CISC vs. RISC: Bridging the Processor Architecture-Software Gap

Ferenc Vajda
KFKI Research Institute for Measurementand Computing Techniques
of the Hungarian Academy of Sciences
Budapest, Hungary

Abstract

This paper was used in a series of two lectures given at the CERN School of Computing at Sopron, Hungary in September 1994. It deals with the basic definitions and roots of RISC and CISC machines, problems of the superscaler and pipeline design approaches especially branch techniques. The paper discusses the requirements and problems associated with cache memories, software support and performance evaluation. Examples of current commercial RISC architectures are also given.

1. Definitions and roots

The *RISC* = Reduced Instruction Set Computer is a recent trend in computer design. The RISC is a design philosophy that attempts to simplify the design of processors by implementing a relatively small number of instructions. These instructions are designed to execute in a few machine cycles. Usually the execution logic for instructions is hardwired rather than microcoded. To simplify the decoding logic for instructions, fixedlenght and fixed format instructions are used. It uses high-speed on-chip registers (in typical design 32 or more) and allows only load and store instructions to access main memory (or more likely, cache memory). All other operations work on data held in registers. This allows more data to be held on-chip for longer and assists the compiler with allocating registers to variable. RISC processors also assist to compiler in holding data values by operating a policy of putting the results of arithmetic operations on values in two registers into a third. RISC architectures depend crucially on optimizing compilers [12] to produce efficient software.

An opposite of a RISC is a *CISC* = Complex Instruction Set Computer. A typical "classical" example of CISCs is the Digital VAX 11/780 computer [1]. It is a rather complex system with all the main features of a CISC. For convenience, the architecture can be devided into two major areas, i.e. application programmer (or user) and system programmer aspects. The most important attributes of the VAX family as parts of the user architecture are: large number of instructions, very many and complex addressing modes including immediate, register, register deferred, autoincrement, autodecrement, autoincrement deferred, absolute, displacement, displacement deferred, indexed. The *VAX* has an uncountable number of instruction formats. It supports a considerable number of data types including five integer and four floating point types, packed decimal, numeric and character strings, variable-length bit field, queue. An operand specifier many be as short as one byte and as long as ten bytes. There may be 0,1 and up to six operand specifiers. As parts of the system programmer architecture the VAX system have many features which support or directly implement High Level Language *(HLL)* or Operating System *(OS)* functions or primitives such as procedure management, array operations and index testing, information typing and protection, memory management, etc. By privileged instructions it supports virtual addressing implementing a so called process and a system space. It has sophisticated exception and interrupt handling methods and direct support for process control. The VAX instruction set includes machine language instructions which are either identical or very similar to some HLL instructions.

There is considerable debate about the relative merits of RISC vs. CISC. Many RISC design features are already incorporated into up-to-date CISC processors such as the Intel *Pentium* microprocessor architecture [2].

There are technologies to *enhance* (both RISC and CISC) processor performance:

- *Pipelining* which is a technique that allows a processor to execute one instruction per cycle even though it may take longer than one cycle for any individual instruction to complete. The pipeline breaks an instruction into part called stages which require one clock cycle (or less) to complete.
- *Superpipelining* which is a technique in which instructions are sent in rapid succession through the pipeline. The processor starts handling the next instruction before the previous instruction is fully executed.
- *Superscalar* organization which enables the processor to execute more than one instruction at the same time in parallel using parallel functional units.

It is used to say that there is no new thing under the sun. The main ideas of CISC i.e. supporting/implementing high level primites on the level of the traditional architecture (microcode) is originated from the technique of *vertical migration.* RISC instruction set structure is a "reincarnation" of the *vertical microinstructions* (Similarly *horizontal microinstruction* principle is also "reincarnated" as *VLIW* = Very Long Instruction Word computer). Basic ideas of techniques as pipeling, superpipeling and superscaler organization have been derived from *parallel and super computers* [3].

2. Instruction execution pipelines

A typical CISC *integer pipeline* [2] has the following stages:

- *Prefetch* (The CPU prefetchs code from the instruction cache)
- *First decode* (The CPU decoders the instruction to generate a control word)
- *Second decode* (The CPU decodes the control words for use in the execute stage and generates addresses for data references)
- *Execute* (The CPU either accesses the data cache or calculates results in the ALU)
- *Write back* (The CPU updates registers and flags with instruction results. All exceptions must be resolved before)

The superscalar organization enables two instructions to execute parallel in two independent integer ALU using two independent pipelines.

