**SET-1 Answers**

**1a Answer:**

**Computer Architecture**: Computer Architecture is a discipline that deals with the design and organisation of the computer systems. It involves understanding about how the hardware and software of the computer systems come together to efficiently and how they can be optimised for high performance within the given constraints.

**Computer Architecture Evolution:** The computer architecture evolved over centuries with the idea of making the processors better and to do the computing efficiently. The evolution of the computer architecture can be described well by the technologies that ushered the growth in their respective time periods which are majorly divided into five generations, with fifth generation is still under the development phase.

* **Zeroth Generation Computers**: The zeroth-generation computers were mainly mechanical machines. The first of them was invented by a French mathematician Blaise pascal in the year 1642 and the machine was called Pascaline. Later in 1830s English mathematician Charles Babbage tried to build digital computer called Analytical Engine, which had ideas of all the components of a modern computer – like inputs, outputs, memory and computation units which made Charles Babbage the father of computers.
* **First Generation Computers:** The first generation of computers and build with the electronic components called Vacuum tubes which are huge and consumed high power and released lot of heat. they were not very reliable and broke down often which made maintenance hard. some of the examples of first-generation computers are ENIAC and IAS which had major five components of a modern computer i.e., ALU (arithmetic and logical unit), the memory unit, the control unit and the input and output units.
* **Second Generation Computers:** with the invention of transistors made out of silicon, the second-generation of the computers was started. The transistors were relatively small, compact (can fit more is less space) and efficient when compared with the vacuum tubes but still they have their own disadvantages like consumed more power and were slower. The examples of second-generation computers are IBM 701, PDP-1 and IBM 650.
* **Third Generation Computers:** The advent of integrated circuits (ICs) heralded the third-generation computers as hundreds of transistors were packed in a small circuit board (IC) making it easier to maintain them. These computers were smaller in size relatively and produced less heat.
* **Fourth Generation Computers:** Around 1978, when thousands of ICs were being able to fit on a small circuit board calling them LSI (large scale integration) the fourth generation of computers started. Later it was found that millions of components could be packed into a single circuit board rising to the VLSI (very large-scale integration) which are the basic building blocks of the present-day modern computers.
* **Fifth Generation Computers:** Fifth generation computers or computers of tomorrow are still in building phase. They are the computers that have artificial intelligence and will be using ULSI (Ultra large-scale integration) in place of VLSI on ICs.

**1b Answer:**  A process can be seen as an individual instance of a program that is being executed. It represents an independent execution environment with its own memory space, resources, and program counter. Each process runs in isolation from other processes, meaning that they do not share memory or resources by default. Processes have their own copies of program code, data, stack, file descriptors, and other resources. They communicate with each other through inter-process communication (IPC) mechanisms provided by the operating system.

On the other hand, a thread is a unit of execution within a process. It can be considered as a lightweight process that shares the same memory space and resources of the parent process. Threads are scheduled and executed concurrently, allowing multiple threads within a process to work on different tasks simultaneously. They share the process's memory, file descriptors, and other resources, which makes communication and data sharing between threads more efficient and faster compared to inter-process communication.

**The process state diagram** provides a visual representation of the various states and transitions a process can go through. It helps the operating system manage and control the execution of processes, allocate resources efficiently, and maintain system stability and responsiveness. Explanation of the process state diagram:

1. New: The process is in the "New" state when it is first created or spawned by the operating system. At this stage, the necessary resources are being allocated to the process, such as memory, process control block (PCB), and other required data structures.

2. Ready: In the "Ready" state, the process is loaded into main memory, and it is waiting to be assigned to a processor for execution. Multiple processes in the ready state compete for the CPU's attention, and the operating system scheduler determines which process will be selected to run next.

3. Running: When a process is selected from the ready state, it enters the "Running" state. The CPU executes the instructions of the process, and the process is actively using the CPU to perform its tasks.

4. Blocked: Sometimes referred to as "Waiting" or "Sleeping," a process enters the "Blocked" state when it is unable to proceed further until a certain event occurs, such as waiting for input/output (I/O) completion or a specific signal. While in the blocked state, the process is temporarily suspended and does not consume CPU resources.

5. Exit: The process enters the "Exit" state when it completes its execution or terminates for any reason. At this stage, the process releases any resources it acquired during its lifetime, and the operating system removes its PCB and other associated data structures.

