UNIT 12 REDUCED INSTRUCTION SET COMPUTER ARCHITECTURE

Structure **Page No.**

12.0 INTRODUCTION

In the previous units, we have discussed the instruction set, register organization and pipelining, and control unit organization. The trend of those years was to have a large instruction set, a large number of addressing modes and about 16 –32 registers. However, their existed a pool of thought which was in favour of having simplicity in instruction set. This logic was mainly based on the type of the programs, which were being written for various machines. This led to the development of a new type of computers called Reduced Instruction Set Computer (RISC). In this unit, we will discuss about the RISC machines. Our emphasis will be on discussing the basic principles of RISC and its pipeline. You may refer to further readings for more details on this architecture.

12.1 OBJECTIVES

After going through this unit, you should be able to:

- define the reason of increasing complexity of instructions;
- explain the reasons for developing RISC;
- define the basic principles of RISC;
- describe the importance of having large register file;
- describe RISC pipelining.

12.2 INTRODUCTION TO RISC

Reduced Instruction set computer architectures, were initially designed to reduce the complexity of the instruction sets, which included very large number of instructions. The purpose was to design an instruction set that could be designed for better performance with no additional cost of the processor. In fact, the aim of computer processor chip architects has been to design processor chips, which are more powerful than their predecessors, yet are not expensive. Thus, the processor chip designer would like to:

- Optimise the hardware manufacturing cost.
- Optimises the cost for programming scalable/ portable architectures that require low costs for debugging the initial hardware and subsequent programs.

If you review the history of computer families, you will find that the most common architectural change is the trend towards even more complex machines.

12.2.1 Importance of RISC Processors

Reduced Instruction Set Computers recognise a relatively limited number of instructions in comparison to the complex instruction set computers. In addition, a RISC processor also has a limited number of addressing modes with most of the instruction using the register operands. Thus, the instructions of RISC processor are simpler, therefore, can be executed faster. Another advantage is that RISC chips cheaper to design and produce.

In general, an instruction on a RISC machine can be executed in one processor cycle, as RISC use instruction pipeline. As discussed in Unit 11, an instruction pipeline enhances the speed of instruction execution. In addition, the control unit of the RISC processor is simpler and smaller than Complex Instruction Set Computers (CISC). This saved space can be used for building additional registers in the processor. This further enhances the processing capabilities of the RISC processor. Most RISC processor uses the register operands, this necessitates that the memory to register "LOAD" and "STORE" are created as separate independent instructions, which use memory reference operation..

Various RISC Processors

RISC has fewer design bugs; its simple instructions reduce design time. Thus, because of all the above important reasons RISC processors have become popular. Some of the uses if RISC processors are in the mobile processors, desktops, workstation and embedded devices. One of the open instructions set architecture for RISC instruction set is RISC V. This architecture uses the principles of RISC, which are discussed in this unit. For more details, you may refer to the further readings.

12.2.2 Reasons for Increased Complexity

The complexity of computer chips kept growing with the advancement in technology. In this section, we discuss the reasons for increased complexity of instruction set of computers.

Speed of Memory Versus Speed of CPU

In the past, there existed a large gap between the speed of a processor and memory. Thus, an execution of instruction using a program, 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 sequences. Thus, a "higher level" instruction can be added to machines in an attempt to improve performance.

However, this assumption is not very valid in the present era where 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. Let us explain it with the help of an example:

Suppose the floating-point operation ADD A, B requires the following steps (assuming the machine does not have floating point registers) and the registers being used for exponent are E1, E2, and EO (output); for mantissa M1, M2 and MO (output):

- Load the exponent of A in E1

- Load the mantissa of A in M1
- Load the exponent of B in E2
- Load the mantissa of B in M2
- Compare E1 and E2

```
    If  E1 = E2 then MO ← M1 + M2 and EO ← E1
        -   Normalise MO and adjust EO
        -   Result will be contained in MO, E1
    else if E1 < E2 then find the difference = E2 – E1
        -   Shift Right M1, by difference
        -   MO ← M1 + M2 and EO ← E2
        -   Normalise MO and adjust EO
        -   Result is contained in MO, EO
    else E2 < E1, if so find the difference = E1 – E2
        -   Shift Right M2 by difference above
        -   MO ← M1 + M2 and EO ← E1
        -   Normalise MO and adjust E1 into EO
        -   Result is contained in MO, EO
```

- Move the mantissa and exponent of the results to A
- Checks overflow underflow if any.

