# Reduced Instruction Set Computers

## 1 Historical Trends

'Traditional' CPU design has tended to utilise advances in chip fabrication technologies to add every more features to CPUs. A major motivating factor here has been the so-called 'software crisis' in which the complexity of software systems tended to grow much faster than our ability to manage that complexity. This quickly led to developers abandoning assembler programming in favour of high-level languages such as C, in which programming is easier and more productive. CPU designers responded by incorporating 'support' for high-level languages – more specifically, their compilers – and instruction sets became large and complex[1].

Around 1980 researchers started looking at the way high-level languages mapped onto CPUs by carrying-out dynamic analyses of running programs. The main findings of this research were that:

- The most common instructions were register-to-register moves
- The most frequently accessed data were scalars
- The *Principle of Locality* holds for data[2].
- Procedure calls are costly operations due to the need for extensive stack operations which involve external memory[3].

These observations gave rise to the *reduced instruction set computer* (RISC) philosophy which dominated the debate around CPU design – often in a very heated manner – in the 1980s and early 1990s. Since many of the more exotic, complex instructions were rarely if ever used, the idea behind the RISC philosophy was to improve CPU performance by tailoring the architecture to what the CPU actually did when it was running real programs. In particular:

**Provide Lots of Registers**

Providing a lot of registers in the CPU directly supports the most common instruction (register-to-register moves) and allows more intermediate results to be held in the CPU rather than written to memory (which is more time-consuming than register access.)

Coupled with larger numbers of registers, high-level language compilers should be optimised to try to keep scalars (the most commonly accessed type of data) in registers. The Principle of Locality implies they will be needed again soon.

Large numbers of registers make it possible to call procedures by passing parameters in registers rather than being pushed/popped onto a stack in external memory. Thus the cost of procedure calling can be reduced.

In fact, RISC machines have been designed with up to 528 registers although somewhere in the range 32-128 is more common.

**Reducing the Number of Instructions**

To distinguish them from RISC processors 'traditional' machines with large instruction sets are usually termed *complex instruction set computers* (CISC), a term which was only invented as part of the RISC debate

[1] An alternative analysis is that because memory was very expensive in the early days of computing, CPU designers tried to pack as much as possible into the available memory by devising ever more complex instructions. Except this only conserves program memory, not data memory. I am not wholly convinced by this point of view. Possibly a more credible view can be illustrated by the way new versions of computer programs 'bloat' by adding more features which people hardly ever use – probably CPU designers added extra instructions because they could and they thought they might be useful...

[2] The Principle of Locality – which is really an observation, not a proper principle – states that if a program accesses a particular memory location, it will tend to access a nearby location fairly soon afterwards.

[3] Principally, because the procedure parameters and results were passed via the stack.

to allow RISC proponents to identify the "enemy"!

Many of the instructions in CISC machines are rarely used but contribute significant complexity to the control path. Removing these rarely-used instructions will make the control path simpler and therefore it may be able to run at a higher clock frequency. Also, the silicon area freed-up could be used for additional registers. Therefore the RISC philosophy aims to use a small but carefully optimised instruction set.

In particular, you will not find indirect addressing modes in RISC machines as the basic idea is to complete each data operation quickly.

**Optimised Instruction Pipeline**

Part of the original RISC philosophy was to pre-fetch instructions and queue them in a pipeline. If the instruction width is fixed – as opposed to comprising an op code followed by optional parameters – this makes the design of the pipeline more straightforward. Therefore, fixed instruction widths tend to be seen as part of the RISC philosophy although this is probably an implementation detail rather than a central part of the RISC idea.

# 2 CISC vs RISC

Are RISC processors better than CISC? This is a hard question to answer definitively since processors are not produced in both RISC and CISC versions which can be compared side-by-side. In addition, differences in compiler construction further cloud the issue.

What can be said is that RISC ideas have been very influential on CPU design. The Intel Pentium – which is regarded by many as the last significant CISC processor in the marketplace – has in fact 'borrowed' many RISC ideas such as pipelining. Although the Intel Pentium still dominates desktop computing, it is probably fair to say that the majority of embedded microprocessors (*e.g.* in mobile phones, *etc.*) are RISC processors. The humble PIC microcontroller is a RISC processor, for example[4]. Arguably, this may be connected to the fact that because RISC processors are simpler, they are quicker and therefore cheaper to design; the *cost* of embedded processors is often a more important factor than raw performance so a cheaper processor may have a big competitive advantage. That said, many observers argue that there has been a tendency in recent years for the size of the instruction sets in RISC processors to increase.

# 3 Are RISC Programs Shorter?

One common piece of false logic is to assume that because the instruction sets of RISC machines are smaller, then a program implemented on a RISC will be shorter than the same program implemented on a CISC machine. In fact, the exact very opposite is true! The RISC program would comprise maybe (many) more instructions than its CISC equivalent. One way of looking at this would be that the RISC instruction set provides a less rich way of coding something and so the RISC implementation would take a larger number of (simpler) instructions to accomplish the same task.

So would the shorter CISC program run faster? No, not necessarily! In the RISC, each (simple) instruction takes only a few clock cycles whereas in a CISC, the more 'sophisticated' instructions may take many clock cycles. So the longer (*i.e.* having more instructions) RISC program may actually run faster than the corresponding CISC program which contains fewer instructions.

[4]The PIC microcontroller family is also unusual in conforming to the Harvard architecture model. But this is coincidental...