Internal Assessment Test 2 Answer– January 2026

Sub:	Digital Design and Computer Organization					Sub Code:	BCS302	Branch :	AIML/CSE AIML
Date:	06.01.2026	Duration:	90 min	Max Marks:	50	Sem/Sec:	III /A, B & C		OBE
Answer any FIVE FULL Questions							**MARKS**	**CO**	**RBT**

1. **Explain about Direct memory access. Also specify the working principles.** **[10]**

Answer: -

Direct memory access (DMA) is a technology that allows hardware devices to transfer data between themselves and memory without involving the central processing unit (CPU). DMA enhances system performance by offloading data transfer tasks from the CPU, enabling it to focus on other critical operations.

DMA acts as a traffic controller for data moving in and out of memory. It efficiently manages these transfers, freeing up the CPU for more complex tasks. This mechanism significantly boosts overall system efficiency and speed.

By utilizing DMA, devices like network cards, graphics cards, device drivers, and storage controllers can directly access memory locations without constant intervention from the processor. This streamlined process accelerates data movement and reduces latency in information exchange within the system.

DMA plays a pivotal role in optimizing data flow within computers and enhancing their operational capabilities by minimizing unnecessary processing overheads.

How Does Direct Memory Access Work?

Step 1. Initiation

Initiation is the first step in the DMA process. It kicks off the data transfer operation between devices without involving the CPU constantly. When a device needs to send or receive data from memory, it initiates a DMA request.

During initiation, the DMA controller identifies and prioritizes incoming requests based on predefined criteria. This ensures efficient utilization of system resources and minimizes delays in data transfer. Once a request is accepted, the DMA controller temporarily takes control of the bus to facilitate direct communication between devices and memory.

Step 2. Request

When a peripheral device needs to transfer data to or from memory, it sends a DMA request signal to the DMA controller. This signal indicates that the CPU is not required for this particular data transfer operation.

Upon receiving the DMA request, the DMA controller checks if the bus is available and then initiates access to memory. By handling these requests independently of the CPU, DMA significantly reduces processor overhead and speeds up data transfers between devices and memory.

The DMA controller manages the timing and prioritization of these requests through efficient arbitration techniques. This ensures that multiple devices can communicate with memory seamlessly without causing conflicts or bottlenecks in data flow.

Step 3: Arbitration

When multiple devices need to access the memory simultaneously, the DMA controller arbitrates between these requests to ensure efficient utilization of the system's resources.

Through arbitration, the DMA controller prioritizes and schedules data transfer tasks based on predefined rules or algorithms. This process helps prevent conflicts and ensures that each device gets fair access to the memory bus without causing bottlenecks or delays in data transfers.

By intelligently managing contention for memory access, the DMA controller optimizes the system's overall performance by minimizing idle time and maximizing throughput. It coordinates communication between different components seamlessly, allowing for smoother operation and improved efficiency in handling large volumes of data.

Step 4: Bus mastering

In this process, a DMA-capable device takes control of the system bus to manage data transfers independently from the CPU. The DMA controller coordinates with other devices on the bus for efficient data movement, ensuring smooth communication flow within the system.

Step 5: Memory access

Once the DMA controller gains control of the system bus, it can directly access the memory without involving the CPU. This direct interaction allows efficient and speedy data transfers between peripherals and memory locations.

During memory access, data is read from or written to specific memory addresses as instructed by the DMA controller. The controller ensures that data is transferred accurately and promptly without requiring constant intervention from the CPU.

Step 6: Data transfer

Once the DMA controller has control of the bus, it initiates the actual data movement between devices and memory. The DMA controller coordinates with the source and destination devices to efficiently transfer data without involving the CPU. During data transfer, information flows directly from one device to another through DMA channels without CPU intervention.

Step 7: Completion and interrupt

Once the data transfer is completed, the DMA controller triggers an interrupt to notify the CPU. This interrupt signals that the DMA operation has finished successfully. The CPU can then resume its tasks or handle any necessary follow-up actions based on the completion of the data transfer.

Interrupts are crucial as they allow efficient communication between the DMA controller and the CPU without constant polling. By using interrupts, system resources are utilized more effectively since the CPU can attend to other processes while waiting for DMA operations to finish.

Upon receiving an interrupt from the DMA controller, the CPU may execute specific routines or procedures set up to handle post-DMA tasks, such as updating status flags, notifying software components about data availability, or initiating further processing steps based on the transferred data.

