

Reduced Instruction Set Computers

Prof. Vojin G. Oklobdzija

University of California

Keywords: IBM 801; RISC; computer architecture; Load/Store architecture; instruction sets; pipelining; super-scalar machines; super-pipeline machines; optimizing compiler; Branch and Execute; Delayed Branch;

Definitions

Main features of RISC architecture

Analysis of RISC and what makes RISC

What brings performance to RISC

Going beyond one instruction per cycle

Issues in super-scalar machines

1. Architecture

The term *Computer Architecture* was first defined in the paper by Amdahl, Blaauw and Brooks of IBM Corporation announcing IBM System/360 computer family on April 7, 1964 [1]. On that day IBM Corporation introduced, in the words of IBM spokesman, "*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 which enables 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* [2]. 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*.

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 embodiment 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 which objective is to remove ambiguities in the definition of the architecture and in some cases, adjust the functions provided [1-3].

1.1. RISC Architecture

A special place in computer architecture has been given to RISC. RISC architecture has been developed as a result of the 801 project which started in 1975 at the IBM T.J.Watson Research Center and was completed by the early 1980s [6]. 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 [7,8]. The term RISC (Reduced Instruction Set Architecture), 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"* [9] after revealing evidence which 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. This evidence 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 reset of 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"* [6]. Therefore RISC architecture starts with a small set of most frequently used instructions which determine 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 12-18 gate levels) for one gate level, than the new instruction should contribute at least 6-8% to the overall performance of the machine.

One cycle per instruction is achieved by exploitation of parallelism through the use of pipelining. It turns out that *parallelism through pipelining* is the single most important characteristic of RISC architecture from which all the rest of the RISC features could be 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 outgrow of Berkeley RISC), MIPS (an outgrow of Stanford MIPS project, used by Silicon Graphics), and a *super-scalar* implementation of RISC architecture, IBM RS/6000 (also known as PowerPC architecture).

1.2. RISC Performance

Since the early beginning, the quest for higher performance has been present in every new computer model and architecture. This has been the driving force behind 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 and 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 take effect. 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 straight forward. Thus, the time required to execute each instruction can be shortened and the number of cycles reduced. Typically the instruction execution time is divided in five stages, *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:

- 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).

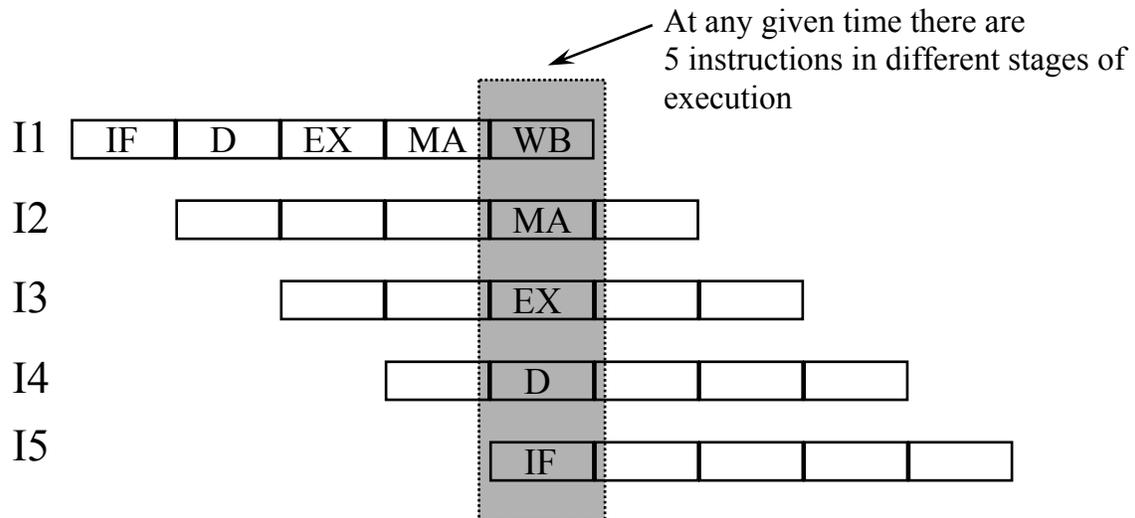


Fig. 1. Typical five stage RISC pipeline

The goal of RISC is to achieve execution rate of one Cycle Per Instruction (CPI=1.0) which would be the case when no interruptions in the pipeline occurs. However, as we will see later, this is not the case.

