Computer Organization and Architecture

Chapter 6

RISC, CISC Architecture and Pipelining

6.1 Characteristic Features of RISC Architecture

RISC (Reduced Instruction Set Computer) is a design philosophy for the Central Processing Unit that favours a small, highly optimised set of instructions, each of which is simple enough to execute in a single clock cycle in most cases. The central idea is to simplify the hardware of the processor so that the compiler takes on more responsibility for generating efficient code, and the resulting simple, regular instruction execution allows aggressive pipelining. The important characteristic features of RISC architecture are listed below.

- Reduced and simple instruction set: Only a small number of simple, frequently used instructions are implemented in hardware; complex operations are built up from these using software.
- Fixed-length, uniform instruction format: Every instruction occupies the same number of bytes, which simplifies instruction fetch and decode and helps pipelining.
- Single-cycle execution: Most instructions execute in one clock cycle, giving a Cycles-Per-Instruction (CPI) close to 1.
- Load-store architecture: Only LOAD and STORE instructions access memory; all other instructions operate on data already present in registers.
- Large general-purpose register file: A large number of registers reduces the number of memory accesses required.
- Hardwired control unit: RISC processors generally use hardwired (rather than micro-programmed) control, which is faster since it avoids the overhead of decoding microcode.
- Few addressing modes: Typically only one or two simple addressing modes are supported, which simplifies instruction decoding.
- Heavy use of pipelining: The regularity and simplicity of RISC instructions make them naturally suited to deep, efficient pipelines.
- Compiler-dependent optimisation: Since the hardware is simple, the compiler performs more of the work of scheduling instructions and optimising register usage.

6.2 Comparison between RISC and CISC

CISC (Complex Instruction Set Computer) follows the opposite philosophy: it provides a large number of instructions, many of which are complex and can perform multi-step operations (such as a single instruction that loads an operand from memory, performs an arithmetic operation on it, and stores the result back to memory) in one instruction. This reduces the number of instructions per program but increases the complexity of the

hardware and the number of clock cycles some instructions require. The key differences between the two approaches are summarised in the table below.

Feature	RISC	CISC
Instruction set	Small, simple set of instructions	Large, complex set of instructions
Instruction length	Fixed length, uniform format	Variable length, non-uniform format
Execution time	Mostly single clock cycle (CPI ≈ 1)	Multiple clock cycles per instruction (CPI > 1)
Memory access	Only LOAD/STORE instructions access memory	Many instructions can access memory directly
Addressing modes	Few (1–2)	Many (5–20)
Control unit	Hardwired control	Micro-programmed control
Register set	Large number of general-purpose registers	Comparatively fewer registers
Pipelining	Simple to pipeline efficiently	Harder to pipeline due to variable-length, complex instructions
Code size	Larger program size (more instructions)	Smaller program size (fewer, powerful instructions)
Compiler role	Compiler does more optimisation work	Hardware does more of the work
Examples	ARM, MIPS, SPARC, RISC-V, POWER	Intel x86, VAX

6.3 Concept of Parallel Processing and Flynn's Classification

Parallel processing refers to the simultaneous execution of multiple instructions, tasks, or data operations in order to increase the overall computational throughput of a system. Instead of executing instructions strictly one after another, a parallel-processing system uses multiple functional units, processors, or processing elements that operate concurrently. Parallelism can be achieved at several levels: within a single instruction (pipelining), across multiple data items (vector/array processing), or across multiple independent programs or threads (multiprocessing).

Michael J. Flynn proposed a widely used classification of computer architectures, known as Flynn's Classification, based on the number of concurrent instruction streams and data streams available in the architecture. It divides architectures into four categories, as shown in the diagram below.