Step 8: Release of bus control

Once the data transfer is complete, the DMA controller releases control of the system bus. This step is crucial as it allows other devices to access the bus for their own operations without any interference from the DMA process.

This final step paves the way for ongoing processes within the computer system to continue smoothly without any hindrance caused by the exclusive use of resources during data transfers.

2. Define addressing modes. Explain any five types of addressing modes with examples.

Addressing modes specify the mode by which a processor fetches and accesses operands necessary for instruction execution. An operand, or data that is to be operated upon, may be fetched from three basic locations:

1. **Register-stored value** - The operand is stored in one of the processor's registers. Register access provides the highest possible operand access speed since registers are provided directly in the CPU core, with single-cycle access and no memory delay.
2. **Location in memory** - The operand is stored in the system's main memory. Memory addressing causes the processor to calculate an effective address and issue a memory fetch, which has variable latency based on cache hierarchy and memory subsystem performance.
3. **Immediate value** - The operand is explicitly contained within the instruction. Immediate operands are very efficient for constant data since they save the instructions from having to look up individual register or memory, though they are constrained by the immediate field size of the instruction.

Types of Addressing Modes

Addressing modes define how processors must handle operand locations within instructions. Addressing modes are central to instruction set architecture (ISA) design and have direct influence on performance, code density, and program flexibility. Contemporary processors use several addressing modes to support diverse access patterns ranging from efficient register accesses to complex memory address techniques.

1. Implied (Implicit) Addressing Mode

In implied addressing mode, the instruction itself holds all information regarding the operand without any explicit mention. It is usually employed for instructions that act on implicit registers or system operations where the destination is to be implied from the instruction itself. The biggest benefit is less instruction encoding as no operand field is needed, but the drawback is less flexibility because operations are constrained to acting on fixed locations.

Example:

- NOP (No Operation) - This instruction does nothing and requires no operands.
- HLT (Halt) - Stops the CPU execution without needing an operand.

2. Immediate Addressing Mode

The immediate addressing places the operand value explicitly in the instruction to be executed immediately without further memory or register access. Immediate addressing is highly effective for loading constants and performing arithmetic calculations on known constants because it avoids memory access latency. Nevertheless, the immediate value width is usually restricted by the instruction bit width, limiting the range of constants that can be accessed directly.

Example:

- MOV R3, #50 - Loads the immediate value **50** into register **R3**.
- ADD R1, R2, #5 - Adds **5** to the value in **R2** and stores the result in **R1**.

3. Register Addressing Mode

In this addressing mode the operands will be in the registers of the processor itself, thus operand access will be as fast as possible because registers are implemented in the CPU itself which means zero wait states. This mode is essential to load-store architectures where all operations must work on the contents of registers. Benefits include one-cycle execution and low power consumption, but the drawback is the limited number of registers available in most architectures.

Example:

- SUB R4, R5 - Subtracts the value in **R5** from **R4** and stores the result in **R4**.
- MUL R0, R1, R2 - Multiplies the values in **R1** and **R2**, storing the result in **R0**.

4. Direct (Absolute) Addressing Mode

Direct addressing finds out where operand resides in the memory and directly writes the operand location in the instruction itself. This gives simple access to global variables and fixed locations in memory but requires the full memory address to be encoded directly into the instruction, thereby making instructions larger. Direct addressing, though not complicated, has fallen out of use among modern architectures since it can be inefficient for both position-independent code and virtual memory systems.

Example:

- LOAD R6, [2000] - Loads data from memory location **2000** into **R6**.
- STORE R7, [3000] - Stores the value in **R7** into memory location **3000**.

5. Indirect Addressing Mode

Indirect addressing uses a memory location or register that contains the actual address of the operand rather than the operand itself. This provides powerful indirection capabilities essential for implementing pointers, data structures, and dynamic memory access. The trade-off is additional memory cycles required to fetch the effective address before accessing the actual operand.

Example:

- MOV R1, [R2] - **R2** contains the address of the operand, which is loaded into **R1**.
- ADD R3, [R4] - Adds the value at the address stored in **R4** to **R3**.

6. Relative Addressing Mode

Relative addressing computes the target address as an offset from the current program counter (PC) value, making it essential for position-independent code and branching instructions. This mode allows code to be relocated in memory without modification since addresses are expressed relative to the instruction's location rather than absolute memory positions.

