REDUCED INSTRUCTION SET COMPUTING

ARCHITECTURE

The term *computer architecture* was first defined in the article by Amdahl, Blaauw, and Brooks of International Business Machines (IBM) Corporation announcing the IBM System/

360 computer family on April 7, 1964 (1,2). On that day, IBM Corporation introduced, in the words of an IBM spokesperson, "the most important product announcement that this corporation has made in its history."

Computer architecture was defined as the attributes of a computer seen by the machine language programmer as described in the *Principles of Operation.* IBM referred to the Principles of Operation as a definition of the machine that enables the machine language programmer to write functionally correct, time-independent programs that would run across a number of implementations of that particular architecture.

The architecture specification covers all functions of the machine that are observable by the program (3). On the other hand, Principles of Operation are used to define the functions that the implementation should provide. In order to be functionally correct, it is necessary that the implementation conforms to the Principles of Operation.

The Principles of Operation document defines computer architecture, which includes:

- Instruction set
- Instruction format
- Operation codes
- Addressing modes
- All registers and memory locations that may be directly manipulated or tested by a machine language program
- Formats for data representation

Machine Implementation was defined as the actual system organization and hardware structure encompassing the major functional units, data paths, and control.

Machine Realization includes issues such as logic technology, packaging, and interconnections.

Separation of the machine architecture from implementation enabled several embodiments of the same architecture to be built. Operational evidence proved that architecture and implementation could be separated and that one need not imply the other. This separation made it possible to transfer programs routinely from one model to another and expect them to produce the same result which defined the notion of *architectural compatibility.* Implementation of the whole line of computers according to a common architecture requires unusual attention to details and some new procedures which are described in the Architecture Control Procedure. The design and control of system architecture is an ongoing process whose objective is to remove ambiguities in the definition of the architecture and, in some cases, adjust the functions provided (1,3,4).

RISC Architecture

A special place in computer architecture is given to RISC. RISC architecture has been developed as a result of the 801 project which started in 1975 at the IBM Thomas J. Watson Research Center and was completed by the early 1980s (5). This project was not widely known to the world outside of IBM, and two other projects with similar objectives started in the early 1980s at the University of California Berkeley and Stanford University (6,7). The term RISC (reduced instruction set computing), used for the Berkeley research project, is the term under which this architecture became widely known and recognized today.

Development of RISC architecture started as a rather "fresh look at existing ideas" (5,8,9) after revealing evidence that surfaced as a result of examination of how the instructions are actually used in the real programs. This evidence came from the analysis of the *trace tapes,* a collection of millions of the instructions that were executed in the machine running a collection of representative programs (10). It showed that for 90% of the time only about 10 instructions from the instruction repertoire were actually used. Then the obvious question was asked: "why not favor implementation of those selected instructions so that they execute in a short cycle and emulate the rest of the instructions?" The following reasoning was used: "If the presence of a more complex set adds just one logic level to a 10 level basic machine cycle, the CPU has been slowed down by 10%. The frequency and performance improvement of the complex functions must first overcome this 10% degradation and then justify the additional cost" (5). Therefore, RISC architecture starts with a small set of the most frequently used instructions which determines the pipeline structure of the machine enabling fast execution of those instructions in one cycle. If addition of a new complex instruction increases the "critical path" (typically 12 to 18 gate levels) for one gate level, then the new instruction should contribute at least 6% to 8% to the overall performance of the machine.

One cycle per instruction is achieved by exploitation of parallelism through the use of pipelining. It is *parallelism through pipelining* that is the single most important characteristic of RISC architecture from which all the remaining features of the RISC architecture are derived. Basically we can characterize RISC as a *performance-oriented architecture based on exploitation of parallelism through pipelining.*

RISC architecture has proven itself, and several mainstream architectures today are of the RISC type. Those include SPARC (used by Sun Microsystems workstations, an outgrowth of Berkeley RISC), MIPS (an outgrowth of Stanford MIPS project, used by Silicon Graphics), and a superscalar implementation of RISC architecture, IBM RS/6000 (also known as PowerPC architecture).

RISC Performance

Since the beginning, the quest for higher performance has been present in the development of every computer model and architecture. This has been the driving force behind the introduction of every new architecture or system organization. There are several ways to achieve performance: technology advances, better machine organization, better architecture, and also the optimization and improvements in compiler technology. By technology, machine performance can be enhanced only in proportion to the amount of technology improvements; this is, more or less, available to everyone. It is in the machine organization and the machine architecture where the skills and experience of computer design are shown. RISC deals with these two levels—more precisely their interaction and trade-offs.

The work that each instruction of the RISC machine performs is simple and straightforward. Thus, the time required to execute each instruction can be shortened and the number of cycles reduced. Typically the instruction execution time is

divided into five stages, namely, machine cycles; and as soon as processing of one stage is finished, the machine proceeds with executing the second stage. However, when the stage becomes free it is used to execute the same operation that belongs to the next instruction. The operation of the instructions is performed in a pipeline fashion, similar to the assembly line in the factory process. Typically, those five pipeline stages are as follows:

IF: Instruction Fetch
ID: Instruction Decode
EX: Execute
MA: Memory Access
WB: Write Back

By overlapping the execution of several instructions in a pipeline fashion (as shown in Fig. 1), RISC achieves its inherent execution parallelism which is responsible for the performance advantage over the complex instruction set architectures (CISC).

The goal of RISC is to achieve an execution rate of one cycle per instruction (CPI = 1.0), which would be the case when no interruptions in the pipeline occurs. However, this is not the case.

