Computer Organization and Architecture

Chapter 6 – RISC, CISC Architecture and Pipelining

Exercise and Exam-Oriented Questions and Answers – Set 2

Section A – Short Answer Questions

Q1. What is meant by Instruction Set Architecture (ISA)?

Ans. The Instruction Set Architecture is the part of a processor's design visible to the programmer or compiler — it defines the available instructions, registers, addressing modes, and data types, and forms the interface between hardware and software.

Q2. Give one advantage of RISC over CISC.

Ans. Because RISC instructions are simple and uniform, they achieve a CPI close to 1 and pipeline very efficiently, giving higher instruction throughput and faster average execution than a comparable CISC design.

Q3. Give one advantage of CISC over RISC.

Ans. CISC instructions are more powerful and code-dense, so a given task can often be expressed in fewer instructions, which reduces program size and the number of instruction fetches from memory.

Q4. Distinguish between throughput and latency in the context of pipelining.

Ans. Throughput is the rate at which instructions are completed (e.g., one instruction per clock cycle once a pipeline is full), whereas latency is the total time a single instruction takes to pass through all pipeline stages; pipelining improves throughput but does not reduce the latency of any individual instruction.

Q5. Name any two examples of RISC processors.

Ans. ARM and RISC-V are two well-known examples of processors based on the RISC design philosophy.

Q6. Name an example of a CISC processor.

Ans. The Intel x86 family of processors is a well-known example of CISC architecture.

Q7. What is vector (array) processing?

Ans. Vector or array processing is a form of SIMD parallelism in which a single instruction is applied simultaneously to an entire array (vector) of data elements using multiple processing elements, rather than processing one data element at a time.

Q8. What is a systolic array?

Ans. A systolic array is a network of simple processing elements that rhythmically compute and pass data through the array; it is one of the few practical examples cited under Flynn's MISD category, used for specialised tasks such as signal processing.

Q9. What is a Branch Target Buffer (BTB)?

Ans. A Branch Target Buffer is a small, fast cache that stores the predicted target addresses of recently executed branch instructions, allowing the processor to quickly redirect instruction fetch to the predicted target without waiting for the branch to be fully resolved.

Q10. Define register renaming.

Ans. Register renaming is a hardware technique that dynamically maps architectural register names to a larger set of physical registers, eliminating false Write-After-Read and Write-After-Write dependencies between instructions that happen to reuse the same register name.

Q11. What is speculative execution?

Ans. Speculative execution is the technique of executing instructions before it is certain they are actually needed (for example, along a predicted branch path), with the ability to discard the results and roll back if the speculation later proves incorrect.

Q12. Define out-of-order execution.

Ans. Out-of-order execution is a technique in which instructions are executed as soon as their operands are ready rather than strictly in program order, allowing the processor to keep functional units busy instead of stalling on a single blocked instruction.

Q13. What is a "bubble" in a pipeline?

Ans. A bubble is an artificially inserted no-operation (idle) slot in the pipeline, introduced to delay an instruction until a hazard is resolved, effectively wasting one or more clock cycles in exchange for correctness.

Q14. What is meant by a "pipeline being full"?

Ans. A pipeline is said to be full when every stage of the pipeline is simultaneously occupied by a different instruction, which is the steady-state condition under which the pipeline completes one instruction every clock cycle.

Section B – Long / Descriptive Answer Questions

Q1. Explain how RISC and CISC design choices affect CPI and code density, and discuss the resulting trade-off.

Ans. CPI (cycles per instruction) and code density are two competing metrics that RISC and CISC optimise in opposite directions.

RISC keeps instructions simple and uniform so that almost every instruction completes in one cycle, driving CPI close to 1; however, because each instruction does very little, a given task typically needs more instructions, increasing code size (lower code density).

CISC instructions can each perform several primitive operations (e.g., load-modify-store in one instruction), so a task needs fewer instructions and code density is higher; but these complex instructions often take several clock cycles to execute, and some cannot be pipelined efficiently, pushing CPI above 1.

The overall performance trade-off is captured by the relation Execution time = Instruction count × CPI × Clock cycle time — RISC reduces CPI at the cost of instruction count, while CISC reduces instruction count at the cost of CPI; in practice, RISC's pipelining advantage has generally given it the performance edge in modern high-performance processors.

Q2. Explain SIMD architecture with a suitable example.

Ans. SIMD (Single Instruction, Multiple Data) is a class of parallel architecture, under Flynn's classification, in which a single instruction is broadcast to and executed simultaneously by many processing elements, each operating on its own, separate data item.