Example:

- BEQ +8 (MIPS) - Branches **8 bytes ahead** if the zero flag is set.
- JMP -12 (x86) - Jumps **12 bytes backward** from the current PC.

7. Auto-increment/Auto-decrement Addressing Mode

These modes automatically update the pointer register after (auto-increment) or before (auto-decrement) accessing the operand, streamlining sequential memory access patterns common in stack operations and array processing. The increment/decrement amount typically matches the operand size for proper alignment.

Example:

- LDR R0, [R1], #4 (ARM) - Loads from **R1**, then increments **R1** by **4**.
- MOV (R2)+, R3 (PDP-11) - Stores **R3** at **R2**, then increments **R2**.

8. Stack Addressing Mode

A specialized form of addressing where an implicit stack pointer register is used to access operands in last-in-first-out (LIFO) order. Stack addressing is fundamental for subroutine calls, parameter passing, and temporary variable storage. The stack pointer is automatically adjusted during push and pop operations.

Example:

- PUSH AX (x86) - Pushes **AX** onto the stack.

o POP R0 (ARM) - Pops the top of the stack into **R0**.

Each addressing mode offers unique advantages for specific programming scenarios, and modern processors typically support multiple modes to handle diverse access patterns efficiently. Understanding these modes is essential for both compiler writers optimizing code generation and assembly programmers squeezing maximum performance from hardware.

3. Illustrate the cache mapping techniques with the help of diagrams.

The *cache mapping* bridges the mismatch of speed between the main memory and the processor. Whenever a cache hit occurs

- The word that is required is present in the memory of the cache.
- Then the required word would be delivered from the cache memory to the CPU.

And, whenever a cache miss occurs,

- The word that is required isn't present in the memory of the cache.
- The page consists of the required word that we need to map from the main memory.
- We can perform such a type of mapping using various different techniques of cache mapping.

Techniques of Cache Mapping

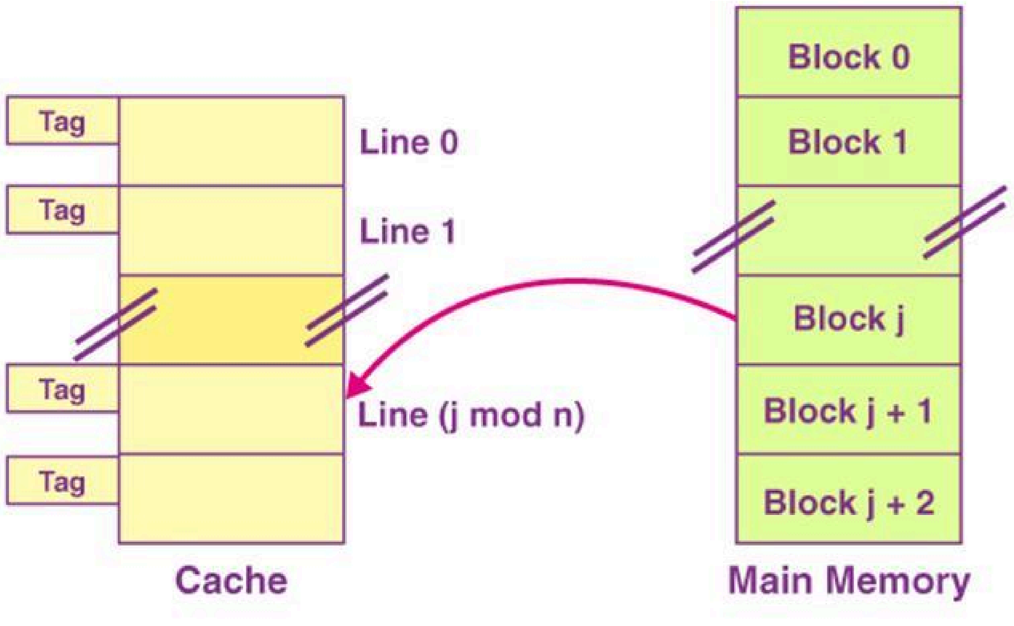
1. Direct Mapping

In the case of direct mapping, a certain block of the main memory would be able to map a cache only up to a certain line of the cache. The total line numbers of cache to which any distinct block can map are given by the following:

Cache line number = (Address of the Main Memory Block) Modulo (Total number of lines in Cache)

For example,

- Let us consider that particular cache memory is divided into a total of 'n' number of lines.
- Then, the block 'j' of the main memory would be able to map to line number only of the cache (j mod n).



