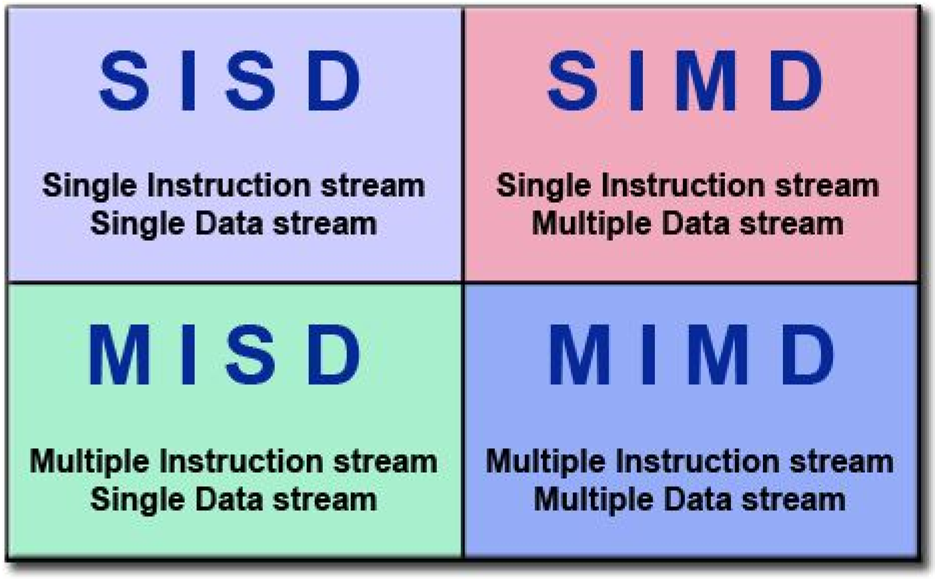
PRN : 1403046 FINAL YEAR BTECH CSE

Parallel Programming lab

SEVENTH SEMESTER 2017 LAB EXPERIMENT 1

INSTRUCTOR: prof. P.R.Gawli

Flynn's Classical Taxonomy

- There are different ways to classify parallel computers.
- One of the more widely used classifications, in use since 1966, is called Flynn's Taxonomy.
- Flynn's taxonomy distinguishes multi-processor computer architectures according to how they can be classified along the two independent dimensions of Instruction Stream and Data Stream. Each of these dimensions can have only one of two possible states: Single or Multiple.
- The matrix below defines the 4 possible classifications according to Flynn:
- 

S I S D Single Instruction stream Single Data stream	**S I M D** Single Instruction stream Multiple Data stream
M I S D Multiple Instruction stream Single Data stream	**M I M D** Multiple Instruction stream Multiple Data stream

Single Instruction, Single Data (SISD):

- A serial (non-parallel) computer
- **Single Instruction**: Only one instruction stream is being acted on by the CPU during any one clock cycle
- **Single Data**: Only one data stream is being used as input during any one clock cycle
- Deterministic execution
- This is the oldest type of computer
- Examples: older generation mainframes, minicomputers, workstations and single processor/core PCs

Single Instruction, Multiple Data (SIMD):

A type of parallel computer

- **Single Instruction**: All processing units execute the same instruction at any given clock cycle
- **Multiple Data**: Each processing unit can operate on a different data element
- Best suited for specialized problems characterized by a high degree of regularity, such as graphics/image processing.
- Synchronous (lockstep) and deterministic execution
- Two varieties: Processor Arrays and Vector Pipelines
- Examples:
- Processor Arrays: Thinking Machines CM-2, MasPar MP-1 & MP-2, ILLIAC IV

- Vector Pipelines: IBM 9000, Cray X-MP, Y-MP & C90, Fujitsu VP, NEC SX-2, Hitachi S820, ETA10

Most modern computers, particularly those with graphics processor units (GPUs) employ SIMD instructions and execution units.