In addition to these primary states, there are also some transition states:

- Ready to Running: When the operating system scheduler selects a process from the ready state to run, it transitions from the ready state to the running state.

- Running to Blocked: If a process needs to perform an I/O operation or waits for an event, it moves from the running state to the blocked state.

- Blocked to Ready: Once the required I/O operation or event occurs, a blocked process transitions back to the ready state, waiting for its turn to be scheduled for execution.

- Running to Ready: The running process may voluntarily release the CPU, move back to the ready state, and wait for its next turn for execution. This can occur when the process completes its time slice or is pre-empted by a higher-priority process.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**2 Answer:**

**2b)**  Addressing modes are techniques used by processors to specify the source and destination operands for instructions. Each addressing mode defines a specific way to access data or operands in memory or registers. The commonly used addressing modes:

1. Immediate Addressing:

In immediate addressing, the operand value is directly specified in the instruction itself. It is typically used for constants or immediate values that do not need to be fetched from memory or registers.

Example: ADD R1, #5 ; Add immediate value 5 to register R1

2. Register Addressing:

In register addressing mode, the operands are specified using registers. The instruction directly operates on the contents of the specified registers.

Example: ADD R1, R2 ; Add contents of registers R1 and R2

3. Direct Addressing:

In direct addressing mode, the address of the operand is given explicitly in the instruction. The instruction fetches the operand from the specified memory location.

Example: LOAD R1, X ; Load the value from memory location X into register R1

4. Indirect Addressing:

Indirect addressing uses a memory location to specify the address of the operand. The instruction fetches the operand from the memory location pointed to by the specified address.

Example: LOAD R1, [X] ; Load the value from the memory location pointed to by X into register R1

5. Indexed Addressing:

Indexed addressing mode adds an offset or index value to a base address to access the operand. It allows efficient access to elements in arrays or data structures.

Example: LOAD R1, A[X] ; Load the value from the memory location A[X] into register R1

6. Relative Addressing:

Relative addressing mode specifies the operand address relative to the program counter (PC) or instruction pointer. It is commonly used for branching instructions or accessing instructions nearby.

Example: JUMP LABEL ; Jump to the instruction at the memory location specified by LABEL

7. Base Addressing:

Base addressing mode uses a base register to hold a starting address. The operand address is calculated by adding an offset to the base register value.

Example: LOAD R1, [BASE+OFFSET] ; Load the value from the memory location calculated by adding BASE and OFFSET into register R1

8. Stack Addressing:

Stack addressing mode is used for stack operations, such as pushing and popping values onto/from the stack. The stack pointer register holds the top of the stack.

Example: PUSH R1 ; Push the value from register R1 onto the stack

The choice of addressing mode depends on the specific requirements of the instruction set architecture and the design goals of the processor.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**3 Answer:**

**3a)** Pipelining is a technique used in computer processors to improve performance by overlapping the execution of multiple instructions. However, pipelining introduces certain challenges known as hazards, which can impact the efficiency and correct execution of instructions. Here are three types of hazards commonly encountered in pipelining:

1. Structural Hazards:

Structural hazards occur when the hardware resources required by multiple instructions overlap in time and are not available simultaneously. These hazards arise due to limitations in the processor's design or resources. Some common examples include:

- Resource conflicts: When multiple instructions require the same functional unit (e.g., arithmetic logic unit) at the same time.

- Memory conflicts: When instructions require simultaneous access to the same memory location or cache, resulting in contention or delays.

- Input/output conflicts: When multiple instructions require access to the same input/output device simultaneously.

Structural hazards can lead to pipeline stalls or delays, reducing the overall performance gains of pipelining.

2. Data Hazards:

Data hazards occur when there are dependencies between instructions that can affect the correct execution or ordering of instructions in a pipeline. Data hazards can be further classified into three types:

- Read-after-write (RAW) hazard: Also known as a true dependency, occurs when an instruction depends on the result of a previous instruction that is still being computed or written to a register. The pipeline must ensure that the dependent instruction does not access the result before it is available.

- Write-after-read (WAR) hazard: Also known as an anti-dependency, occurs when an instruction writes to a register or memory location that a subsequent instruction is trying to read. The pipeline must ensure that the read operation does not fetch incorrect data.

- Write-after-write (WAW) hazard: Also known as an output dependency, occurs when two instructions attempt to write to the same register or memory location. The pipeline must ensure the correct order of write operations to avoid data corruption.

