increasingly inexpensive) hardware level. If there was a common function or operation for which a programmer had to write out all the steps every time he or she used it, why not just implement that function in hardware and make everyone’s life easier? After all, hardware was cheap (relatively speaking) and programmer time wasn’t. This idea of moving complexity from the software realm to the hardware realm is the driving idea behind CISC, and almost everything that a true CISC machine does is aimed at this end.

Some researchers suggested that the way to make programmers and compiler-writers jobs easier was to "close the semantic gap" between statements in a high-level language and the corresponding statements in assembler. "Closing the semantic gap" was a fancy way of saying that system designers should make assembly code look more like C or PASCAL code. The most extreme proponents of this kind of thinking were calling for the move to a High-Level Language Computing Architecture (HLLCA).

The HLLCA was CISC taken to the extreme. Its primary motivation was to reduce overall system costs by making computers easy to program for. By simplifying the programmer's and compiler's jobs, it was thought that software costs could be brought under control. Here's a list of some of the most commonly stated reasons for promoting HLLCAs [5]:

- Reduce the difficulty of writing compilers.
- Reduce the total system cost.
- Reduce software development costs.
- Eliminate or drastically reduce system software.
- Reduce the semantic gap between programming and machine languages.
- Make programs written in a HLL run more efficiently.
- Improve code compaction.
- Ease debugging.

To summarize the above, if a complex statement in a HLL were to translate directly into exactly one instruction in assembler, then

- Compilers would be much easier to write. This would save time and effort for software developers, thereby reducing software development costs.
- Code would be more compact. This would save on RAM, thereby reducing the overall cost of the system hardware.
- Code would be easier to debug. Again, this would save on software development and maintenance costs.

At this point in our discussion, it's important to note that I'm not asserting that the flurry of literature published on HLLCAs amounted to a "CISC movement" by some "CISC camp", in the way that there was a RISC movement led by the Berkeley, IBM, and Stanford groups. There never actually was any sort of "CISC movement" to speak of. In fact, the term "CISC" was invented only after the RISC movement had started. "CISC" eventually came to be a pejorative term meaning "anything not RISC." So I'm not equating the HLLCA literature with "CISC literature" produced by a "CISC movement", but rather I'm using it to exemplify one of the main schools of thought in computer architecture at the time, a school of thought to which RISC was a reaction. We'll see more of that in a bit.

B. CISC and the performance equation

The discussion so far has focused more on the economic advantages of CISC, while ignoring the performance side of the debate. The CISC approach to increasing performance is rooted in the same philosophy that I've been referring too thus far: move complexities from software to hardware. To really understand how this affords a performance advantage, lets look at the performance equation.

time/program = [(instructions/program) **x** (cycles/instruction) **x** (time/cycle)]

The above equation is a commonly used metric for gauging a computer's performance. Increasing performance means reducing the term on the left side of the "=", because the less time it take to run a program, the better the machine's performance. A CISC machine tries to reduce the amount of time it takes to run a program by reducing the first term to the right of the "=", that is, the number of instructions per program. Researchers thought that by reducing the overall number of instructions that the machine executes to perform a task you could reduce the overall amount of time it takes to finish that task, and thus increase performance.

So decreasing the size of programs not only saved memory, it also saved time because there were fewer lines of code to execute. While this approach proved fruitful, it had its drawbacks. In a moment we'll discuss those drawbacks, and how they led researchers to focus on other terms in the equation in their efforts to increase performance.

C. Complex instructions: an example

Before we go any further, lets look at an example so we can better understand the motivations behind increasing the complexity of a machine's instructions. If you think you have the above down pat, you can go ahead and skip to the next section.

Consider the situation where we want to take the cube of 20 and store it in a variable. To do this, we write some code in a hypothetical high-level language called "H." (For the sake of simplicity, all variables in H designate specific architectural registers.) Our H compiler translates this code into assembler for the ARS-1 platform. The ARS-1 ISA only has two instructions:

MOVE [destination register, integer or source register]. This instruction takes a value, either an integer or the contents of another register, and places it the destination register. So MOVE [D, 5] would place the number 5 in register D. MOVE [D, E] would take whatever number is stored in E and place it in D. MUL [destination register, integer or multiplicand register]. This instruction takes the contents of the destination register and multiplies it by either an integer or the contents of multiplicand register, and places the result in the destination register. So MUL [D, 70] would multiply the contents of D by 70 and place the results in D. MUL [D, E] would multiply the contents of D by the contents of E, and place the result in D.