Multiple Instruction, Single Data (MISD):

A type of parallel computer

- **Multiple Instruction:** Each processing unit operates on the data independently via separate instruction streams.
- **Single Data:** A single data stream is fed into multiple processing units.
- Few (if any) actual examples of this class of parallel computer have ever existed.
- Some conceivable uses might be:
- multiple frequency filters operating on a single signal stream
- multiple cryptographic algorithms attempting to crack a single coded message.

Multiple Instruction, Multiple Data (MIMD):

A type of parallel computer

- **Multiple Instruction**: Every processor may be executing a different instruction stream
- **Multiple Data**: Every processor may be working with a different data stream
- Execution can be synchronous or asynchronous, deterministic or non-deterministic
- Currently, the most common type of parallel computer - most modern supercomputers fall into this category.
- Examples: most current supercomputers, networked parallel computer clusters and "grids", multi-processor SMP computers, multi-core PCs.
- Note: many MIMD architectures also include SIMD execution sub-components

Multiprocessing , Multitasking and Multiprogramming

Multiprogramming - A computer running more than one program at a time (like running Excel and Firefox simultaneously).

Multiprocessing - A computer using more than one CPU at a time.

Multiprocessing refers actually to the CPU units rather than running processes. If the underlying hardware provides more than one processor then that is multiprocessing. There are many variations on the basic scheme for example having multiple cores on one die or multiple dies in one package or multiple packages in one system. In summary, multiprocessing refers to the underlying hardware (multiple CPUs, Cores) while multiprogramming refers to the software (multiple programs, processes). Note that a system can be both multi-programmed by having multiple programs running at the same time and multiprocessing by having more than one physical processor.

Multitasking - Tasks sharing a common resource (like 1 CPU).

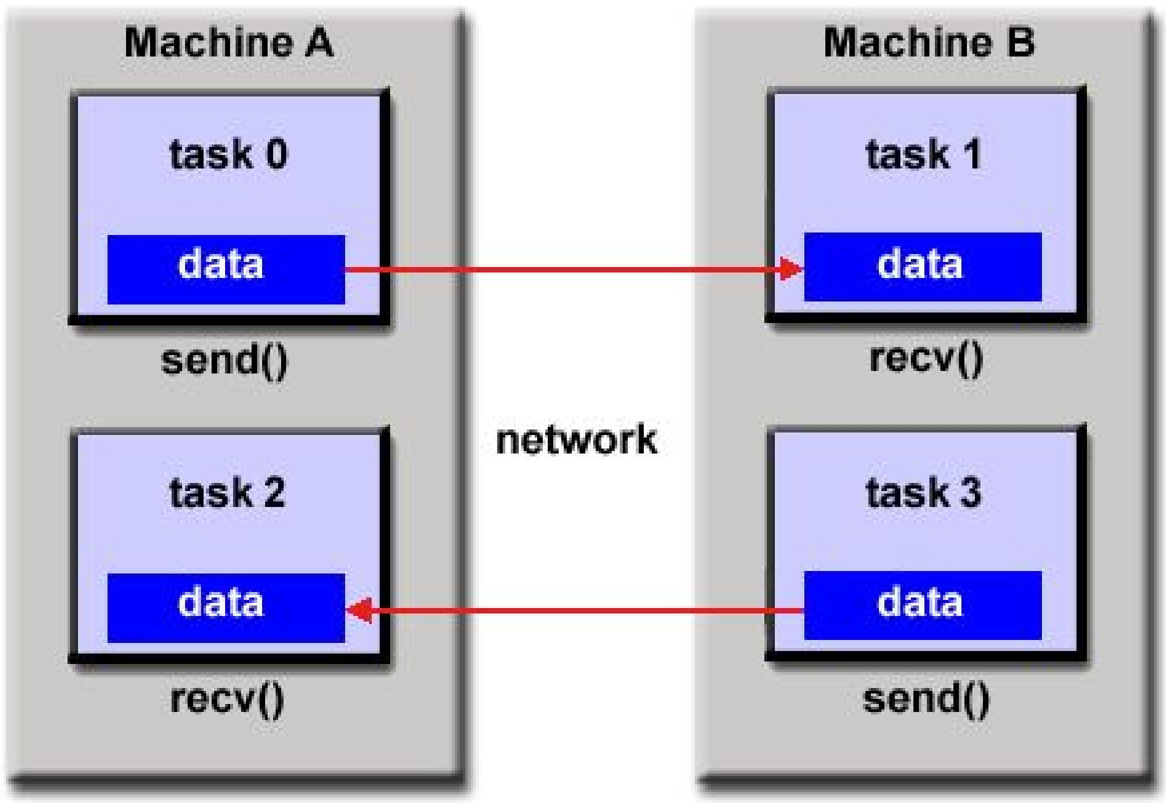
Multitasking is the term used in modern operating systems when multiple tasks share a common processing resource (CPU and Memory). At any point in time the CPU is executing one task only while other tasks waiting their turn. The illusion of parallelism is achieved when the CPU is reassigned to another task (context switch). There are few main differences between multitasking and multiprogramming (based on the definition provided in this article). A task in a multitasking operating system is not a whole application program (recall that programs in modern operating systems are divided into logical pages). Task can also refer to a thread of execution when one process is divided into sub tasks (will talk about multi threading later). The task does not hijack the CPU until it finishes like in the older multiprogramming model but rather have a fair share amount of the CPU time called quantum (will talk about time sharing later in this article). Just to make it easy to remember, multitasking and multiprogramming refer to a similar concept (sharing CPU time) where one is used in modern operating systems while the other is used in older operating systems.

