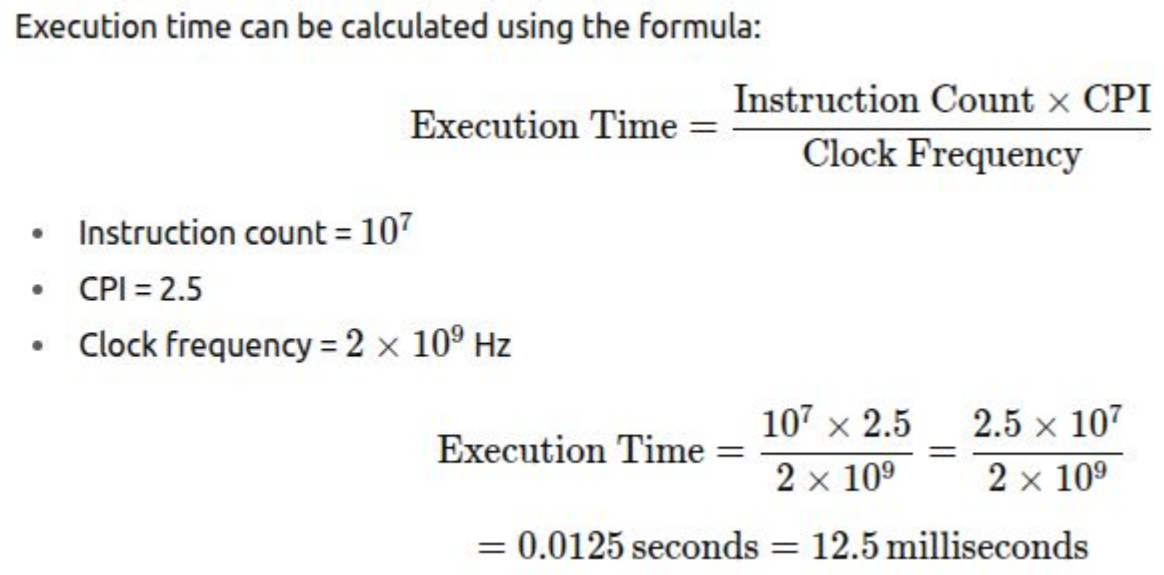
Cloud Infrastructure Past Paper

Q&A

Q.1. A processor executes 10 million (10^7) instructions for a given program. The processor has an average CPI of 2.5 and operates at a clock frequency of 2 GHz. Calculate the total execution time of the program in milliseconds.



Execution time can be calculated using the formula:

$$\text{Execution Time} = \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Frequency}}$$

- Instruction count = 10^7
- CPI = 2.5
- Clock frequency = 2×10^9 Hz

$$\text{Execution Time} = \frac{10^7 \times 2.5}{2 \times 10^9} = \frac{2.5 \times 10^7}{2 \times 10^9}$$

$$= 0.0125\,\text{seconds} = 12.5\,\text{milliseconds}$$

Q.2. Differentiate between ‘Computer Architecture & Computer Organization’ by taking appropriate example.

Answer:

- **Computer Architecture** refers to the conceptual design and functional behaviour of a computer system as seen by a programmer. It defines the **instruction set architecture (ISA), addressing modes, data types, and system design principles**. Example: The x86 and ARM instruction sets are part of computer architecture.

- **Computer Organization** deals with the **implementation details** of how the architecture is realized in hardware, such as control signals, microprogramming, ALU design, pipeline structure, and memory hierarchy. Example: The difference between Intel Core i7 (superscalar, pipelined) and ARM Cortex-A processors lies in organization.

Q.3. Explain the correlation between response time and waiting time in process scheduling. Additionally, draw their relationship using a Venn diagram.

