BCA Part I

Paper-VII

Topic: Reduced Instruction Set Computer Architecture

Prepared by: Dr. Kiran Pandey
(School of Computer Science)

Email-id: kiranpandey.nou@gmail.com

Introduction to RISC

The RISC stands for Reduced Introduction Set Computer. To put simply, RISC is a microprocessor which runs using a pipelining architecture to improve the performance of a processor. Generally speaking this means faster machine, mostly by improving MIPS (which stands for millions of instructions per sec, meaning higher MIPS are better). It is important to note that improvement of MIPS isn't always result in faster machine, as measurement of MIPS alone isn't good enough to measure the processor. Some well known microprocessors such as SUN Microsystems' SPARC microprocessor or DEC's Alpha microchips uses the RISC concept to develop their microprocessors.

Few more improvements by using RISC processors besides improving MIPS are :

1. A new microprocessor can be developed and tested more quickly if one of its aims is to be less complicated.
2. Operating system and application programmers who use the microprocessor's instructions will find it easier to develop code with a smaller instruction set.
3. The simplicity of RISC allows more freedom to choose how to use the space on a microprocessor.

4. Higher-level language compilers produce more efficient code than formerly because they have always tended to use the smaller set of instructions to be found in a RISC computer.

Importance of RISC Processors

RISC processors exhibit certain distinct features compared to the CISC processors. They include :

- ✶ Few Powerful Instructions
- ✶ Single clock Cycle execution (for most instruction)
- ✶ Register-based execution
- ✶ Highly Pipelined Superscalar Architecture
- ✶ Fixed Instruction Format and fixed length instruction
- ✶ Large register set or register windows
- ✶ Hardwired control Unit
- ✶ Highly Integrated Architecture

Examples of RISC processor

RISC has fewer design bugs, its simple instructions reduce design time. Thus, because of all the above important reasons RISC processors have become very popular. Some of the RISC processors are :

i860, SPARC Processors

Sun 4/350 SPARCserver 350, Sun 4/360 SPARCserver 360, Sun 4/370 SPARCserver 370, Sun 4/20, SPARCstation SLC, Sun 4/40 SPARCstation IPC, Sun 4/75, SPARCstation 2.

PowerPC Processors (Power refers to Optimisation with enhanced RISC).

MPC755, MPC7400/7410, MPC745X, MPC7450, MPC8240, MPC8245.

Titanium-IA64 Processor

RISC processors are used to build supercomputers for high performance computing. For example, i860 family of processors are used to build PARAM (PARALLEL Machine), PARALLEL Super Computer of Centre for Development of Advanced Computing (C-DAC), India.

Reasons for Increased Complexity

The reasons for increased complexity are :

Speed gap between Memory and CPU

In the past, there was a big gap between the speed of a processor and memory. Thus, a subroutine execution for an instruction, for example floating point

addition, may have to follow a lengthy instruction sequence. The question is, if we make it a machine instruction then only one instruction fetch will be required and rest will be done with control unit sequence. Thus, a "higher level" instruction can be added to machines in a attempt to improve performance.

However, today the Main memory is supported with Cache technology. Cache memories have reduced the difference between the CPU and the memory speed and, therefore, an instruction execution through a subroutine step may not be that difficult. Pipelining can further enhance such speed.

Microcode technology versus VLSI Technology

The control unit of a computer can be constructed using two ways: create micro-program that execute micro-instructions or build circuits (hardwired) for each instruction execution. Micro-programmed control allows the implementation of complex architectures more cost effective than hardwired control as the cost to expand an instruction set is very small, only a few more microinstructions for the control store. Thus it may be reasoned that moving subroutines like string editing, integer to floating point number conversion and mathematical evaluations such as polynomial evaluation to control unit micro-program is more cost effective.

Code Density versus Fast execution of programs

The memory was very large and expensive in the older computer. Thus there was a need of less memory utilization, that is, it was cost effective to have smaller compact programs. Thus, it was thought that the instruction set should be more complex, so that programs are smaller. However, increased complexity of instruction sets had resulted in instruction sets and addressing modes requiring more bits to represent them. It is stated that the code compaction is important, but the cost of 10 percent more memory is often far less than the cost of reducing code by 10 percent out of the CPU architecture innovations.

The smaller programs are advantageous because they require smaller RAM space. However, today memory is very inexpensive, this potential advantage today is not so compelling. More important, small programs should improve performance because fewer instructions mean fewer instruction bytes to be fetched.