The *floating-point pipeline* [2] consists of eight stages (the first two stages are processed by the integer-common pipeline resources):

- *Prefetch*
- *First decode*
- *Second decode*
- *Operand fetch*
- *First execute*
- *Second execute*
- *Write float*
- *Error reporting*

This pipeline allows single cycle throughput for floating point add, subtract, multiply and compare.

Pipeline hazards interrupt smooth pipeline flow, causing stalls, slips or exceptions. In *stall* cycles, the pipeline does not advances. For *slips* (e.g. load interlocks [14]) only a subset of the stages advances. When the slip condition is resolved, the instructions in the pipeline resume from whatever stage they are in. For *exceptions,* the processor suspends the normal sequence of instruction execution and transfers control to an exception handler.

3. Problems of multiple-execution unit (superscalar) and pipeling design approach

The two different design philosophies used to exploit *concurrency* are also called as *spatial parallelism* and *temporal parallelism* (pipelining). To use the concurrency techniques to execute a program the dependencies between instructions must be taken into account. It is called a hazard that presents an instruction from issuing. There are three kinds of *hazards* (or dependencies):

- A necessary *resource* is not available or is busy. It is called *structural hazard* or resource dependency. Structural hazards occur because of pipeline resource or instruction class

conflict. A typical pipeline conflict is referring to the same register. It can be omitted by multiported register files. Instruction class conflict occur when two instructions requiring the same functional unit on the same clock cycle. It can be partly resolved by duplicating the most important functional units (e.g. integer ALU [4]) or by the logic in the instruction issue algorithm [2].

- Instruction issue can also stall because of *data hazards* or with other words *data-dependency.* Given any two instructions the following *dependencies* may exist between them:
  - *RAW:* Read After Write
  - *WAR:* Write After Read
  - *WAW:* Write After Write

The WAW and the WAR dependencies can be considered as false ones since they involve only a *register name conflict.* The logic of the instruction sequencer can ensure that the source and destination register of the instruction issued in one functional unit pipe differ from the destination register of the instruction issued to the other functional unit pipe. This arrangement eliminates the RAW and WAW hazards [2]. *Register reservation* can be handled by the *scoreboard* method. At register-busy state the sequencer automatically interlocks the pipeline against incorrect data on hazards [4].

- A *control dependency* (or control hazard) occurs when the result of one instruction determines whether another instruction will be executed (Jump, branch). Handling the branches is so important that we discuss it later separately.

The different *techniques for eliminating* or reducing the effect of hazards on the performance can be divided into:

- *static* techniques that are used at *compilation time* and
- *dynamic* (hardware) techniques based on special parts of the architecture used to detect and resolve hazards at *execution time.*

4. Branch techniques

To execute a branch, the processor must perform two *operations:*

- *compute* the target address
- *evaluate* the condition (except at unconditional branches) to determine whether the branch is to be taken.

Depending on the way the branch condition is *evaluated* there are architectures:

- *with* condition codes
- *without* condition codes.

In the first case the branch condition is stored in a special purpose (condition code) store while in the second case it is stored in any general purpose register [5]. In the later case the branch instruction itself specifies a certain operation (usually a comparison). This type of instruction is called *compare-and-branch.*

Branch hazards (caused by RAW dependencies) my produce a substantial reduction in a pipelined processor performance. Therefore several mechanisms are used to reduce the "cost" of branches [8].

At *delayed branch* the effect of a branch is delayed by n instructions. By this way there is no direct dependency between the branch and the next instruction. The number of delay slots after branch is determined by the position of the stages that evaluate the condition and compute the target address in the pipeline. This slots are filled with NOP instructions or the compiler can schedule some useful computation (optimized delayed branches). This technique used e.g. at the MIPS [14] processors.

The fact that the instructions in the delay slots are always executed (regardless of the outcome of the branch) restricts compiler scheduling. At an improved scheme called *delayed branch with squashing,* the branch instruction can cancel (nullify) the instructions in the delay slots. For example at the SPARC and at the 88100 architectures [4] one bit in the branch instruction specifies "always execute" or "cancel if the branch is not taken" delay slots. At the Precision Architecture [6] the nullification depends on both the branch outcome and the target address. At forward branch it is cancelled if the branch is taken while at backward branch it is cancelled if the branch is not taken (The first usually corresponds to an *IF-THEN-ELSE* structure while the second is usually used as a loop closing branch).