Statements in H	**Statements in ARS-1 Assembly**
1. A = 20	1. MOVE [A, 20]
2. B = CUBE (A)	2. MUL [A, A]
	3. MUL [A, A]
	4. MOVE [B, A]

[Editor's note: this example actually finds 20^4, not 20^3. I'll correct it when the load on the server goes down. Still, it serves its purpose.] Notice how in the above example it takes four statements in ARS-1 assembly to do the work of two statements in H? This is because the ARS-1 computer has no instruction for taking the CUBE of a number. You just have to use two MUL instructions to do it. So if you have an H program that uses the CUBE( ) function extensively, then when you compile it the assembler program will be quite a bit larger than the H program. This is a problem, because the ARS-1 computer doesn't have too much memory. In addition, it takes the compiler a long time to translate all those CUBE( ) statements into MUL[ ] instructions. Finally, if a programmer decides to forget about H and just write code in ARS-1 assembler, he or she has more typing to do, and the fact that the code is so long makes it harder to debug.

One way to solve this problem would be to include a CUBE instruction in the next generation of the ARS architecture. So when the ARS-2 comes out, it has an instruction that looks as follows:

- CUBE [destination register, multiplicand register]. This instruction takes the contents of the multiplicand register and cubes it. It then places the result in the destination register.
  So CUBE [D, E] takes whatever value is in E, cubes it, and places the result in D.

Statements in H	**Statements in ARS-2 Assembly**
1. A = 20	1. MOVE [A, 20]
2. B = CUBE (A)	2. CUBE [B, A]

So now there is a one-to-one correspondence between the statements in H, on the right, and the statements in ARS-2, on the left. The "semantic gap" has been closed, and compiled code will be smaller -- easier to generate, easier to store, and easier to debug. Of course, the ARS-2 computer still cubes numbers by multiplying them together repeatedly in hardware, but the programmer doesn't need to know that. All the programmer knows is that there is an instruction on the machine that will cube a number for him; how it happens he doesn't care. This is a good example of the fundamental CISC tenet of moving complexity from the software level to the hardware level.

D. Complex addressing modes

Besides implementing all kinds of instructions that do elaborate things like cube numbers, copy strings, convert values to BCD, etc., there was another tactic that researchers used to reduce code size and complication: complex addressing modes. The picture below shows the storage scheme for a generic computer. If you want to multiply two numbers, you would first load each operand from a location in main memory (locations 1:1 through 6:4) into one of the six registers (A, B, C, D, E, or F). Once the numbers are loaded into the registers, they can be multiplied by the execution unit (or ALU).

Since ARS-1 has a simple, load/store addressing scheme, we would use the following code to multiply the contents of memory locations 2:3 and 5:2, and store the result in address 2:3.

1. MOVE [A, 2:3]
2. MOVE [B, 5:2]
3. MUL [A, B]
4. MOVE [2:3, A]

The above code spells out explicitly the steps that ARS-1 has to take to multiply the contents of the two memory locations together. It tells the computer to load the two registers with the contents of main memory, multiply the two numbers, and store the result back in main memory.

If we wanted to make the assembly less complicated and more compact, we could modify the ARS architecture so that when ARS-2 is released, the above operation can be done with only one instruction. To do this, we change the MUL instruction so that it can take two memory addresses as its operands. So the ARS-2 assembler for the memory-to-memory multiply operation would look like this:

1. MUL [2:3, 5:2]

Changing from four instructions to one is a pretty big savings. Now, the ARS-2 still has to load the contents of the two memory locations into registers, multiply them, and write them back out—there's no getting around all that—but all of those lower-level operations are done in hardware and are invisible to the programmer. So all that complicated work of shuffling memory and register contents around is hidden; the computer takes care of it behind the scenes. This is an example of a complex addressing mode. That one assembler instruction actually carries out a "complex" series of operations. Once again, this is an example of the CISC philosophy of moving functionality from software into hardware.

