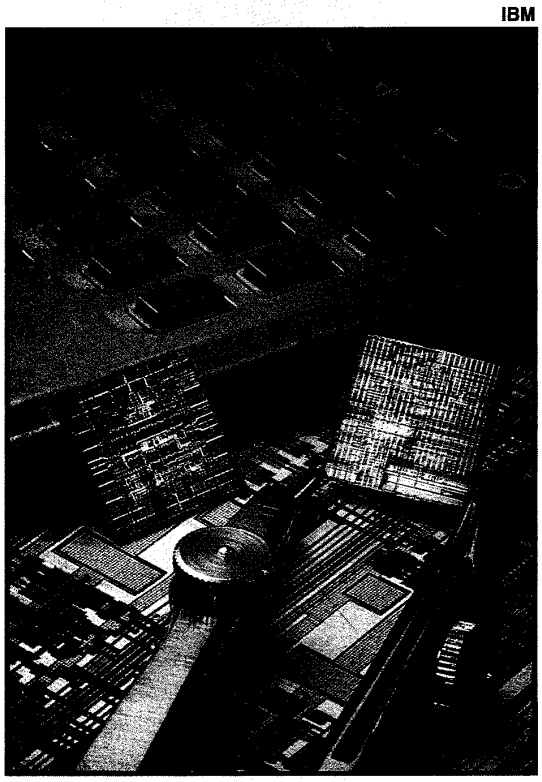
RISC

(Reduced Instruction Set Computers)

Paul Chow

IBM



IBM RISC System/6000* VLSI CMOS logic chips

Reduced Instruction Set Computers or RISC. Those of you who follow the marketing propaganda about the latest microprocessors are probably very familiar with the term. Steven Przybylski, a designer of the MIPS RISC processor at Stanford University, now defines a RISC as "any computer announced after 1985."

The original term, RISC, came from the RISC project at Berkeley, led by Professor David Patterson. The first processor, RISC I, was designed in 1982 and was followed by the RISC II processor in 1984. During the same period, Professor John Hennessy at Stanford University was leading the MIPS project. Although these projects are probably the most well known, there was prior work done at IBM, Yorktown Heights, on a machine known as the 801 Minicomputer. These projects resulted in machines with smaller numbers of instructions than the popular architectures of that time, thus the name. However, this was really a result of the design methodology used, rather than a necessary and sufficient feature of the architecture.

The RISC approach

The design of a RISC processor begins with the definition of the computer system being designed. For most RISC machines today, this means a system running the *UNIX** operating system with programs written in high-level languages such as *C*, *Fortran*, or *Pascal*. The compilers generate the instructions that are executed by the hardware in the CPU. The operating system manages the hardware resources of the system. To build an efficient system, the designers must pay close attention to the compiler/hardware interface, and the operating system/hardware interfaces. For this discussion, let us focus on the compiler/hardware interface.

Early programmers used assembly language because of limited memory and inefficient compiler technology. With VLSI technology, limited memory is essentially gone, and compilers are good enough so assembly language is not needed. We measure the performance of a computing system by how long it takes to execute a program. Define the execution time as:

$$\text{Execution Time} = \text{Path Length (Instructions)} \times \frac{\text{Cycles}}{\text{Instruction}} \times \text{Clock Period} \left(\frac{\text{seconds}}{\text{cycle}}\right)$$

By examining this equation we can look for ways to reduce the total execution time.

Complex instruction set computers (CISC) attempt to reduce the semantic gap between high-level languages and the machine instructions implemented in the hardware. This is done by using instructions that can support many of the constructs found in high-level languages. For example, accessing an element in an array requires the computation of a memory offset from the starting address of the array in memory. The size of this offset is dependent on the size of the array elements and the array index. Complex addressing modes (ways to specify memory addresses) make it possible to access the elements of an array using a single operand specifier. For example, consider the high-level language statement

ARRAYX(50) = 0

where ARRAYX is an array of integers starting at address 1000. An example CISC instruction might be:

clear.word 1000 (r2)

where Register r2 contains the index, 50 in this case. Assume each array element is a word that requires four memory locations, and the starting address is 1000. To compute the operand address, this instruction first multiplies the contents of the register by four, and adds 1000 to it. This philosophy attempts to reduce execution time by minimizing the path length, or number of instructions executed.

