black N White

NAME	
ROLL NUMBER	
SEMESTER	**II**
COURSE CODE	**DCA6210**
COURSE NAME	**COMPUTER ARCHITECTURE**

SET-I

Q.1) What is meant by Direct Mapping? Discuss the various types of Mapping.

Answer . :-

Cache memory is a high-speed memory unit that stores frequently accessed data and instructions to reduce the average time to access data from the main memory. However, due to limited size, the organization of cache memory plays a vital role in improving system performance. **Mapping techniques** define how blocks from the main memory are placed into the cache. One of the most basic and widely used techniques is **Direct Mapping**.

Direct Mapping:

Direct Mapping is the simplest form of cache memory mapping technique. In this method, **each block of main memory is mapped to exactly one specific cache line**. This is achieved using a formula:

Cache Line Number = (Block Address) MOD (Number of Cache Lines)

For example, if the cache has 8 lines and the block address is 10, then it will be placed in line 10 mod 8 = 2.

Each cache line stores not only the data but also a **tag** to identify which block from the main memory is stored there. When a memory address is requested, the system checks the specific cache line using the formula, compares the tag, and either finds the data (**hit**) or goes to the main memory (**miss**).

Advantages of Direct Mapping:

- Simple to implement.
- Faster access due to direct indexing.
- Requires less complex hardware.

Disadvantages:

- High chances of **conflict misses**, especially if different blocks map to the same cache line.

Types of Cache Mapping Techniques:

Besides Direct Mapping, there are two other major types of mapping techniques:

1. Associative Mapping:

In this method, **any block of main memory can be placed in any cache line**. There is no restriction like in direct mapping.

Each cache line stores a **tag** along with the data to identify the block stored.

- **Advantages:**
    - Very flexible.
    - Minimizes conflict misses.
- **Disadvantages:**
    - Needs complex hardware to search all cache lines.

- Slower than direct mapping.

2. Set-Associative Mapping:

This is a combination of both Direct and Associative mapping. The cache is divided into multiple **sets**, and each set has multiple lines (e.g., 2-way, 4-way).

A block from main memory maps to a specific set using modulo operation, but within that set, it can occupy **any line**.

- **Advantages:**
  - Reduces conflict misses compared to direct mapping.
  - More efficient than associative mapping in terms of speed.
- **Disadvantages:**
  - More complex than direct mapping.
  - Slightly slower due to searching within the set.

Cache mapping techniques determine how efficiently memory blocks are transferred and stored in the cache. **Direct mapping** is easy and fast but suffers from high conflict misses. **Associative mapping** is more flexible but costlier. **Set-associative mapping** offers a good balance between performance and complexity. The choice of mapping depends on system requirements and performance needs.

Q.2) What are addressing modes? Explain various addressing modes with example.

Answer .:-

Addressing modes are techniques used in computer architecture to specify how the operand (data) is accessed or referred to by an instruction. In other words, addressing modes define the rules or methods through which the CPU identifies the location of data that it needs to process.

Addressing modes are important because they increase the flexibility and efficiency of the instruction set. They allow the use of constants, memory locations, registers, and more, with a single instruction format.

Types of Addressing Modes

There are several types of addressing modes used in assembly language programming. Each mode has its own way to specify the operand.

1. Immediate Addressing Mode

In this mode, the operand is directly specified in the instruction itself.

Example:

MOV A, #5

Here, #5 is the actual data. The value 5 is directly moved to register A.

Advantage: Fast execution, no memory lookup required.

2. Register Addressing Mode

In this mode, the operand is in a register, and the instruction refers to the register.

Example:

MOV A, B

This copies the contents of register B into register A.

Advantage: Very fast, as it operates directly within CPU registers.

3. Direct Addressing Mode

The memory address of the operand is given directly in the instruction.

Example:

MOV A, 3000H

This means copy the contents of memory location 3000H into register A.

Advantage: Simple and easy to understand.

4. Indirect Addressing Mode

Here, the address of the operand is stored in a register or memory location. The instruction refers to that register or memory to get the actual address.

Example:

MOV A, @R0

If R0 = 40H, then contents of memory location 40H are moved to A.

Advantage: Allows flexible memory access.

5. Register Indirect Addressing Mode

The memory address is stored in a register pair (like HL), and the operand is fetched from that memory location.

Example:

MOV A, M ; if HL = 2500H → A = [2500H]

Advantage: Efficient access to arrays or dynamic memory locations.

6. Indexed Addressing Mode

Here, the effective address is calculated by adding a constant value (index) to a base register.

Example:

MOV A, 1000H(R1)

If R1 = 05H, effective address = 1005H, so A = [1005H]

Advantage: Useful for accessing array elements.

