

RISC versus CISC

During the late 70s there was experimentation with very complex instructions, made possible by the interpreter. Designers tried to close the “semantic gap” between what machines could do and what high-level programming languages required. Hardly anyone thought about designing simpler machines, just as now not a lot of research goes into designing less powerful spreadsheets, networks, Web servers, etc. (perhaps unfortunately).

One group that bucked the trend and tried to incorporate some of Seymour Cray’s ideas in a high-performance minicomputer was led by John Cocke at IBM. This work led to an experimental minicomputer, named the **801**. Although IBM never marketed this machine and the results were not published until years later (Radin, 1982), word got out and other people began investigating similar architectures.

In 1980, a group at Berkeley led by David Patterson and Carlo Se’quin began designing VLSI CPU chips that did not use interpretation (Patterson, 1985, Patterson and Se’quin, 1982). They coined the term **RISC** for this concept and named their CPU chip the RISC I CPU, followed shortly by the RISC II. Slightly later, in 1981, across the San Francisco Bay at Stanford, John Hennessy designed and fabricated a somewhat different chip he called the **MIPS** (Hennessy, 1984). These chips evolved into commercially important products, the SPARC and the MIPS, respectively.

These new processors were significantly different than commercial processors of the day. Since they did not have to be backward compatible with existing products, their designers were free to choose new instruction sets that would maximize total system performance. While the initial emphasis was on simple instructions that could be executed quickly, it was soon realized that designing instructions that could be **issued** (started) quickly was the key to good performance. How long an instruction actually took mattered less than how many could be started per second.

At the time these simple processors were being first designed, the characteristic that caught everyone’s attention was the relatively small number of instructions available, typically around 50. This number was far smaller than the 200 to 300 on established computers such as the DEC VAX and the large IBM mainframes. In fact, the acronym RISC stands for **Reduced Instruction Set Computer**, which was contrasted with CISC, which stands for **Complex Instruction**

Set Computer (a thinly veiled reference to the VAX, which dominated university Computer Science Departments at the time). Nowadays, few people think that the size of the instruction set is a major issue, but the name stuck.

To make a long story short, a great religious war ensued, with the RISC supporters attacking the established order (VAX, Intel, large IBM mainframes). They claimed that the best way to design a computer was to have a small number of simple instructions that execute in one cycle of the data path of Fig. 2-2 by fetching two registers, combining them somehow (e.g., adding or ANDing them), and storing the result back in a register. Their argument was that even if a RISC machine takes four or five instructions to do what a CISC machine does in one instruction, if the RISC instructions are 10 times as fast (because they are not interpreted), RISC wins. It is also worth pointing out that by this time the speed of main memories had caught up to the speed of read-only control stores, so the interpretation penalty had greatly increased, strongly favoring RISC machines.

One might think that given the performance advantages of RISC technology, RISC machines (such as the Sun UltraSPARC) would have mowed down CISC machines (such as the Intel Pentium) in the marketplace. Nothing like this has happened. Why not?

First of all, there is the issue of backward compatibility and the billions of dollars companies have invested in software for the Intel line. Second, surprisingly, Intel has been able to employ the same ideas even in a CISC architecture. Starting with the 486, the Intel CPUs contain a RISC core that executes the simplest (and typically most common) instructions in a single data path cycle, while interpreting the more complicated instructions in the usual CISC way. The net result is that common instructions are fast and less common instructions are slow. While this hybrid approach is not as fast as a pure RISC design, it gives competitive overall performance while still allowing old software to run unmodified.