In the RISC philosophy of design, a more quantitative approach is taken. Since compilers are generating the machine code, we examine the kinds of

* UNIX is a trademark of AT&T

00278-6648/91/0010-0028 $1.00

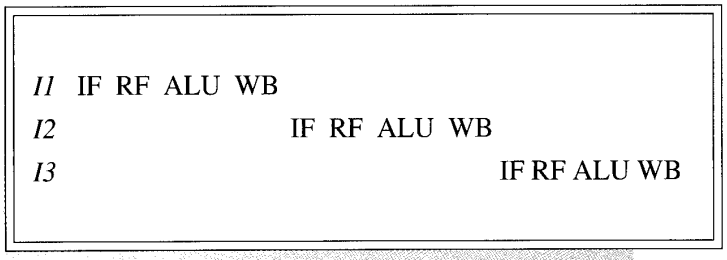


Fig. 1: Timing for a nonpipelined CPU

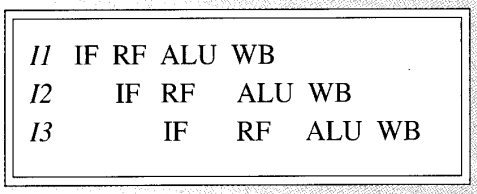


Fig. 2 Timing for a pipelined processor

```
I1          beq  r1,r2,equal   ; branch to equal if r1 = r2
I2          add  r3,r4,r5      ; r5← r3 + r4
                 .
                 .
                 .
In  equal   add  r5,r5,r7      ; r7← r5 + r5
```

Fig. 3 Using a branch instruction

```
                 .                              .
                 .                              .
equal   add      r7,r1,r1                       add    r7,r1,r1
        add      r1,r2,r3              equal    add    r1,r2,r3
        beq      r3,r5,equal                    beq    r3,r5,equal
        nop                                     add    r7,r1,r1
                 .                              .
                 .                              .
                 .                              .
        Program 1                      Program 2
   4 cycles per iteration         3 cycles per iteration
```

Fig. 4 Using the branch delay slot

instructions actually generated by a compiler. This is done by collecting statistics showing the frequency of instruction usage as programs execute. Data from programs compiled for machines, such as the VAX, show that most instructions are simple. If we examine the equation for Execution Time, a different tradeoff from the CISC approach can be seen. Instead of focusing on minimizing the path length, we reduce the average number of cycles per instruction. This can be achieved by implementing just the simple instructions in hardware. An added benefit is that simpler hardware is required so a shorter clock period is feasible. However, using only simpler instructions means the path length will increase; because complex operations that could have been done in one instruction, now require several instructions. This effect is small compared to the reduction in average instruction cycles and cycle time.

Pipelining

Several features that characterize most RISC processors result from using pipelining. Although CISC machines are often pipelined, the instruction set of a RISC processor has been designed specifically to take advantage of pipelining.

Pipelining is a technique for increasing the effective throughput of a system. Consider the steps that must be performed to execute a typical add instruction. Assume that the operands are available in the CPU registers. The instruction must be fetched and decoded (**IF**); the operands must be fetched from the registers (**RF**); the addition is performed in the ALU (**ALU**), and then the result must be written back to the registers (**WB**). If each step requires one clock cycle, then it will take four cycles to execute this instruction. Figure 1 shows the timing for a sequence of these instructions. Each instruction completes before the next one begins so the instructions are issued at a rate of one every four clock cycles.