Direct Mapping

The Need for Replacement Algorithm

In the case of direct mapping,

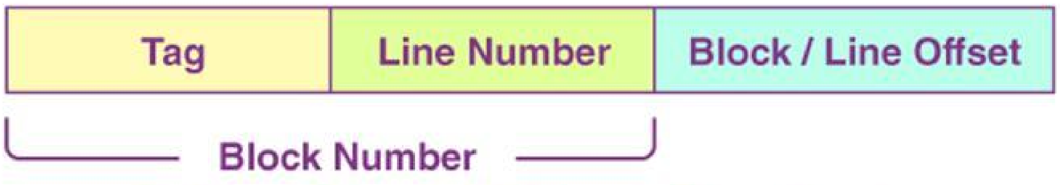
· There is no requirement for a replacement algorithm.

· It is because the block of the main memory would be able to map to a certain line of the cache only.

· Thus, the incoming (new) block always happens to replace the block that already exists, if any, in this certain line.

Division of Physical Address

In the case of direct mapping, the division of the physical address occurs as follows:



Division of Physical Address in Direct Mapping

2. Fully Associative Mapping

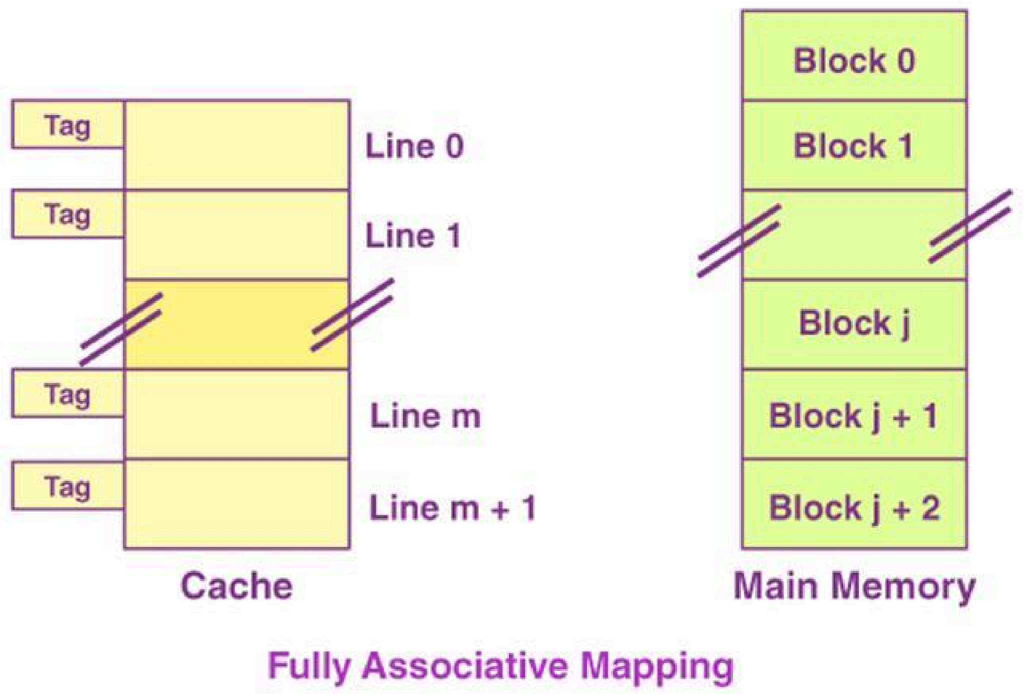
In the case of fully associative mapping,

· The main memory block is capable of mapping to any given line of the cache that's available freely at that particular moment.

· It helps us make a fully associative mapping comparatively more flexible than direct mapping.

For Example

Let us consider the scenario given as follows:



Here, we can see that,

- Every single line of cache is available freely.
- Thus, any main memory block can map to a line of the cache.
- In case all the cache lines are occupied, one of the blocks that exists already needs to be replaced.