If all these steps are coded as one machine instruction, then this simple instruction will require several instruction cycles. If this instruction is made a part of the machine instruction set architecture as an instruction: ADDF A, B (Add floating point numbers A and B and store results in A), then it will just be a single machine instruction. All the above steps required will then be coded with the help of micro-operations in the form of Control Unit Micro-Program. Thus, just one instruction cycle (although a long one) may be needed. This cycle will require just one instruction fetch. Whereas in the program memory instructions will be fetched.

However, faster cache memory for Instruction and data stored in registers can create an almost similar instruction execution environment. Pipelining can further enhance such speed. Thus, creating an instruction as above may not result in faster execution.

Microcode and VLSI Technology

It is considered that the control unit of a computer be constructed using two ways; create micro-program that execute micro-instructions or build circuits 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 micro-instructions 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. However, such a mechanism may result in slightly slower execution of commonly used instructions.

Code Density and Smaller Faster Programs

The memory was very 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, more complex instruction sets were designed, 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. Fewer instructions mean fewer instruction bytes to be fetched. But this does not ensures that program written for CISC machines would be smaller in size than that of programs written for RISC machine. It may be possible that a CISC program is smaller in number of instructions, yet the overall size, in terms of number of bytes, may not be small. This may result from the reason that in RISC we use register addressing and less instruction, which require fewer bits in general. In addition, you may please note that even the compliers on CISC machine favours simpler instructions. Let us explain this with the help of the following example:

Assume a CISC machine has a 4GB byte addressable RAM (2^{32}) and 32 registers (2^5). A machine instruction consists of two operands, one of which should be a register operand. Almost similar RISC machine having the same size of RAM and active registers. Further, the CISC machine uses 16 bit to represent opcode and addressing modes and the RISC machine uses 8 bit to represent opcode and addressing modes. Figure 12.1(a) shows ADD and MOV instructions for the CISC machine. On the other hand, RISC machine would have at least two instruction formats (First to load the data from RAM to register or store the register to memory; or addition operation on register. Figure 12.2 (b) shows these two instruction formats for RISC machine.

16	5	32	
ADD	R1	A	; R1←R1+[A]
ADD	A	R1	; [A]←R1+[A]
MOV	R1	A	; R1← [A]
MOV	A	R1	; [A]←R1

(a) Sample instructions of CISC

8	5	32	
LDA	R1	A	; R1← [A]
STR	R1	A	; [A] ←R1

8	5	5	
ADD	R1	R2	; R1← R1+R2

(b) A sample Load, Store and ADD instruction of RISC

Figure 12.1: Sample machine instructions

Figure 12.1 shows the instructions for a CISC and RISC machine. The size of CISC ADD instruction is 53 bits, therefore, it will be stored in 7 bytes or 56 bits. The load (LDA) and store (STR) instructions of RISC machines are 45 bits long, so would be stored in 6 bytes. In addition, in RISC machine the ADD instruction would use 18 bits or 3 bytes. Consider the following sequence of operations on these two machines:

C=A+B

16	5	32	
MOV	R1	A	; R1← [A]
ADD	R1	B	; R1←R1+[A]
MOV	C	R1	; [C]← R1

Program segment size = 7 × 3 = 21 Bytes

Size in bits = 53 × 3 = 159 bits

(a) Segment to compute C=A+B using sample CISC ISA

8	5	32	
LDA	R1	A	; R1← [A]
LDA	R2	B	; R2← [B]
ADD	R1	R2	; R1←R1+R2 (18 bits)
STR	R1	C	; [C] ←R1

Program segment size = 6 × 3 + 3= 21 Bytes

Size in bits = 45 × 3 + 18 = 153 bits

(b) Segment to compute C=A+B using sample RISC ISA

Figure 12.2: Execution of C=A+B on hypothetical machines

So, the expectation that a CISC will produce smaller programs may not be correct. In addition, in the present time memory is inexpensive, this potential advantage of smaller programs is not so compelling these days.

Support for High-Level Language

With the increasing use of more and higher-level languages, manufacturers 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 compilers 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 instruction 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. However, because of the bias of programmers towards the use of simpler instructions, it may turn out otherwise. CISC makes the more complex control unit with larger microprogram 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. This has led a number of groups to pursue the opposite path.

