What is High Performance Computing - Why CISC?*

Charles Severance
Kevin Dowd

You might ask, "If RISC is faster, why did people bother with CISC designs in the first place?" The short answer is that in the beginning, CISC *was* the right way to go; RISC wasn't always both feasible and affordable. Every kind of design incorporates trade-offs, and over time, the best systems will make them differently. In the past, the design variables favored CISC.

1 Space and Time

To start, we'll ask you how well you know the assembly language for your work- station. The answer is probably that you haven't even seen it. Why bother? Compilers and development tools are very good, and if you have a problem, you can debug it at the source level. However, 30 years ago, "respectable" programmers understood the machine's instruction set. High-level language compilers were commonly available, but they didn't generate the fastest code, and they weren't terribly thrifty with memory. When programming, you needed to save both space and time, which meant you knew how to program in assembly language. Accordingly, you could develop an opinion about the machine's instruction set. A good instruction set was both easy to use and powerful. In many ways these qualities were the same: "powerful" instructions accomplished a lot, and saved the programmer from specifying many little steps — which, in turn, made them easy to use. But they had other, less apparent (though perhaps more important) features as well: powerful instructions saved memory and time.

Back then, computers had very little storage by today's standards. An instruction that could roll all the steps of a complex operation, such as a do-loop, into single opcode[1] was a plus, because memory was precious. To put some stakes in the ground, consider the last vacuum-tube computer that IBM built, the model 704 (1956). It had hardware floating-point, including a division operation, index registers, and instructions that could operate directly on memory locations. For instance, you could add two numbers together and store the result back into memory with a single command. The Philco 2000, an early transistorized machine (1959), had an operation that could repeat a sequence of instructions until the contents of a counter was decremented to zero — very much like a do-loop. These were complex operations, even by today's standards. However, both machines had a limited amount of memory — 32-K words. The less memory your program took up, the more you had available for data, and the less likely that you would have to resort to overlaying portions of the program on top of one another.

Complex instructions saved time, too. Almost every large computer following the IBM 704 had a memory system that was slower than its central processing unit (CPU). When a single instruction can perform several

*Version 1.3: Aug 25, 2010 11:24 am -0500
†http://creativecommons.org/licenses/by/3.0/
[1]Opcode = operation code = instruction.

operations, the overall number of instructions retrieved from memory can be reduced. Minimizing the number of instructions was particularly important because, with few exceptions, the machines of the late 1950s were very sequential; not until the current instruction was completed did the computer initiate the process of going out to memory to get the next instruction.[2] By contrast, modern machines form something of a bucket brigade — passing instructions in from memory and figuring out what they do on the way — so there are fewer gaps in processing.

If the designers of early machines had had very fast and abundant instruction memory, sophisticated compilers, and the wherewithal to build the instruction "bucket brigade" — cheaply — they might have chosen to create machines with simple instruction sets. At the time, however, technological choices indicated that instructions should be powerful and thrifty with memory.

2 Beliefs About Complex Instruction Sets

So, given that the lot was cast in favor of complex instruction sets, computer architects had license to experiment with matching them to the intended purposes of the machines. For instance, the do-loop instruction on the Philco 2000 looked like a good companion for procedural languages like FORTRAN. Machine designers assumed that compiler writers could generate object programs using these powerful machine instructions, or possibly that the compiler could be eliminated, and that the machine could execute source code directly in hardware.

You can imagine how these ideas set the tone for product marketing. Up until the early 1980s, it was common practice to equate a bigger instruction set with a more powerful computer. When clock speeds were increasing by multiples, no increase in instruction set complexity could fetter a new model of computer enough so that there wasn't still a tremendous net increase in speed. CISC machines kept getting faster, in spite of the increased operation complexity.

As it turned out, assembly language programmers used the complicated machine instructions, but compilers generally did not. It was difficult enough to get a compiler to recognize when a complicated instruction could be used, but the real problem was one of optimizations: verbatim translation of source constructs isn't very efficient. An optimizing compiler works by simplifying and eliminating redundant computations. After a pass through an optimizing compiler, opportunities to use the complicated instructions tend to disappear.

[2] In 1955, IBM began constructing a machine known as Stretch. It was the first computer to process several instructions at a time in stages, so that they streamed in, rather than being fetched in a piece- meal fashion. The goal was to make it 25 times faster than the then brand-new IBM 704. It was six years before the first Stretch was delivered to Los Alamos National Laboratory. It was indeed faster, but it was expensive to build. Eight were sold for a loss of $20 million.