As a complementary to the delayed branch there is a technique called *early computation*

of branches. Another way to advance the target address computation is by mean of a special store *(branch target buffer)* which can be accessed parallel with the fetch of the branch instruction. An alternative use of a buffer is to store some first instruction from the taken part. This buffer is called *branch target instruction memory* [4].

Another way of anticipating the result of a branch is by predicting it. The technique (using *static* prediction made by the compiler or *dynamic* one carried out by the architecture based on the previous execution of the branch) is called *branch prediction.* Based on some assumption at branch, the processor proceeds to conditionally execute the instruction from the taken path or which follows in sequence.

An alternative that eliminates the need of predict is to execute *concurrently* both paths and when the outcome of the branch condition is available, discard one of them. This technique is called *multiple prefetch.* This implies that some part of the architecture of the processor must be duplicated. This technique can be combined with the prediction technique. In this case one path is only fetched while the other one is executed. Using this technique (always executing the instructions that follow in sequence)the branch "cost" is one cycle if not taken and three cycles if taken [2]. At the IBM RS/6000 processor [7] the instruction unit detect branches in advance by means of prefetching. The compiler tries to separate the compare and branch instructions and is able to execute the branch with *zero cost* regardless of its outcome [9].

5. Caches

Cache memories are major contributions to the performance of the RISC (and CISC) processors. The most important cache parameters are the cache size (considering on-chip cache memories and at two-level cache technique on-chip and off-chip caches), the *block (line) size* and *associativity* (degree of associativity or set size). The later determines the *mapping* from main memory to the cache location in frame of the *placement* policy. The n-way set-associative policy is a good compromise between the *direct* and *fully associative* mapping [3].

If a line in the cache is *replaced,* information may be lost if the line is modified. Therefore the architecture should take care to ensure data *coherence* between the cache and main memory. The two most important policies to solve this problem are the *write-through* policy (where write operations always transmit changes to both the cache and main memory) and the *copy-back* policy (which only transmits a cache line to memory if it has to be replaced in the cache). The later policy has two version, i.e. *always* copy-back and *flagged* copy back (using a "dirty" flag to indicate modification).

For example the PowerPC 601 processor provides a unified, eight-way set-associative cache [10]. Each line holds two 32-byte sectors. Cache *indexing* and cache *tag comparisons* use the physical (real) address. In general, the cache make use of a copy-back policy but allows page- and block-level write-through. To improve the cache bandwidth the processor employs several techniques including 8 words wide fetch unit and memory queue, complete real-modify-write every cycle, to prevent cache arbitration from stalling the execution units operate queues for functional units and for storage, etc.

The 88110 processor has a Harward-style internal architecture [4] i.e. *separate* and *independent* instruction and data paths. The on-chip instruction cache feeds the instruction unit (which dispatches two instructions each clock cycle into an array of ten concurrent execution units) and the data cache feeds the load/store unit. The write-back data cache follows a four state (MESI = Modified, Exclusive, Shared, Invalid) *coherence protocol* (for multiprocessor systems). The caches are fed by two independent fully associative (40 entry) address *TLBs* (Translation Look-aside Buffers) supporting a *demand paged* virtual memory environment. A second level cache can be implemented at the processor level using a *dedicated* cache controller.

6. Current commercial RISC architectures

The most important current RISC architectures are as follows (Apologies are extended to any company whose name and product has been inadvertently and unintentionally omitted):

Trade name	Manufacturer	Types
PA-RISC: Precision Architecture	Hewlett Packard	PA 7100
Alpha AXP	Digital Equipment	DECchip 21064
SPARC: Sun Microsystem's Scalable Processor Architecture	Sun	SPARC, MicroSPARC, SuperSPARC
POWER: Performance Optimization With Enhanced RISC	IBM	RS/6000
Power PC	IBM, Apple, Motorola	PC 601, PC 603, PC 604, PC G20
Clipper	Intergraph Corp.	C 400
MIPS	Mips Computer Systems/Division of Silicon Graphics	R 4000, R 4400
88000	Motorola	88100/200, 88110
TRON	Hitachi	Gmicro/500
Transputer	INMOS Limited	T9000

7. Software support for RISC processors

For fast execution on a RISC processor, instructions should be arranged in an order that uses the arithmetic units as efficiently as possible. It means that the role of the compiler and the operating system in managing the hardware architecture is increased. One of the key issues is *scheduling* [11] i.e. arrange the instructions in an order in which they execute the fastest (with minimum delays or hold-offs). It processes the output of the optimizer (code motion, commoning, strength reduction, etc. are done).