12.2.3 High Level Language Program Characteristics

The new architectures should support high-level language programming. 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 were:

Variables	Operations	Procedure Calls
Integral Constants 15-25% Scalar Variables 50-60% Array/ structure 20-30%	Simple assignment 35-45% Looping 2-6% Procedure call 10-15% IF 35-45% GOTO FEW Others 1-5%	Most time-consuming operation. FACTS: Most of the procedures are called with fewer than 6 arguments. Most of these have fewer than 6 local variables

Figure 12.3: Typical Program Characteristics

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 optimized instructional pipeline such that any change in flow of execution is taken care of.

The Origin of RISC

In the 1980s, a new philosophy evolved having optimizing compilers that could be used to compile "normal" programming languages down to instructions that were as simple as equivalent micro-operations in a large virtual address space. This made the instruction cycle time as fast as the technology would permit. These machines should have simple instructions such that it can harness the potential of simple instruction execution in one cycle – thus, having reduced instruction sets – hence the reduced instruction set computers (RISCs).

Check Your Progress 1

1. List the reasons of increased complexity.
   ....
   ....
   ....
2. State True or False

	T	F
a) The instruction cycle time for RISC is equivalent to CISC.		
b) CISC yields smaller programs than RISC, which improves its performance; therefore, it is very superior to RISC.		
c) CISC emphasizes optional use of register while RISC does not.		

12.3 RISC ARCHITECTURE

Let us first list some important considerations of RISC architecture:

1. The RISC functions are kept simple unless there is a very good reason to do otherwise. A new operation that increases execution time of an instruction by 10 per cent can be added only if it reduces the size of the code by at least 10 per cent. Even greater reductions might be necessary if the extra modification necessitates a change in design.
2. Micro-instructions stored in the control unit cannot be faster than simple instructions, as the cache is built from the same technology as writable control unit store, a simple instruction may be executed at the same speed as that of a micro-instruction.
3. Microcode is not magic. Moving software into microcode does not make it better; it just makes it harder to change. The runtime library of RISC has all the characteristics of functions in microcode, except that it is easier to change.

4. Simple decoding and pipelined execution are more important than program size. 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. Compiler should simplify instructions rather than generate complex instructions. 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 registers, rather than repeating a memory access or a calculation. They use LOADs and STOREs to access memory so that operands are not implicitly discarded after being fetched. (Refer to Figure 12.1(b) for a simple illustration).

Thus, the RISC were designed having the following:

- **One instruction per cycle**: A machine cycle is the time taken to fetch two operands from registers, perform the ALU operation on them and store the result in a register. Thus, RISC instruction execution takes about the same time as the micro-instructions 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.

12.4 THE USE OF LARGE REGISTER FILE

In general, the register storage is faster than the main memory and the cache. Also the register addressing uses much shorter addresses than the addresses for main memory and the cache. However, the numbers of registers in a machine are less as generally the same chip contains the ALU and control unit. Thus, a strategy is needed that will optimize the register use and, thus, allow the most frequently accessed operands to be kept in registers in order to minimize register-memory operations.

Such optimisation can either be entrusted to an optimising complier, which requires techniques for program analysis; or we can follow some hardware related techniques. The hardware approach will require the use of more registers so that more variables can be held in registers for longer periods of time. This technique is used in RISC machines.

It may seem that a large number of registers would lead to fewer memory accesses, however in general, about 32 registers were considered optimum. So how does this large register file further optimize the program execution?

Since most operand references are to local variables of a function in C they are the obvious choice for storing in registers. Some registers can also be used for global variables. However, the problem here is that the program follows function call - return so the local variables are related to most recent local function, in addition this call - return expects saving the context of calling program and return address. This also requires parameter passing on call. On return, from a call the variables of the calling program must be restored and the results must be passed back to the calling program.

RISC register file provides a support for such call-returns with the help of register windows. Register files are broken into an overlapping set of smaller group of registers, as shown in Figure 12.4. Each of these register set can be used for different function/subroutine. A function call automatically changes the set being used. The use from one fixed size window of registers to another, rather than saving registers in memory as done in CISC. Windows for adjacent procedures are overlapped. This feature allows parameter passing without moving the variables at all. The following figure tries to explain this concept:

Assumptions:

Register file contains 138 registers. Let them be called by register number 0 – 137. Further, a program has three functions, viz. main, sorting and Xchange. The operating system calls function main (f_{main}) which calls function sorting ($f_{sorting}$) and function sorting calls function Xchage (f_{Xchage}).