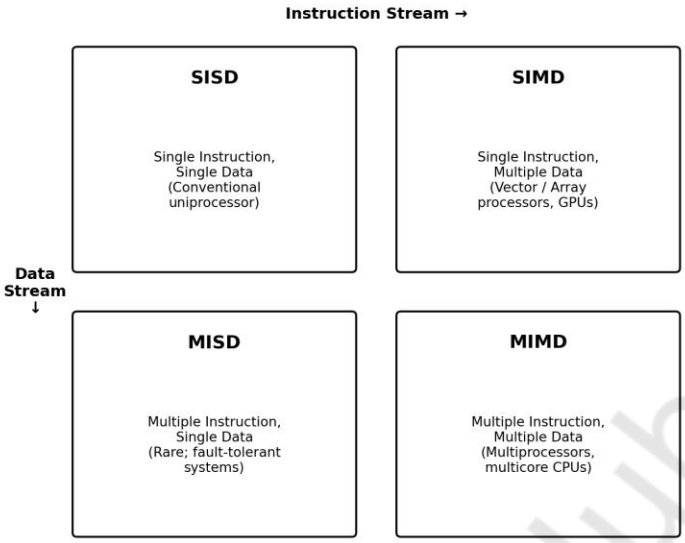

Fig. 6.1 – Flynn's Classification of Computer Architectures

- SISD (Single Instruction, Single Data): A single processor executes a single instruction stream operating on a single data stream. This is the traditional von Neumann, sequential uniprocessor model.
- SIMD (Single Instruction, Multiple Data): A single instruction is broadcast to multiple processing elements, each of which operates on its own data item simultaneously. Vector processors, array processors, and modern GPUs follow this model.
- MISD (Multiple Instruction, Single Data): Multiple processing units execute different instructions on the same data stream. This category is largely of theoretical interest; systolic arrays and certain fault-tolerant systems are cited as rare examples.
- MIMD (Multiple Instruction, Multiple Data): Multiple autonomous processors execute different instructions on different data streams simultaneously. Most modern multiprocessor and multicore systems fall into this category.

6.4 Concept of Instruction Pipelining

Instruction pipelining is a technique that overlaps the execution of multiple instructions by dividing the processing of every instruction into a series of smaller, independent steps called stages (or segments), each handled by a separate piece of hardware. As soon as one instruction moves from the first stage into the second, the next instruction can enter the first stage. This is analogous to an assembly line in a factory, where several units of a product are simultaneously at different stages of production.

A typical RISC processor uses a five-stage instruction pipeline, consisting of the following stages:

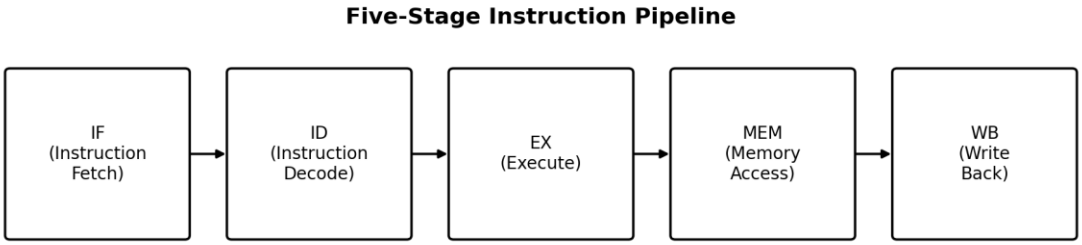

Fig. 6.2 – Five-Stage Instruction Pipeline

1. IF (Instruction Fetch): The instruction is fetched from instruction memory (or instruction cache) using the program counter.
2. ID (Instruction Decode): The fetched instruction is decoded, and the required operand registers are read from the register file.
3. EX (Execute): The arithmetic/logic unit performs the required operation, such as an arithmetic computation or an address calculation.
4. MEM (Memory Access): If the instruction needs to read from or write to data memory (as in LOAD/STORE instructions), this is done in this stage.
5. WB (Write Back): The result of the instruction is written back into the destination register in the register file.

Because each stage is handled by dedicated hardware and operates on a different instruction at any given time, the throughput of the processor increases significantly even though the time taken to execute any single instruction (latency) is not reduced.

6.5 Space-Time Diagram, Speed-up due to Pipelining

A space-time diagram (also called a reservation table or pipeline execution diagram) is a graphical representation that shows which pipeline stage each instruction occupies during each clock cycle. Instructions are listed along one axis and clock cycles along the other; the diagram makes the overlapped, staggered execution of instructions clearly visible.

