1.

RISC	CISC
It is Reduced Instruction Set Computer. It has a limited and simple instruction set.	It is Complex Instruction Set Computer. It has a complex instruction set to provide support for High Level Languages
It emphasis more on software to optimize the performance	It emphasis more on the hardware to optimize the performance
It requires multiple register sets to store the instruction. It uses multiple registers and compiler technology to optimize register usage	It requires single register set to store the instruction. It ease the task of compile writer.
It requires less time to execute the instruction.	It requires more time to execute the system.
The addressing modes are less	The addressing modes are more
It has a hard-wired unit of programming.	It has a micro-programming unit.
Use of pipeline is simple in RISC	Use of Pipeline is complex in CISC
It has a fixed format instruction	It has a variable format instruction
No indirect addressing and hence it has no memory unit and uses separate hardware to implement instructions.	It has a memory unit to implement complex instructions.
RISC architecture is used in high-end applications such as video processing, telecommunications, and image processing.	CISC architecture is used in low-end applications such as security systems, home automation, etc
Examples of RISC are ARM, PA-RISC, SPARC etc,	Examples of CISC are VAX, AMD, Intel x86 etc.
It uses LOAD-STORE that are independent in register-register program interaction	It uses LOAD-STORE in the memory - memory interaction of program.

2.

The register storage is a fastest available storage device, faster to access than the main memory and cache. The register file is physically present on the same chip as the ALU and control unit, and employs much shorter addresses than the addresses for cache and main memory. Thus, a strategy is needed that will allow the most frequently accessed operands to be kept in registers and to minimize register-memory operations. A large set of registers decrease access to memory. Most operand references are to local scalars, if we store these local scalars in the registers we can reduce the access time. With each procedure call and return operations that occur frequently local changes are made. If we store the local values in the registers instead of the main memory the procedure call can reuse these local variables and thus increase performance. A typical procedure employs only a few passed parameters and local variables, the depth of procedure activation fluctuates within a

relatively narrow range. To take advantage of this we need multiple small sets of registers, each assigned to a different procedure.

Register Pros:
Access time will be less in registers
Registers hold all the local scalar variables.
Register file contains only those variables in use.
To reference a local scalar in a window based register file, a virtual register number and a window number are used. These can pass through a relatively simple decoder to select one of the physical registers.

Register Cons:
A register file may make inefficient use of space, because not all procedures will need the full window space allotted to them.
Registers doesn't treat all the memory references alike.
If register file is supplemented with global registers, it can hold some global scalars. when program modules are separately compiled, it is impossible for the compiler to assign global values to registers; the linker must perform this task.

Cache Pros:
Cache may make more efficient use of space, because it is reacting to the situation dynamically.
Caches generally treat all memory references alike, including instructions and other types of data. Thus, savings in these other areas are possible with a cache.
The cache is capable of handling global as well as local variables. There are usually many global scalars, but only a few of them are used, cache will dynamically discover these variables and hold them.

Cache Cons:
Data are read into the cache in blocks, which may not be completely used.
Cache Hold only the recently used scalar variables.
To reference a memory location in cache, a full-width memory address must be generated. The complexity of this operation depends on the addressing mode.

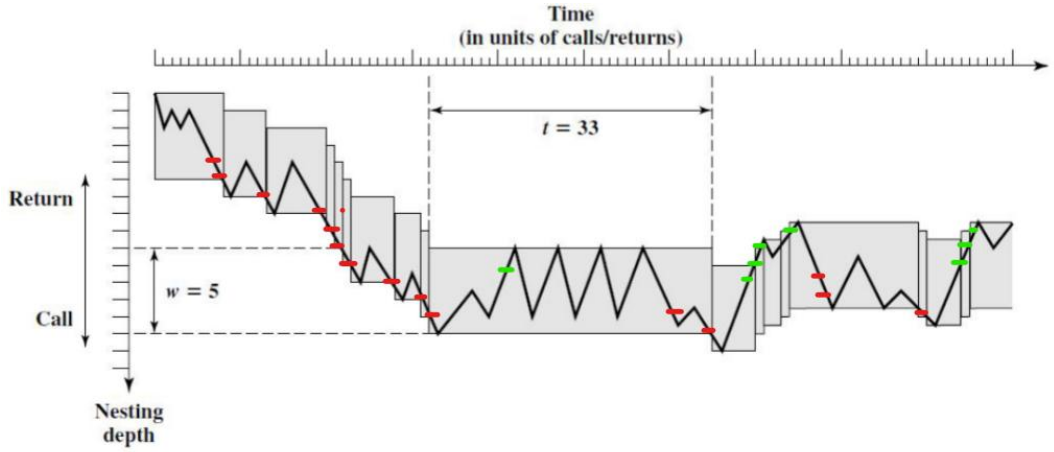
3.

The Weighted relative dynamic frequency of dynamic occurrence of most frequently occurring instruction is given in the table below

Instruction	Dynamic Occurrence	
	Pascal	C
ASSIGN	45%	38%
LOOP	5%	3%
CALL	15%	12%
IF	29%	43%
GOTO	-	3%
OTHER	6%	1%

4.