Data hazards can lead to pipeline stalls, data forwarding (bypassing), or the need for additional instructions to resolve dependencies.

3. Control Hazards:

Control hazards arise when the pipeline encounters branches or conditional instructions that change the program flow. These hazards can result in incorrect branch predictions or delays in fetching and executing instructions. Control hazards include:

- Branch hazards: Occur when a branch instruction is encountered, and the pipeline needs to fetch instructions from a different location based on the branch outcome. If the branch prediction is incorrect, pipeline bubbles (stalls) occur, wasting clock cycles.

- Delayed branch hazards: Occur when a branch instruction is executed after several instructions, causing potential pipeline stalls until the branch outcome is determined.

Various techniques, such as branch prediction and branch target buffers, are used to mitigate control hazards and minimize pipeline stalls.

Overall, hazards in pipelining can reduce the performance gains achieved by parallel execution. Techniques such as forwarding, buffering, and careful instruction scheduling can be employed to mitigate hazards and optimize pipeline efficiency.

**SET - 2 Answers**

**4 Answer:**

**4a)** Direct and associative mapping are two commonly used techniques for mapping data from main memory to cache memory. Both techniques aim to improve the overall performance of the system by reducing the memory access time.

1. Direct Mapping: Direct mapping is the simplest and most straightforward mapping technique. In direct mapping, each block of main memory is mapped to a specific block in the cache. The mapping is based on a simple mathematical formula, typically using the modulo operation.

Direct mapping has several advantages, including simplicity and low complexity, which makes it more cost-effective. However, it has some drawbacks as well. Since each main memory block can only be mapped to a specific cache block, multiple memory blocks may collide and compete for the same cache block, causing frequent cache misses.

2. Associative Mapping: Associative mapping, also known as fully associative mapping, is a more flexible and efficient technique compared to direct mapping. In associative mapping, each block of main memory can be mapped to any block in the cache. This means that there is no predetermined rule or formula for the mapping.

In associative mapping, the cache controller searches the entire cache for a match whenever a memory access occurs. It compares the memory block address with the stored addresses in the cache tags. If a match is found, it is a cache hit. If no match is found, it is a cache miss.

Associative mapping offers several advantages, such as flexibility in mapping and a reduced chance of cache conflicts. It allows multiple memory blocks to be mapped to different cache blocks simultaneously, which helps in reducing cache misses. However, associative mapping requires additional hardware complexity and can be more expensive compared to direct mapping.

**4b)** Memory hierarchy refers to the organization and arrangement of different types of memory in a computer system, with the aim of providing fast and efficient data storage and retrieval. The memory hierarchy consists of multiple levels, each with varying capacities, access speeds, and costs. The hierarchy is designed to exploit the trade-off between cost and performance, where faster and more expensive memories are used for frequently accessed data, while larger and slower memories are used for less frequently accessed data.

The different levels in a typical memory hierarchy:

1. Registers: These are the fastest and smallest storage units located directly within the CPU. Registers hold instructions and data that are currently being processed by the CPU. They have extremely fast access times but have very limited capacity.

2. Cache Memory: Cache memory is a small but fast memory located between the CPU and the main memory. It stores frequently accessed instructions and data to minimize the time taken to access data from the slower main memory. Cache memory is organized into multiple levels, such as L1, L2, and sometimes L3 caches, each with increasing size and latency. L1 cache is the fastest but smallest, while L3 cache is slower but larger.

3. Main Memory (RAM): The main memory, typically implemented using Random Access Memory (RAM), is larger than cache memory and holds the instructions and data that are actively used by the CPU. It provides a much larger storage capacity but has higher latency compared to cache memory.

4. Secondary Storage: Secondary storage devices, such as hard disk drives (HDDs) and solid-state drives (SSDs), are non-volatile storage mediums used for long-term data storage. They have significantly larger capacities but slower access times compared to main memory. Secondary storage is used to store the operating system, applications, and data that are not actively being used.

5. Tertiary Storage: Tertiary storage includes devices like magnetic tapes and optical discs. They have even larger capacities but even slower access times compared to secondary storage. Tertiary storage is typically used for archival purposes and long-term backup.

The memory hierarchy operates based on the principle of locality, which refers to the tendency of programs to access a relatively small portion of data and instructions repeatedly over a short period. The cache memory takes advantage of temporal and spatial locality by storing frequently accessed data and instructions, reducing the need to fetch them from slower memory levels.