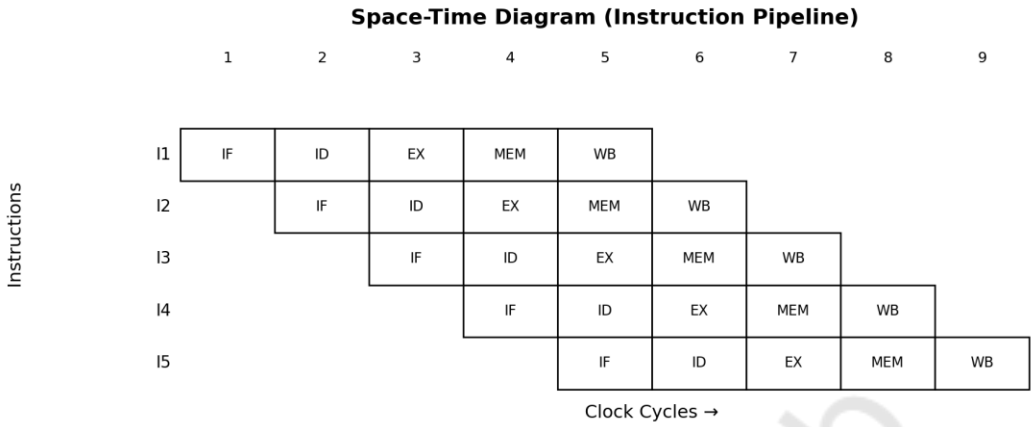

Fig. 6.3 – Space-Time Diagram for Five Instructions in a Five-Stage Pipeline

As the diagram shows, once the pipeline is full, one instruction completes (reaches the WB stage) every single clock cycle, even though each individual instruction still takes five clock cycles to pass through the whole pipeline.

Speed-up due to pipelining

Consider a pipeline with k stages, where each stage takes one clock cycle. For n instructions executed without pipelining (strictly one after another), the total time required is:

$$T_{non\text{-}pipeline} = n \times k \text{ cycles}$$

With pipelining, the first instruction still takes k cycles to complete, but every subsequent instruction completes one cycle later than the one before it. Hence the total time required is:

$$T_{pipeline} = k + (n - 1) \text{ cycles}$$

The speed-up achieved due to pipelining is defined as the ratio of the non-pipelined execution time to the pipelined execution time:

$$\textbf{Speed-up (S)} = \textbf{(n} \times \textbf{k)} \,/\, \textbf{(k + n - 1)}$$

As the number of instructions n becomes very large, the speed-up approaches the number of pipeline stages k, i.e. S → k. This represents the ideal, maximum possible speed-up of a k-stage pipeline. In practice, the actual speed-up achieved is always lower than this ideal value because of pipeline hazards, unequal stage delays, and the overhead of latching data between stages.

6.6 Running the Pipeline with Minimum Idling

Pipeline idling (also called pipeline stalling) occurs when one or more stages of the pipeline are forced to wait — doing no useful work — because the next instruction cannot proceed, typically due to a hazard. Since idling directly reduces the throughput and defeats the purpose of pipelining, several techniques are employed to keep the pipeline as full and productive as possible.

- Instruction scheduling / reordering: The compiler rearranges independent instructions so that dependent instructions are placed further apart, giving earlier instructions time to complete before their results are needed.
- Operand forwarding (bypassing): Results computed in one stage are fed directly to an earlier stage that needs them, instead of waiting for the value to be written back to the register file first.
- Delayed branching: The instruction(s) immediately following a branch (the “branch delay slot”) are always executed regardless of whether the branch is taken, so the compiler fills these slots with useful, independent instructions instead of NOPs.
- Branch prediction: The hardware predicts the outcome of a branch (taken or not taken) in advance and begins fetching instructions along the predicted path, so the pipeline is not left idle while the branch is being resolved.
- Multiple/parallel functional units: Providing several copies of frequently used functional units (such as separate integer and floating-point ALUs) avoids structural hazards that would otherwise stall the pipeline.
- Out-of-order execution: Instructions that are ready to execute are allowed to proceed ahead of earlier instructions that are stalled waiting for operands, keeping functional units busy.
- Register renaming: Eliminates false dependencies (name dependencies) between instructions that reuse the same register name for unrelated values, allowing more instructions to proceed without stalling.
- Pipeline interleaving with multiple independent instruction streams: In some designs, instructions from more than one independent stream are interleaved in the pipeline so that a stall in one stream does not idle the entire pipeline.

6.7 RISC Architecture and Pipelining

RISC architecture and pipelining are closely linked in processor design: RISC instruction sets were deliberately designed to make deep, efficient pipelining practical, and pipelining is one of the principal reasons RISC processors are able to achieve high performance despite their simple instructions. The following features of RISC directly support efficient pipelining.