The instructions and the addressing modes in RISC architecture are carefully selected and tailored upon the most frequently used instructions, in a way that will result in a most efficient execution of the RISC pipeline.

The simplicity of the RISC instruction set is traded for more parallelism in execution. On average, a code written for RISC will consist of more instructions than the one written for CISC. The typical trade-off that exists between RISC and CISC can be expressed in the total time required to execute a certain task:

$$\text{Time (task)} = I \times C \times P \times T_0$$

where

I = number of instructions/task
C = number of cycles/instruction
P = number of clock periods/cycle (usually $P = 1$)
T_0 = clock period (ns)

While CISC instruction will typically have less instructions for the same task, the execution of its complex operations will require more cycles and more clock ticks within the cycle as compared to RISC (11). On the other hand, RISC requires more instructions for the same task. However, RISC executes its instructions at the rate of one instruction per cycle, and its machine cycle requires only one clock tick (typically). In addition, given the simplicity of the instruction set, as reflected in simpler machine implementation, the clock period T_0 in RISC can be shorter, allowing the RISC machine to run at the higher speed as compared to CISC. Typically, as of today, RISC machines have been running at the frequency reaching 1 GHz, while CISC is hardly at the 500 MHz clock rate.

The trade-off between RISC and CISC can be summarized as follows:

1. CISC achieves its performance advantage by denser program consisting of a fewer number of powerful instructions.
2. RISC achieves its performance advantage by having simpler instructions resulting in simpler and therefore faster implementation allowing more parallelism and running at higher speed.

RISC MACHINE IMPLEMENTATION

The main feature of RISC is the architectural support for the exploitation of parallelism on the instruction level. Therefore all distinguished features of RISC architecture should be considered in light of their support for the RISC pipeline. In addition to that, RISC takes advantage of the principle of locality: *spatial* and *temporal*. What that means is that the data that was used recently is more likely to be used again. This justifies the implementation of a relatively large general-purpose register file found in RISC machines as opposed to CISC. Spatial locality means that the data most likely to be referenced is in the neighborhood of a location that has been referenced.

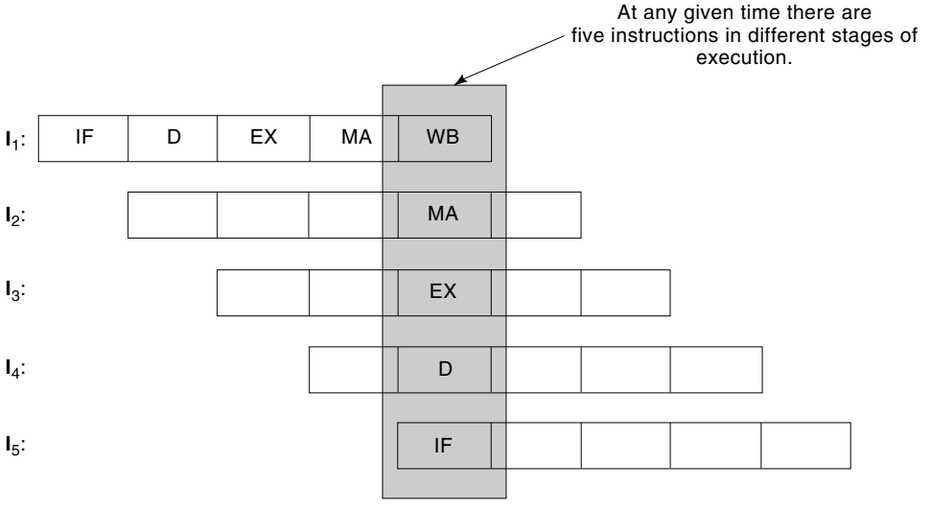

Figure 1. Typical five-stage RISC pipeline.