Multi threading is an execution model that allows a single process to have multiple code segments (threads) run concurrently within the context of that process. You can think of threads as child processes that share the parent process resources but execute independently. Multiple threads of a single process can share the CPU in a single CPU system or (purely) run in parallel in a multiprocessing system. A multitasking system can have multi threaded processes where different processes share the CPU and at the same time each has its own threads.

- Thus, something like multithreading is an extension of multitasking.

Distributed Memory / Message Passing Model

- **This model demonstrates the following characteristics:**



  - A set of tasks that use their own local memory during computation. Multiple tasks can reside on the same physical machine and/or across an arbitrary number of machines.
  - Tasks exchange data through communications by sending and receiving messages.
  - Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation.

Implementations:

- From a programming perspective, message passing implementations usually comprise a library of subroutines. Calls to these subroutines are embedded in source code. The programmer is responsible for determining all parallelism.
- Historically, a variety of message passing libraries have been available since the 1980s. These implementations differed substantially from each other making it difficult for programmers to develop portable applications.
- In 1992, the MPI Forum was formed with the primary goal of establishing a standard interface for message passing implementations.
- Part 1 of the Message Passing Interface (MPI) was released in 1994. Part 2 (MPI-2) was released in 1996 and MPI-3 in 2012. All MPI specifications are available on the web at http://www.mpi-forum.org/docs/.
- MPI is the "de facto" industry standard for message passing, replacing virtually all other message passing implementations used for production work. MPI implementations exist for virtually all popular parallel computing platforms. Not all implementations include everything in MPI-1, MPI-2 or MPI-3.