For Window size 4:

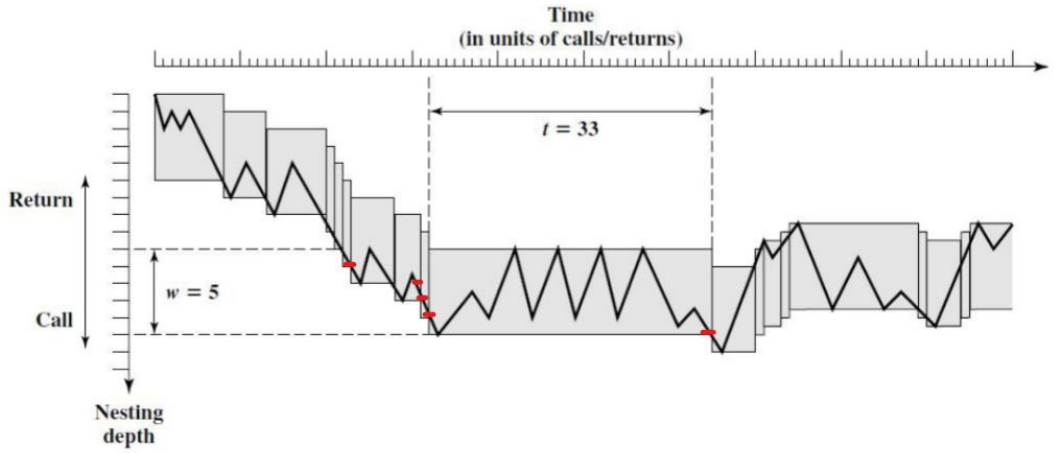


The red dash represents overflows and the green dash represents underflow.
The y co-ordinates of the start and end of the window will be as follows

0-4 O
1-5 O
2-6 O
3-7 O
4-8 O
5-9 O
6-10 O
7-11 O
8-12 O
9-13 O
10-14 U
9-13 O
10-14 O
11-15 U
10-14 U
9-13 U
8-12 U
7-11 O
8-12 O
9-13 O
8-12 U
7-11 U
6-10 U

U = underflow, O = 0verflow
There are total 15 Overflows and 8 Underflow

For Window size 10:



The red dash represents overflows and the green dash represents underflow.
The y co-ordinates of the start and end of the window will be as follows

0-10 O
1-11 O
2-12 O
3-13 O
4-14 O

U = underflow, O = 0verflow
There are total 5 Overflows and 0 Underflow

Window Size	Overflows	Underflows	Total
4	15	8	23
10	5	0	5

5.

Basically a window is divided into three fixed-sized areas. It has the Parameter register that holds the parameters passed down from the procedure that called the current procedure. Local registers are used for local variables. Temporary registers are used to exchange parameters and results with the next procedure call. The temporary registers at one level is physically the same as the parameter register at the next lower level. It is for this reason we can only store the first two portions of the window, that is parameter register and local register. The temporary register is available as the parameter register at the next level hence it need not be saved to avoid miscalculations and save memory. The programmer doesn't have access to the temporary registers and hence a programmer cannot manipulate the temporary register values, so storing it will be of no use to the programmer as well.
The organization of register files is a circular buffer of overlapping windows. The Current Window Pointer**(CWP)** points to the window of the currently active procedure. Register references by a machine instruction are offset by this pointer to determine the actual physical register. The Saved Window Pointer **(SWP)** identifies the window most recently saved in memory. If we consider a circular buffer with 6 windows and assume that the buffer is filled until a depth 4. The current active procedure will be D which is stored in current window pointer. If D call E then the parameters to pass to E are stored in D's temporary register and passed to parameter register of E, and hence the CWP is incremented by 1. Now the CWP is pointed to E. In the next call E calls procedure F, the call cannot be made with the

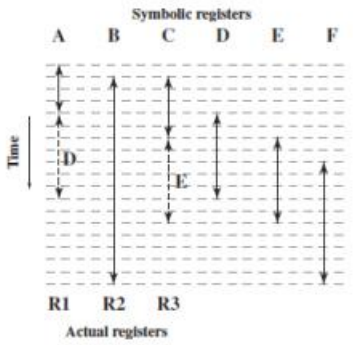
current status of the buffer since F's window overlaps A's window. If F begins to load its temporary registers, preparatory to a call, it will overwrite the parameter registers of A. Thus, when CWP is incremented so that it becomes equal to SWP, an interrupt occurs, and A's window is saved. Then, the SWP is incremented and the call to F proceeds. A similar interrupt can occur on returns, when B returns to A, CWP is decremented and becomes equal to SWP. This causes an interrupt that results in the restoration of A's window.

6.