Support for High-Level Language

With the advent and use of more and more higher level languages, manufactures had provided more powerful instructions to support them. It was argued that a stronger instruction set would reduce the software crisis and would simplify the compilers. Another important reason for such a movement was the desire to improve performance.

However, even though the instructions that were closer to the high level languages were implemented in Complex Instruction Set Computers (CISCs), still it was hard to exploit these instructions since the compliers were needed to find those conditions that exactly fit those constructs. In addition, the task of optimising the generated code to minimise code size, reduce instruction execution count, and enhance pipelining is much more difficult with such a complex instruction set.

Another motivation for increasingly complex instructions sets was that the complex HLL operation would execute more quickly as a single machine instruction rather than as a series of more primitive instructions. CISC makes the more complex control unit with larger micro program control store to accommodate a richer instruction set. This increases the execution time for simpler instructions. Thus, it is far from clear that the trend to complex instruction sets is appropriate.

High Level Language Program Characteristics

A high-level language system can be implemented mostly by hardware or mostly by software, provided the system hides any lower level details from the programmer. Thus, a cost-effective system can be built by deciding what pieces of the system should be in hardware and what pieces in software. To ascertain the above, it may be a good idea to find program characteristics on general computers. Some of the basic findings about the program characteristics are :

Variables	Operations	Procedure Calls
Integer Constants 10-25%	Simple assignment 35-45%	Most time consuming operation
Scalar Variables 50-60%	Function call 10-15%	Most of the functions are called with fewer arguments and have fewer local variables
Data Structures 15-30% (Array, Stacks, Queues etc)	Looping constructs 2-6%	
	Conditional statements 35-45%	
	GOTO FEW Others 1-5%	

Figure 1 : Common Program Characteristics

*** (The data used above are approximate figures)**

Observations

- Integer constants appeared almost as frequently as arrays or structures.
- Most of the scalars were found to be local variables whereas most of the arrays or structures were global variables.

- Most of the dynamically called procedures pass lower than six arguments.
- The numbers of scalar variables are less than six.
- A good machine design should attempt to optimize the performance of most time consuming features of high-level programs.
- Performance can be improved by more register references rather than having more memory references.
- There should be an optimied instructional pipeline such that any change in flow of execution is taken care of.

RISC Architecture

Some important considerations of RISC architecture are :

1. The RISC functions are to be kept simple as possible until there is the requirement of complexity. If new operations need to be added then it should be evaluated that if execution time of an instruction increases by 10 per cent then the size of the code should reduce 10 per cent. This is required to balance the simplicity of the code.
2. A simple instruction may be executed at the same speed as that of a micro-instruction because micro-instructions stored in the control unit cannot be faster than simple instructions. This is due to the fact that the cache is built on the same technology as control unit.
3. The runtime library of RISC has all the characteristics of functions in microcode, except that it is easier to change. In general micro codes are difficult to change.
4. Pipelined execution gives a peak performance of one instruction every step. The longest step determines the performance rate of the pipelined machine, so ideally each pipeline step should take the same amount of time.
5. RISC compilers try to remove as much work as possible during compile time so that simple instructions can be used. For example, RISC compilers try to keep operands in registers so that simple register-to-register instructions can be used. RISC compilers keep operands that will be reused in register, rather than repeating a memory access or a calculation. They, therefore, use LOADs and STOREs to access memory so that operands are not implicitly discarded after being fetched.

Therefore, the RISC architecture were designed with the following features :

One instruction per cycle : A machine cycle is the time taken to fetch two operands from register, perform the ALU operation on them and store the result in a

register. Thus, RISC instruction execution takes about the same time as the microinstructions on CISC machines. With such simple instruction execution rather than micro-instructions, it can use fast logic circuits for control unit, thus increasing the execution efficiency further.

Register-to-register operands : In RISC machines the operation that access memories are LOAD and STORE. All other operands are kept in registers. This design feature simplifies the instruction set and, therefore, simplifies the control unit. For example, a RISC instruction set may include only one or two ADD instructions (e.g. integer add and add with carry); on the other hand a CISC machine can have 25 add instructions involving different addressing modes. Another benefit is that RISC encourages the optimization of register use, so that frequently used operands remain in registers.

Simple addressing modes : Another characteristic is the use of simple addressing modes. The RISC machines use simple register addressing having displacement and PC relative modes. More complex modes are synthesized in software from these simple ones. Again, this feature also simplifies the instruction set and the control unit.