E. Microcode vs. direct execution

Microprogramming was one of the key breakthroughs that allowed system architects to implement complex instructions in hardware [6]. To understand what microprogramming is, it helps to first consider the alternative: direct execution. With direct execution, the machine fetches an instruction from memory and feeds it into a hardwired control unit. This control unit takes the instruction as its input and activates some circuitry that carries out the task. For instance, if the machine fetches a floating-point ADD and feeds it to the control unit, there's a circuit somewhere in there that kicks in and directs the execution units to make sure that all of the shifting, adding, and normalization gets done. Direct execution is actually pretty much what you'd expect to go on inside a computer if you didn't know about microcoding.

The main advantage of direct execution is that it's fast. There's no extra abstraction or translation going on; the machine is just decoding and executing the instructions right in hardware. The problem with it is that it can take up quite a bit of space. Think about it. If every instruction has to have some circuitry that executes it, then the more instructions you have, the more space the control unit will take up. This problem is compounded if some of the instructions are big and complex, and take a lot of work to execute. So directly executing instructions for a CISC machine just wasn't feasible with the limited transistor resources of the day.

Enter microprogramming. With microprogramming, it's almost like there's a mini-CPU on the CPU. The control unit is a microcode engine that executes microcode instructions. The CPU designer uses these microinstructions to write microprograms, which are stored in a special control memory. When a normal program instruction is fetched from memory and fed into the microcode engine, the microcode engine executes the proper microcode subroutine. This subroutine tells the various functional units what to do and how to do it.

As you can probably guess, in the beginning microcode was a pretty slow way to do things. The ROM used for control memory was about 10 times faster than magnetic core-based main memory, so the microcode engine could stay far enough ahead to offer decent performance [7]. As microcode technology evolved, however, it got faster and faster. (The microcode engines on current CPUs are about 95% as fast as direct execution [10].) Since microcode technology was getting better and better, it made more and more sense to just move functionality from (slower and more expensive) software to (faster and cheaper) hardware. So ISA instruction counts grew, and program instruction counts shrank.

As microprograms got bigger and bigger to accommodate the growing instructions sets, however, some serious problems started to emerge. To keep performance up, microcode had to be highly optimized with no inefficiencies, and it had to be extremely compact in order to keep memory costs down. And since microcode programs were so large now, it became much harder to test and debug the code. As a result, the microcode that shipped with machines was often buggy and had to be patched numerous times out in the field. It was the difficulties involved with using microcode for control that spurred Patterson and others began to question whether implementing all of these complex, elaborate instructions in microcode was really the best use of limited transistor resources [11].

III. The RISC solution

For reasons we won't get into here, the "software crisis" of the 60's and 70's never quite hit. By 1981, technology had changed, but architectures were still following the same old trend: move complexity from software to hardware. As I mentioned earlier, many CISC implementations were so complex that they spanned multiple chips. This situation was, for obvious reasons, not ideal. What was needed was a single-chip solution—one that would make optimal use of the scarce transistor resources available. However, if you were going to fit an entire CPU onto one chip, you had to throw some stuff overboard. To this end, there were studies done that were aimed at profiling actual running application code and seeing what types of situations occurred most often. The idea was to find out what the computer spent the most time working on, and optimize the architecture for that task. If there were tradeoffs to be made, they should be made in favor of speeding up what the computer spends the most time on, even if it means slowing down other, less commonly done tasks. Patterson summed up this quantitative approach to computer design in the famous dictum: make the common case fast.

As it turned out, making the common case fast meant reversing the trend that CISC had started: functionality and complexity had to move out of the hardware and back into the software. Compiler technology was getting better and memory was getting cheaper, so many of the concerns that drove designers towards more complex instruction sets were now, for the most part, unfounded. High-level language support could be better done in software, reasoned researchers; spending precious hardware resources on HLL support was wasteful. Those resources could be used in other places to enhance performance.