The goal of the memory hierarchy is to bridge the performance gap between the CPU and slower memory levels, minimizing the overall system latency and maximizing data throughput. By organizing memory into multiple levels with varying access speeds and capacities, the memory hierarchy provides a balance between speed and cost, optimizing the overall performance of the computer system.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**5 Answer:**

**5a)**  Fine-grained and coarse-grained architectures are two approaches to designing parallel computing systems. These architectures differ in how they divide and allocate tasks for execution across multiple processing units.

**Fine-grained architecture** refers to a design where tasks or operations are divided into smaller, more granular units. In this approach, the system assigns these smaller units of work to individual processing elements, such as threads or cores. Each processing element performs a fraction of the overall task, resulting in parallel execution.

**Coarse-grained architecture**, on the other hand, involves dividing tasks or operations into larger units. In this approach, the system assigns larger chunks of work to processing elements, such as processes or nodes. Each processing element handles a significant portion of the overall task, allowing for parallel processing

One key difference between fine-grained and coarse-grained architectures is the size of the tasks or operations assigned to the processing elements. Fine-grained architectures divide the workload into smaller units, while coarse-grained architectures allocate larger units of work. This difference impacts how tasks are managed and how efficiently parallelism can be achieved.

Fine-grained architectures provide finer task granularity, enabling more fine-tuned control over parallel execution. Smaller tasks offer greater potential for load balancing and resource utilization, as they can be dynamically scheduled and distributed across available processing elements. Fine-grained architectures are well-suited for workloads with a high degree of inherent parallelism, as they can exploit the parallelism at a fine level of detail. However, managing the overhead associated with the coordination and synchronization of numerous small tasks can be challenging.

On the other hand, coarse-grained architectures deal with larger tasks, resulting in less frequent communication and synchronization between processing elements. This can reduce the overhead associated with task coordination. Coarse-grained architectures are beneficial when tasks have dependencies that span multiple processing elements or when the communication overhead outweighs the benefits of fine-grained parallelism. However, coarse-grained architectures may not fully exploit the available parallelism in workloads that exhibit significant fine-grained parallelism.

Another difference lies in the scalability of the architectures. Fine-grained architectures have higher potential for scalability due to their ability to distribute small tasks across numerous processing elements. This allows for efficient utilization of a large number of cores or threads. Coarse-grained architectures, although limited by the larger task size, can also achieve scalability by adding more nodes to the system, making them suitable for distributed computing scenarios.

Fine-grained and coarse-grained architectures differ in terms of task granularity, task management, parallelism exploitation, overhead, and scalability. Fine-grained architectures focus on dividing tasks into smaller units to achieve high parallelism, while coarse-grained architectures allocate larger units of work with reduced coordination overhead.

**5b)** The storage devices are the hardware components that are used to read or write to the storage media. The major types of these storage devices are categorised into:

* Magnetic storage
* Optical Storage

**Magnetic Storage Devices:**

Magnetic storage devices use magnetized materials to store data. The data is recorded and read through magnetic fields. Here are two common examples of magnetic storage devices:

- Hard Disk Drives (HDDs): Hard disk drives consist of spinning disks, known as platters, coated with a magnetic material. The data is written and read using read/write heads that move across the platters. HDDs offer high storage capacities and are commonly used for primary storage in computers and servers.

- Magnetic Tape Drives: Magnetic tape drives use magnetic tape reels to store data. The tape is passed over read/write heads to record and retrieve data. Magnetic tape drives offer high storage capacities and are often used for long-term archival storage and data backup purposes.

Magnetic storage devices provide relatively high storage densities and are cost-effective for large-scale storage requirements. However, they have moderate access times and are more susceptible to mechanical failures compared to other storage technologies.

**Optical Storage Devices:**

Optical storage devices use laser technology to record and read data from optical discs. The data is stored as pits and lands on the disc's surface. Here are two common examples of optical storage devices:

- Compact Discs (CDs): CDs were one of the earliest optical storage media. They can store data, audio, and video content. CDs have a lower storage capacity compared to more modern optical discs.

- Digital Versatile Discs (DVDs) and Blu-ray Discs: DVDs and Blu-ray discs offer higher storage capacities compared to CDs. They are commonly used for storing high-quality video content, software installations, and data backups. Blu-ray discs have significantly higher storage capacities than DVDs.