Simple instruction formats : RISC uses simple instruction formats. Generally, only one or a few instruction formats are used. In such machines the instruction length is fixed and aligned on word boundaries. In addition, the field locations can also be fixed. Such an instruction format has a number of benefits. With fixed fields, opcode decoding and register operand accessing can occur in parallel. Such a design has many advantages. These are :

- It simplifies the control unit
- Simple fetching as memory words of equal size are to be fetched
- Instructions are not across page boundaries.

Thus, RISC is potentially a very strong architecture. It has high performance potential and can support VLSI implementation. Let us discuss these points in more detail.

Performance using optimizing compilers : As the instructions are simple the compilers can be developed for efficient code organization also maximizing register utilization etc. Sometimes even the part of the complex instruction can be executed during the compile time.

High performance of Instruction execution : While mapping of HLL to machine instruction the compiler favours relatively simple instructions. In addition, the control unit design is simple and it uses little or no micro-instructions, thus could

execute simple instructions faster than a comparable CISC. Simple instructions support better possibilities of using instruction pipelining.

VLSI Implementation of Control Unit : A major potential benefit of RISC is the VLSI implementation of microprocessor. The VLSI Technology has reduced the delays of transfer of information among CPU components that resulted in a microprocessor. The delays across chips are higher than delay within a chip; thus, it may be a good idea to have the rare functions built on a separate chip. RISC chips are designed with this consideration. In general, a typical microprocessor dedicates about half of its area to the control store in a micro-programmed control unit. The RISC chip devotes only about 6% of its area to the control unit. Another related issue is the time taken to design and implement a processor. A VLSI processor is difficult to develop, as the designer must perform circuit design, layout, and modeling at the device level. With reduced instruction set architecture, this processor is far easier to build.

The Use of Large Register File

The register storage is the faster storage device, faster than even the main memory and the cache. Thus, a strategy is needed that will allow the most frequently accessed operands to be kept in registers and to minimize register memory operations.

Two basic approaches are possible, one based on the software and the other based on the hardware, The software approach is to rely on the compiler to maximize register usage. The compiler will attempt to allocate the registers to those variables that will be used most in the given time period. This approach requires the use of sophisticated program-analysis algorithms. The hardware approach is simply to use more registers so that more variables can be held in registers for longer periods of time. RISC follows the hardware approach.

Since most operand references are to local scalars, the obvious approach is to store these in registers, with perhaps a few registers reserved for global variables. The problem is the definition of local changes with each procedure call and return, operations that occur frequently. On every call, local variables must be saved from the registers into memory, so that the registers can be reused by the called program. Furthermore, the parameters must be passed. On return, the variables of the parent program must be restored and the results must be passed back to the parent program.

RISC takes care of these with the help of register windows. Multiple small sets of registers are used, each assigned to different procedure. A procedure call automatically switches the CPU to use a different fixed size window of registers,

rather than saving registers in memory. Windows for adjacent procedures are overlapped to allow parameter passing.