Thus the instructions and the addressing modes in RISC architecture are carefully selected and tailored upon the most frequently used ones, but in a way that will satisfy the 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:

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

I = no. of instructions / task

C = no. of cycles / instruction

P = no. of clock periods / cycle (usually P=1)

T₀ = clock period (nS)

While CISC instruction will typically have fewer 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. On the other hand RISC will require more instructions for the same task, however, it executes its instructions at the rate of one 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 T₀ in RISC can be shorter allowing RISC machine to run at the higher speed as compared to CISC. Typically as of today RISC machines have been running at the rate in excess of 667 MHz reaching 1 GHz, while CISC is hardly at 500MHz clock rate.

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

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

2. RISC

Another distinguished feature of RISC is exploitation of locality ...Given that the main feature of RISC is the architectural support for the exploitation of parallelism on the instruction level, we should revisit all the eight remaining features in light of their flow through the RISC pipeline.

2.1. Load / Store Architecture

Often RISC is referred to as Load/Store architecture or the operations in its instruction set are defined as Register-to-Register operations. This is because all the operations performed by RISC are between the operands that reside in the *General Purpose Register File* (GPR) where the result of the operation is written back to. Limiting the locations of the operands to the GPR allows for determinism in the RISC operation. In the other words, a potentially multi-cycle and unpredictable access to memory has been de-coupled from the operation. Once the operands are available in the GPR the operation can proceed. It is almost certain that once commenced the operation will be completed and the result being written into the GPR. Of course, there are possible conflicts for the operands which can, never the less, be easily handled in hardware. The execution flow in the pipeline for a register-to-register operation is shown in Fig. 1.

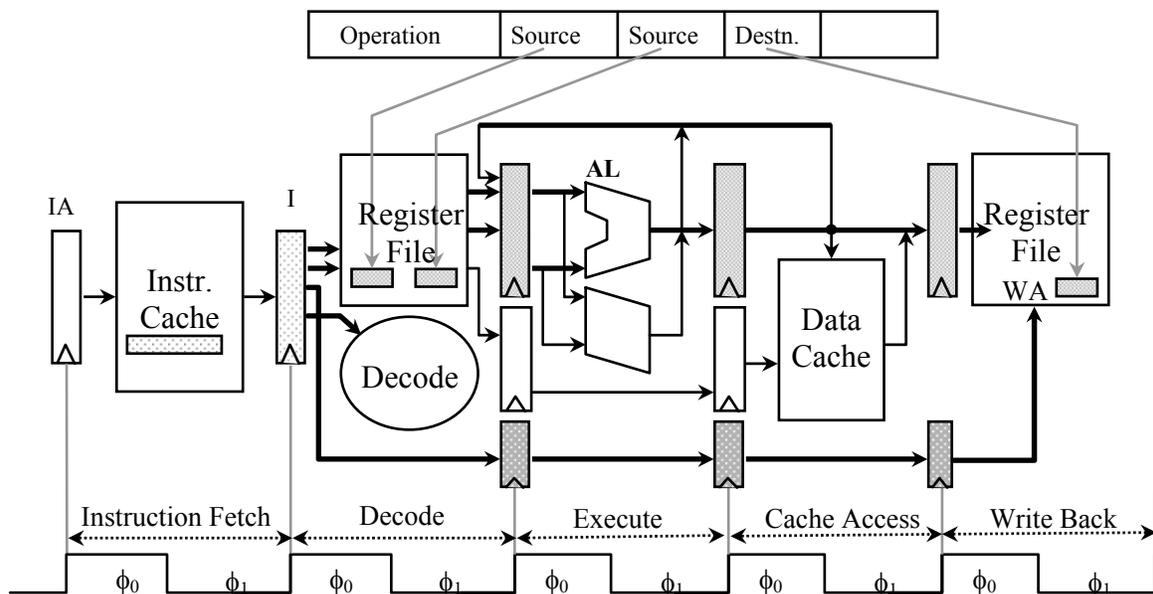


Fig. 1. Pipeline Flow of a Register-to-Register operation

On the other hand Memory Access is achieved only through Load and Store instructions (thus the term Load/Store Architecture often is used to refer to RISC). The pipeline is

tailored in a way in which it accommodates both: operation and memory access with equal efficiency.