- Fixed-length instructions: Because every instruction is the same length, the Instruction Fetch stage can always fetch the next instruction without first having to decode the current one to know its length, which is essential for a smooth, predictable pipeline flow.
- Uniform instruction format: A small number of regular instruction formats simplifies the decode logic and allows decoding to be completed quickly and predictably in a single stage.
- Load-store architecture: Since only LOAD and STORE instructions access memory, most instructions need at most one memory access (or none), and the Memory Access stage is only meaningfully used by a subset of instructions, simplifying pipeline control.
- Single-cycle execution of most instructions: Because nearly every instruction takes the same, small number of cycles, all instructions can move through the pipeline stages at a uniform rate, avoiding irregular stage delays.

- Few addressing modes: This reduces the complexity (and hence the time) of address computation in the Execute stage, keeping stage delays balanced across the pipeline.
- Large register file: A large number of registers reduces the frequency of memory operand references, which in turn reduces the number of pipeline stalls caused by memory-access delays.

Because CISC instructions are of variable length, involve complex, multi-cycle operations, and can access memory in many different ways, they are considerably harder to pipeline efficiently — this is one of the main practical motivations behind the original shift towards RISC design in high-performance processors.

6.8 Different Pipeline Hazards and Their Detection and Minimization

A pipeline hazard is any situation that prevents the next instruction in the instruction stream from executing in its designated clock cycle, forcing the pipeline to stall (idle) or produce incorrect results if not properly handled. Pipeline hazards are broadly classified into three types.

(a) Structural Hazards

A structural hazard occurs when two or more instructions in the pipeline require the same hardware resource (such as a single shared memory port, a single ALU, or a single register-file write port) at the same time, but only one instance of that resource is available.

- Detection: Occurs when hardware resource conflicts are identified during pipeline scheduling — for example, when both the Instruction Fetch stage of one instruction and the Memory Access stage of another instruction need to access memory in the same cycle.
- Minimization: Provide duplicate hardware resources (e.g. separate instruction and data caches/memories, multiple ALUs), or use resource scheduling with pipeline stalls only when duplication is not economically feasible.

(b) Data Hazards

A data hazard occurs when an instruction depends on the result of a previous instruction that has not yet completed (i.e. has not yet been written back), and that instruction is still in the pipeline.

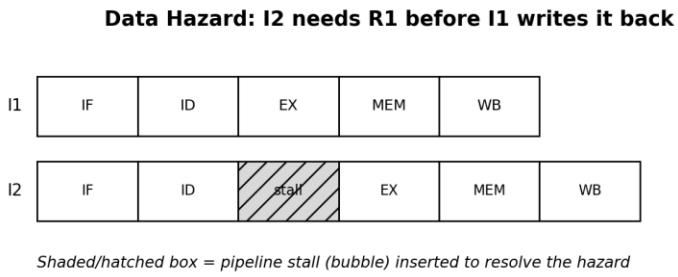

Fig. 6.4 – Example of a Data Hazard and the Resulting Pipeline Stall

- Read-After-Write (RAW): An instruction needs to read a register before a previous instruction has written the new value into it — the most common type of data hazard, as illustrated above.
- Write-After-Read (WAR) and Write-After-Write (WAW): These are “false” or “name” dependencies that can occur when instructions execute out of the original program order.
- Detection: Hardware comparators (hazard-detection units) compare the source register numbers of an instruction entering the pipeline with the destination register numbers of instructions already in the pipeline.
- Minimization: Operand forwarding/bypassing, compiler-based instruction scheduling to separate dependent instructions, register renaming, and, as a last resort, inserting a pipeline stall (bubble).

(c) Control Hazards

A control hazard (also called a branch hazard) arises from branch and jump instructions, since the address of the next instruction to be fetched is not known with certainty until the branch instruction itself has been executed and resolved.

- Detection: The hazard is inherent to any conditional or unconditional branch instruction; the pipeline control logic identifies it as soon as a branch instruction is decoded.
- Minimization: Branch prediction (static or dynamic) to guess the branch outcome in advance, delayed branching to make productive use of the branch delay slot, branch target buffers to quickly supply the predicted target address, and speculative execution with the ability to roll back (flush the pipeline) if the prediction is later found to be wrong.