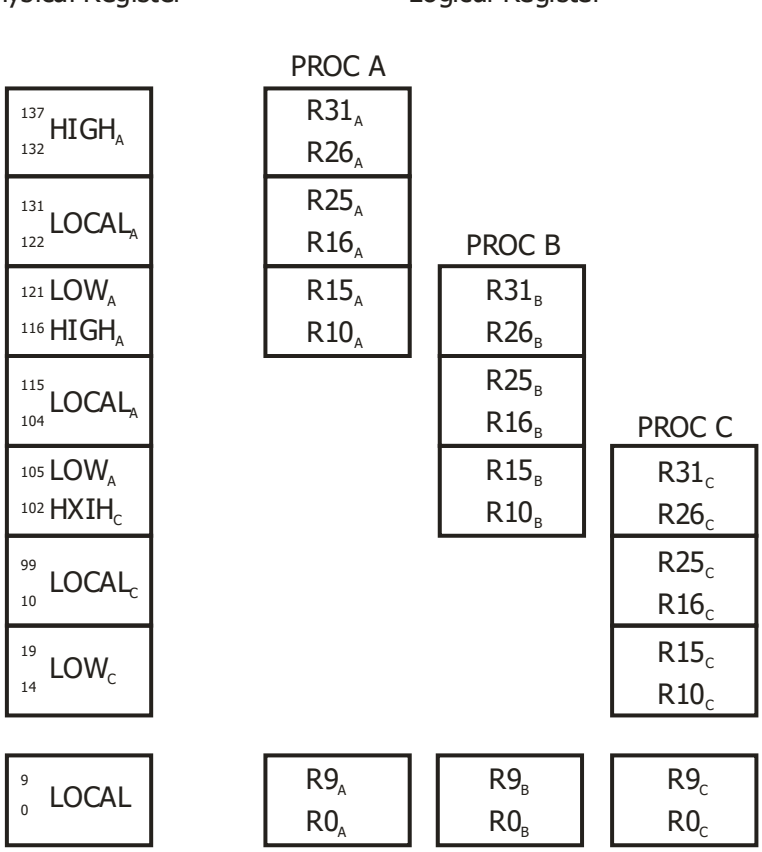


Figure 2 : Use of three overlapped register windows

Thus the register file, organized in the form as above, is small fast register buffer that holds most of the variables that are likely to be used heavily. From this point of view the register file acts almost like a caches memory.

Characteristics of large-register-file and cache organizations

Large Register File	Cache
Hold local variables for almost all functions. This saves time.	Recently used local variables are fetched from main memory for any further use. Dynamic use optimises memory.
The variables are individual.	The transfer from memory is block wise.
Global variables are assigned by the compilers.	It stores recently used variables. It cannot keep track of future use.
Save/restore needed only after the maximum call nesting is over (that is n–1 open windows).	Save/restore based on cache replacement algorithms.

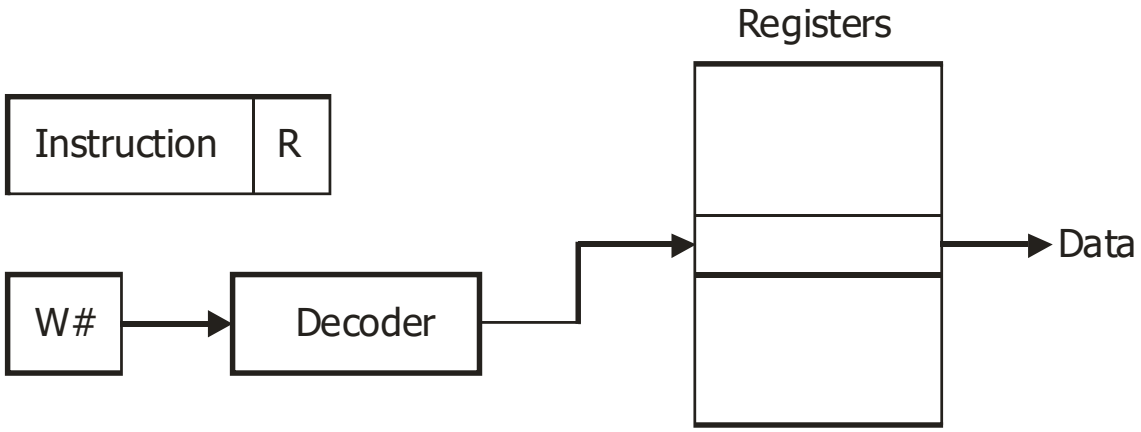
It follows faster register addressing	It is memory addressing.

Table 1 : Characteristics of Large-Register file and cache organisation

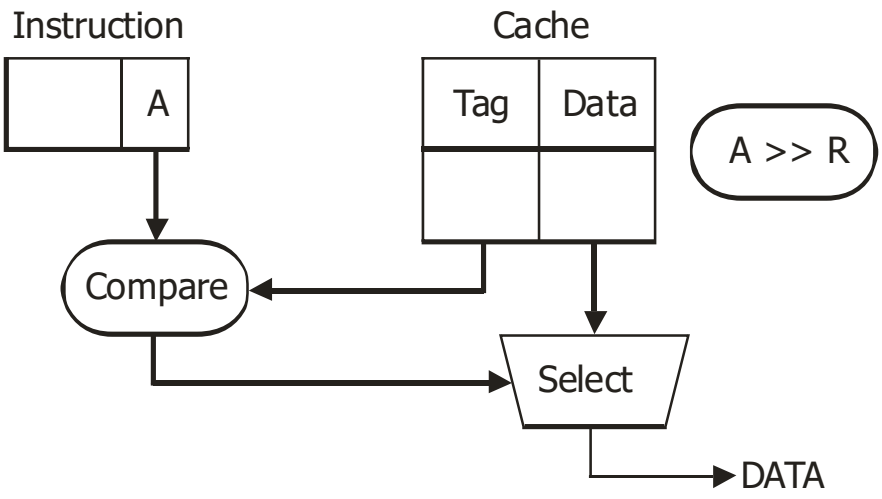
All points above basically show comparative equality. The basic difference is due to addressing overhead of the two approaches.