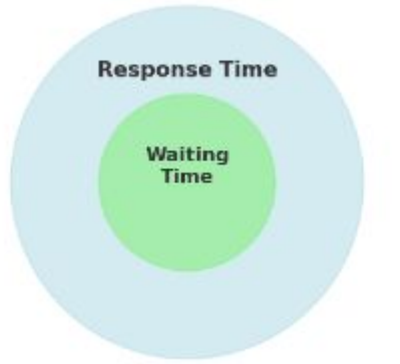
Answer:

- **Response Time** is the time interval from when a request is submitted until the first response is produced. It indicates system interactivity.
- **Waiting Time** is the total time a process spends in the ready queue, waiting to be executed.

Correlation: Waiting time directly contributes to response time. Specifically,

Response Time= Waiting Time + Time for first CPU burst

- Waiting time (part of total turnaround).
- Response time (includes waiting time + initial CPU allocation).



Q.4. Explain the role and physical location of each level of cache (L1, L2, L3) in the memory hierarchy. Discuss the potential advantages and disadvantages of adding additional levels of cache (e.g., L4) to the memory hierarchy. Consider factors such as (not limiting to) access time, power consumption, and cost in your answer.

Answer:

- **L1 (on-core, closest):** Smallest and fastest; typically split I-cache and D-cache per core; minimises load/use latency for the current instructions and data.
- **L2 (on-core or per-cluster):** Bigger and slightly slower than L1; buffers working sets that don’t fit in L1; still private (often) to a core or a small cluster.
- **L3 (last-level cache, on-die, shared):** Much larger, higher latency; shared across cores on the socket; reduces traffic to DRAM and mitigates inter-core interference.

Adding an L4 (e.g., on-package eDRAM or stacked SRAM):

- **Pros:** Fewer DRAM accesses; better throughput for memory-intensive or multiprogrammed workloads; can smooth NUMA effects.
- **Cons:** Higher access latency than L3; extra **power** and **area** cost; design complexity (coherence, inclusion policy); diminishing returns if DRAM is fast (DDR5/HBM) or workloads are cache-unfriendly.

Trade-off summary: More cache levels usually improve hit rates and performance, but each added level increases latency, silicon area, leakage power, and cost—so value depends on workload mix and platform power/area budgets.

Q.5. If mapping techniques such as direct mapping, associative mapping, and set-associative mapping are applied to L3 cache and RAM, explain how is data handled when it resides in L1 or L2 cache?

Answer:
When data is already present in **L1 or L2 cache**, the CPU serves it directly from there. In this case, **L3 and RAM mapping techniques (direct, associative, or set-associative) are not involved at all**, since mapping only comes into play when the processor must fetch from lower levels after an L1/L2 miss.

The handling of data across levels depends on the **Last-Level Cache (LLC, usually L3)** inclusion policy:

- **Inclusive L3:** Everything in L1/L2 must also exist in L3. Evicting a line from L3 forces back-invalidations from L1/L2.
- **Exclusive L3:** Data resides in only one cache at a time. Lines evicted from L2 are placed in L3, avoiding duplication.
- **Non-Inclusive/Non-Exclusive (NI/NE):** No strict duplication rules—data may or may not appear in multiple levels simultaneously.

In all cases, the key point is that **an L1/L2 hit bypasses L3 and RAM entirely**. L3 mapping only matters when the CPU cannot find the requested data in the upper levels.

Q.6. Discuss the importance of Instruction Set Architecture (ISA) in processor design and software compatibility. Consider the elements such as system performance, portability, and power efficiency etc. Also, provide a real-world example where ISA selection played a crucial role in technological advancement or transition?

Answer:
The **Instruction Set Architecture (ISA)** defines the set of instructions, data types, registers, addressing modes, and memory model that a processor supports. It serves as the interface between hardware and software, shaping how programs are written and executed.

Its importance lies in several aspects:

- **System Performance:** The richness and efficiency of the instruction set influence how effectively compilers can translate programs into machine code.
- **Portability and Compatibility:** Since software is compiled against the ISA, applications can run on any processor that implements the same ISA, regardless of internal hardware design.
- **Power Efficiency:** The complexity of an ISA impacts how the processor is built. A simpler ISA often allows for more energy-efficient implementations.