Addressing modes provide a systematic way to locate operands needed by instructions. They are crucial in low-level programming and computer architecture as they allow instructions to access data in various flexible ways. The choice of addressing mode impacts program size, speed, and performance. Different modes are suited for different types of operations like working with constants, registers, or memory.

Q.3) What are pipelining hazards? Explain various hazards in detail.

Answer .:-

Pipelining is a technique used in modern CPUs to improve performance by executing multiple instructions in overlapping phases. Just like an assembly line, while one instruction is being executed, the next one is being decoded, and another is being fetched.

However, pipelining doesn't always run smoothly. Sometimes, certain conditions prevent the next instruction from executing in its designated clock cycle. These issues are called **pipelining hazards** or **pipeline stalls**.

Types of Pipelining Hazards

There are mainly **three types** of pipelining hazards:

1. Structural Hazards:

A structural hazard occurs when **two or more instructions require the same hardware resource** at the same time.

Example:

If the pipeline does not have separate hardware units for instruction fetch and memory access, both operations cannot happen simultaneously. This causes a stall.

Solution:

- Use **separate hardware units** for fetch, decode, execute, and memory access.
- Improve hardware resource availability.

2. Data Hazards:

A data hazard arises when **an instruction depends on the result of a previous instruction** that is still in the pipeline and not yet completed.

Types of Data Hazards:

- **Read After Write (RAW):**

  Also known as a **true dependency**.

  Example:
- ADD R1, R2, R3 ; R1 = R2 + R3
- SUB R4, R1, R5 ; R4 = R1 - R5

Here, the second instruction needs the value of R1, which is not ready yet.

- **Write After Read (WAR):**

  Happens when a later instruction writes to a register before the earlier one has read from it.
- **Write After Write (WAW):**

  Occurs when two instructions write to the same register and the order gets messed up.

Solution:

- **Forwarding (bypassing):** Pass the result directly to the next stage.
- **Stalling:** Delay the instruction until the data is available.
- **Register renaming:** Avoid write conflicts.

3. Control Hazards:

Control hazards happen due to **branching or jumping** in code. The CPU may not know which instruction to fetch next until the branch is resolved.

Example:

BEQ R1, R2, LABEL ; Branch if R1 = R2

Until the condition is checked, the next instruction cannot be decided.

Solution:

- **Branch prediction:** CPU predicts the next instruction path.
- **Delayed branching:** Execute next instruction regardless of branch decision.
- **Pipeline flushing:** Remove wrong instructions from the pipeline.

Pipelining improves performance by overlapping instruction execution. However, **pipelining hazards** reduce efficiency by causing delays. These hazards—**structural**, **data**, and **control**—must be handled carefully using techniques like **hardware enhancements, stalling, forwarding, and branch prediction** to ensure smooth pipeline operation. Understanding and managing hazards is key to designing efficient and fast processors.

SET-II

Q.4) What is Amdahl's Law, and how does it relate to multicore systems?

Answer .:-

Amdahl's Law is a formula used in parallel computing to predict the theoretical maximum speedup of a task when using multiple processors or cores. The law highlights the limitations of parallelizing certain portions of a task, providing insight into how much improvement can be gained by adding more processors.

It was formulated by **Gene Amdahl** in 1967, and is widely used in computer science to understand the **scalability** of programs, especially when transitioning from single-core to multi-core systems.

Amdahl's Law Formula

Amdahl's Law can be mathematically expressed as:

$$s = \frac{1}{(1 - P) + \frac{P}{N}}$$

Where:

- SS is the **speedup** of the program when using **N** processors.
- PP is the **fraction of the program** that can be parallelized (expressed as a value between 0 and 1).
- NN is the **number of processors** or cores being used.

Interpretation of Amdahl's Law

- **Speedup**: Amdahl's Law shows the relationship between the speedup and the fraction of the program that can be parallelized.
- **Parallelizable portion**: If a portion of the program can be executed in parallel, Amdahl's Law tells you how much faster the program can run by adding more processors.
- **Non-parallelizable portion**: The non-parallelizable portion of the program (1 - P) will always remain a bottleneck, limiting the overall speedup regardless of how many processors are added.

How Amdahl's Law Relates to Multicore Systems

In a **multicore system**, multiple processors or cores work together to execute a program, with the goal of improving performance. However, **Amdahl's Law** presents a limit to the potential speedup that can be achieved, particularly when the program has a significant sequential component.

Multicore Systems and Scalability:

- **Parallelism Limitation**: If only a small portion of the program can be parallelized, adding more cores will not lead to significant speedup. For example, if 90% of the program can be parallelized, and the remaining 10%

must run sequentially, adding more cores will help, but the overall speedup will still be limited due to the sequential part.