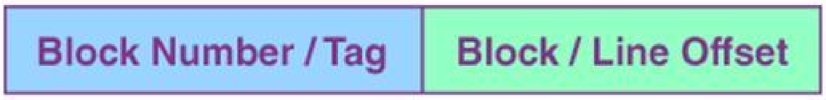
The Need for Replacement Algorithm

In the case of fully associative mapping,

- The replacement algorithm is always required.
- The replacement algorithm suggests a block that is to be replaced whenever all the cache lines happen to be occupied.
- So, replacement algorithms such as LRU Algorithm, FCFS Algorithm, etc., are employed.

Division of Physical Address

In the case of fully associative mapping, the division of the physical address occurs as follows:



Division of Physical Address in Fully Associative Mapping

3. K-way Set Associative Mapping

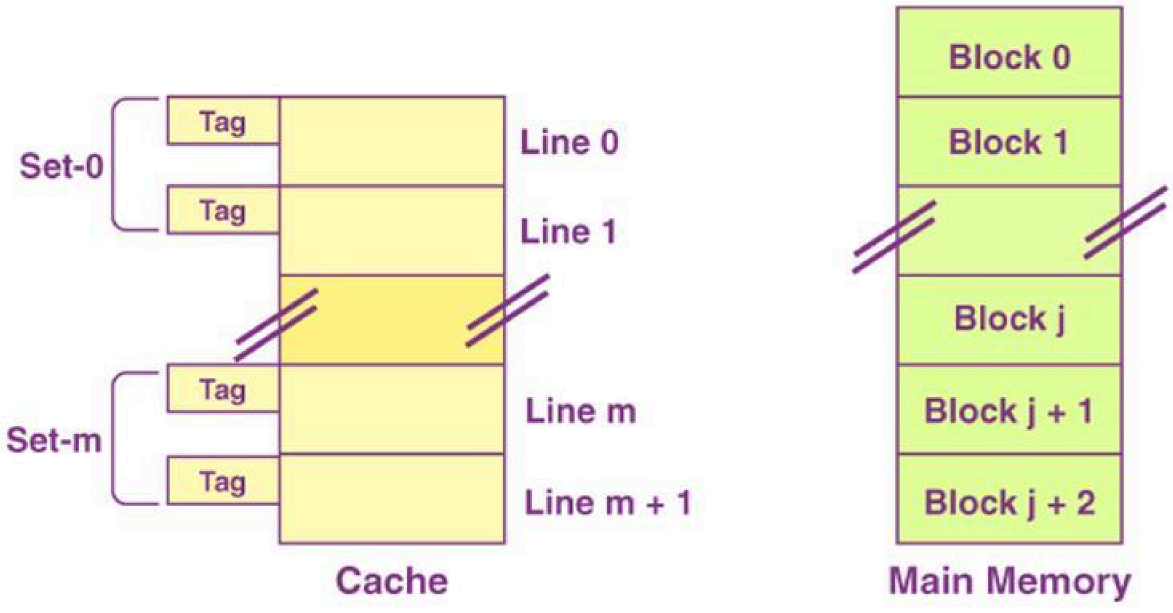
In the case of k-way set associative mapping,

- The grouping of the cache lines occurs into various sets where all the sets consist of k number of lines.
- Any given main memory block can map only to a particular cache set.
- However, within that very set, the block of memory can map any cache line that is freely available.
- The cache set to which a certain main memory block can map is basically given as follows:

Cache set number = (Block Address of the Main Memory) Modulo (Total Number of sets present in the Cache)

For Example

Let us consider the example given as follows of a two-way set-associative mapping:



2-Way Set Associative Mapping

In this case,

- k = 2 would suggest that every set consists of two cache lines.

· Since the cache consists of 6 lines, the total number of sets that are present in the cache = 6 / 2 = 3 sets.

· The block ‘j’ of the main memory is capable of mapping to the set number only (j mod 3) of the cache.

· Here, within this very set, the block ‘j’ is capable of mapping to any cache line that is freely available at that moment.

· In case all the available cache lines happen to be occupied, then one of the blocks that already exist needs to be replaced.

The Need for Replacement Algorithm

In the case of k-way set associative mapping,

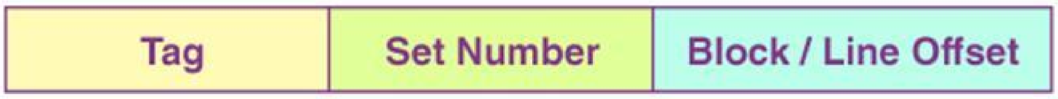
· The k-way set associative mapping refers to a combination of the direct mapping as well as the fully associative mapping.