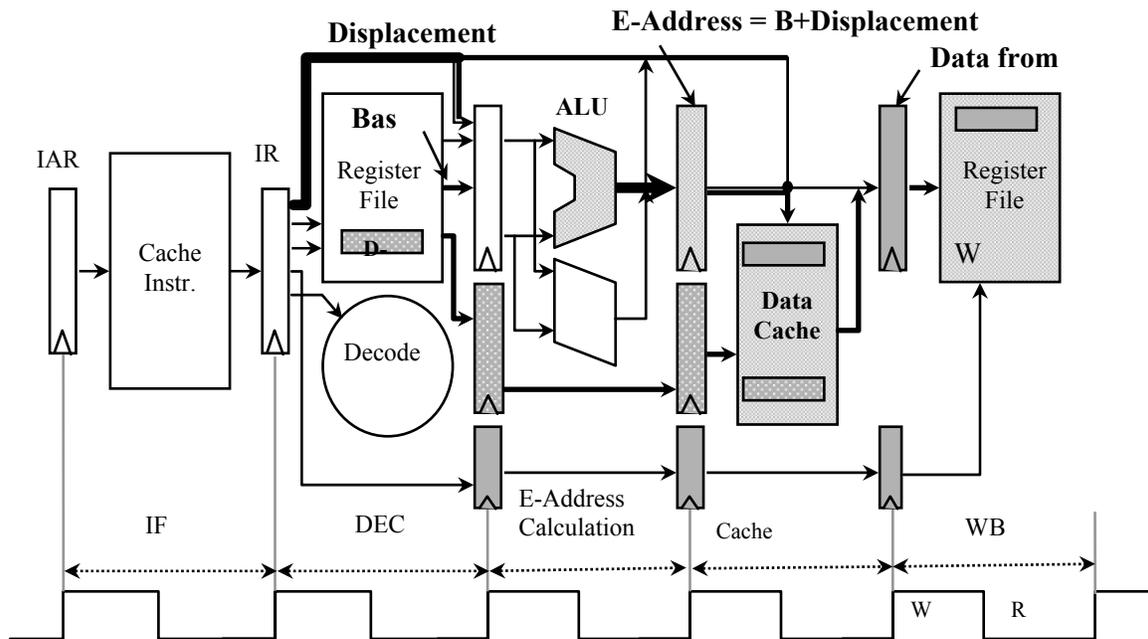


Fig. 2. Load/Store Pipeline

2.2. Carefully Selected Set of Instructions

The principle of locality is applied throughout RISC. The fact that only a small set of instructions is used most frequently is used to determine the most efficient pipeline organization that would exploit instruction level parallelism of those instructions in the most efficient way. The pipeline is “tailored” for that group of instructions. Such derived pipeline must serve efficiently the three main classes of instructions:

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

Given the simplicity of the pipeline the control part of RISC is implemented in hardware. However, this is the most misunderstood part of RISC architecture which has even given it the inappropriate name. RISC as it stands out: Reduced Instruction Set Computer, does not necessarily imply that the number of instructions in RISC has to be small. The number of instructions in the instruction set of RISC can grow until the complexity of the control starts to impose an increase in the clock period. In practice this point is further beyond the number of instructions commonly used and we have witnessed a paradox that several of today’s RISC machines have an instruction set larger than that of CISC.

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 IBM System/370 containing 208. The first two are representatives of RISC architecture while the later two are not.

2.3. Fixed format instructions

What really matters in RISC is that the instructions have 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 (32-bits and 16-bits). The fixed format feature is very important if it is required from RISC to decode its instruction in one cycle. This feature is very valuable for super-scalar implementations. Fixed size instruction allow Instruction Fetch to be pipelined (know next address without decoding the current one). This guarantees only single I-TLB access per instruction.

Specifically, this is especially important if we want to determine the outcome of the Branch instruction and clock the new instruction target address in only one cycle. The operation associated with determining a Branch instruction during the Decode cycle is illustrated in Fig. 3. In order to minimize the number of lost cycles, Branch has to be resolved during Decode stage. This requires a separate address adder as well as comparator which are used during Decode stage. In the best case one cycle will be lost when Branch instruction is encountered. (this *instruction slot* is used for an independent instruction which is scheduled in this slot – as we will see later in “branch and execute”)