- **Diminishing Returns**: As you keep increasing the number of cores in a multicore system, the benefit from each additional core diminishes. For example, adding 2 cores will give a better speedup than adding 10 cores if the parallelizable portion of the program is limited.
- **Practical Implication**: In practical applications, the performance gains from adding more cores are often less than expected due to **Amdahl's Law**. Developers must strive to optimize the parallelizable portion of their programs to achieve better scalability.

Example:

Let's say 80% of a program is parallelizable (P = 0.8), and 20% must run sequentially (1 - P = 0.2). If we use 4 cores (N = 4), according to Amdahl's Law:

$$s = \frac{1}{(1-0.8)+\frac{0.8}{4}} + \frac{1}{0.2+0.2} = \frac{1}{0.4} = 2.5$$

This means the program can achieve at most a **2.5x speedup** even with 4 cores.

Amdahl's Law teaches that the performance improvement from adding more processors to a system is limited by the **non-parallelizable portion** of a program. Even in multicore systems, where multiple cores work together, the overall speedup is constrained if a significant part of the task cannot be parallelized. Therefore, while multicore systems offer advantages, understanding and optimizing the parallelizable part of the code is essential for achieving the best performance gains.

Q.5) Describe in brief the architecture of vector processor. What are some of the key limitations of this architecture?

Answer .:- :

Architecture of a Vector Processor

A **vector processor** is a specialized CPU designed to handle vector computations efficiently. Vector processing is highly suitable for applications that require processing large sets of data simultaneously, such as scientific computing, simulations, and graphics processing. The architecture of a vector processor is specifically optimized to perform vector operations (i.e., operations on arrays of data) in parallel, leading to significant performance improvements in certain types of workloads.

Key Components of a Vector Processor:

1. **Vector Registers**:
    - These are special registers used to hold vectors (large arrays of data). Unlike scalar processors, which use a single register for a single value, vector processors have **multiple vector registers** to store entire vectors of data.
    - These registers allow the processor to access and process large sets of data quickly, avoiding frequent memory accesses.
2. **Vector Units (Functional Units)**:

- The vector processor has specialized **vector functional units**, such as **adders, multipliers, and dividers**, that can perform vectorized operations. These units can perform arithmetic operations on multiple data elements in parallel.
- The vector units work directly with the vector registers to carry out computations on vectors rather than on single data elements.

3. **Control Unit**:
   - The **control unit** coordinates the execution of vector instructions and manages the flow of data between vector registers and functional units. It issues vector instructions to the functional units, and ensures synchronization during vector operations.
4. **Memory System**:
   - A vector processor often uses a high-bandwidth memory system that allows for fast access to the large data sets being processed. The memory system must be capable of delivering data to the vector registers in a continuous stream to avoid bottlenecks.
5. **Vector Pipelines**:
   - Vector processors typically implement **pipelining** for vector operations. Each stage of the pipeline handles a specific part of the vector operation (e.g., fetching data, performing arithmetic, and writing results). This pipelining helps improve throughput and efficiency.

Key Limitations of Vector Processor Architecture

Despite their power in certain applications, vector processors have a number of limitations:

1. **Limited Applicability**:
   - Vector processors are highly effective for **data-parallel tasks**, where the same operation is applied to large datasets. However, they are **not ideal for tasks** with complex control dependencies or tasks that involve a lot of branching and conditional operations.
   - Programs that do not have significant parallelism in their data processing (i.e., scalar tasks) cannot benefit from vector processing.
2. **Complexity in Software Development**:
   - Writing software that can take full advantage of vector processors can be complex. Developers must restructure their programs to take advantage of **vectorized instructions**, which often involves **rearranging data** and ensuring operations are parallelizable.
3. **Cost of Implementation**:
   - Vector processors often require **specialized hardware** for vector registers, vector units, and memory systems, which can be expensive to design and manufacture.
   - This makes vector processors less cost-effective for general-purpose computing, and they are often used in niche applications like supercomputers.
4. **Memory Bottlenecks**:
   - While vector processors are designed to handle large datasets, they still face potential **memory bottlenecks**. If the memory bandwidth cannot keep up with the rate at which data is being processed, performance can

degrade significantly. Effective memory hierarchy design is crucial for the performance of vector processors.

5. **Parallelism Limitations**:
    - Vector processors excel in **single-instruction multiple-data (SIMD)** operations but can struggle with more complex parallelism schemes that involve multiple instruction streams (SIMT, MIMD). This makes vector processors less flexible compared to general-purpose processors like multicore CPUs or GPUs that can handle a wider variety of parallel tasks.
6. **Scaling Issues**:
    - As the number of processors in a system increases, managing vector parallelism becomes more difficult. The overhead of coordinating multiple vector units or managing memory access can become significant, limiting scalability.