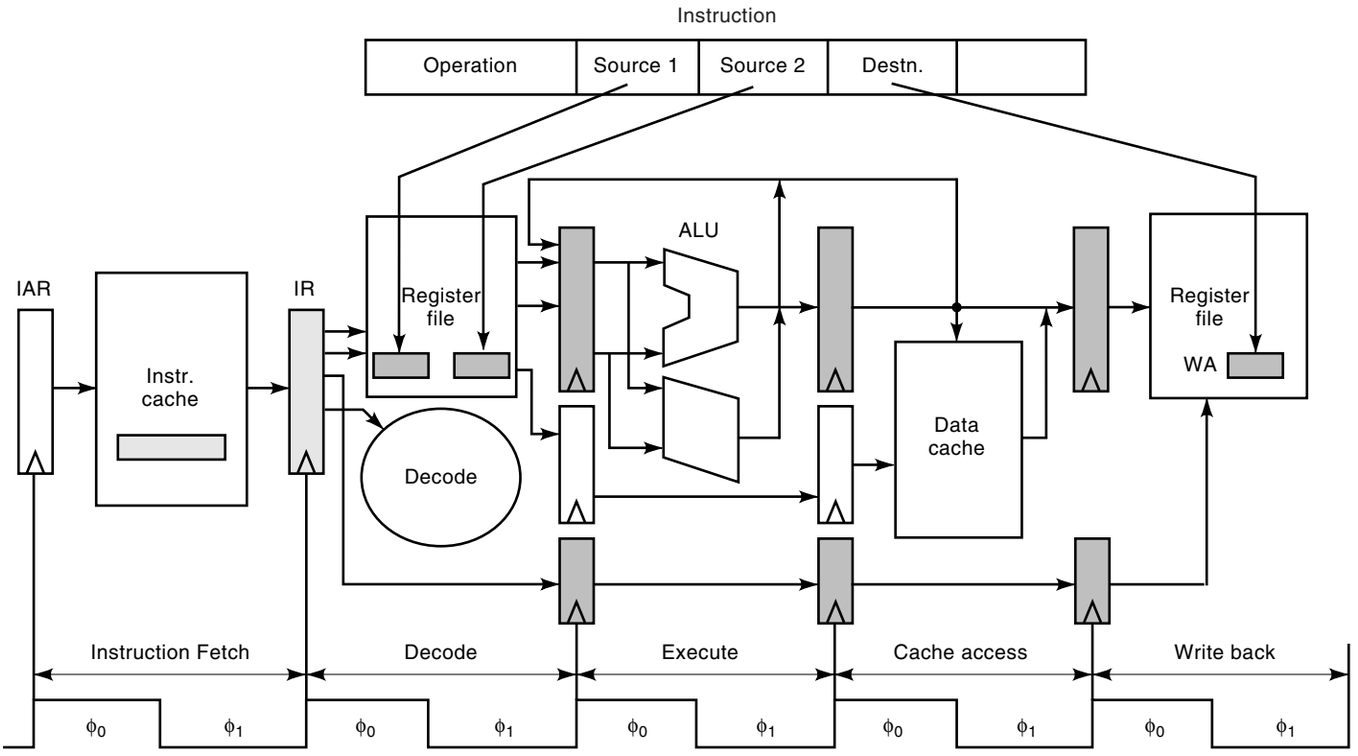

Figure 2. Pipeline flow of a Register-to-Register operation.

It is not explicitly stated, but that implies the use of caches in RISC.

Load/Store Architecture

Often, RISC is referred to as Load/Store architecture. Alternatively the operations in its instruction set are defined as Register-to-Register operations. The reason is that all the RISC machine operations are between the operands that reside in the General Purpose Register File (GPR). The result of the operation is also written back to GPR. When restricting the locations of the operands to the GPR only, we allow for determinism in the RISC operation. In the other words, a potentially multicycle and unpredictable access to memory has been separated from the operation. Once the operands are available in the GPR, the operation can proceed in a deterministic fashion. It is almost certain that once commenced, the operation will be completed in the number of cycled determined by the pipeline depth and the result will be written back into the GPR. Of course, there are possible conflicts for the operands which can, nevertheless, be easily handled in hardware. The execution flow in the pipeline for a Register-to-Register operation is shown in Fig. 2.

Memory Access is accomplished through Load and Store instructions only; thus the term *Load/Store Architecture* is often used when referring to RISC. The RISC pipeline is specified in a way in which it must accommodate both operation and memory access with equal efficiency. The various pipeline stages of the Load and Store operations in RISC are shown in Fig. 3.

Carefully Selected Set of Instructions

The principle of locality is applied throughout RISC. The fact that only a small set of instructions is most frequently used, was used in determining the most efficient pipeline organization with a goal of exploiting instruction level parallelism in the most efficient way. The pipeline is "tailored" for the most frequently used instructions. Such derived pipelines must serve efficiently the three main instruction classes:

- Access to Cache: Load/Store
- Operation: Arithmetic/Logical
- Branch

Given the simplicity of the pipeline, the control part of RISC is implemented in hardware—unlike its CISC counterpart, which relies heavily on the use of microcoding.

However, this is the most misunderstood part of RISC architecture which has even resulted in the inappropriate name: RISC. Reduced instruction set computing implies that the number of instructions in RISC is small. This has created a widespread misunderstanding that the main feature characterizing RISC is a small instruction set. This is not true. The number of instructions in the instruction set of RISC can be substantial. This number of RISC instructions can grow until the complexity of the control logic begins to impose an increase in the clock period. In practice, this point is far beyond the number of instructions commonly used. Therefore we have reached a possibly paradoxical situation, namely, that several of representative RISC machines known today have an instruction set larger than that of CISC.