A. Simple instructions and the return of direct execution

When RISC researchers went looking for excess functionality to throw overboard, the first thing to go was the microcode engine, and with the microcode engine went all those fancy instructions that allegedly made programmer's and compiler-writer's jobs so much easier. What Patterson and others had discovered was that hardly anyone was using the more exotic instructions. Compiler-writers certainly weren't using them—they were just too much of a pain to implement. When compiling code, compilers forsook the more complex instructions, opting instead to output groups of smaller instructions that did the same thing. What researchers learned from profiling applications is that a small percentage of an ISA's instructions were doing the majority of the work. Those rarely-used instructions could just be eliminated without really losing any functionality. This idea of reducing the instruction set by getting rid of all but the most necessary instructions, and replacing more complex instructions with groups of smaller ones, is what gave rise to the term Reduced Instruction Set Computer. By including only a small, carefully-chosen group of instructions on a machine, you could get rid of the microcode engine and move to the faster and more reliable direct execution control method.

Not only was the number of instructions reduced, but the size of each instruction was reduced as well [18]. It was decided that all RISC instructions were, whenever possible, to take one and only one cycle to complete. The reasoning behind this decision was based on a few observations. First, researchers realized that anything that could be done with microcode instructions could be done with small, fast, assembly language instructions. The memory that was being used to store microcode could just be used to store assembler, so that the need for microcode would be obviated altogether. Therefore many of the instructions on a RISC machine corresponded to microinstructions on a CISC machine. [12]

The second thing that drove the move to a one-cycle, uniform instruction format was the observation that pipelining is really only feasible to implement if you don't have to deal with instructions of varying degrees of complexity. Since pipelining allows you to execute multiple pieces of the different instructions in parallel, a pipelined machine has a drastically lower average number of cycles per instruction (CPI). (For an in-depth discussion of pipelining, check out my K7 design preview). Lowering the average number of cycles that a machine's instructions take to execute is one very effective way to lower the overall time it takes to run a program.

IV. RISC and the performance equation

Our discussion of pipelining and its effect on CPI brings us back to a consideration of the performance equation,

time/program = [(instructions/program) **x** (cycles/instruction) **x** (time/cycle)]

RISC designers tried to reduce the time per program by decreasing the second term to the right of the "=", and allowing the first term to increase slightly. It was reasoned that the reduction in cycles-per-instruction achieved by reducing the instruction set and adding pipelining and other features (about which we'll talk more in a moment) would more than compensate for any increase in the number of instructions per program. As it turns out, this reasoning was right.

A. LOAD/STORE and registers

Besides pipelining, there were two key innovations that allowed RISC designers to both decrease CPI and keep code bloat to a minimum: the elimination of complex addressing modes and the increase in the number of architectural registers. In a RISC architecture, there are only register-to-register operations, and only LOADs and STOREs can access memory. Recall the ARS-1 and ARS-2 example architectures we looked at earlier.

In a LOAD/STORE architecture, an ARS-2 instruction like MUL [2:3, 5:2] couldn't exist. You would have to represent this instruction with two LOAD instructions (used to load operands from memory into the registers), one register-to-register MUL instruction (like MUL [A, B] and a STORE instruction (used to write the result back to memory). You would think that having to use LOADs and STOREs instead of a single, memory-to-memory instruction would increase the instruction count so much that memory usage and performance would suffer. As it turns out, there a few reasons why the code doesn't get as bloated as you might expect.

The aforementioned profiles of HLL application code showed Patterson and his colleagues that local scalars are by far the most frequent operands to appear in a program; they found that over 80% of the scalar values in a program are local variables [13]. This meant that if they added multiple banks of registers to the architecture, they could keep those local scalars right there in the registers, and avoid having to LOAD them every time. Thus, whenever a subroutine is called, all the local scalars are loaded into a bank of registers and kept there as needed [18]. In contrast, my hypothetical ARS-2 machine uses microcode operand-specifiers to carry out the loads and stores associated with memory-to-memory operations (much like the VAX). What this means is that whenever the ARS-2 encounters something like the MUL [2:3, 5:2] instruction, its microcode engine translates this MUL into a set of microinstructions that

1. LOAD the contents of 2:3 into a register,
2. LOAD the contents of 5:2 into a register,
3. MUL the two registers, and
4. STORE the result back in 2:3.