A vector processor is designed to efficiently handle large datasets and perform arithmetic operations in parallel. This architecture is ideal for applications such as scientific simulations and graphics processing. However, its limitations, including complexity in software development, limited applicability to specific tasks, and memory bottlenecks, restrict its use in general-purpose computing. The effectiveness of vector processors depends heavily on the nature of the workload and the design of the system.

Q.6) What is the difference between SMP and AMP in multiprocessor systems .

Answer .:-

In multiprocessor systems, multiple processors or cores work together to perform tasks faster and more efficiently. **SMP (Symmetric Multiprocessing)** and **AMP (Asymmetric Multiprocessing)** are two common architectures used to implement multiprocessor systems. While both systems utilize multiple processors, they differ significantly in terms of their structure, management, and application.

1. Symmetric Multiprocessing (SMP)

In an **SMP system**, all processors are **equal** and have **access to the same shared memory**. Each processor is capable of executing tasks independently, and the system operates under a **single control unit**.

Key Characteristics of SMP:

- **Equal Access to Resources**: All processors have equal access to the system's memory and I/O devices. This means that any processor can read from or write to the shared memory.
- **Shared Memory Architecture**: The processors share a common memory space, and this shared memory allows for communication between the processors. Any processor can access data stored in the memory.
- **Independent Processors**: All processors are capable of running independent tasks or collaborating on tasks. They can execute instructions simultaneously in parallel, which improves performance.

- **Centralized Control**: Typically, there is one **operating system (OS)** managing the entire system, and it treats all processors as equal entities.

Advantages of SMP:

- **Scalability**: It's easier to add more processors to the system to increase performance.
- **Flexibility**: Any processor can handle any task, making it easier to distribute workload efficiently.
- **Redundancy**: If one processor fails, the other processors can continue to work, providing fault tolerance.

Disadvantages of SMP:

- **Memory Contention**: Since all processors access the same memory, **memory contention** can occur, causing delays if multiple processors attempt to access memory simultaneously.
- **Complexity in Management**: Managing synchronization and ensuring data consistency between processors can become complex.

2. Asymmetric Multiprocessing (AMP)

In an **AMP system**, there is an **asymmetry** between the processors. One processor, called the **master processor**, controls the system and is responsible for running the operating system and managing tasks. The other processors, called **slave processors**, are primarily used for executing tasks assigned by the master processor.

Key Characteristics of AMP:

- **Master-Slave Relationship**: The system consists of one **master processor** that handles the control and management tasks, while the **slave processors** perform the actual computations.
- **Dedicated Tasks**: Slave processors are typically assigned specific tasks by the master processor and do not have access to the operating system or the entire memory.
- **Non-Shared Memory**: In some AMP systems, slave processors may have local memory that is not shared with the master processor.

Advantages of AMP:

- **Simplicity**: The architecture is simple since the master processor controls all the tasks, and the slave processors execute the commands given by the master.
- **Efficient for Specific Tasks**: AMP is suitable for systems where one processor needs to control other processors and assign them specific tasks (e.g., real-time systems).
- **Lower Cost**: Since the system doesn't require multiple processors to manage the OS, it may be less costly to implement.

Disadvantages of AMP:

- **Single Point of Failure**: If the master processor fails, the entire system becomes inoperable.
- **Limited Scalability**: Adding more slave processors does not necessarily improve the performance since the master processor still manages all tasks.
- **Inefficient Resource Utilization**: Since only one processor handles the control tasks, the slave processors might be underutilized when they don't have tasks to perform.

Key Differences Between SMP and AMP

Feature	SMP (Symmetric Multiprocessing)	AMP (Asymmetric Multiprocessing)
Processor Role	All processors have equal roles and responsibilities.	One master processor controls the system, while others are slaves.
Memory Architecture	Shared memory where all processors access the same memory.	Non-shared memory, with slave processors often having local memory.
System Management	Managed by a single OS that treats all processors equally.	Managed by the master processor, which runs the OS.
Scalability	High scalability, additional processors can be added easily.	Limited scalability due to the master-slave relationship.
Fault Tolerance	Fault tolerance is higher, as other processors can take over.	If the master processor fails, the system fails.
Complexity	More complex to manage due to synchronization between processors.	Simple to implement and manage due to master-slave setup.

SMP is ideal for systems requiring high performance and scalability, as all processors share the workload and can access memory equally. It is widely used in modern server systems and high-performance computing tasks. **AMP**, on the other hand, is suitable for specialized systems where one processor can control others, and the system is less focused on scalability and parallelism. While AMP offers simplicity, it is not as flexible or efficient as SMP in handling complex, parallel tasks.

chahiye ya koi aur question ho, toh batao bhai ji!