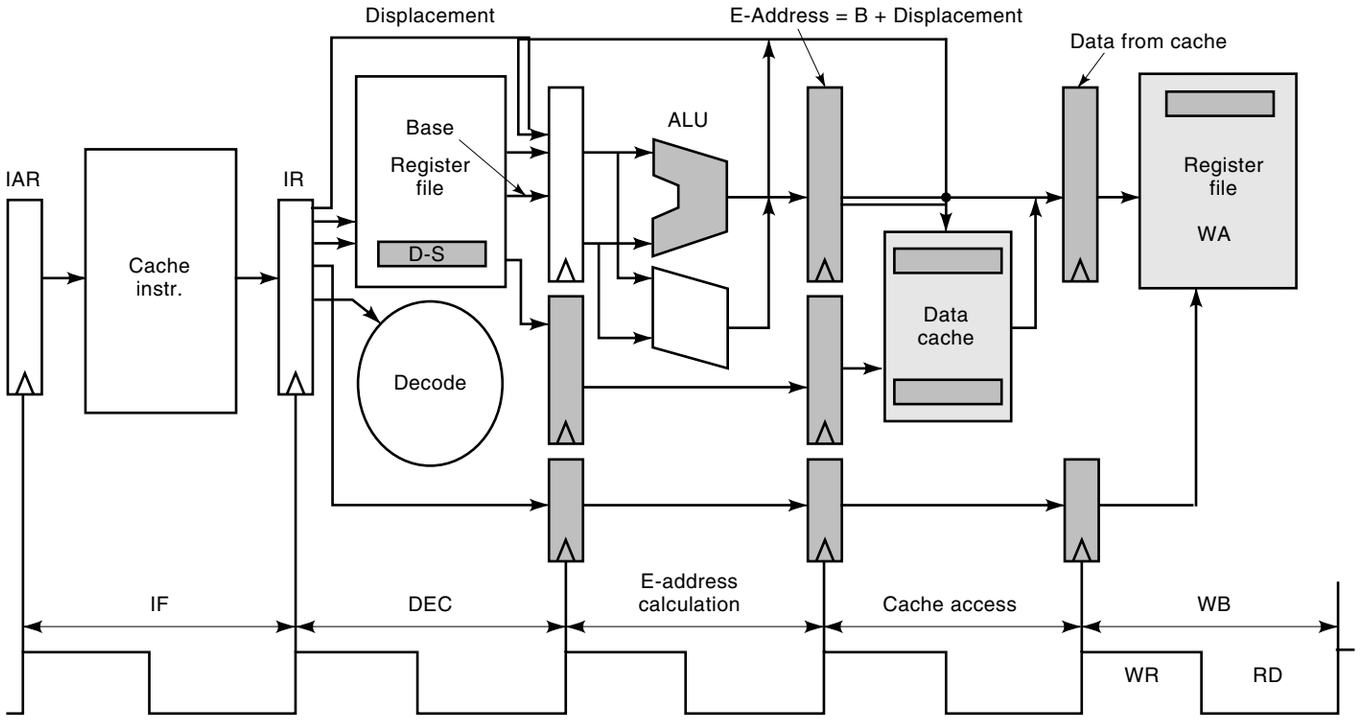

Figure 3. The operation of Load/Store pipeline.

For example: IBM PC-RT Instruction architecture contains 118 instructions, while IBM RS/6000 (PowerPC) contains 184 instructions. This should be contrasted to the IBM System/360 containing 143 instructions and to the IBM System/370 containing 208. The first two are representatives of RISC architecture, while the latter two are not.

Fixed Format Instructions

What really matters for RISC is that the instructions have a fixed and predetermined format which facilitates decoding in one cycle and simplifies the control hardware. Usually the size of RISC instructions is also fixed to the size of the word (32 bits); however, there are cases where RISC can contain two sizes of instructions, namely, 32 bits and 16 bits. Next is the case of the IBM ROMP processor used in the first commercial RISC IBM PC/RT. The fixed format feature is very important because RISC must decode its instruction in one cycle. It is also very valuable for superscalar implementations (12). Fixed size instructions allow the Instruction Fetch Unit to be efficiently pipelined (by being able to determine the next instruction address without decoding the current one). This guarantees only single I-TLB access per instruction.

One-cycle decode is especially important so that the outcome of the Branch instruction can be determined in one cycle in which the new target instruction address will be issued as well. The operation associated with detecting and processing a Branch instruction during the Decode cycle is illustrated in Fig. 4. In order to minimize the number of lost cycles, Branch instructions need to be resolved, as well, during the Decode stage. This requires a separate address adder as well as comparator, both of which are used in the Instruction Decode Unit. In the best case, one cycle must be lost when Branch instruction is encountered.

Simple Addressing Modes

Simple Addressing Modes are the requirements of the pipeline. That is, in order to be able to perform the address calculation in the same predetermined number of pipeline cycles in the pipeline, the address computation needs to conform to the other modes of computation. It is a fortunate fact that in real programs the requirements for the address computations favors three relatively simple addressing modes:

1. Immediate
2. Base + Displacement
3. Base + Index

Those three addressing modes take approximately over 80% of all the addressing modes according to Ref. 3: (1) 30% to 40%, (2) 40% to 50%, and (3) 10% to 20%. The process of calculating the operand address associated with Load and Store instructions is shown in Fig. 3.

Separate Instruction and Data Caches

One of the often overlooked but essential characteristics of RISC machines is the existence of cache memory. The second most important characteristic of RISC (after pipelining) is its use of the locality principle. The locality principle is established on the observation that, on average, the program spends 90% of the time in the 10% of the code. The instruction selection criteria in RISC is also based on that very same observation that 10% of the instructions are responsible for 90%