Figure 2 shows how a pipelined version of this processor would work. The hardware is designed so that when I1 has completed the **IF** step, then the **IF** for I2 can begin. Similarly, all other steps in the instructions can be sequenced in the same way. In this scheme, the instruction throughput is one instruction per cycle, assuming no pipeline hazards.

A pipeline hazard occurs when an instruction needs the result of a previous instruction not yet completed. An important hazard has to do with branch instructions. A branch instruction is a decision point in the program and generally means that the flow of the program can take one of two possible paths. Figure 3 shows how a branch instruction (*beq*) might be used. A simple implementation of a branch instruction will read two register values and check whether they are equal. If they are equal, then the branch is taken. In Figure 3, we would like a taken branch to continue execution at instruction *In*, whereas a non-taken branch should fall through to instruction I2. Assume in Figure 2 that Instruction I1 is a branch instruction. At the end of **RF**, the branch instruction has determined whether the branch is to be taken. However, by this time I2 has already been fetched and the **IF** in I3 is the first one that occurs after the **RF** in I1. In this example, the hazard occurs because the branch decision is made too late to affect the **IF** in I2.

There are two solutions. One is to always wait for the branch instruction to determine if a branch is going to be taken before fetching the next instruction. This means that all branch instructions will effectively take two cycles to execute. The other technique is to use a delayed branch, which allows the instruction following the branch to always execute, independent of the outcome of the branch instruction. This instruction is said to be in the branch

delay slot. The simplest way to fill the delay slot is to put in an instruction that does nothing (*nop*). If this is done, the *nop* does not affect the program and the correct sequence of instructions is always executed. If we want to be more clever, we can try to use the instruction in the delay slot to do something useful. This requires a more sophisticated compiler to predict what instructions can go into the slot. If the compiler guesses correctly then the cycle is not wasted. If the compiler guesses wrong, the instruction should be one that does not affect the program's outcome. Figure 4 shows an example of how a branch delay slot can be used. Program 1 takes four cycles per loop iteration because of the *nop* that is used in the branch delay slot. Program 2 rearranges the code slightly to use the branch delay slot, and requires only three cycles per iteration. It also assumes that the contents of Register R1 are not used after the loop is exited since the value will be different than in Program 1.

Simple instruction encoding

Pipelining and the desire to reduce cycle time, makes it important that instructions can be decoded quickly. This leads to a few other characteristics.

RISC processors have fixed-length instructions. In CISC machines, instructions can vary in length from a fraction of a memory word to several memory words. All instructions in a RISC are the same length, usually one memory word. This makes the decoding logic simpler, because it does not have to figure out how much has to be fetched from memory before the actual decoding can begin.

RISC processors have a simple instruction encoding. For example, each instruction needs to specify the registers that are to be used. If the register specifiers always appear in the same location in the instruction encoding, they can be used immediately without the instruction decoder needing to figure out that one instruction type specifies registers in bits 0 to 14, and another instruction type specifies registers in bits 10 to 24. A simple encoding means less hardware is needed for decoding.

A symmetric use of the registers is also characteristic of RISC instruction sets. This means that all registers can be used by any instruction. This makes the compilation process easier; because it does not have to keep track of what registers can be used in each instruction. It also makes decoding easier.

Memory instructions

An important aspect of any computer architecture is how memory is accessed. This is reflected in the types of instructions and the addressing modes. A CISC processor generally has very sophisticated addressing modes and instructions that can read operands from memory, do an operation, and store the result back to memory. These are called memory-to-memory instructions. For a high-performance CPU, operations that access operands in memory always take longer to execute than operands found in the CPU registers. This is because of the electrical properties of accessing memory compared with accessing the registers. That's why modern compilers attempt to keep variables in the CPU registers as much as possible. This also means memory-to-memory instructions become less useful.

RISC processors use a load-store, or register-register, architecture. The only instructions that access memory are load and store instructions. Load instructions read a value from memory into a register and store instructions, take a value in a register and write it to memory. All instructions that operate on data, read the operands from the registers and store the results into registers.