This series of LOADs and STOREs takes multiple cycles, just like it would on a RISC machine. The only difference is that those cycles are charged to the MUL instruction itself, making the MUL a multicycle instruction. And after the MUL is over and the result is written to memory, the ARS-2's microcode program will write over the contents of the two registers it just used with whatever data is needed next, instead of keeping it around for reuse. This means that the ARS-2 is actually doing more LOADs and STOREs than would a RISC machine, because it can't split the memory accesses off from the MUL instruction and manage them intelligently.

Since those LOADs and STOREs are tied to the MUL instruction, the compiler can't shuffle them around and rearrange them for maximum efficiency. In contrast, the RISC's separation of LOADs and STOREs from other instructions allows the compiler to schedule an operation in the delay slot immediately after the LOAD. So while it's waiting a few cycles for the data to get loaded to the register, it can do something else instead of sitting idle. Many CISC machines, like the VAX, take advantage of this LOAD delay slot also, but this has to be done in microcode.

B. The changed role of the compiler

As you can see from the above discussion, the compiler's role in managing memory accesses is quite different on a RISC machine than it is on a CISC one. Patterson notes:

> "RISC compilers try to keep operands in registers so that simple register-to-register instructions can be used. Traditional compilers, on the other hand, try to discover the ideal addressing mode and the shortest instruction format to add the operands in memory. In general, the designers of RISC compilers prefer a register-to-register model of execution so that compliers can keep operands that will be reused in registers, rather than repeating a memory access of a calculation. They therefore use LOADs and STOREs to access memory so that operands are not implicitly discarded after being fetched, as in the memory-to-memory architecture." [16]

In a RISC architecture, the compiler's role is much more prominent. The success of RISC actively depends on intelligent, optimizing compilers that can take the increased responsibilities that RISC gives them and put out optimal code. This act of shifting the burden of code optimization from the hardware to the compiler was one of the key advances of the RISC revolution. Since the hardware was now simpler, this meant that the software had to absorb some of the complexity by aggressively profiling the code and making judicious use of RISC's minimal instruction set and expanded register count. Thus RISC machines devoted their limited transistor resources to providing an environment in which code could be executed as quickly as possible, trusting that the compiler had made the code compact and optimal.

V. RISC and CISC -- Side by Side?

By now, it should be apparent that the acronyms "RISC" and "CISC" belie the fact that both design philosophies deal with much more than just the simplicity or complexity of an instruction set. In the table below, I summarize the information that I've presented so far, beginning with each philosophy's general strategy for increasing performance and keeping costs down. I hope you've seen enough by now to understand, however, that any approach that affects price will affect performance, and vice versa, so my division of RISC and CISC design strategies into "price" and "performance" is somewhat arbitrary and artificial. In fact, because the RISC and CISC design philosophies developed within a matrix defined by the price and performance of the technologies we've discussed (VLSI, compilers, memory/storage), the following summary of RISC and CISC strategies and features should only be understood as a set of heuristics for helping you organize and develop your own thoughts on the design decisions that CPU architects make, and not as hard-and-fast rules or definitions for determining exactly what does and does not constitute RISC and/or CISC [17].

CISC	RISC
Price/Performance Strategies	
Price: move complexity from software to hardware.	Price: move complexity from hardware to software.
Performance: make tradeoffs in favor of decreased code size, at the expense of a higher CPI.	Performance: make tradeoffs in favor of a lower CPI, at the expense of increased code size.
Design Decisions	
A large and varied instruction set that includes simple, fast instructions for performing basic tasks, as well as complex, multi-cycle instructions that correspond to statements in an HLL.	Simple, single-cycle instructions that perform only basic functions. Assembler instructions correspond to microcode instructions on CISC machine.
Support for HLLs is done in hardware.	All HLL support is done in software.
Memory-to-memory addressing modes.	Simple addressing modes that allow only LOAD and STORE to access memory. All operations are register-to-register.
A microcode control unit.	Direct execution control unit.
Spend fewer transistors on registers.	Spend more transistors on multiple banks of registers. Use pipelined execution to lower CPI.