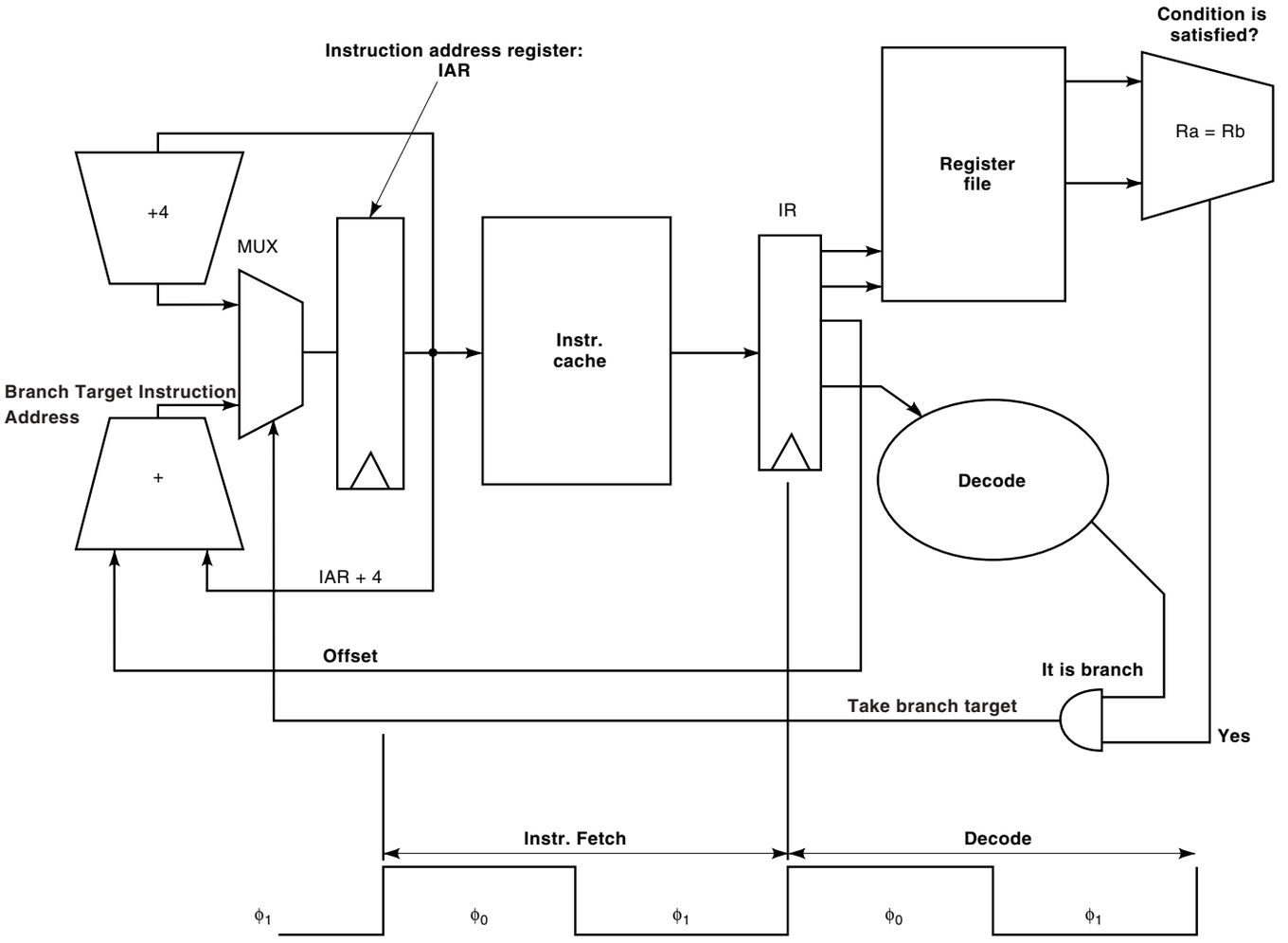

Figure 4. Branch instruction.

of the code. Often the principle of the locality is referred to as a 90–10 rule (13).

In case of the cache, this locality can be spatial and temporal. *Spatial locality* means that the most likely location in the memory to be referenced next will be the location in the neighborhood of the location that was just referenced previously. On the other hand, *temporal locality* means that the most likely location to be referenced next will be from the set of memory locations that were referenced just recently. The cache operates on this principle.

The RISC machines are based on the exploitation of that principle as well. The first level in the memory hierarchy is the general-purpose register file GPR, where we expect to find the operands most of the time. Otherwise the Register-to-Register operation feature would not be very effective. However, if the operands are not to be found in the GPR, the time to fetch the operands should not be excessive. This requires the existence of a fast memory next to the CPU—the Cache. The cache access should also be fast so that the time allocated for Memory Access in the pipeline is not exceeded. One-cycle cache is a requirement for RISC machine, and the performance is seriously degraded if the cache access requires two or more CPU cycles. In order to maintain the required one-cycle cache bandwidth the data and instruction access should not collide. It is from there that the separation of instruction and data caches, the so-called Harvard architecture, is a must feature for RISC.

Branch and Execute Instruction

Branch and Execute or Delayed Branch instruction is a new feature of the instruction architecture that was introduced and fully exploited in RISC. When a Branch instruction is encountered in the pipeline, one cycle will be inevitably lost. This is illustrated in Fig. 5.

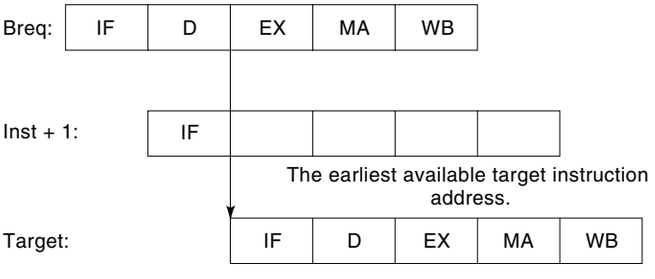

Figure 5. Pipeline flow of the Branch instruction.

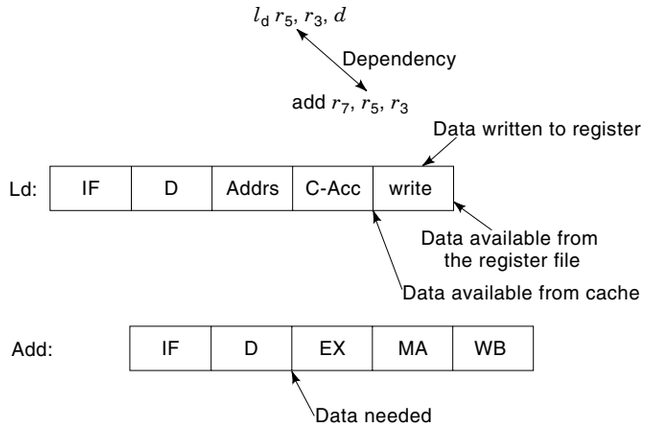

Figure 6. Lost cycle during the execution of the load instruction.

RISC architecture solves the lost cycle problem by introducing Branch and Execute instruction (5,9) (also known as Delayed Branch instruction), which consists of an instruction pair: *Branch* and the *Branch Subject* instruction which is always executed. It is the task of the compiler to find an instruction which can be placed in that otherwise wasted pipeline cycle.