Effective handling of all three categories of hazards — through a combination of hardware techniques (forwarding, prediction, renaming) and compiler techniques (instruction scheduling) — is essential to achieving a speed-up close to the ideal value of k for a k-stage pipeline.

Multiple Choice Questions (MCQs)

1. Which of the following is a characteristic feature of RISC architecture?

(A) Fixed-length instruction format (B) Large number of complex instructions (C) Micro-programmed control unit (D) Variable number of clock cycles per instruction

2. CISC architecture typically has:

(A) Simple instruction set (B) Large and complex instruction set (C) Hardwired control unit only (D) Fewer addressing modes

3. In Flynn's classification, a GPU executing the same operation on multiple data elements is an example of:

(A) SISD (B) MISD (C) SIMD (D) MIMD

4. Which Flynn's classification category is considered mainly of theoretical interest with very few practical implementations?

(A) SISD (B) SIMD (C) MIMD (D) MISD

5. The main purpose of instruction pipelining is to:

(A) Increase instruction throughput by overlapping execution of instructions (B) Reduce instruction set size (C) Increase memory capacity (D) Reduce the number of registers

6. In a pipeline with n instructions and k stages, the total number of clock cycles required (ignoring stalls) is:

(A) $n \times k$ (B) $n + k - 1$ (C) $n - k + 1$ (D) $k - n + 1$

7. The ideal speed-up of a k-stage pipeline over non-pipelined execution, for a very large number of instructions, approaches:

(A) n (B) 1 (C) k (D) k / n

8. A hazard that occurs when two instructions attempt to use the same hardware resource at the same time is called:

(A) Data hazard (B) Control hazard (C) Name hazard (D) Structural hazard

9. A hazard arising due to branch instructions changing the flow of control is known as:

(A) Control hazard (B) Structural hazard (C) Data hazard (D) Resource hazard

10. Forwarding (or bypassing) is a technique primarily used to minimize:

(A) Structural hazards (B) Data hazards (C) Control hazards (D) Instruction cache misses

11. Branch prediction is a technique used to reduce the impact of:

(A) Data hazards (B) Structural hazards (C) Control hazards (D) Memory hazards

12. Which of the following is NOT typically a feature of RISC architecture?

(A) Load-store architecture (B) Large number of general-purpose registers (C) Single-cycle execution of most instructions (D) Complex addressing modes

13. Inserting a "bubble" or no-operation (NOP) into the pipeline to resolve a hazard is called:

(A) Pipeline stalling (B) Forwarding (C) Out-of-order execution (D) Instruction scheduling

14. Which of the following best defines "speed-up" due to pipelining?

(A) Ratio of pipelined execution time to non-pipelined execution time (B) Ratio of non-pipelined execution time to pipelined execution time (C) Difference between clock cycles of two instructions (D) Number of stages in the pipeline

15. RISC processors typically achieve efficient pipelining mainly because:

(A) They have variable-length, complex instructions (B) They use micro-programmed control (C) They have simple, uniform, fixed-length instructions that mostly execute in one cycle (D) They require multiple memory accesses per instruction

Answer Key

Q1. A	Q2. B	Q3. C	Q4. D	Q5. A
Q6. B	Q7. C	Q8. D	Q9. A	Q10. B
Q11. C	Q12. D	Q13. A	Q14. B	Q15. C

Broad / Long-Answer Questions

1. Explain the characteristic features of RISC architecture with suitable examples.
2. Compare and contrast RISC and CISC architectures on the basis of instruction set, addressing modes, control unit design, and execution speed.
3. What is parallel processing? Explain Flynn's classification of computer architectures with a neat diagram.
4. Define instruction pipelining. Explain the working of a five-stage instruction pipeline with a suitable diagram.
5. Draw and explain the space-time diagram for a pipelined processor executing five instructions.
6. Derive the expression for speed-up achieved due to pipelining. Discuss the factors that limit the ideal speed-up.
7. Explain the various techniques used to run a pipeline with minimum idling (minimum stalling).
8. Discuss why RISC architecture is well suited for pipelining. Support your answer with valid reasons.
9. What are pipeline hazards? Explain structural, data, and control hazards with suitable examples.
10. Explain various techniques for detecting and minimizing pipeline hazards, such as forwarding, stalling, branch prediction, and out-of-order execution.