For example, if an array of 8 numbers needs to be multiplied by a constant, a SIMD unit with 8 processing elements can issue one “multiply” instruction that is applied to all 8 numbers in parallel in roughly the same time a scalar processor would take to multiply just one pair of numbers.

Modern examples of SIMD include vector/array processors, GPUs (which apply the same shader instruction to thousands of pixels or data elements), and SIMD instruction extensions in general-purpose CPUs (such as SSE/AVX), which are especially effective for graphics, scientific computing, and multimedia processing where the same operation is repeated over large data sets.

Q3. Explain MIMD architecture and distinguish between shared-memory and distributed-memory MIMD systems.

Ans. MIMD (Multiple Instruction, Multiple Data) is the Flynn classification category in which multiple autonomous processors simultaneously execute different instructions on different data, making it the most general and most widely used form of parallel architecture in modern computing.

In a shared-memory MIMD system, all processors access a single, common main memory, which simplifies communication between processors (they simply read and write shared variables) but requires careful synchronisation to avoid conflicts; symmetric multiprocessors (SMPs) and modern multicore CPUs are typical examples.

In a distributed-memory MIMD system, each processor has its own private, local memory, and processors communicate explicitly by passing messages over an interconnection network; this scales to a very large

number of processors (as in computing clusters) but requires the programmer or system software to manage communication explicitly.

Q4. Distinguish between pipeline throughput and latency with the help of an example.

Ans. Throughput refers to the number of instructions completed per unit time, whereas latency refers to the total time a single instruction takes from start to finish; pipelining is a technique aimed at improving throughput, not at reducing the latency of any one instruction.

For example, in a 5-stage pipeline where each stage takes 1 clock cycle, any individual instruction still takes 5 clock cycles to fully complete — its latency remains 5 cycles. However, once the pipeline is full, a new instruction completes every single clock cycle, giving a steady-state throughput of one instruction per cycle, which is five times higher than what a non-pipelined design (which must wait for one instruction to fully finish before starting the next) could achieve.

Q5. Explain why increasing the number of stages in a pipeline (superpipelining) does not always give a proportional increase in performance.

Ans. In theory, the ideal speed-up of a k-stage pipeline approaches k as the number of instructions grows large, which suggests that simply adding more, finer-grained stages should keep increasing performance. In practice, several factors prevent this proportional gain.

First, dividing work into more stages adds fixed overheads at each stage boundary (such as latch/register delays for holding data between stages), so beyond a certain point, added stages contribute more overhead than useful work. Second, more and shorter stages increase the number of instructions in flight at once, which increases the frequency and cost of hazards — branch mispredictions become more expensive because more in-flight instructions must be flushed, and data hazards become more likely to require stalls or complex forwarding paths. Third, unequal work across stages means the pipeline's cycle time is set by the slowest stage, so a poorly balanced deeper pipeline may not actually run at a proportionally higher clock frequency.

For these reasons, real processor designs choose the number of pipeline stages as an engineering trade-off between clock speed, hazard penalties, and design complexity, rather than simply maximising the number of stages.

Q6. Differentiate between static and dynamic branch prediction.

Ans. Branch prediction techniques are divided into static and dynamic approaches, both aimed at reducing control hazards by guessing a branch's outcome before it is actually resolved.

Static branch prediction makes a fixed prediction that does not change during program execution, based on simple rules decided at compile time or by the hardware — for example, always predicting “not taken,” always predicting “taken,” or predicting backward branches (typical of loops) as taken and forward branches as not taken.

Dynamic branch prediction, on the other hand, uses hardware that observes the actual run-time behaviour of each branch (often using history bits or counters stored in a branch prediction table) and adapts its prediction as the program runs, allowing it to achieve much higher prediction accuracy than static schemes, especially for branches whose outcome depends on run-time data.

Q7. Explain how register renaming helps overcome WAR and WAW hazards.

Ans. Write-After-Read (WAR) and Write-After-Write (WAW) hazards are “false” or “name” dependencies that occur not because one instruction genuinely needs a value from another, but simply because two unrelated instructions happen to use the same architectural register name; they typically arise when instructions are allowed to execute out of program order.

Register renaming solves this by maintaining a larger pool of physical registers than the architecture exposes, and dynamically mapping each new write to a fresh physical register rather than reusing the same physical location. Since each instruction's destination is given its own physical register, a later instruction's write can no longer accidentally overwrite a value that an earlier instruction still needs to read (WAR) or interfere with another instruction's write to the “same” architectural register (WAW), because they are, in reality, writing to different physical registers altogether.

By eliminating these false dependencies, register renaming allows the processor's out-of-order execution hardware far greater freedom to reorder and overlap instructions, improving overall pipeline utilisation.