The addressing modes of RISC machines are also very simple. The simple modes fit well in the pipelined model where time is limited to compute the address of an operand in memory. Complex addressing modes are replaced with a sequence of instructions. Analysis of compiler output also supports this design decision.

Optimizing compilers

When examining the components of a RISC system, don't forget software. The design of a RISC system must involve the design of the hardware and the software at the same time. For example, deciding if an instruction should be implemented in the CPU hardware must be based on whether the overall system performance is improved. An add instruction is important because it is frequently used and difficult to do without having the hardware. On the other hand, an instruction such as divide is not used frequently, and requires significant hardware to implement. This makes it a good candidate for being left out of the hardware, and the equivalent function being done in software.

The analysis of the hardware/software tradeoffs must also consider the compiler technology available, and how this technology can effectively use the hardware. For example in pipelining, we described how the slot in a delayed branch can be used more effectively by reordering instructions. If the compiler technology for determining how to fill this slot did not exist, a different method for implementing branches in the hardware might have been developed.

We mentioned how it was desirable to keep operands in registers as much as possible. To make this possible, RISC architectures usually have a larger number of general purpose registers, usually 32 or more, compared with CISC architectures, which often have less. Manipulating operands in registers means the instructions should also help. RISC architectures use a three-address architecture, as shown in the add instructions used in Figure 3. In this architecture, the two source operands, and the register for the result can all be different registers. This makes it easier for the compiler to use the registers efficiently.

RISC architectures rely heavily on the compilers to achieve their best performance. These compilers are generally known as optimizing compilers. Most of the optimizations used can be found in any textbook on compiler design. The key difference in a RISC system is that the hardware has been designed so that these optimizations can be used effectively.

Implementation

The key point here is that the only features implemented in hardware are those contributing significant performance benefits. This is very important because adding hardware will often impact the cycle time of the processor. For example, assume an instruction implemented in hardware reduces the number of instructions executed by 1%, but causes the cycle time to increase by 4%. The net effect of this is that the execution time will increase by a factor of $0.99 \times 1.04 = 1.03$ or 3%!

Also, using only simple instructions in the hardware leaves room for other features. On a VLSI chip, which is the main implementation technology, this room can be used to add features such as an on-chip instruction cache, or float-

ing-point hardware.

Performance

We will now demonstrate with numbers why RISC processors are able to outperform CISC processors of the same generation. Consider the three factors in the equation for Execution Time.

Path length. A RISC architecture will need to execute more instructions than a CISC architecture to get the same job done. Studies have shown this is about a factor of 1.2 more instructions.

Cycles/instruction. The goal of a RISC architecture is to execute one instruction every cycle. Due to effects like not filling all of the branch-delay slots, and cache misses, the average is slightly greater than one Cycle/instruction, and closer to 1.3. In a CISC processor, this number can be quite large, and varies across all of the architectures. For this example, assume it to be four Cycles/instruction.

Clock period. This is a very tricky number to estimate because it is also technology dependent. What we are looking for is how much the architecture affects the clock period. By looking at processors currently available on the market, we can estimate that a RISC machine could run about 1.5 times faster than a CISC machine in the same technology.

Using these numbers to compute the ratio of execution times, we can see that the execution time on a RISC processor is about

$$\frac{1}{1.2} \times \frac{4}{1.3} \times 1.5 = 3.8$$

times better than on a CISC processor. This suggests that a RISC processor has a performance advantage of about a factor of four over a CISC processor purely from using a better architecture. This number will vary depending on the exact processor models used.

Potential disadvantages

There are some potential disadvantages of RISC architectures compared to CISC architectures. Some are true problems; some will also appear in CISC machines as they strive to increase performance.