The subject instruction can be found in the instruction stream preceding the Branch instruction, in the target instruction stream, or in the fall-through instruction stream. It is the task of the compiler to find such an instruction and to fill-in this execution cycle (14).

Given the frequency of the Branch instructions, which varies from 1 out of 5 to 1 out of 15 (depending on the nature of the code), the number of those otherwise lost cycles can be substantial. Fortunately a good compiler can fill-in 70% of those cycles which amounts to an up to 15% performance improvement (13). This is the single most performance contributing instruction from the RISC instruction architecture.

However, in the later generations of superscalar RISC machines (which execute more than one instruction in the pipeline cycle), the *Branch and Execute* instructions have been abandoned in favor of *Brand Prediction* (12,15).

The Load instruction can also exhibit this lost pipeline cycle as shown in Fig. 6.

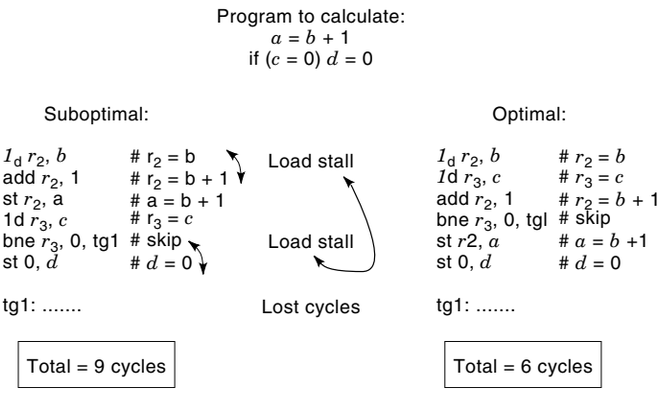

Figure 7. An example of instruction scheduling by compiler.

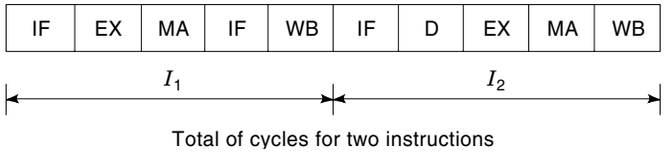

Figure 8. Instruction execution in the absence of pipelining.

The same principle of scheduling an independent instruction in the otherwise lost cycle, which was applied for in Branch and Execute, can be applied to the Load instruction. This is also known as delayed load.

An example of what the compiler can do to schedule instructions and utilize those otherwise lost cycles is shown in Fig. 7 (13,14).

Optimizing Compiler

A close coupling of the compiler and the architecture is one of the key and essential features in RISC that was used in order to maximally exploit the parallelism introduced by pipelining. The original intent of the RISC architecture was to create a machine that is only *visible through the compiler* (5,9). All the programming was to be done in High-Level Language and only a minimal portion in Assembler. The notion of the "Opti-

Table 1. Features of RISC Architecture

Feature	Characteristic
Load/store architecture	All operations are Register to Register, so *Operation* is decoupled from *access to memory*
Carefully selected subset of instructions	Control implemented in hardware (no microcoding in RISC); set of instructions not necessarily small[a]
Simple addressing modes	Only most frequently used addressing modes used; important to fit into existing pipeline
Fixed size and fixed field instructions	Necessary to decode instruction and access operands in one cycle (there are, however, architectures using two sizes for instruction format (IBM PC-RT))
Delayed branch instruction (known also as *Branch* and *Execute*)	Most important performance improvement through instruction architecture (no longer true in new designs)
One instruction per cycle execution rate (CPI = 1.0)	Possible only through use of pipelining
Optimizing compiler	Close coupling between architecture and compiler (compiler *knows* about pipeline)
Harvard architecture	Separation of *Instruction* and *Data Cache* resulting in increased memory bandwidth

[a] IBM PC-RT Instruction architecture contains 118 instructions, while IBM RS/6000 (PowerPC) contains 184 instructions. This should be contrasted to the IBM System/360 containing 143 instructions and IBM System/370 containing 208. The first two are representatives of RISC architecture; the latter two are not.

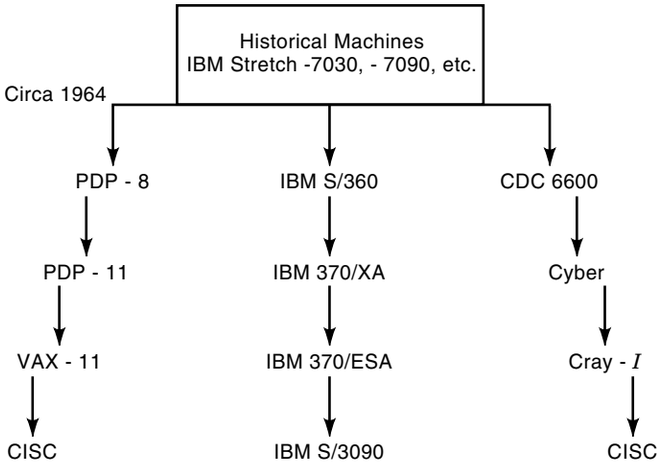

Figure 9. Main branches in development of computer architecture.

mizing Compiler" was introduced in RISC (5,9,14). This compiler was capable of producing a code that was as good as the code written in assembler (the hand-code). Though there was strict attention given to the architecture principle (1,3), adhering to the absence of the implementation details from the principle of the operation, this is perhaps the only place where this principle was violated. Namely, the optimizing compiler needs to "know" the details of the implementation, the pipeline in particular, in order to be able to efficiently schedule the instructions. The work of the optimizing compiler is illustrated in Fig. 7.