Q8. Explain the concept of superscalar architecture and how it differs from simple pipelining.

Ans. Simple (scalar) pipelining improves performance by overlapping the execution of instructions, but it can still only fetch, decode, and complete at most one instruction per pipeline stage in a given clock cycle.

Superscalar architecture goes a step further by providing multiple parallel pipelines (and multiple copies of key functional units) within a single processor, so that more than one instruction can be fetched, decoded, executed, and completed in the same clock cycle, subject to the availability of independent instructions and hardware resources.

In effect, a scalar pipeline aims for a CPI of 1, while a superscalar processor aims for a CPI below 1 (equivalently, an Instructions-Per-Cycle, or IPC, greater than 1). This requires more sophisticated hardware for detecting hazards among the multiple instructions being processed simultaneously, and often relies heavily on out-of-order execution, register renaming, and dynamic scheduling to keep the extra pipelines productively occupied.

Section C – Solved Numerical Problems

Q1. A processor has a 6-stage pipeline, and each stage takes one clock cycle of 15 ns. If 12 instructions are executed, find (a) the non-pipelined execution time, (b) the pipelined execution time, and (c) the speed-up due to pipelining.

Solution:

Given: k = 6 stages, n = 12 instructions, clock cycle = 15 ns.

(a) Non-pipelined time = n × k × cycle time = 12 × 6 × 15 = 1080 ns.

(b) Pipelined time = (k + n − 1) × cycle time = (6 + 12 − 1) × 15 = 17 × 15 = 255 ns.

(c) Speed-up S = Non-pipelined time / Pipelined time = 1080 / 255 = 4.24 (approximately).

Q2. A non-pipelined processor executes an instruction in 300 ns. It is redesigned into a 6-stage pipeline with equal stage delays. Find (a) the time per stage, (b) the pipeline cycle time, and (c) the time taken to execute 25 instructions on the pipelined processor.

Solution:

Given: total instruction time (non-pipelined) = 300 ns, k = 6 equal stages, n = 25 instructions.

(a) Time per stage = 300 / 6 = 50 ns.

(b) Pipeline cycle time = time of the slowest stage = 50 ns (since all stages are equal here).

(c) Pipelined execution time = (k + n − 1) × cycle time = (6 + 25 − 1) × 50 = 30 × 50 = 1500 ns.

Q3. For a 3-stage pipeline, find the number of clock cycles needed to execute 10 instructions, and calculate the resulting speed-up over non-pipelined execution.

Solution:

Given: k = 3 stages, n = 10 instructions.

Total clock cycles required (pipelined) = k + n − 1 = 3 + 10 − 1 = 12 cycles.

Non-pipelined cycles required = n × k = 10 × 3 = 30 cycles.

Speed-up S = (n × k) / (k + n − 1) = 30 / 12 = 2.5.

Note that the speed-up (2.5) is noticeably below the number of stages (3), showing that with a small n, the pipeline does not yet approach its ideal speed-up of k.

Q4. A 4-stage pipelined processor runs at a clock frequency of 1.5 GHz. Out of 2000 instructions executed, 15% are branch instructions, each causing a stall of 3 clock cycles due to a control hazard. Assuming no other stalls, find the effective CPI and the total execution time.

Solution:

Given: clock frequency = 1.5 GHz → clock cycle time = 1 / (1.5×10^9) ≈ 0.667 ns; total instructions n = 2000; branch instructions = 15% of 2000 = 300; stall per branch = 3 cycles.

Ideal CPI (no stalls) = 1 (typical for a RISC pipeline once it is full).

Total stall cycles = 300 × 3 = 900 cycles.

Total clock cycles = (ideal cycles for 2000 instructions) + stall cycles = 2000 + 900 = 2900 cycles (using the simplifying assumption of 1 cycle/instruction as the base rate).

Effective CPI = Total cycles / Total instructions = 2900 / 2000 = 1.45.

Total execution time = Total cycles × cycle time = 2900 × 0.667 ns ≈ 1934.7 ns.

Q5. In a program of 800 instructions, 12% are LOAD instructions that are immediately followed by an instruction depending on the loaded value, causing a 2-cycle stall (data hazard) each time forwarding cannot fully resolve it. Find the total number of stall cycles and the effective CPI, assuming an ideal CPI of 1.

Solution:

Given: total instructions n = 800; LOAD instructions causing a dependent stall = 12% of 800 = 96; stall per occurrence = 2 cycles.

Total stall cycles = 96 × 2 = 192 cycles.

Total clock cycles = ideal cycles + stall cycles = 800 + 192 = 992 cycles.

Effective CPI = Total cycles / Total instructions = 992 / 800 = 1.24.