A. Post-RISC architectures and the current state of the art

A group at Michigan State University's Department of Computer Science published an excellent paper called Beyond RISC - The Post-RISC Architecture [16]. In this paper, they argue that today's RISC processors have departed from their RISC roots to the point where they can no longer rightly be called RISC. (I'll be drawing on a number of ideas from this paper to make my points, so before writing me with corrections/questions/flames/etc., you should read their paper for a full explanation and defense of some of the following assertions.) The paper notes that since the first RISC designs started to appear in the 80's, transistor counts have risen and architects have taken advantage of the increased transistor resources in a number of ways.

- additional registers
- on-chip caches which are clocked as fast as the processor
- additional functional units for superscalar execution
- additional "non-RISC" (but fast) instructions
- on-chip support for floating-point operations
- increased pipeline depth
- branch prediction

To the above list, we should add

- out-of-order execution (OOO)
- on-chip support for SIMD operations.

The first two items, additional registers and on-chip caches, seem to be very much in line with traditional RISC thinking. I'll therefore focus on a few of the other items in making my case.

As you can see from the above list, post-RISC architectures take many of the RISC features as their basis (simple instructions, a large number of registers, software support for HLLs, LOAD/STORE addressing) and either expand on them or add wholly new features. The reason most post-RISC architectures still get called "RISC" is because of those features that they still share (or seem to share) with the first RISC machines. It's interesting to note that the post-RISC architectures that get called CISC are so called only because of the ISA that's visible to the programmer; implementation is ignored almost entirely in the discussion.

Now lets look in more detail at the post-RISC features that were added to RISC foundations to produce today's CPUs.

B. Superscalar execution

When the first RISC machines came out, Seymore Cray was the only one really doing superscalar execution. One could argue that since superscalar execution drastically lowers the average CPI, then it's keeping in the spirit of RISC. Indeed, superscalar execution seems to be such an essential and common component of all modern CPUs, that it doesn't quite seem fair to single it out and call it "non-RISC." After all, whether RISC or CISC, modern processors use this technique. Now, reread that last sentence, because this is precisely the point. Superscalar execution is included in today's processors not because it's part of a design philosophy called "RISC," but because it enhances performance, and performance is all that matters. Concerning all of the items in the above feature list, the Michigan group notes,

> "Thus, the current generation of high performance processors bear little resemblance to the processors which started the RISC revolution. Any instruction or feature which improves the overall price/performance ratio is considered for inclusion."

Superscalar execution has added to the complexity of today's processors--especially the ones that use scoreboarding techniques and special algorithms to dynamically schedule parallel instruction execution on the fly (which is almost all of them but the Alpha). Recall the comment from Ditzel, which I quoted at the beginning of this paper, where he identifies superscalar execution as one of the complexities that are impeding performance gains in today's RISC machines.

C. Branch prediction

Branch prediction is like superscalar execution in that it's one of those things that just wasn't around in '81. Branch prediction is a feature that adds complexity to the on-chip hardware, but it was included anyway because it has been shown to increase performance. Once again, what matters is performance and not principle.

D. Additional instructions

Many would disagree that the addition of new instructions to an ISA is a "non-RISC" tendency. "They" insist that the number of instructions were never intended to be reduced, but rather it was only the individual instructions themselves that were to be reduced in cycle time and complexity. Invariably, the folks who protest this way are Mac users who know that the G3 has more instructions than the PII, yet they still want to insist that the G3 is a pure RISC chip (because RISC = good) and the PII is a pure CISC chip (because CISC = bad). The following quote from Patterson should put this to rest once and for all:

> "A new computer design philosophy evolved: Optimizing compilers could be used to compile "normal" programming languages down to instructions that were as unencumbered as microinstructions in a large virtual address space, and to make the instruction cycle time as fast as the technology would allow. These machines would have fewer instructions—a reduced set—and the remaining instructions would generally execute in one cycle—reduced instructions—hence the name reduced instruction set computers (RISCs)." [Patterson, RISCs, p. 11]

Current RISC architectures like the G3, MIPS, SPARC, etc., have what the Michigan group calls a FISC (Fast Instruction Set Computer) ISA. Any instructions, no matter how special-purpose and rarely-used, are included if the cycle-time can be kept down. Thus the number of instructions is not reduced in modern, post-RISC machines -- only the cycle time.