One Instruction per Cycle

The objective of *one instruction per cycle* (CPI = 1) execution was the ultimate goal of RISC machines. This goal can be theoretically achieved in the presence of infinite size caches and thus no pipeline conflicts, which is not attainable in practice. Given the frequent branches in the program and their interruption to the pipeline, Loads and Stores that cannot be scheduled, and finally the effect of finite size caches, the number of "lost" cycles adds up, bringing the CPI further away from 1. In the real implementations the CPI varies and a CPI = 1.3 is considered quite good, while CPI between 1.4 to 1.5 is more common in single-instruction issue implementations of the RISC architecture.

However, once the CPI was brought close to 1, the next goal in implementing RISC machines was to bring CPI below 1 in order for the architecture to deliver more performance. This goal requires an implementation that can execute more than one instruction in the pipeline cycle, a so called *superscalar* implementation (12,16). A substantial effort has been made on the part of the leading RISC machine designers to build such machines. However, machines that execute up to four instructions in one cycle are common today, and a machine that executes up to six instructions in one cycle was introduced in 1997.

Pipelining

Finally, the single most important feature of RISC is pipelining. The degree of parallelism in the RISC machine is determined by the depth of the pipeline. It could be stated that all the features of RISC (that were listed in this article) could easily be derived from the requirements for pipelining and maintaining an efficient execution model. The sole purpose of many of those features is to support an efficient execution of RISC pipeline. It is clear that without pipelining, the goal of CPI = 1 is not possible. An example of the instruction execution in the absence of pipelining is shown in Fig. 8.

We may be led to think that by increasing the number of pipeline stages (the pipeline depth), thus introducing more parallelism, we may increase the RISC machine performance further. However, this idea does not lead to a simple and straightforward realization. The increase in the number of pipeline stages introduces not only an overhead in hardware (needed to implement the additional pipeline registers), but also the overhead in time due to the delay of the latches used to implement the pipeline stages as well as the cycle time lost due to the clock skews and clock jitter. This could very soon

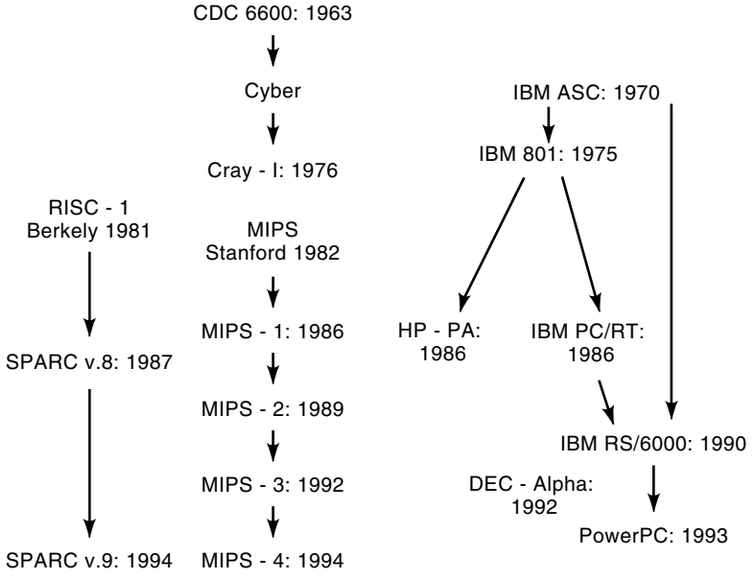

Figure 10. History of RISC development.

Table 2. Some features of RISC Processors

Feature	Digital 21164	MIPS 10000	PowerPC 620	HP 8000	Sun UltraSparc
Frequency (MHz)	500	200	200	180	250
Pipeline stages	7	5–7	5	7–9	6–9
Issue rate	4	4	4	4	4
Out-of-order execution	6 Loads	32	16	56	None
Register renaming (int/FP)	None/8	32/32	8/8	56	None
Transistors/logic transistors	9.3 M/1.8 M	5.9 M/2.3 M	6.9 M/2.2 M	3.9 M[a]/3.9 M	3.8 M/2.0 M
SPEC95 (Intg/FlPt)	12.6/18.3	8.9/17.2	9/9	10.8/18.3	8.5/15
Performance/log-trn (Intg/FP)	7.0/10.2	3.9/7.5	4.1/4.1	2.77[a]/4.69	4.25/7.5

[a] No cache.

bring us to the point of diminishing returns where further increase in the pipeline depth would result in less performance. An additional side effect of deeply pipelined systems is hardware complexity necessary to resolve all the possible conflicts that can occur between the increased number of instructions residing in the pipeline at one time. The number of the pipeline stages is mainly determined by the type of the *instruction core* (the most frequent instructions) and the operations required by those instructions. The pipeline depth depends, as well, on the technology used. If the machine is implemented in a very high speed technology characterized by the very small number of gate levels (such as GaAs or ECL), and a very good control of the clock skews, it makes sense to pipeline the machine deeper. The RISC machines that achieve performance through the use of many pipeline stages are known as *superpipelined* machines.

Today the most common number of pipeline stages encountered is five (as in the examples given in this text). However, 12 or more pipeline stages are encountered in some machine implementations.

The features of RISC architecture that support pipelining are listed in Table 1.

HISTORICAL PERSPECTIVE

The architecture of RISC did not come about as a planed or a sudden development. It was rather a long and evolutionary process in the history of computer development in which we learned how to build better and more efficient computer systems. From the first definition of the architecture in 1964 (1), there are the three main branches of the computer architecture that evolved during the years. They are shown in Fig. 9.