Parallel programming

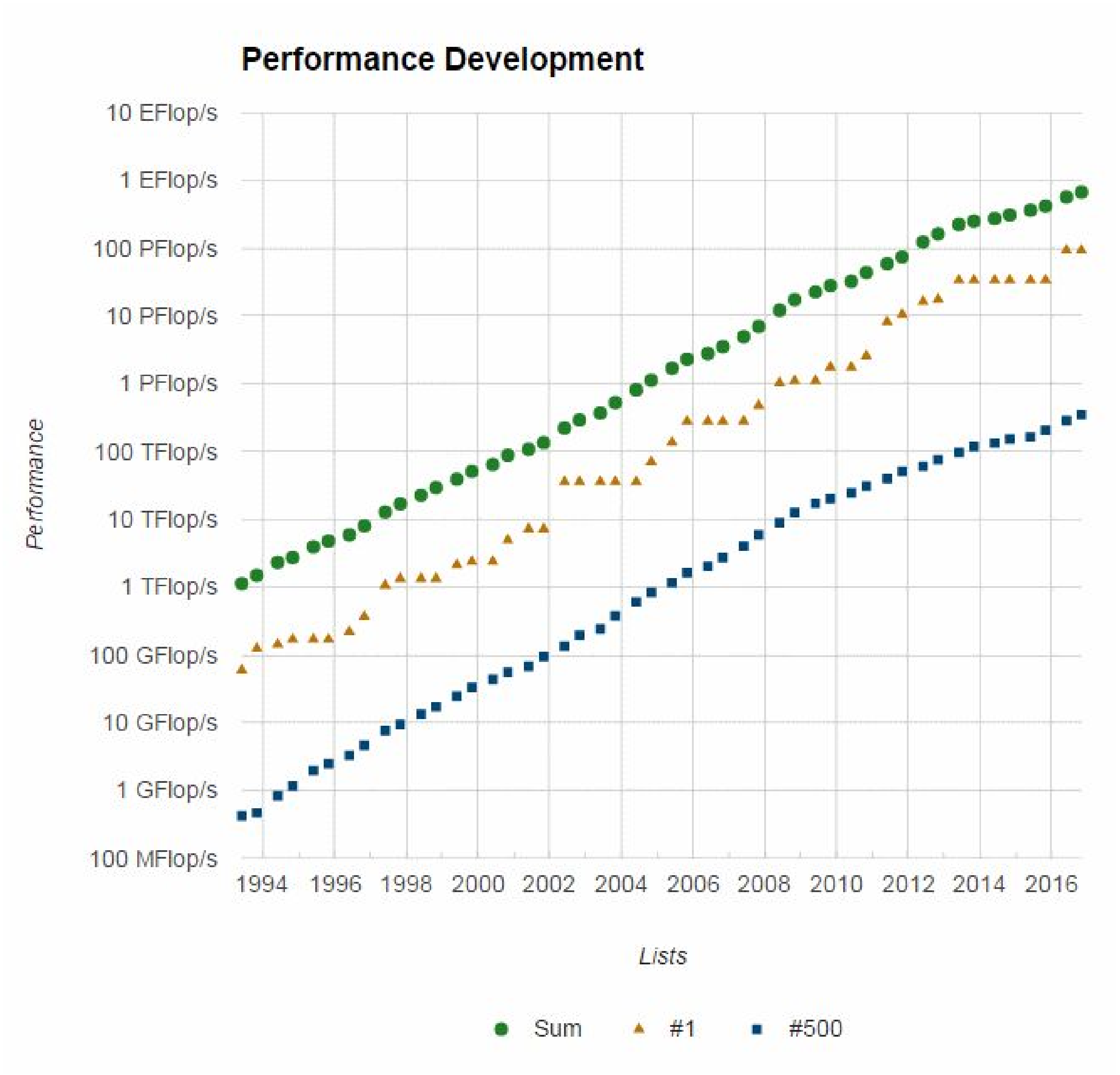
Parallel computing is a type of computation in which many calculations or the execution of processes are carried out simultaneously. Large problems can often be divided into smaller ones, which can then be solved at the same time.

- **Why parallel programming is used?**
    - Save Time And/Or Money.
    - Solve Larger / More Complex Problems.
    - Provide Concurrency.
    - Take Advantage Of Non-local Resources.
    - Make Better Use Of Underlying Parallel Hardware.

The Future:

- During the past 20+ years, the trends indicated by ever faster networks, distributed systems, and multiprocessor computer architectures (even at the desktop level) clearly show that parallelism is the future of computing.