E. On-chip floating-point and vector processing units

In fact, with the addition of SIMD and floating-point execution units, sometimes the cycle time isn't even really "reduced." Not only do some SIMD and FP instructions take multiple cycles to complete, but neither the on-chip SIMD unit nor the on-chip FP unit was around in the first RISC machines. Like superscalar execution, this complex functionality was added not because it fit in with some overarching RISC design principles, but because it made the machine faster. And like superscalar execution, SIMD and FP units are now common on both "RISC" and "CISC" processors. Processors with these features, especially the SIMD unit, would be better termed "post-RISC" than "RISC."

I also should mention here that the addition of FP and especially SIMD units expands the instruction set greatly. One of the reasons the G4 has such a huge number of instructions is because the SIMD unit adds a whole raft of them.

F. Out-of-order execution

OOO is one of the least RISC-like features that modern processors have; it directly contradicts the RISC philosophy of moving complexity from hardware to software. In a nutshell, a CPU with an OOO core uses hardware to profile and aggressively optimize code by rearranging instructions and executing them out of program order. This aggressive, on-the-fly optimization adds immense amounts of complexity to the architecture, increasing both pipeline depth and cycle time, and eating up transistor resources.

Not only does OOO add complexity to the CPU's hardware, but it simplifies the compiler's job. Instead of having the compiler reorder the code and check for dependencies, the CPU does it. This idea of shifting the burden of code optimization from the compiler to the hardware sounds like the exact opposite of an idea we've heard before. According to Ditzel, this is a step in the wrong direction [19].

VI. Conclusion

Let's now briefly consider the current state of the three parameters that defined the technological matrix from which RISC arose, in light of the preceding discussion of post-RISC advances.

A. Storage and Memory

Today's memory is fast and cheap; anyone who's installed a Microsoft program in recent times knows that many companies no longer consider code bloat an issue. Thus the concerns over code size that gave rise to CISC's large instruction sets have passed. Indeed, post-RISC CPUs have ever-growing instruction sets of unprecedented size and diversity, and no one thinks twice about the effect of this on memory usage. Memory usage has all but ceased to be a major issue in the designing of an ISA, and instead memory is taken for granted.

B. Compilers

Compiler research has come a long way in the past few years. In fact, it has come so far that next-generation architectures like Intel's IA-64 (which I talk about here) depend wholly on the compiler to order instructions for maximum throughput; dynamic, OOO execution is absent from the Itanium. The next generation of architectures (IA-64, Transmeta, Sun's MAJC) will borrow a lot from "very long instruction word" (VLIW) designs. VLIW got a bad wrap when it first came out, because compilers weren't up to the task of ferreting out dependencies and ordering instructions in packets for maximum ILP. Now however, it has become feasible, so it's time for a fresh dose of the same medicine that RISC dished out almost 20 years ago: move complexity from hardware to software.

C. VLSI

Transistor counts are extremely high, and they're getting even higher. The problem now is not how do we fit needed functionality on one piece of silicon, but what do we do with all these transistors. Stripping architectures down and throwing rarely-used functionality overboard is not a modern design strategy. In fact, designers are actively looking for things to integrate onto the die to make use of the wealth of transistor resources. They're asking not what they can throw out, but what they can include. Most of the post-RISC features are a direct result of the increase in transistor counts and the "throw it in if it increases performance" approach to design.

E. The guilty parties

For most of the aforementioned post-RISC transgressions, the guilty parties include such "RISC" stalwarts as the MIPS, PPC, and UltraSPARC architectures. Just as importantly, the list also includes so-called "CISC" chips like AMD's Athlon and Intel's P6. To really illustrate the point, it would be necessary to take two example architectures and compare them to see just how similar they are. A comparison of the G4 and the Athlon would be most appropriate here, because both chips contain many of the same post-RISC features. The P6 and the Athlon are particularly interesting post-RISC processors, and they deserve to be treated in more detail. (This, however, is not the place to take an in-depth look at a modern CPU. I've written a technical article on the Athlon that should serve to illustrate many of the points I've made here. I hope to start work soon on an in-depth look at the G4, comparing it throughout to the Athlon and P6.) Both the Athlon and the P6 run the CISC x86 ISA in what amounts to hardware emulation, but they translate the x86 instructions into smaller, RISC-like operations that fed into a fully post-RISC core. Their cores have a number of RISC features (LOAD/STORE memory access, pipelined execution, reduced instructions, expanded register count via register renaming), to which are added all of the post-RISC features we've discussed. The Athlon muddies the waters even further in that it uses both direct execution and a microcode engine for instruction decoding. A crucial difference between the Athlon (and P6) and the G4 is that, as already noted, the Athlon must translate x86 instructions into smaller RISC ops.