The CISC development was characterized by (1) the PDP-11 and VAX-11 machine architecture that was developed by Digital Equipment Corporation (DEC) and (2) all the other architectures that were derived from that development. The middle branch is the IBM 360/370 line of computers, which is characterized by a balanced mix of CISC and RISC features. The RISC line evolved from the development line characterized by Control Data Corporation CDC 6600, Cyber, and ultimately the CRAY-I supercomputer. All of the computers belonging to this branch were originally designated as *supercomputers* at the time of their introduction. The ultimate quest for performance and excellent engineering was a characteristic of that branch. Almost all of the computers in the line preceding RISC carry the signature of one man: Seymour Cray, who is by many given the credit for the invention of RISC.

History of RISC

The RISC project started in 1975 at the IBM Thomas J. Watson Research Center under the name of the 801. 801 is the number used to designate the building in which the project started (similar to the 360 building). The original intent of the 801 project was to develop an emulator for System/360 code (5). The IBM 801 was built in ECL technology and was completed by the early 1980s (5,8). This project was not known to the world outside of IBM until the early 1980s, and the results of that work are mainly unpublished. The idea of a simpler computer, especially the one that can be implemented on the single chip in the university environment, was appealing; two other projects with similar objectives started in the early 1980s at the University of California Berkeley and Stanford University (6,7). These two academic projects had much more influence on the industry than the IBM 801 project. Sun Microsystems developed its own architecture currently known as SPARC as a result of the University of California Berkeley work. Similarly, the Stanford University work was directly transferred to MIPS (17).

The chronology illustrating RISC development is illustrated in Fig. 10.

The features of some contemporary RISC processors are shown in Table 2.

BIBLIOGRAPHY

1. G. M. Amdahl, G. A. Blaauw, and F. P. Brooks, Architecture of the IBM System/360, *IBM J. Res. Develop.,* **8**: 87–101, 1964.
2. D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples,* Advanced Computer Science Series, New York: McGraw-Hill, 1982.
3. G. A. Blaauw and F. P. Brooks, The structure of System/360, *IBM Syst. J.,* **3**: 119–135, 1964.
4. R. P. Case and A. Padegs, Architecture of the IBM System/370, *Commun. ACM,* **21**: 73–96, 1978.
5. G. Radin, The 801 Minicomputer, IBM Thomas J. Watson Research Center, Rep. RC 9125, 1981; also in *SIGARCH Comput. Archit. News,* **10** (2): 39–47, 1982.
6. D. A. Patterson and C. H. Sequin, A VLSI RISC, *IEEE Comput. Mag.,* **15** (9): 8–21, 1982.
7. J. L. Hennessy, VLSI processor architecture, *IEEE Trans. Comput.,* **C-33**: 1221–1246, 1984.

8. J. Cocke and V. Markstein, The evolution of RISC technology at IBM, *IBM J. Res. Develop.,* **34**: 4–11, 1990.
9. M. E. Hopkins, A perspective on the 801/reduced instruction set computer, *IBM Syst. J.,* **26**: 107–121, 1987.
10. L. J. Shustek, Analysis and performance of computer instruction sets, PhD thesis, Stanford Univ., 1978.
11. D. Bhandarkar and D. W. Clark, Performance from architecture: Comparing a RISC and a CISC with similar hardware organization, *Proc. 4th Int. Conf. ASPLOS,* Santa Clara, CA, 1991.
12. G. F. Grohosky, Machine organization of the IBM RISC System/6000 processor, *IBM J. Res. Develop.,* **34**: 37, 1990.
13. J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach,* San Mateo, CA: Morgan Kaufman.
14. H. S. Warren, Jr., Instruction scheduling for the IBM RISC System/6000 processor, *IBM J. Res. Develop.,* **34**: 37, 1990.
15. J. K. F. Lee and A. J. Smith, Branch prediction strategies and branch target buffer design, *Comput.,* **17** (1): 1984, 6–22.
16. J. Cocke, G. Grohosky, and V. Oklobdzija, *Instruction control mechanism for a computing system with register renaming, MAP table and queues indicating available registers,* U.S. Patent No. 4,992,938, 1991.
17. G. Kane, *MIPS RISC Architecture,* Englewood Cliffs, NJ: Prentice-Hall, 1988.

Reading List

D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, The IBM 360 Model 91: Machine philosophy and instruction handling, *IBM J. Res. Develop.,* **11**: 8–24, 1967.

Digital RISC Architecture Technical Handbook, Digital Equipment Corporation, 1991.

V. G. Oklobdzija, Issues in CPU—coprocessor communication and synchronization, *EUROMICRO '88, 14th Symp. Microprocessing Microprogramming,* Zurich, Switzerland, 1988, p. 695.

R. M. Tomasulo, An efficient algorithm for exploring multiple arithmetic units, *IBM J. Res. Develop.,* **11**: 25–33, 1967.

VOJIN G. OKLOBDZIJA
Integration Corporation

REDUNDANT SYSTEMS ANALYSIS. See RELIABILITY OF REDUNDANT AND FAULT-TOLERANT SYSTEMS.

RE-ENGINEERING. See BUSINESS PROCESS RE-ENGINEERING; SOFTWARE MAINTENANCE, REVERSE-ENGINEERING AND RE-ENGINEERING; SYSTEMS RE-ENGINEERING.

REFLECTANCE. See GONIOMETERS.

REFLECTION MEASUREMENT. See STANDING WAVE METERS AND NETWORK ANALYZERS.