More instructions. This is manifested in two ways. First, there is the static size of a program, which is the amount of memory and disk space needed to hold the program. The increase for RISC processors can be in the range of 40-50%, putting a greater demand on the instruction memory and secondary storage. Secondly, there is the increased dynamic size, which is the actual number of instructions executed. However, this can be compensated by other factors.

High memory bandwidth. Memory bandwidth is the measure of how fast data can be transferred between the memory and the CPU. This is a major bottleneck in most architectures, and having to execute more instructions means that the average banDwidth must be higher for a RISC processor. However, the complexity of a memory system is determined by the peak bandwidth needed. To achieve its highest performance, the program on a CISC processor will use mostly simple load, store, ALU, and branch instructions, just as in a RISC machine. These instructions put the highest demand on the memory system and thus determine the peak bandwidth needed. The peak bandwidth needed by a CISC processor will be comparable to that of a RISC machine.

Complex software. RISC machines may require more complex software because implementation details such as pipeline hazards must be handled. However, the use of simpler instructions actually makes code generation easier since there are less choices to make. It is also easier for an optimizer to find and remove redundant operations. Many optimizations used in RISC compilers can also be used in compilers for CISC architectures. The difference is that the RISC architecture has been engineered so these optimizations are used more effectively.

Future trends

Current projections suggest that RISC architectures will continue to double their performance every year or so for several years. The performance of current CISC architectures will also improve as they use lessons learned with RISC machines. For example, the Motorola 68040 and the Intel 486 both attempt to execute the common instructions in a single cycle. However, improvements in these architectures will be constrained by the desire to maintain machine code compatibility between generations.

The next step in RISC architectures will be to break the one instruction per cycle barrier. This will be done by executing more than one instruction per cycle. Analysis of code from compilers and existing compiler technology has shown that this is feasible. Some recent commercial architectures already have this feature (Intel i960 and i860, and the IBM RS/6000). These are called superscalar architectures.

Summary

To describe a reduced instruction set computer as one that uses a small number of simple instructions really misses the significance of this design methodology. The important lesson is that it is possible to measure and evaluate the performance of a computer system, and use these numbers in making design decisions. The design of a computer system is a quantitative science, not an art.

Read more about it

- Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley Publishing Company, 1986.
- John Hennessy, Norman Jouppi, Forest Baskett, Thomas Gross, and John Gill. Hardware/Software Tradeoffs for Increased Performance. In *Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 2-11, Palo Alto, March 1982. SIGARCH/SIGPLAN
- John Hennessy, Norman Jouppi, John Gill, Forest Baskett, Alex Strong, Thomas Gross, Chris Rowen, and Judson Leonard. The MIPS Machine. In *Compcon Spring*, pages 2-7. IEEE, 1982.
- John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufman Publishers, Inc. 1990.
- David A. Patterson. Reduced Instruction Set Computers. *Communications of the ACM*, 28(1):8-21, January 1985.
- David A. Patterson and Carlo H. Sequin. A VLSI RISC. *IEEE Computer*, 15(9):8-212, September 1982.
- G. Radin. The 801 minicomputer. In *Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 39-47, Palo Alto, March 1982. SIGARCH/SIGOPLAN.
- Cheryl A. Wiecek. A Case Study of VAX-11 Instruction Set Usage for Compiler Execution. In *Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 177-184, Palo Alto, March 1982. SIGARCH/SIGPLAN.

About the author

Paul Chow received the B.A.Sc. degree with honours in Engineering Science, and the M.A.Sc. and Ph.D. degrees in electrical engineering from the University of Toronto, Toronto, Ont. in 1977, 1979 and 1984, respectively. He is a member of the IEEE and ACM.

In 1984 he joined the Computer Systems Laboratory at Standford University, CA, as a Postdoctoral Fellow and later as a Research Associate. He was a major contributor to the MIPS-X RISC microprocessor project. Since January 1988, he has been an Assistant Professor in the Department of Electrical Engineering at the University of Toronto. ■