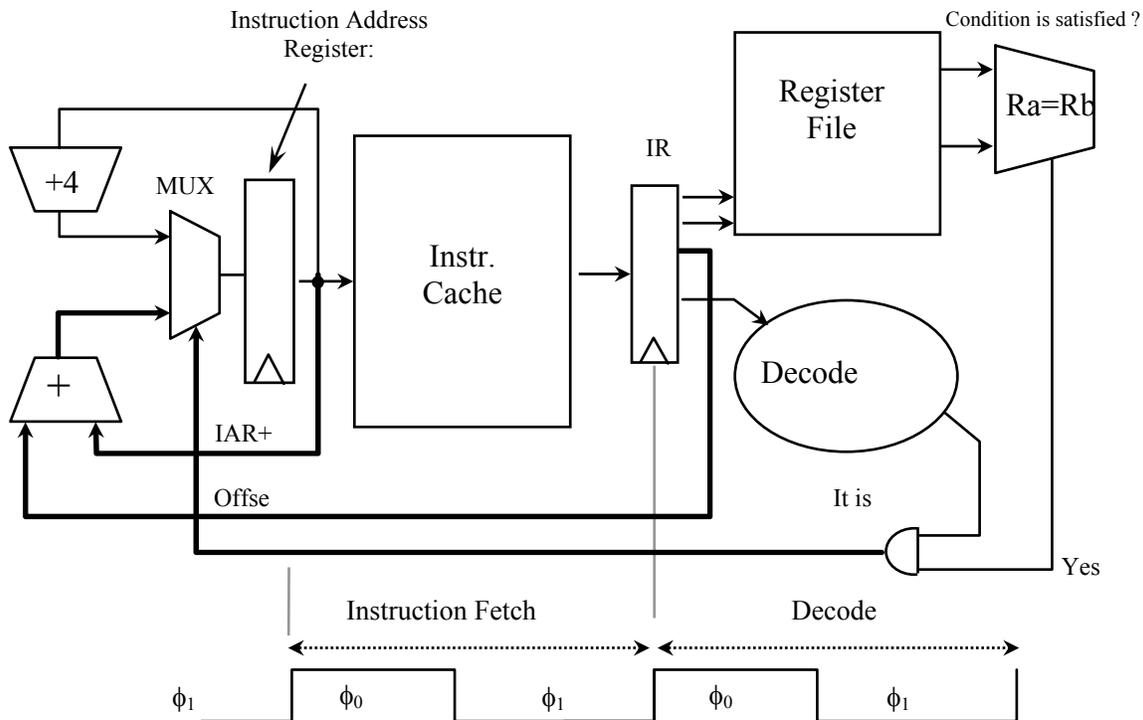


Fig. 3. Branch Instruction

2.4. Simple Addressing Modes

Simple Addressing Modes are the requirements of the pipeline. Namely, in order to be able to perform the address calculation in the pre-determined number of pipeline cycles and in the pre-determined pipeline, the address computation need to conform to the other modes of computation. It is a fortunate fact that in the real programs the requirements for the address computations favors three main addressing modes:

- (a.) Immediate
- (b.) Base + Displacement
- (c.) Base + Index

Those three addressing modes take approximately over 80% of all the addressing modes:

(a.) 30-40% (b.) 40-50% (c.) 10-20% according to [11]. The process of calculating the operand address associated with Load and Store instructions is shown in Fig. 2.

2.5. Separate Instruction and Data Caches

One of the often overlooked but essential characteristics of RISC machines is existence of Cache memory. The second most important characteristic of RISC (after pipelining) is exploitation of the *locality principle*. The locality principle is established in the observation that on the 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 the 10% of the instructions are responsible for 90% of the code. Often the principle of the locality is referred too as a 90-10 rule [11].

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, the *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. 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 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.

2.6. Branch and Execute Instruction

Branch and Execute or *Delayed Branch* instruction is a new instruction architecture feature 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. 4.

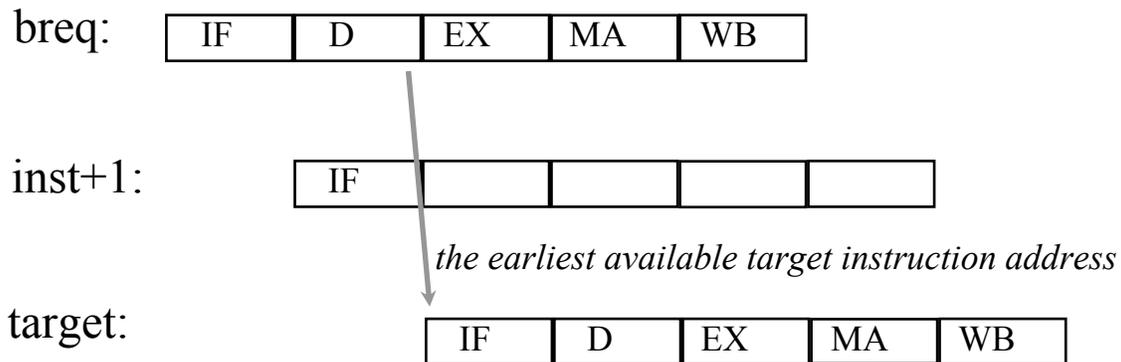


Fig. 4. Branch Instruction Pipeline Flow

RISC architecture solves this problem by introducing *Branch and Execute* instruction (also known as *Delayed Branch Instruction*), which consists of a 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 come from the instruction stream preceding the branch instruction, from the target instruction stream or from the fall-through instruction stream. It is the task of the compiler to find such an instruction and to fill-in this execution cycle. Given the frequency of the branch instructions which varies from one out of five to one out of fifteen (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. This is the single most performance contributing instruction from the RISC instruction architecture.