Registers Nos.	Used for			
0 – 9	Global variables required by f_{main}, $f_{sorting}$, and f_{Xchage}	Function main	Function sorting	Function Xchage
10 – 83	Unused			
84 – 89 (6 Registers)	Used by parameters of f_{Xchage} that may be passed to next call			Temporary variables of function Xchage
90 – 99 (10 Registers)	Used for local variable of f_{Xchage}			Local variables of function Xchage
100 – 105 (6 Registers)	Used by parameters that were passed from $f_{sorting} \rightarrow f_{Xchage}$		Temporary variables of function sorting	Parameters of function Xchage
106 – 115 (10 Registers)	Local variables of $f_{sorting}$		Local variables of function sorting	
116 – 121 (6 Registers)	Parameters that were passed from f_{main} to $f_{sorting}$	Temporary variables of function main	Parameters of function sorting	
122 – 131 (10 Registers)	Local variable of f_{main}	Local variables of function main		
132 – 138 (6 Registers)	Parameter passed to f_{main}	Parameters of function main		

Figure 12.4: Use of three Overlapped Register Windows

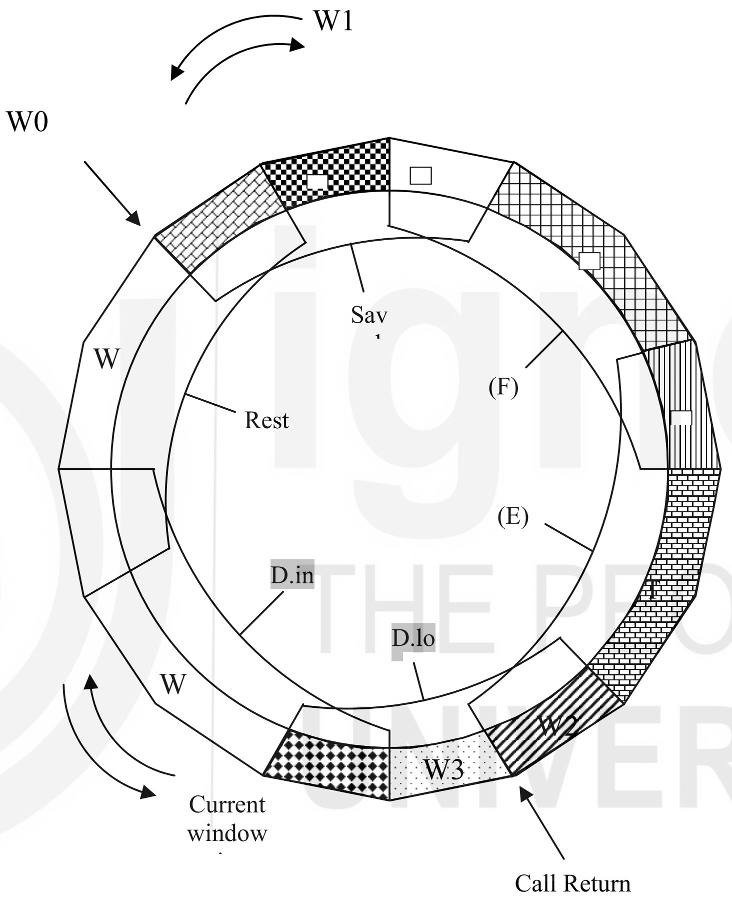
Functioning of the registers: at any point of time the global registers and one set of register being used for a specific function would be active for execution of the program. Thus, for programming purpose there may be only 32 registers. Window in the above example although has a total of 138 registers. This window consists of:

- Global registers which are shareable by all functions.
- Parameters registers for holding parameters passed from the previous function to the current function. They also hold the results that are to be passed back.
- Local registers that hold the local variables, as assigned by the compiler.
- Temporary registers: The temporary registers are used to pass parameters to the function that is called by the presently executing function. The parameters to passed are stored in these temporary registers and passed as parameters to the function that is called by the presently executing function.

But what is the maximum function calls nesting can be allowed through RISC? Let us describe it with the help of a circular buffer diagram, technically the registers as above have to be circular in the call return hierarchy.

This organization is shown in the following figure. The register buffer is filled as function A called function B, function B called function C, function C called function

D. The function D is the current function. The current window pointer (CWP) points to the register window of the most recent function (function D in this case). Any register references by a machine instruction is added with the contents of CWP to compute the address of the registers holding the operands for the executing function. The other register, i.e., the saved window pointer registers points to the most recent register window that has be saved in the memory of the computer. This action will be needed if a further call is made and there is no space for that call. If function D now calls function E arguments for function E are placed in D's temporary registers indicated by D temp and the CWP is advanced by one window.