Optical storage devices are popular for their portability, durability, and compatibility with various devices. However, they generally have slower access times compared to magnetic storage devices and limited rewrite capabilities.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**6 Answer:**

**6a) RAID (Redundant Array of Independent Disks)** is a technology used in computer storage to improve data reliability, performance, or a combination of both. It involves combining multiple physical disk drives into a logical unit to provide enhanced storage capabilities. RAID levels define various configurations or layouts of these disk drives, each offering different characteristics in terms of data redundancy, performance, and capacity. The different levels of RAID are:

RAID 0 (Striping):

Data Striping: RAID 0 distributes data across multiple disks in "stripes" or segments, improving performance by allowing concurrent read and write operations on different disks.

Data Redundancy: RAID 0 does not provide redundancy or fault tolerance. If one disk fails, it can result in data loss for the entire array.

Performance: RAID 0 offers high performance and increased data throughput, especially for large sequential read/write operations. However, it does not provide data protection.

RAID 1 (Mirroring):

Data Mirroring: RAID 1 duplicates data across multiple disks, creating an exact copy or mirror of each disk. Every write operation is performed on both disks simultaneously.

Data Redundancy: RAID 1 provides excellent data redundancy because if one disk fails, the mirrored disk contains an exact copy of the data, ensuring no data loss.

Performance: RAID 1 generally has lower performance compared to RAID 0 since write operations require data to be written on both disks. However, read performance can be improved as data can be read from either disk.

RAID 5 (Block-Level Striping with Parity):

Data Striping with Parity: RAID 5 distributes data across multiple disks in blocks, similar to RAID 0. However, it also calculates parity information for each block and stores it on different disks in a rotating manner.

Data Redundancy: RAID 5 provides fault tolerance by using parity information. If one disk fails, the missing data can be reconstructed using the parity information stored on other disks.

Performance: RAID 5 offers a good balance between performance and data redundancy. It provides improved read performance due to striping and reasonable write performance due to the distributed parity calculations.

**6b)** Scalable computers, which are designed to handle increasing workloads and accommodate growing demands, However, they can encounter various problems as the system scales up. Some common problems and potential ways to resolve them:

* Scalability Bottlenecks:
  + Problem: Scalable systems may face bottlenecks due to limitations in network bandwidth, storage capacity, or processing power, hindering their ability to handle increased workloads.
  + Resolution: To resolve bottlenecks, one can employ techniques such as load balancing to distribute work evenly across multiple nodes, implementing faster networking technologies, upgrading storage devices, or adding more processing resources.
* Data Consistency:
  + Problem: Maintaining data consistency across multiple nodes in a scalable system can be challenging. Concurrent updates and distributed transactions may result in conflicts and data inconsistencies.
  + Resolution: Techniques like distributed locking, conflict resolution algorithms, or adopting distributed database management systems can help ensure data consistency and coherence across nodes.
* Fault Tolerance:
  + Problem: Scalable systems are susceptible to hardware or software failures, which can lead to system downtime or data loss.
  + Resolution: Employing fault-tolerant mechanisms like redundancy, data replication, and implementing fault detection and recovery mechanisms such as backup and restore procedures, data mirroring, or RAID configurations can enhance system resilience and minimize disruptions.
* Scalable Communication:
  + Problem: Efficient communication between nodes in a scalable system becomes crucial as the number of nodes increases. High communication overhead can limit scalability.
  + Resolution: Using scalable communication protocols, optimizing network configurations, employing message queues, or implementing parallel processing and data streaming techniques can improve communication efficiency and overall system scalability.
* Load Balancing:
  + Problem: Uneven distribution of workloads among nodes can lead to performance bottlenecks and reduced scalability.
  + Resolution: Implementing load balancing techniques such as round-robin, least connection, or dynamic load balancing algorithms can evenly distribute workloads across nodes, ensuring optimal resource utilization and improved scalability.
* Management and Monitoring:
  + Problem: As the system scales, managing and monitoring a large number of nodes becomes challenging, making it difficult to identify and address performance issues or failures.
  + Resolution: Employing centralized monitoring systems, implementing automated management tools, utilizing distributed logging and analytics, or adopting cloud-based management platforms can help streamline system management and monitoring processes, facilitating scalability.

It's important to note that these are general problem areas and resolution approaches. The specific solutions may vary depending on the architecture, technology stack, and requirements of the scalable computer system being implemented.