The following figure shows the difference. Small register (R) address is added with current window Pointer W#. This generates the address in register file, which is decoded by decoder for register access. On the other hand Cache reference will be generated from a long memory address, which first goes through comparison logic to ascertain the presence of data, and if the data is present it goes through the select circuit. Thus, for simple variables access register file is superior to cache memory.

However, even in RISC computer, performance can be enhanced by the addition of instruction cache.



(a) Windows based Register file



(b) Cache Reference

Figure 3 : Referencing a local Simple Variables

RISC Pipelining

Pipelining, a standard feature in RISC processors, is much like an assembly line. Because the processor works on different steps of the instruction at the same time more instructions can be executed in a shorter period of time. Different processors have different numbers of steps, they are basically variations of these five, used in the MIPS R3000 processor :

1. fetch instructions from memory
2. read registers and decode the instruction
3. execute the instruction or calculate an address
4. access an operand in data memory
5. write the result into a register

Pipelining is used for enhancing the overall performance. Let us consider Instruction in the context of RISC architecture. In RISC machines most of the operations are register. Therefore, the instructions can be executed in two phases :

F : Instruction Fetch to get the instruction.

E : Instruction Execute on register operands and store the results in register.

In general, the memory access in RISC is performed through LOAD and STORE operations. For such instructions the following steps may be needed :

F : Instruction Fetch to get the instruction

E : Effective address calculation for the desired memory operand

D : Memory to register or register to memory data transfer through bus.

Please note that the pipeline above is not running at its full capacity. This is because of the following problems :

- We are assuming a single port memory thus only one memory access is allowed at a time. Thus, Fetch and Data transfer operations cannot occur at the same time. Thus, you may notice blank in the time slot 3, 5 etc.
- The last instruction is an unconditional jump. Please note that after this instruction the next instruction of the calling program will be executed. Although not visible in this example a branch instruction interrupts the sequential flow of instruction. Thus, causing inefficiencies in the pipelined execution.

This pipeline can simply be improved by allowing two memory accesses at a time.

Pipeline optimization

In order to make processors even faster, various methods of optimizing pipelines have been devised.

Super-pipelining refer to dividing the pipeline into more steps. The more pipe stages there are, the faster the pipeline is because each stage is then shorter. Ideally, a pipeline with five stages should be five times faster than a non-pipelined processor (or rather, a pipeline with one stage). The instructions are executed at the speed at which each stage is completed, and each stage takes one fifth of the amount of time that the non-pipelined instruction takes. Thus, a processor with an 8-step pipeline (the MIPS R4000) will be even faster than its 5-step counterpart. The MIPS R4000 chops its pipeline into more pieces by dividing some steps into two. Instruction fetching, for example, is now done in two stages rather than one. The states are as shown :

1. Instruction Fetch (First Half)
2. Instruction Fetch (Second Half)
3. Register Fetch
4. Instruction Execute
5. Data Cache Access (First Half)
6. Data Cache Access (Second Half)
7. Tag Check
8. Write Back

Superscalar pipelining involves multiple pipelines in parallel. Internal components of the processor are replicated so it can launch multiple instructions in some or all of its pipeline stages. The RISC System/6000 has a forked pipeline with different paths for floating-point and integer instructions. If there is a mixture of both types in a program, the processor can keep both forks running simultaneously. Both types of instructions share two initial stages (Instruction Fetch and Instruction Dispatch) before they fork. Often, however, superscalar pipelining refers to multiple copies of all pipeline. Many of today's machines attempt to find two to six instructions that it can execute in every pipeline stage. If some of the instructions are dependent, however, only the first instruction or instructions are issued.

Dynamic pipelines have the capability to schedule around stalls. A dynamic pipeline is divided into three units: **the instruction fetch** and **decode unit,** five to

ten **execute or functional units,** and **a commit unit.** Each execute unit has reservation stations, which act as buffers and hold the operands and operations.