Real-world example: A notable case is the adoption of **ARM (RISC) ISA** in mobile devices and later in cloud servers. ARM's emphasis on energy efficiency and scalability made it the dominant choice for smartphones and has enabled its transition into laptops and cloud platforms (e.g., Apple's M-series chips, AWS Graviton instances). This shift highlights how ISA selection can drive both technological advancement and market adoption.

Q.7. Why are RISC-based processors increasingly being adopted by public cloud providers? Also, list the challenges in complete adoption of RISC for cloud workloads?

Why cloud providers are moving to RISC (like ARM):

- **Better efficiency:** RISC chips give more performance per watt, lowering power and cooling costs in massive data centres.
- **Custom silicon freedom (via ARM's licensing):** Cloud providers can design their own processors (e.g., AWS Graviton) and add features such as higher memory bandwidth or accelerators.
- **Cheaper to run:** Efficiency translates into lower cost per instance for customers, while still delivering good performance for scale-out workloads.
- **Ecosystem ready:** Compilers, Linux, containers, and cloud platforms already support ARM, making adoption practical.

Challenges to full adoption:

- **Legacy software:** Many enterprise apps are still x86-only, so they need porting or emulation.
- **Tuning gap:** Workloads optimised for Intel/AMD vector units (AVX) may require re-optimisation for ARM's NEON/SVE.
- **Mixed environments:** Running both x86 and ARM creates extra complexity in builds, testing, and deployment.
- **Feature parity:** Some specialised hardware extensions and vendor ecosystems remain stronger on x86.

Q.8. In a modern operating system, three types of schedulers—Long-Term, Medium-Term, and Short-Term—work together to optimize CPU and memory usage. Explain the role of each scheduler in process management, highlighting their impact on CPU scheduling, memory allocation, and system performance.
Also, considering a cloud-based workload management system where user applications are executed in virtualized environments.

- **How should each scheduler be designed to efficiently allocate resources and handle dynamic workload variations?**

Answer:

Long-Term (job) scheduler: Controls how many jobs enter the system, regulating multiprogramming and CPU/memory usage.

- **In AWS terms:** This is like **Auto Scaling Groups** or **EKS cluster autoscaling**, which decide when to launch or stop new EC2 instances/containers based on demand.

Medium-Term scheduler: Manages memory residency by suspending/resuming processes. In virtualized clouds, it handles memory ballooning, live migration, and overcommit.

- **In AWS terms:** This is like **EC2 live migration** (to handle hardware faults or balance workloads) or **pause/resume in ECS/Fargate**, where workloads can be shifted around to optimise utilisation.

Short-Term (CPU) scheduler: Picks which process (or VM vCPU) runs next, affecting latency, throughput, and fairness.

- **In AWS terms:** This is similar to how **EC2 vCPUs are time-sliced** between tenants, enforced with **cgroups/quotas** under the hypervisor, ensuring fair CPU shares while meeting SLAs.

Cloud-oriented design guidelines:

- **Long-Term Scheduler:** Should decide *when new workloads are admitted*. In the cloud this means:
    - Only let in as many VMs or containers as the system can handle (SLA-aware admission).
    - Scale out or scale in automatically when demand changes.
    - Place workloads carefully on NUMA nodes/servers for efficiency.
    - Keep extra capacity ready for sudden demand spikes.
- **Medium-Term Scheduler:** Should *adjust workloads dynamically* to avoid bottlenecks. In the cloud this means:
    - Suspend or resume workloads quickly if memory is tight.
    - Live migrate VMs away from hot spots.
    - Use memory optimisation tricks (ballooning, page sharing).
    - Preempt or pause lower-priority workloads (e.g., spot instances) if needed.
- **Short-Term Scheduler:** Should *share CPU fairly and responsively*. In the cloud this means:
    - Allocate CPU time fairly across tenants (quotas/weights).
    - Give different latency guarantees for interactive vs batch jobs.
    - Be aware of NUMA/cache layout for performance.
    - Co-schedule I/O with CPU to avoid stalls.