In the end, I'm not calling the Athlon or P6 "RISC," but I'm also not calling them "CISC" either. The same goes for the G3 and G4, in reverse. Indeed, in light of what we now know about the historical development of RISC and CISC, and the problems that each approach tried to solve, it should now be apparent that both terms are equally nonsensical when applied to the G3, G4, MIPS, P6, or K7. In today's technological climate, the problems are different, so the solutions are different. Current architectures are a hodge-podge of features that embody a variety of trends and design approaches, some RISC, some CISC, and some neither. In the post-RISC era, it no longer makes sense to divide the world into RISC and CISC camps. Whatever "RISC vs. CISC" debate that once went on has long been over, and what must now follow is a more subtle and far more interesting discussion that takes each platform -- hardware and software, ISA and implementation -- on its own merits.

Bibliography:

The papers cited below are all available through the ACM's Digital Library.

[1] David A. Patterson and Carlo H. Sequin. RISC I: A Reduced Instruction Set VLSI Computer. 25 years of the international symposia on Computer architecture (selected papers), 1998, Pages 216 – 230

[2] John L. Hennessy and David A. Patterson, Computer Architecture: A Quantitative Approach, Second Edition. Morgan Kaufmann Publishers, Inc. San Francisco, CA. 1996. p 9.

[3] David A. Patterson and Carlo H. Séquin. Retrospective on RISC I. 25 years of the international symposia on Computer architecture (selected papers) , 1998, p.25

[4] Hennessy and Patterson, p. 14

[5] David R. Ditzel and David A. Patterson. Retrospective on HLLCA. 25 years of the international symposia on Computer architecture (selected papers) , 1998, p.166

Bibliography: (cont.)

[6] David A. Patterson, Reduced Instruction Set Computers. Commun. ACM 28, 1 (Jan. 1985), p. 8

It was the microcode engine, first developed by IBM in the 60s, that first enabled architects to abstract the instruction set from its actual implementation. With the IBM System/360, IBM introduced a standard instruction set architecture that would work across an entire line of machines; this was quite a novel idea at the time.

[7] ibid.

[10] From a lecture by Dr. Alexander Skavantzos of LSU in a 1998 course on computer organization.

[11] Patterson, Reduced Instruction Set Computers, p. 11

[12] ibid.

[13] Patterson and Sequin, RISC I: A Reduced Instruction Set Computer. p. 217

[14] ibid. p. 218

[15] Patterson, Reduced Instruction Set Computers, p. 11

[16] I first came across this paper through a link on Paul Hseih's page. There seem to be two different versions of this paper, one later than the other. Both contain slightly different (though not contradictory) material though, so I'll be drawing on both of them in my discussion as I see fit.

[17] A question that often comes up in RISC vs. CISC discussions is, doesn't the term "RISC" properly apply only to an ISA, and not to a specific implementation of that ISA? Many will argue that it's inappropriate to call any machine a "RISC machine," because there are only RISC ISAs. I disagree. I think it should be apparent from the discussion we've had so far that RISC is about both ISA and implementation. Consider pipelining and direct execution -- don't both of these fall under the heading of "implementation"? We could easily imagine a machine who's instruction set looks just like a RISC instruction set, but that isn't pipelined and uses microcode execution. Just like with price and performance, the line between ISA and implementation isn't always clear. What should be clear though is that the RISC design philosophy encompasses both realms.

[18] Some have actually debated me on the above points, insisting that the number of instructions was never intended to be reduced. See my discussion of post-RISC architectures for a rebuttal.

[19] See the comment by Ditzel that I opened with.