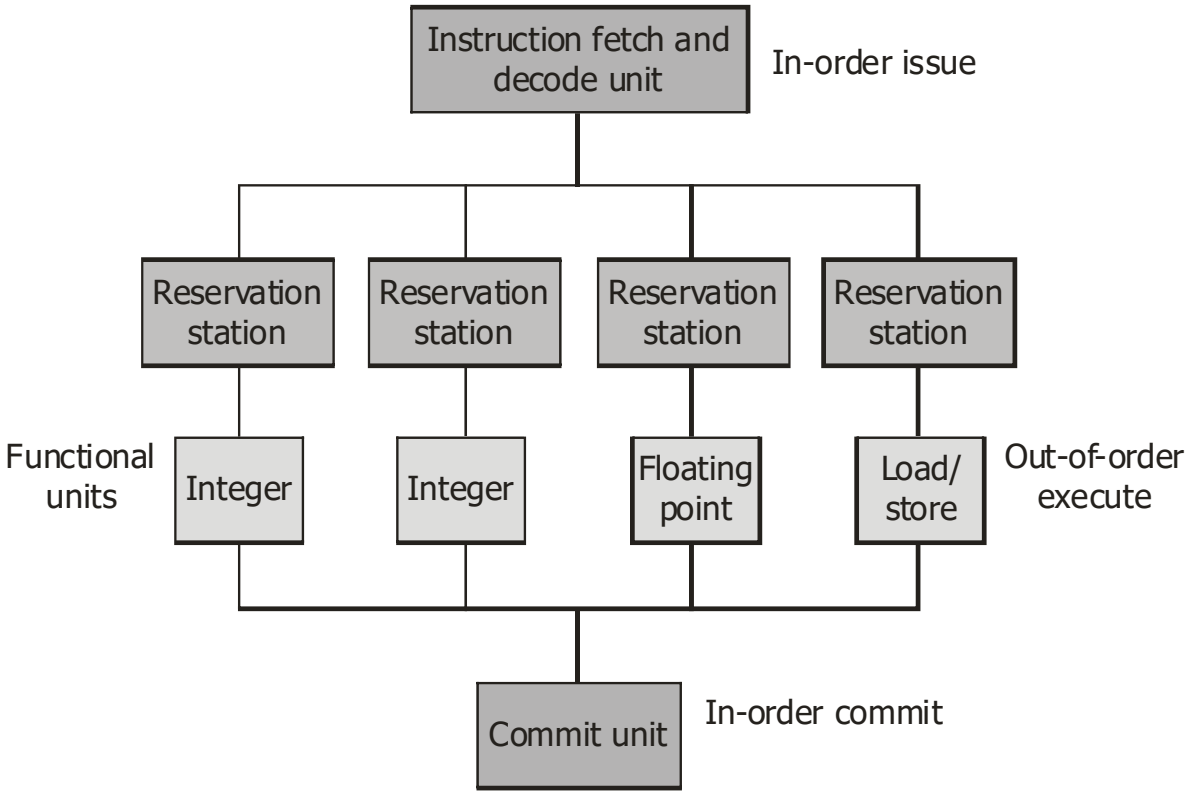


Figure 4 : Dynamic Pipelining

While the functional units have the freedom to execute out of order, the instruction fetch/decode and commit units must operate in-order to maintain simple pipeline behaviour. When the instruction is executed and the result is calculated, the commit unit decides when it is safe to store the result. If a stall occurs, the processor can schedule other instructions to be executed until the stall is resolved. This, coupled with the efficiency of multiple units executing instructions simultaneously, makes a dynamic pipeline an attractive alternative.

RISC machines can employ a very efficient pipeline scheme because of the simple and regular instructions. Like all other instruction pipelines RISC pipeline suffer from the problems of data dependencies and branching instructions. RISC optimizes this problem by using a technique called delayed branching.

One of the common techniques used to avoid branch penalty is to pre-fetch the branch destination also. RISC follows a branch optimization technique called delayed jump as shown in the example given below :

LOAD R1 ← M (1)	F	E	D

LOAD R2 ← M (2)		F	E	D				
SUB RS ← R1 – R2			F	E				
IF RS < 0 Return				F	E			
ADD RA ← R1 + R2					F	E		
STOR → M(A)						F	E	D
RETURN							F	E

Figure 5 : Delayed Branch

Questions

1. What is RISC ? Describe the importance of RISC processor.
2. What is the for Increased complexity ? Explain.
3. Explain some important considerations of RISC architecture.
4. What is Large Register File ? Describe the characteristics of Large Register file and cache organisation.
5. Explain RISC Pipelining with an example. What is delayed branching ?

• • •