- In this same time period, there has been a greater than 500,000x increase in supercomputer performance, with no end currently in sight.
- The race is already on for Exascale Computing!
    - Exaflop = 10^{18} calculations per second



Source: [Top500.org]

Sequential programming

It is clear that the final values of the variables in the example program depend on the order that statements are executed in. In general, given the same input data, a sequential program will always execute the same sequence of instructions and it will always produce the same results. Sequential program execution is deterministic.

The sequential paradigm has the following two characteristics:

- the textual order of statements specifies their order of execution ,successive statements must be executed without any overlap (in time) with one another.
- Neither of these properties applies to concurrent programs.

Difference between Sequential and Parallel Programming

Parallel **programming** involves the concurrent computation or simultaneous execution of processes or threads at the same time.

sequential programming, processes are run one after another in a succession fashion while in parallel computing, you have multiple processes execute at the same time.

It is much easier to sequentialize parallel code than it is to parallelise sequential code.

Process, Program and Thread

- A standalone piece of code which is not under execution is called a **program**. A single **Program** can have multiple **process.**
- A **process**, in the simplest terms, is an executing **program**. One or more **threads** run in the context of the **process**.
- A **thread** is the basic unit to which the operating system allocates processor time. A **thread** can execute any part of the **process** code, including parts currently being executed by another **thread**.

RISC(reduced instruction set computing) Vs CISC(complex instruction set computing)

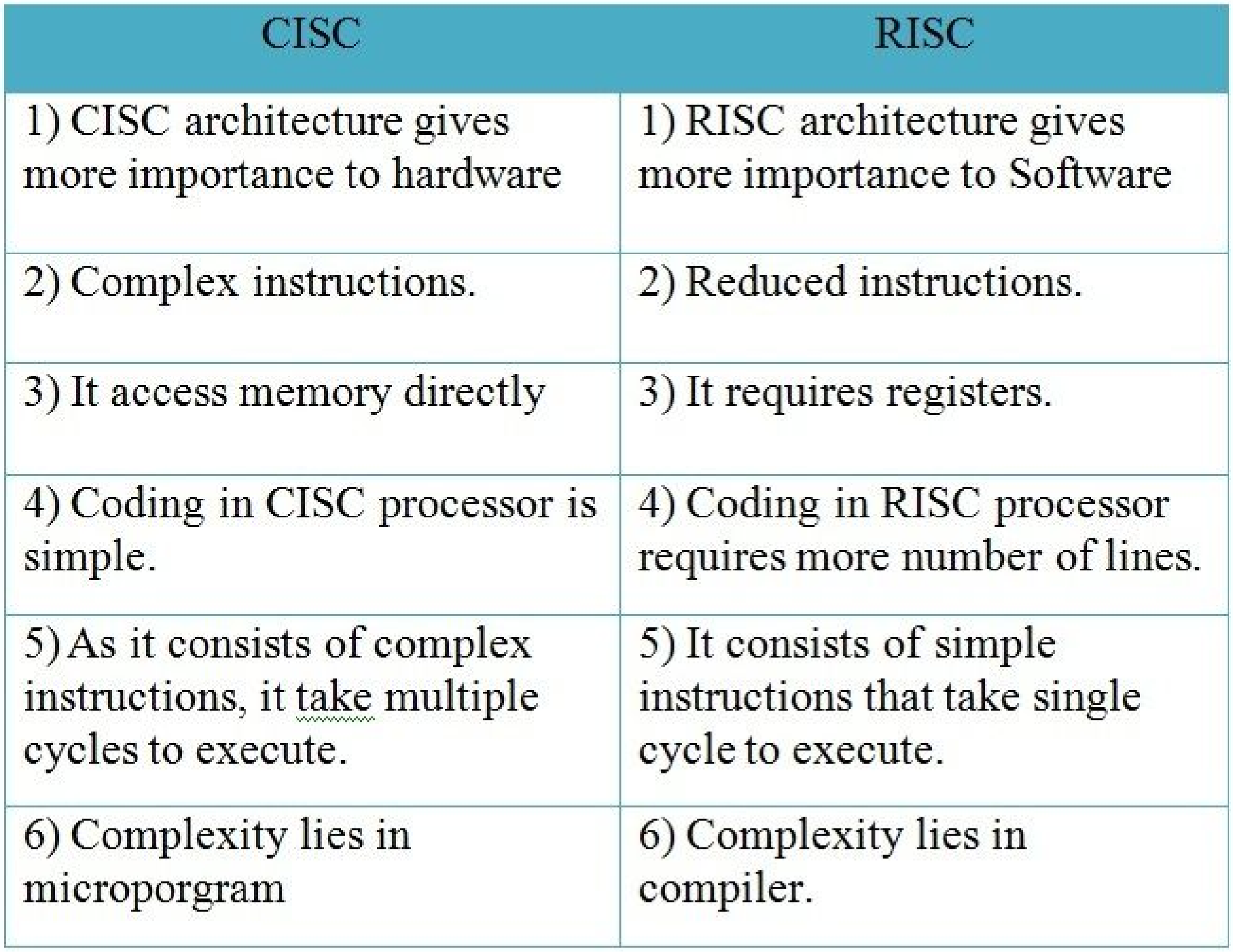
CISC:- In the early days machines were programmed in assembly language and the memory access is also slow. To calculate complex arithmetic operations, compilers have to create long sequence of machine code.