A.in:	Input register parameters/argument of function A
A.loc:	Local variables of function A
B.in or A.temp	Parameters with which function B is to be called B.in It is same as A.temp which are parameters passed by function A to function B

Figure 12.5: Circular-Buffer Organization of Overlapped Windows

Assume that now the function E calls function F. This call cannot be serviced as the circular buffer already has allowed number of function call, unless you free space equivalent to exactly one window. This condition can easily be determined as current window pointer on incrementing will be equal to saved window pointer. Now, we need to create space; how can we do it? The simplest way will be to swap F_A register to memory and use that space. Thus, an N window register file can support N –1 level of function calls.

Thus, the register file, organized in the form as above, is a 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 cache memory. So let us find how the two approaches are different.

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.

The basic difference is due to addressing overhead of the two approaches. Small Register (R) address are smaller than the Cache reference, which are generated from a long memory address. 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.

Check Your Progress 2

1. State True or False in the context of RISC architecture:

   a. RISC has a large register file so that more variables can be stored in register or longer periods of time.

   b. Only global variables are stored in registers.

   c. Variables are passed as parameters in registers using temporary registers in a window.

   d. Cache is superior to a large register file as it stores most recently used local scalars.

2. An overlapped register window RISC machine is having 32 registers. Suppose 8 of these registers are dedicated to global variables and the remaining 24 are split for incoming parameters, local and scalar variables and outgoing parameters. What are the ways of allocating these 24 registers in the three categories?
   ....
   ....
   ....

12.5 COMMENTS ON RISC

Let us now try and answer some of the comments that are asked for RISC architectures. Let us provide our suggestions on those.

CISCs provide better support for high-level languages as they include high-level language constructs such as CASE, CALL etc.

Yes CISC architecture tries to narrow the gap between assembly and High-Level Language (HLL); however, this support comes at a cost. If the architect provides a feature that looks like the HLL construct but runs slowly, or has many options, the compiler writer may omit the feature, or even, the HLL programmer may avoid the construct, as it is slow and cumbersome. Thus, the comment above does not hold.

It is more difficult to write a compiler for a RISC than a CISC.

The studies have shown that it is not so due to the following reasons:

If an instruction can be executed in more ways than one, then more cases must be considered. For it the compiler writer needed to balance the speed of the compilers to get good code. In CISCs compilers need to analyze the potential usage of all available instruction, which is time consuming. Thus, it is recommended that there is at least one good way to do something. In RISC, there are few choices; for example, if an operand is in memory, it must first be loaded into a register. Thus, RISC requires simple case analysis, which means a simple compiler, although more machine instructions will be generated in each case.

RISC is tailored for C language and will not work well with other high level languages.

But the studies of other high-level languages found that the most frequently executed operations in other languages are also the same as simple HLL constructs found in C, for which RISC has been optimized. Unless a HLL changes the paradigm of programming you may get similar result.

The good performance is due to the overlapped register windows; the reduced instruction set has nothing to do with it.

Certainly, a major portion of the speed is due to the overlapped register windows of the RISC that provide support for function calls. However, please note this register windows are possible due to reduction in control unit size. In addition, the control is simple in RISC than CISC, thus further helping the simple instructions to execute faster.

12.6 RISC PIPELINING

Instruction pipelining is often used to enhance performance. A RISC machine, in general, consists of two types of instructions:

a) The memory reference instructions, which are load and store instructions (Figure 12.1(b)). These instructions are used to bring/send data in registers from/to memory.

b) Data processing instructions (ADD instruction in Figure 12.1(b)), which perform the operation on register operands.

A memory reference instruction may be divided into the following pipeline stages:

FI: Fetch the Instruction from a memory address.
EI: Compute the effective address of the operand in the memory using the addressing modes. This may be similar to execution of an instruction.
TD: Transfer data from/to register to/from memory location

The data processing instruction would just require two pipeline stages:
FI: Fetch the Instruction from a memory address.

EI: Execute the instruction on register operands, result is stored in register.