However, in the later generations of super-scalar RISC machines (that execute more than one instruction in the pipeline cycle) Branch and Execute instruction has been abandoned in favor of *Branch Prediction*.

The Load instruction is also exhibiting this lost pipeline cycle as shown in Fig. 5.

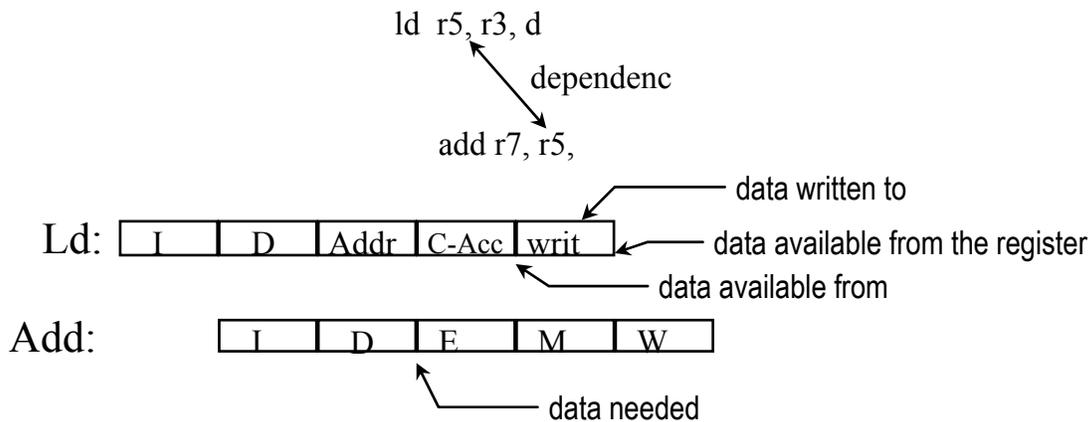


Fig. 5. Lost cycle during the execution of the Load Instruction

The same principle that was applied for alleviating branch problem can be applied to 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. 6.

Program to calculate:

```
a = b + 1
if (c=0) d = 0
```

Sub-optimal:

```
ld r2, b    # r2=b
add r2, 1  # r2=b+1
st r2, a    # a=b+1
ld r3, c    # r3=c
bne r3,0, tg1 #
skip
st 0, d     # d=0
```

Annotations:
 - load: from add to ld r3, c
 - load: from bne to ld r3, c
 - lost: from skip to ld r3, c

Total = 9 cycles

Optimal

```
ld r2, b    # r2=b
ld r3, c    # r3=c
add r2, 1  # r2=b+1
bne r3,0, tg1 #
skip
st r2, a    # a=b+1
st 0, d     # d=0
```

Total = 6 cycles

Fig. 6. An Example of Instruction Scheduling by Compiler

2.7. 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 RISC architecture is originally intended to create a machine that is only “*visible through the compiler*” [5,7]. All the programming is to be done in High-Level Language and only a minimal portion in assembler. The notion of the “*optimizing compiler*” was introduced in RISC [5,7,8]. This compiler was capable of producing a code that was as good as the code written in assembler (the hand-code). Though there was a strict attention given to the *architecture principle* [1,2] as far as absence of the implementation details from the principle of the operation, this is perhaps the only place where this principle was close to being 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. 6.

2.8. One Instruction per Cycle

The objective of one instruction per cycle: $CPI = 1$ execution is the ultimate goal of RISC machines. This goal assumes infinite caches and no pipeline conflicts thus, it is not attainable. Given the frequent branches in the program and their interruption to the pipeline, loads and stores that can not be scheduled and finally the effect of finite size caches, the number of “lost” cycles accumulates 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 one, the next goal in implementing RISC machines was to bring CPI below one. This goal requires an implementation that can execute more than one instruction in the pipeline cycle a so called *Super-Scalar* implementation. A substantial effort has been done on the part of the leading RISC machine implementers to build such machines. However, machines that execute up to four instructions in one cycle are quite common today and a machine that executes up to six instructions in one cycle has been introduced last year.

2.9. Pipelining

Finally, the single most important feature of RISC is pipelining. 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 here, can be derived from pipelining. 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. 7.