Optimizing compilers (e.g. [12]) have an important contribution to application performance on RISC processors. There are different *transformation* phases used e.g. loop invariant code motion, register promotion, peephole optimization, graph colouring register allocation, branch optimization, etc. Loop scheduling can be improved using the software pipelining technique [13] by generating codes that overlaps instructions to exploit the available instruction-level parallelism.

Because much of the architecture design of the RISC processors was influenced by the system's software architecture the new features of the hardware architecture could be used by the kernel of the operating systems realizing an *integrated system approach.* The details of these topics are beyond the scope of this paper.

8. Computer performance

Performance is a key criterion in the design, procurement and use of computer systems. As such, the goal of computer systems engineers, scientists, analysts and users is to get the highest performance for a given cost.

The key measure is the performance as seen by the user.. The *end-user performance* (with the simplifying assumption of infinite main memory and ignoring the effects of I/O and paging) [15]:

$$T_E = L_p * t_{cyc} * CPI$$

where:

T_E is the execute time

L_p is the path length (The number of machine instructions required to execute a given program)

t_{cyc} is the cycle time (e.g. for the Alpha processor t_{cyc} = 6.25 ns)

CPI: Cycles Per Instruction (The average number of cycles required to execute an instruction)

The three most important techniques for *performance evaluation* are
- analytical modelling
- simulation and
- measurement

Computer system performance measurements involve monitoring the system while is being subjected to a particular *workload*. The process of performance comparison for two or more systems by measurement is called *benchmarking*. The workloads used in the measurements are called benchmarks. They are well-known *benchmarks* used for different applications as Sieve, Ackermann, Whetstone, Dhrystone etc. There is a new trend to develop *standard benchmarks* for comparing computer systems for various engineering and scientific applications including networks, image processing systems and databases. Typical examples for standardized set of benchmarks are the different releases of the *SPEC benchmarks* (SPEC: System Performance Evaluation Cooperative is a nonprofit cooporation formed by leading computer vendors).

9. References

1. VAX Architecture Handbook, Digital Equipment Corporation, 1981.
2. D. Alpert and D. Avnon: Architecture of the Pentium Microprocessor, IEEE Micro, June 1993, 11-21.
3. K. Hwang and F.A. Briggs: Computer Architecture and Parallel Processing, McGraw-Hill Book Company, 1984.
4. K. Diefendorff and M. Allen: Organization of the Motorola 88110 Superscalar RISC Processor, IEEE Micro, April 1992, 40-63.
5. R.L. Sites: Alpha AXP Architecture, Digital Technical Journal, Special Issue 1992, 19-29.
6. T. Asprey et al.: Performance Features of the PA7100 Microprocessor, IEEE Micro, June 1993, 22-35.
7. R.R. Oeler and R.D. Groves: IBM RISC System/6000 processor architecture, IBM Journal of Research and Development, January 1990, 23-36.
8. A.M. Gonzalez: A survey of branch techniques in pipelined processors, Microprocessing and Microprogramming, 36(1993), 243-257.
9. K. Uchiyama et. al.: The Gmicro/500 Superscalar Microprocessor with Branch Buffers, IEEE Micro, October 1993, 12-22.
10. M.C. Becker et. al.: The PowerPC 601 Microprocessor, IEEE Micro, October 1993, 54-68.
11. M.S. Warren, Jr.: Instruction Scheduling for the IBM RISC System/6000 processor, IBM Journal of Research and Development, January 1990, 85-92.
12. R.C. Hansen: New Optimizations for PA-RISC Compilers, Hewlett-Packard Journal, June 1992, 15-23.
13. S. Ramakrishnan: Software Pipelining in PA-RISC Compilers, Hewlett-Packard Journal, June 1992, 39-45.
14. S. Mirapuri et. al.: The Mips 4000 Processor, IEEE Micro, April 1992, 10-22.
15. S.W. White et. al.: How Does Processor MHz Relate to End-User Performance?, Part.1.: IEEE Micro, August 1993, 8-16, Part.2.: IEEE Micro, October 1993, 79-89.