The RISC style of architecture

A.H.J. Sale
Electrical Engineering & Computer Science,
University of Tasmania.

In this short tutorial paper the RISC style of computer architecture is discussed. Consequences of the style are brought out, together with the reasons for its relatively recent appearance in commercially available processors.
Keywords and phrases: RISC, computer architecture.
CR categories: C-1-2, C-3.

Manuscript received February 1989, revised September 1989.

WHAT IS A RISC?

Most people who have encountered the term know that RISC stands for Reduced Instruction Set Computer. It is not difficult to deduce that RISCs, such as the Motorola 88100 and the Am29000, have fewer kinds of instructions than the ones with which they are contrasted. These are labelled Complex Instruction Set Computers (CISC). Most processors in the commercial marketplace (ranging from supercomputers through to microprocessors) are classified as CISCs, for example the Cray-1; Cyber 205; IBM S/38 and 9000 series; Digital PDP-11 and VAX ranges; the Motorola 68020, Intel 30386, Zilog Z-80 and related processors. What is less well-known is why fewer kinds of instructions should now be considered good, and what the consequences of RISC architecture are.

First consider the previously dominant architectural style: CISC. As CISCs evolved, new instructions and addressing modes were added to architectures to support commonly required high-level languages. The aim was to reduce the gap between the hardware and its system software by using increasing transistor counts on a chip to build more sophisticated processors. The conventional wisdom was that by migrating frequently used operations into hardware, programs become faster to execute. Readers may recall advertisements for microprocessor chips where the number of different instructions on the advertised processor is held out as a figure of merit.

Several computer designers began to challenge this approach during the 1980s. In this they were influenced by the increasing difficulty of designing correct chips, and the longer and longer lead times to bring a new processor to market. The reducing size of transistors and thus increasing transistor count in VLSI chips caused rapidly escalating design effect, and the consequence of lower and lower confidence in a complex design being correct first-time (or ever). But to what functions should the silicon area be devoted, if not to supporting complex instruction sets? With this question, RISC architecture really begins.

The question highlights an important architectural point: each included feature carries with it a cost as well as a benefit. For example, an instruction set which caters for different instruction execution times must have a more complex clock, and this may result in a slight slowing down of the basic clock frequency, as well as consuming some extra chip area. Similar considerations apply to instruction sets whose instructions are of different lengths. Note that these are some of the costs; the benefits of the two features mentioned are to let simple instructions execute faster than the slowest instruction, and to economize

on program size.

RISC architecture emphasizes identifying the most common features and implementing them very fast, so that we may end up with a faster computer than its competitors. Each operation which is a candidate for inclusion has to be examined to see if its inclusion will increase or decrease the overall execution speed. For example an early RISC machine [Katevenis, 1985] incorporated a barrel shifter in its data path. Subsequent evaluation suggested that while this speeded up a few operations, the overall slowing down of all other machine cycles (even by as little as 5%) far outweighed the gain.

This re-examination of operations eventually results in a small number of instructions (say 16–32), all of the same size, which control a very simple and very fast data-path and ALU. All instructions execute in the same time and operate on registers, so the control logic is simple. Complex operations are achieved by executing a sequence of simpler operations, so that RISC machines execute more instructions to achieve the same result as a CISC machine.

One problem that has been glossed over is the matching of the raw speed of RISC instruction execution (say 20–50 ns) with that of the external memory (80–100 ns). This first shows up in retrieving data values from memory. A RISC instruction may initiate a memory transfer such as a *read*, but stalling its completion until the transfer is complete will waste possibly useful machine cycles. Faced with this mismatch, conventional responses fall into three categories:

- Put up with the wasted time.
- Insert a cache between the processor and the memory so that the average cache access time is well-matched to the processor cycle time.
- Initiate a memory transfer and continue instruction execution, providing hardware to stall any instruction which attempts to access a register which is the destination of a transfer until the data arrives.

RISC architecture rejects all three approaches. A typical RISC machine will initiate the transfer and continue execution, but there will be no hardware interlock. The compiler is responsible for generating code which ensures that the destination register of a *read* is not accessed until the value has arrived. To do this, firstly it must know the timing characteristics of the processor and the memory, and secondly it must be able to insert useful work in the gap between the initiation and completion of a memory operation. There is a transfer of responsibility from the hardware to the software.

Further, RISC architectures tend to have many registers (say 128–512). Such a large register set can minimize memory accesses through two mechanisms.