Let us explain pipelining in RISC with an example program that uses instruction set given in Figure 12.1(b), with few additional instruction MUL, which use the same format as ADD instruction. The program segment implements the following expression:

Z = (A + B) × C

(1)	LDA R1, A	(Load memory location A to R1)
(2)	LDA R2, B	(Load memory location B to R2)
(3)	ADD R1, R2	(R1 ← R1 + R2)
(4)	LDA R2, C	(Load memory location C to R2)
(5)	MUL R1, R2	(R1 ← R1 × R2)
(6)	STR R1, Z	(Store result in in memory location Z)

As discussed earlier, each of the instructions (1), (2), (4) and (6) will be processed in three stages and each of the instructions (3) and (5) will be processed in two stages. Assuming that one stage is executed in one clock cycle, a total of 4×3+2×2=16 clock cycles would be required if these instructions are not executed without using an instruction pipeline. However, if a pipeline that allows various overlapping stages to execute in parallel would result in execution of these instructions in only 8 clock cycles (Please refer to Figure 12.6)

(1)	LDA R1, A	FI	EI	TD					
(2)	LDA R2, B		FI	EI	TD				
(3)	ADD R1, R2			FI		EI			
(4)	LDA R2, C				FI	EI	TD		
(5)	MUL R1, R2					FI		EI	
(6)	STR R1, Z						FI	EI	TD
	Clock Cycles	1	2	3	4	5	6	7	8
	Time = 8 units								

Figure 12.6: RISC Pipelining

You may please note that the pipeline as shown above suffers from data dependencies at instruction (3) and (5). In both these cases, the preceding data access instruction must complete to allow the execution of these instructions. In addition, an instruction pipeline may suffer due to presence of branch instruction penalties. Next, we discuss about how such problems can be minimized

Optimization of Pipelining

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. The data dependency problem can be handled using an optimizing compiler, which can reschedule some of the instructions. For example, in the given program segment the, following changes may minimize the data dependency:

Interchange instruction (3) and instruction (4), in addition, instead of loading memory location C in R2 use R3 register. Accordingly, in instruction (5) change R2 to R3. The new instruction pipeline is shown in Figure 12.7.

(1)	LDA R1, A	FI	EI	TD					
(2)	LDA R2, B		FI	EI	TD				
(4)	*LDA* ***R3****, C*			FI	EI	TD			
(3)	*ADD R1, R2*				FI	EI			
(5)	MUL R1, **R3**					FI	EI		
(6)	STR R1, Z						FI	EI	TD
	Clock Cycles	1	2	3	4	5	6	7	8
	Time = 8 units								

Figure 12.7: Pipeline without data dependencies

The second problem of Branch instruction penalty can be optimised in RISC by using several techniques. For example, consider that a conditional branch instruction: "In case after the computation, the value R1 register is zero, then instruction 6 is not executed and the program jumps to instruction 7" exists after instruction 5. This modified instruction sequence is shown in Figure 12.8.

(1)	LDA R1, A	FI	EI	TD						
(2)	LDA R2, B		FI	EI	TD					
(4)	*LDA* ***R3****, C*			FI	EI	TD				
(3)	*ADD R1, R2*				FI	EI				
(5.a)	MUL R1, **R3**					FI	EI			
(5.b)	If R1=0 JMP to (7)						FI	EI		
(6)	STR R1, Z							FI	EI	TD
(7)	ADD R2, R3								FI	EI
	Clock Cycles	1	2	3	4	5	6	7	8	9

Figure 12.8: Pipeline with Brach Instruction

The problem with this instruction cycle is that the instruction 6 has already been fetched to the pipeline, therefore, in case R1 has zero value the pipeline should be emptied, i.e., instruction 6 will be removed from the pipeline. After that the instruction (7) should be stated again. There are two possible solutions to this problem. First the fetch of instruction (6) may be delayed for a cycle, so that the decision, whether the branch is to be taken or not would be made. Based on that the next instruction would be fetched. This is shown in Figure 12.9.

(1)	LDA R1, A	FI	EI	TD							
(2)	LDA R2, B		FI	EI	TD						
(4)	LDA R3, C			FI	EI	TD					
(3)	ADD R1, R2				FI	EI					
(5.a)	MUL R1, R3					FI	EI				
(5.b)	If R1=0 JMP to (7)						FI	EI			
(5.c)	*DO NOTHING instruction*							FI	EI		
(6) or (7)	STR R1, Z ADD R2, R3								FI FI	EI EI	TD
	Clock Cycles	1	2	3	4	5	6	7	8	9	10

Figure 12.9: Pipeline with reduced Brach penalty

Another way to handle the branch penalty is by moving the branch instruction, so that the branch decision is known prior to the execution of next instruction after the branch instruction. Figure 12.10 shows this solution.