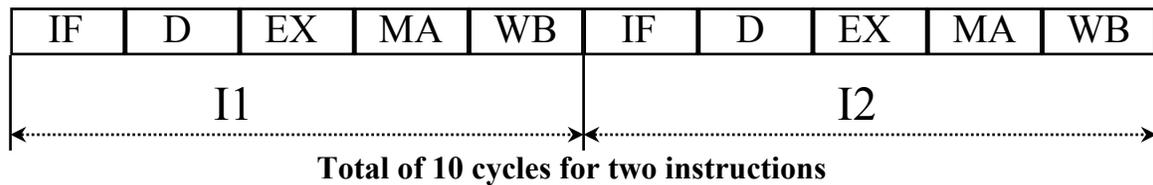


Fig. 7. Instruction execution without pipelining

We may think that by increasing the number of pipeline stages (the pipeline depth) we may increase the RISC machine performance further by introducing more parallelism. However, this is not so simple and straight forward. The increase in the number of pipeline stages brings with it an overhead not only 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 stage as well as the clock skews and clock jitter. All of that will soon 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 also depends 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) it makes sense to pipeline the machine deeper. The RISC machines that achieve performance through the use of many pipeline stages are known as *super-pipelined* machines.

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

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

Table 1. Features of RISC Architecture

Feature	Characteristic
Load / Store Architecture	All of the operations are Register to Register. In this way <u>Operation</u> is decoupled from the <u>access to memory</u>
Carefully selected sub-set of instructions	Control is implemented in hardware. There is no microcoding in RISC. Also this set of instructions is not necessarily small*
Simple Addressing Modes	Only the most frequently used addressing modes are used. Also it is important that <u>they can fit into the existing pipeline.</u>
Fixed size and fixed fields instructions	This is necessary to be able to decode instruction and access operands in one cycle. Though there are architectures using two sizes for the instruction format (IBM PC-RT)
Delayed Branch Instruction (known also as Branch and Execute)	The 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 the use of pipelining
Optimizing Compiler	Close coupling between the architecture and the compiler. Compiler " <i>knows</i> " about the pipeline.
Harvard Architecture	Separation of Instruction and Data Cache resulting in increased memory bandwidth.

* 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 while the later two are not.

3. Historical Perspective

The RISC architecture was not a sudden development, but it rather a long and evolutionary process in which more was learned about computer systems and how to build them efficiently. From the first definition of the architecture in 1964 [1] we can distinguish the three main branches of the computer architecture development, as shown in Fig. 8.