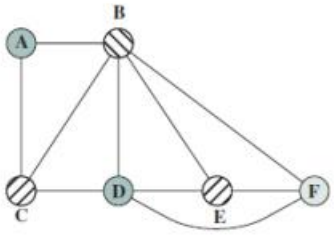
If we have only a small number of register files on the target machine, a program written in high level language will not have explicit reference to the register. In such cases it is the responsibility of the compiler to make the optimized use of the registers available. The objective of the compiler is to keep the operands for as many computations as possible in registers rather than main memory, and to minimize load-and-store operations. The compiler assigns each program quantity that is a candidate for residing in a register to a symbolic or virtual register. The compiler then maps the unlimited number of symbolic registers into a fixed number of real registers. Symbolic registers whose usage does not overlap can share the same real register. If there are more quantities to deal with than real registers, then some of the quantities are assigned to memory locations. Load-and-store instructions are used to position quantities in registers temporarily for computational operations.

The technique most commonly used in RISC compilers is known as graph coloring. This is used by the register interference graph. The technique does the following, Given a graph consisting of nodes and edges, assign colors to nodes such that adjacent nodes have different colors, and do this in such a way as to minimize the number of different colors. The compiler adapts this as follows. First, the program is analyzed to build a register interference graph. The nodes of the graph are the symbolic registers. If two symbolic registers are live during the same program fragment, then they are joined by an edge to depict interference. An attempt is then made to color the graph with *n* colors, where *n* is the number of registers. Nodes that share the same color can be assigned to the same register. If this process does not fully succeed, then those nodes that cannot be colored must be placed in memory, and loads and stores must be used to make space for the affected quantities when they are needed.

For Example assume a program has 6 symbolic registers to be compiled into three actual registers. The time sequence of active use of the register is given by the image



The register interference graph would be as follows



A possible coloring with three colors is indicated. Because symbolic registers A and D do not interfere, the compile can assign both of these to physical register R1. Similarly, symbolic registers C and E can be assigned to register R3. One symbolic register, F, is left uncolored and must be dealt with using loads and stores.

7.

RISC architecture can achieve the maximum speedup by implementing the Pipelining structure and dealing with branch instructions. For a hardware designer it is easy to implement a fast Pipelining structure given the simple nature of RISC. However branch dependencies reduce the overall execution rate, to compensate for these dependencies, code reorganization techniques have been developed. To deal with branch dependencies we use Delayed Branch technique. Delayed Branch is a way of increasing the efficiency of the pipeline. It makes use of a branch that does not take effect until after execution of the following instruction. The instruction location immediately following the branch is referred to as the *delay slot. Example for the same is as follows*

Normal Branch :

100 LOAD X,rA
101 ADD 1,rA
102 JUMP 105
103 ADD rA,rB
104 SUB rC,rB
105 STORE rA,Z

Using Delayed Branch:

100 LOAD X,rA
101 ADD 1,rA
102 JUMP 106
103 NOOP
104 ADD rA,rB
105 SUB rC,rB
106 STORE rA,Z

Using Optimized Delayed BRanch:

100 LOAD X,rA
101 JUMP 105
102 ADD 1,rA
103 ADD rA,rB
104 SUB rC,rB
105 STORE rA,Z

The following figures can be used to understand the pipelining and delayed branch exectutions.

Traditional Pipeline execution:

		1	2	3	4	5	6	7	8
100	*LOAD X,rA*	*I*	*E*	*D*					
101	*ADD 1,rA*		*I*		*E*				
102	*JUMP 105*				*I*	*E*			
103	*ADD rA,rB*					*I*	*E*		
105	*STORE rA,Z*						*I*	*E*	*D*

RISC Pipeline with inserted NOOP:

		1	2	3	4	5	6	7	8
100	*LOAD X,rA*	*I*	*E*	*D*					
101	*ADD 1,rA*		*I*		*E*				
102	*JUMP 106*				*I*	*E*			
103	*NOOP*					*I*	*E*		
106	*STORE rA,Z*						*I*	*E*	*D*

Reversed Instruction:

		1	2	3	4	5	6
100	*LOAD X,Ar*	*I*	*E*	*D*			
101	*JUMP 105*		*I*	*E*			
102	*ADD rA,rB*			*I*	*E*		
105	*STORE rA,Z*				*I*	*E*	*D*

If we have a look at the above figure in the Traditional pipeline the JUMP instruction is fetched at time 4 and executed at time 5. At the same time the instruction 103 is fetched. Since a JUMP occurred, which updates the program counter, the pipeline must clear the instruction 103 and at time 6, instruction 105, which is the target of the JUMP, is loaded. For the RISC Pipeline, The timing is the same. However, because of the insertion of the NOOP instruction, we do not need special circuitry to clear the pipeline; the NOOP simply executes with no effect. Now in the case of the reversed instruction where the JUMP instruction is swapped with the ADD instruction. The JUMP instruction is fetched before the ADD instruction at time 2, and the ADD instruction is fetched at time 3. The ADD instruction is fetched before the execution of the JUMP instruction has a chance to alter the program counter. The ADD instruction is executed at the same time that instruction 105 is fetched, that is at time 4. The original semantics of the program are retained but two fewer clock cycles are required for execution. Thus using a reversed instruction is an advantage and improving the speed. This interchange of instructions will work successfully for unconditional branches, calls, and returns. For conditional branches, this procedure cannot be blindly applied.