This made the designers to build an architecture , which access memory less frequently and reduce burden to compiler. Thus this lead to very powerful but complex instruction set.

RISC:- Although CISC reduces usage of memory and compiler, it requires more complex hardware to implement the complex instructions.

In RISC architecture, the instruction set of processor is simplified to reduce the execution time. It uses small and highly optimized set of instructions which are generally register to register operations.

The speed of the execution is increased by using smaller number of instructions .This uses pipeline technique for execution of any instruction.



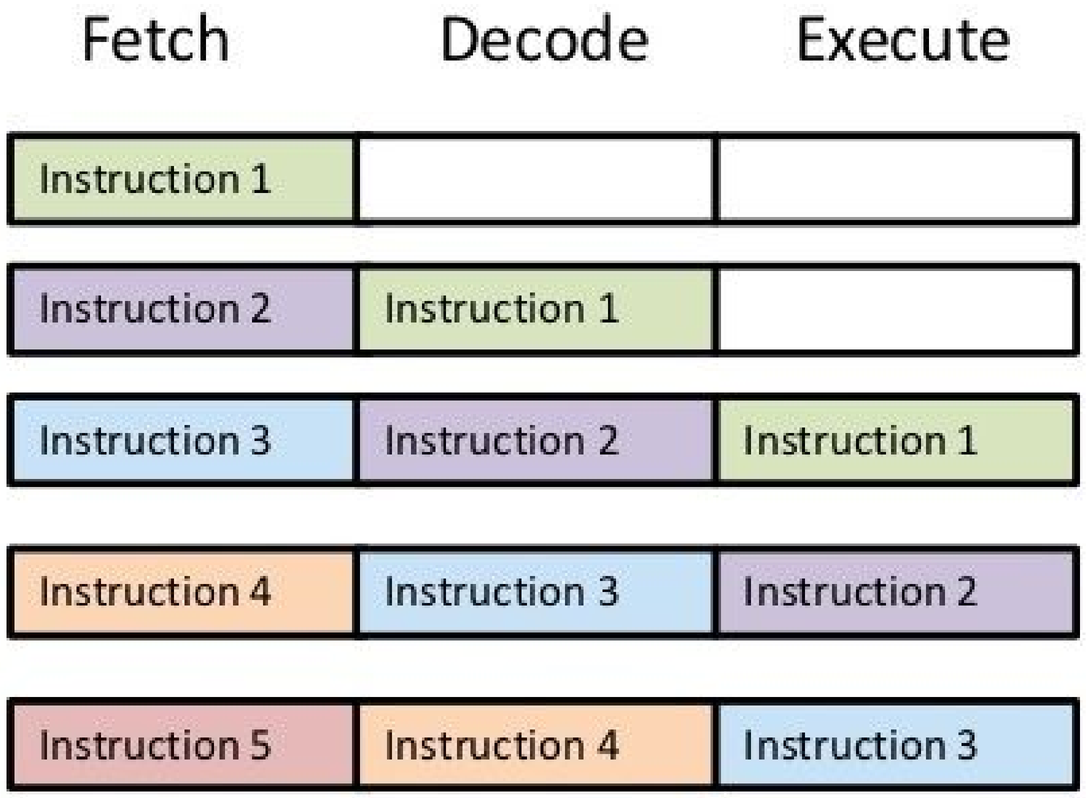
CISC	RISC
1) CISC architecture gives more importance to hardware	1) RISC architecture gives more importance to Software
2) Complex instructions.	2) Reduced instructions.
3) It access memory directly	3) It requires registers.
4) Coding in CISC processor is simple.	4) Coding in RISC processor requires more number of lines.
5) As it consists of complex instructions, it take multiple cycles to execute.	5) It consists of simple instructions that take single cycle to execute.
6) Complexity lies in microporgram	6) Complexity lies in compiler.