(1)	LDA R1, A	FI	EI	TD						
(2)	LDA R2, B		FI	EI	TD					
(4)	LDA R3, C			FI	EI	TD				
(3)	ADD R1, R2				FI	EI				
(5.b)	*If R1=0 or R3 = 0 JMP to (7)*					FI	EI			
(5.a)	*MUL R1, R3*						FI	EI		
(6) or (7)	STR R1, Z ADD R2, R3							FI FI	EI EI	TD
	Clock Cycles	1	2	3	4	5	6	7	8	9

Figure 12.10: Pipeline with optimised Brach penalty

Please note that the instruction at (5.b) shows a hypothetical instruction that checks two conditions at a time. The purpose here is to demonstrate the concept, therefore, such instruction has been shown. Please also note the change in the sequence of instructions (5.a) and (5.b). Please also note that decision to take the branch has been taken at clock cycle 6, therefore, at lock cycle 7, it will be known, which of the two instructions (6) or (7) is to be fetched.

Finally, let us summarize the basic differences between CISC and RISC architecture. The following table lists these differences:

CISC	RISC
1. In general, large number of instructions	1. Relatively fewer instructions than CISC
2. Employs a variety of data types and a large number of addressing modes.	2. Relatively fewer addressing modes.
3. Variable-length instruction formats.	3. Fixed-length instructions, easy to decode instruction format.
4. Instructions manipulate operands residing in memory.	4. Mostly register-register operations. The only memory access is through explicit load and store instructions.
5. Number of Cycles Per Instruction varies from 1-20 depending upon the instruction.	5. Number of cycles per instruction is one as it uses pipelining. Pipeline in RISC is optimised because of simple instructions and instruction formats.
6. About 32 general purpose register, but no support is available for the parameter passing and function calls.	6. Large number of registers, which are used as Global registers and as a register based procedural call and parameter passing.
7. Microprogrammed Control Unit.	7. Hardwired Control Unit.

Check Your Progress 3

1. What are the problems, which prevent RISC pipelining to achieve maximum speed?
   ....
   ....
   ....
2. How can the above problems be handled?

....

....

....

3. What are the problems of RISC architecture? How are these problems compensated such that there is no reduction in performance?

   ....

   ....

   ....

12.7 SUMMARY

RISC represents new styles of computers that take less time to build yet provide a higher performance. While traditional machines support HLLs with instruction that look like HLL constructs, this machine supports the use of HLLs with instructions that HLL compilers can use efficiently. The loss of complexity has not reduced RISC's functionality; the chosen subset, especially when combined with the register window scheme, emulates more complex machines. Thus, we see that because of all the features discussed above, the RISC architecture should prove to better for certain applications.

In this unit we have also covered the details of the pipelined features of the RISC architecture, which support this architecture to show better performance.

12.8 SOLUTIONS/ ANSWERS

Check Your Progress 1

1.
   - Speed of memory is slower than the speed of CPU.
   - Microcode implementation is cost effective and easy.
   - The intention of reducing code size.
   - For providing support for high-level language.

2.
   a) False
   b) False
   c) False

Check Your Progress 2

1.
   (a) True
   (b) False
   (c) True
   (d) False

2. Assume that the number of incoming parameters is equal to the number of outgoing parameters.

   Therefore, Number of locals = 24 –(2 × Number of incoming parameters)

   Return address is also counted as a parameter, therefore, number of incoming parameters is more than or equal to 1 or in other terms the possible combination, are:

C.loc

Incoming Parameter Registers	Outgoing Parameter Registers	No. of Local Registers
1	1	22
2	2	20
3	3	18
4	4	16
5	5	14
6	6	12
7	7	10
8	8	8
9	9	6
10	10	4
11	11	2
12	12	0

Check Your Progress 3

1. The following are the problems:
   - Branch instruction
   - The data dependencies between the instructions

2. It can be improved by:
   - Changing the sequence of some instruciton
   - causing optimized/ delayed jumps/loads etc.

3. The problems of RISC architecture are:

   - More instructions to achieve the same amount of work as CISC.
   - Higher instruction traffic
   - However, the cycle time of one instruction per cycle and instruction cache in the chip may compensate for these problems.