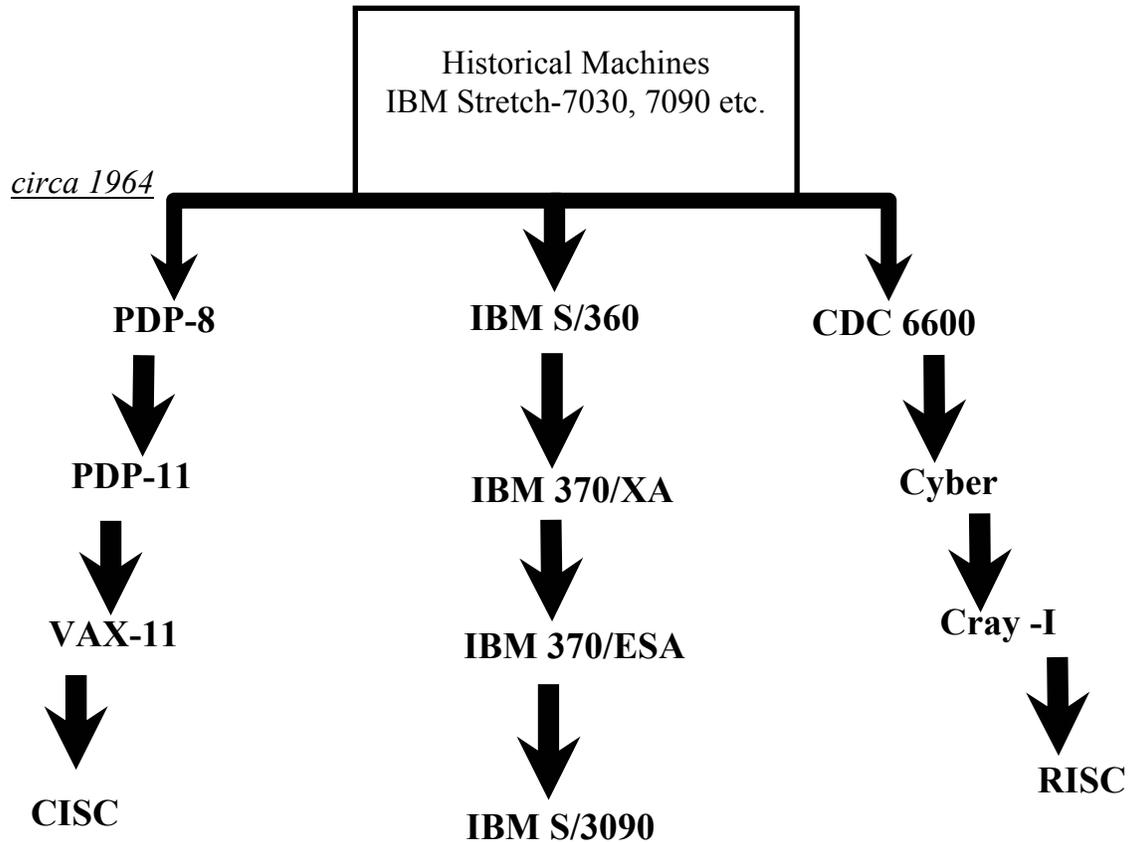


Fig. 8. Main Branches in Development of Computer Architecture

The CISC development can be characterized by the PDP-11 and VAX-11 architecture and all the architectures derived from that development. The middle branch is the IBM 360/370 line of computers which is characterized with a balance of CISC and RISC features. The RISC line really evolved from the line characterized by CDC 6600, Cyber and ultimately CRAY super-computer. All of the computers in this branch were originally designated as super-computers 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 credited with RISC ideas.

3.1. History of RISC

The RISC project started in 1975 at the IBM T.J.Watson Research Center under the name of the 801. The original intent of the 801 project was to develop an emulator for System/360 code. It was built in ECL technology and was completed by the early 1980s [6]. This project was not known to the world outside of IBM until early 1980s and the result of that work are mainly unpublished. The idea of simpler computer especially the one that can be implemented on the single chip in the university environment was appealing and two other projects with similar objectives started in the early 1980s at the University of California Berkeley and Stanford University [7,8]. The 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.

The chronology illustrating RISC development is illustrated in Fig. 9.

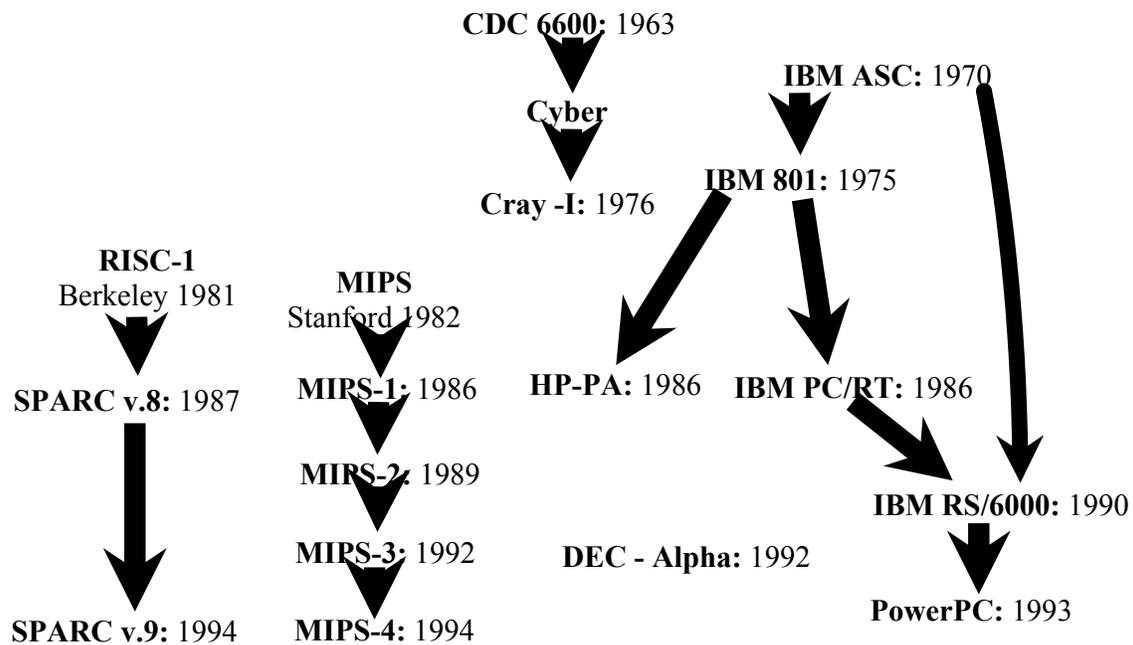


Fig. 9. History of RISC development

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

Table 2: Features of contemporary RISC processors:

Feature	Digital 21164	MIPS 10000	PowerPC 620	HP 8000	Sun UltraSpar c
<i>Frequency</i>	500 MHz	200 MHz	200 MHz	180 MHz	250 MHz
<i>Pipeline Stages</i>	7	5-7	5	7-9	6-9
<i>Issue Rate</i>	4	4	4	4	4
<i>Out-of-Order Exec.</i>	6 loads	32	16	56	none
<i>Register Renam. (int/FP)</i>	none/8	32/32	8/8	56	none
<i>Transistors/ Logic transistors</i>	9.3M/ 1.8M	5.9M/ 2.3M	6.9M/ 2.2M	3.9M*/ 3.9M	3.8M/ 2.0M
<i>SPEC95 (Intg/FlPt)</i>	12.6/18.3	8.9/17.2	9/9	10.8/18.3	8.5/15
<i>Perform./ Log-trn (Intg/FP)</i>	7.0/10.2	3.9/7.5	4.1/4.1	2.77*/4.69	4.25/7.5