Pipeline Architecture:-

The pipelining technique allows the processor to work on different steps of instruction like fetch, decode and execute instructions at the same time. Below is image showing execution of instructions in pipelining technique.

Generally, execution of second instruction is started, only after the completion of the first instruction. But in pipeline technique, each instruction is executed in number of stages simultaneously.

When the first stage of first instruction is completed, next instruction is enters into the first stage. This process continues until all the instructions are executed.



Steps in Instruction Execution by CPU:

Six steps are involved in execution an instruction by CPU. However, not each of them are required for any instructions.

Step 1: Fetch instruction

Execution cycle starts with fetching instruction from main memory. The instruction at the current program counter (PC) will be fetched and will be stored in instruction register (IR).

Step 2: Decode instruction

During this cycle the encoded instruction present in the IR (instruction register) is interpreted by the decoder.

Step 3: Perform ALU operation

ALU (Arithmetic Logic Unit) is where two operands in the instruction will be operated on given operator in the instructions. Such as, if the instruction was to add two numbers, then here the addition will happen. ALU take two values and output one, the result of the operation.

Step 4: Access memory

There are only two kind of instructions that access memory: LOAD and STORE. LOAD copies a value from memory to a register and STORE copies a register value to memory. Any other instruction skips this step.

Step 5: Update Register File

In this step, the output/result of the ALU is written back to the register file to update the register file. The result could also be due to a LOAD from memory. Some instructions don't have results to store. For example, BRANCH and JUMP instructions do not have any results to store.

Step 6: Update the PC (Program Counter)

Ultimately, at the end of the execution of the current instruction, we need to update the program counter (PC) to the address of the next instruction, so that we can go back to step 1 where the CPU will fetch instruction. However, the program counter might need to be set to other memory address than the next one if the instruction was BRANCH or JUMP

Shared memory and Distributed memory

Shared memory and distributed memory are low-level programming abstractions that are used with certain types of parallel programming. Shared memory allows multiple processing elements to share the same location in memory (that is to see each others reads and writes) without any other special directives, while distributed memory requires explicit commands to transfer data from one processing element to another.

In the past there were true shared memory cache-coherent multiprocessor systems. The systems communicated with each other and with shared main memory over a shared bus. This meant that any access from any processor to main memory would have equal latency. Today these types of systems are not manufactured. Instead there are various point-to-point links between processing elements and memory elements (this is the reason for non-uniform memory access, or NUMA).

Distributed memory has traditionally been associated with processors performing computation on local memory and then once it using explicit messages to transfer data with remote processors. This adds complexity for the programmer, but simplifies the hardware implementation because the system no longer has to maintain the illusion that all memory is actually shared. This type of programming has traditionally been used with supercomputers that have hundreds or thousands of processing elements. A commonly used technique is MPI.