- Data does not have to be moved into or out of registers simply to free up some temporary space.
- Many objects can be allocated register space rather than memory space for the whole of their lifetimes, for example temporary results, stack frames, local and global variables, parameter pointers.

Again the compiler has the responsibility for allocating these objects. A large register set is an ideal candidate to fill up silicon area with useful hardware since it is regular and easily designed.

CONSEQUENCES

The major consequences of RISC architecture are easy to deduce.

- Code size is probably larger than in a comparable CISC.
- Program execution is probably faster than a comparable CISC.
- Compilers for RISCs are more complex and incorporate sophisticated optimization schemes.
- Assembly language programmers are unlikely to produce code which is as good as or better than that produced by compilers, except for trivial programs.
- Object code may not be portable across models or even configurations of the same model since it may include timing dependencies.

Two other consequences are less obvious. The first is a direct attack on the 'range' concept, at least at the object code level. As technology changes, a different instruction set architecture may become optimal. Compatibility between RISCs must be sought at a higher level. One possibility is to provide only high-level language (HLL) compatibility, but a more usual approach is to provide compatibility at the level of an intermediate level language (ILL) into which all HLL programs are translated. A machine-specific optimizer and code-generator then translates the ILL form into object code for a specific machine and configuration. The optimizer becomes a very important intellectual property for the manufacturer, since the performance of the machine depends so much on it.

Secondly, a RISC machine requires to fetch more instructions than a CISC machine, and at a faster rate. While it is possible to incorporate an off-chip cache, or to build an on-chip instruction-only cache, it is simpler to initiate the fetching of instructions before they are needed. This is easy for sequentially allocated instructions, but fails to cope with branching. On identifying a branch to be taken, the designer could flush all the pre-fetched instructions,

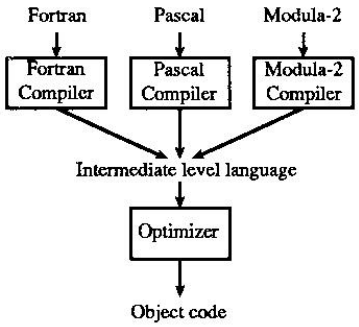

Figure 1 — Compilation for a RISC

and initiate a new instruction fetch at the new address, but this would waste possibly useful cycles. As you expect by now, a RISC continues to execute the fetched instructions with the branch being delayed by the depth of the prefetch pipeline (say 2–5 instructions). Again, the optimizer is given the task of finding useful work that can be done between the branch instruction and its delayed effect.

α–1	MOVE	R7, (R91)
α	BRANCH	B
α+1	ADD	=1, R5
α+2	ADD	=4, R91
B	...	

Figure 2 — Execution trace with prefetch pipeline

SUMMARY

Possession of a small instruction set does not constitute a RISC architecture, despite possible sales claims. The RISC style involves a fundamental rethink of many assumptions and every RISC machine may not have all the features described here. It is the architectural approach rather than any specific feature that constitutes RISC architecture.

Probably the two most consistent themes are the migration of some hardware functions to compilers, and the ruthless excising of any function that does not pay its way. As yet, it is not possible to judge whether RISCs will supplant CISCs, or whether the two will continue to co-exist in particular niches.

FURTHER READING

GREY, G. (1988): The 88000 faces of Multibus II, ESD: The Electronic Systems Design Magazine, September 1988, pp. 45–50.

JOHNSON, T. (1988): Raising the stakes with optimizing compilers. ESD: The Electronic Systems Design Magazine, September 1988, pp. 64–66.

KATEVENIS, M.G.H. (1985): Reduced instruction set architectures for VLSI, MIT Press.

PATTERSON, D.A. & SÉQUIN, C.H. (1982): A VLSI RISC, IEEE Computer, Vol. 15, pp. 8–21.

RADIN, J. (1983): The 801 minicomputer, IBM Journal of Research & Development, Vol. 27, pp. 237–246.

SALE, A.H.J. (1989): The Architecture of the PCM–1. *The Australian Computer Journal*, to appear in Vol. 21, No. 2.

BIOGRAPHICAL NOTE

Arthur Sale is head of the Department of Electrical Engineering & Computer Science at the University of Tasmania. He was appointed to the University in 1974 as Foundation Professor of Information Science. His research interests cover programming methodology, programming languages, silicon chip design, and processor architecture; currently he is involved in the design and implementation of innovative processors for executing functional languages.