* no cache

Conclusion

Difficult competition and complex designs are ahead, yet: *“Risks are incurred not only by undertaking a development, but also by not undertaking a development”*

Super-scalar techniques will help performance to grow faster, with less expense as compared to the use of new circuit technologies and new system approaches such as multiprocessing.*

Ultimately, super-scalar techniques buy time to determine the next cost-effective techniques for increasing performance.*

*Mike Johnson (Super-scalar Microprocessor Design, Prentice-Hall 1991)

References

- [1] G.M.Amdahl, G.A. Blaauw, F.P. Brooks, "Architecture of the IBM System/360, IBM Journal of Research and Development, Vol.8, No.2, p.87-101, April 1964.
- [2] G.A. Blaauw, F.P. Brooks, "*The Structure of System/360*", IBM Systems Journal, Vol.3, No.2, p.119-135, 1964.
- [3] R.P.Case, A.Padegs, "Architecture of the IBM System/370", Communications of ACM, Vol.21, No.1, p. 73-96, January 1978.
- [4] D.W.Anderson, F.J.Sparacio, and R.M.Tomasulo, "The IBM 360 Model 91: Machine philosophy and instruction handling," *IBM Journal of Research and Development*, Vol.11, No.1, January 1967, p.8-24.
- [5] G. Radin, "The 801 Minicomputer", IBM T.J.Watson Research Center, Report RC 9125, November 11, 1981, also in SIGARCH Computer Architecture News 10, No.2, p.39-47, March 1982.
- [6] John Cocke andViky Markstein, "The Evolution of RISC Technology at IBM," *IBM Journal of Research and Development*, Vol.34, No.1, pp.37, January 1990.
- [7] M. E. Hopkins, "*A Perspective on the 801 / Reduced Instruction Set Computer*", IBM Systems Journal, Vol. 26, No.1, 1987.
- [8] Henry S. Warren, Jr., "Instruction scheduling for the IBM RISC System/6000 processor," *IBM Journal of Research and Development*, Vol.34, No.1, pp.37, January 1990.
- [9] D.A. Patterson, C.H.Sequin, "A VLSI RISC", IEEE Computer Magazine, September 1982.
- [10] J. L. Hennessy, "*VLSI Processor Architecture*", IEEE Transactions on Computers, Vol. C-33, No.12, December 1984.
- [11] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan & Kaufman Publishers, San Mateo, California.
- [12] L.J.Shustek, "Analysis and Performance of Computer Instruction Sets," PhD. Thesis, Stanford University, May 1978.
- [13] Gregory F. Grohosky, "Machine Organization of the IBM RISC System/6000 processor," *IBM Journal of Research and Development*, Vol.34, No.1, pp.37, January 1990.
- [14] V.G.Oklobdzija, "Issues in CPU-Coprocessor Communication and Synchronization," EUROMICRO '88, Fourteenth Symposium on Microprocessing and Microprogramming, pp. 695., Zurich, Switzerland, August 1988.
- [15] R.M.Tomasulo, "An Efficient Algorithm for Exploring Multiple Arithmetic Units," *IBM Journal of Research and Development*, Vol.11. No.1. p.25-33.
- [16] John Cocke, Gregory Grohosky, and Vojin 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, February 12, 1991.
- [17] D.P. Siewiorek, C.G. Bell, A. Newell, "Computer Structures: Principles and Examples", McGraw-Hill Advanced Computer Science Series, 1982.
- [18] "Digital RISC Architecture Technical Handbook," Digital Equipment Corporation 1991.
- [19] D. Bhandarkar and D.W. Clark, "Performance from Architecture: Comparing a RISC and a CISC with Similar Hardware Organization," *Proceedings of the 4th Int'l. Conference on ASPLOS*, Santa Clara, California, April 8-11, 1991.
- [20] Gery Kane, *MIPS RISC Architecture*, Prentice-Hall, New Jersey, 1988.
- [21] SPARC Architecture definition.
- [22] J.K.F.Lee and A.J.Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *Computer*, Vol.17,No.1.,1984, p.6-22.
- [23] J.E.Smith, S. Weiss, and N.Y. Pang, "A Simulation Study of Decoupled Architecture Computers, *IEEE Transaction on Computers*, Vol C-35, No.8, August 1986, p.692.