· It makes use of the fully associative mapping that exists within each set.

· Therefore, the k-way set associative mapping needs a certain type of replacement algorithm.

Division of Physical Address

In the case of fully k-way set mapping, the division of the physical address occurs as follows:



Division of Physical Address in K-way Set Associative Mapping

Special Cases

· In case k = 1, the k-way set associative mapping would become direct mapping. Thus,

Direct Mapping = one-way set associative mapping

· In the case of k = The total number of lines present in the cache, then the k-way set associative mapping would become fully associative mapping.

4. **Explain instruction pipelining and analyze the operation of a five-stage instruction pipeline with the help of a timing diagram**.

An instruction in a pipeline refers to a single operation or command that is executed by the processor, which passes through multiple sequential stages (e.g., fetch, decode, execute) as part of a pipelined execution process.

- In a pipelined processor, instructions are divided into smaller steps, and each step is handled by a different part of the CPU simultaneously.
- Each instruction, whether it's a computation, memory access, or branch operation, flows through these stages like an item on an assembly line.
- The purpose of pipelining is to increase instruction throughput that is to complete more instructions in less time by overlapping their execution.

1. Instruction Fetch (IF)

The Instruction Fetch stage is the first step in the pipeline process. Here, the CPU retrieves an instruction from the program memory. The primary tasks in this stage include:

-*Program Counter (PC) Management:* The Program Counter holds the address of the next instruction to be fetched. It increments after each fetch, pointing to the subsequent instruction in the sequence.
 - *Instruction Memory Access:* The instruction is fetched from the instruction memory (usually a cache or RAM). This access is typically very fast due to the high-speed memory technologies used.
 - *Buffering*: The fetched instruction is placed into an Instruction Register (IR) or buffer to be used in the subsequent stages.

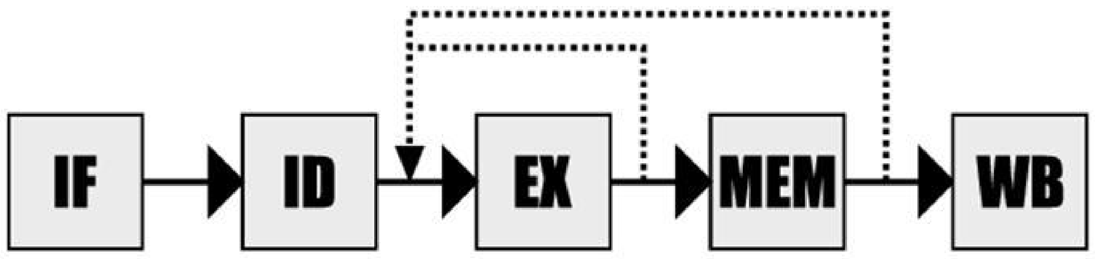
Efficiency at this stage is paramount as delays here can stall the entire pipeline, leading to performance bottlenecks. Advanced CPUs often use techniques like prefetching to mitigate potential delays.

2. Instruction Decode (ID)

Once the instruction is fetched, it enters the Instruction Decode stage. This stage involves interpreting the fetched instruction and preparing the necessary operands for execution. Key activities include:

- *Opcode Decoding:* The instruction is broken down into its constituent parts, such as the operation code (opcode), source operands, and destination operands.
 - *Register Read:* The required operands are read from the register file. Modern CPUs often have multiple read ports to facilitate simultaneous access to several registers.
 - *Instruction Classification:* Instructions are classified into various types (e.g., arithmetic, logical, load/store, control) to determine the subsequent steps in the pipeline.

This stage is also responsible for hazard detection and forwarding to resolve data dependencies and prevent pipeline stalls.


IF
ID
EX
MEM
WB
IF: Instruction fetch
ID: Instruction decode and register read
EX: Execute, address generation
MEM: Memory access
WB: Write back to registers

3. Execute (EX)

The Execute stage is where the actual computation or operation specified by the instruction takes place. This stage includes:

- *ALU Operations:* Arithmetic and logical operations are performed using the Arithmetic Logic Unit (ALU). This includes operations like addition, subtraction, AND, OR, and shifts.
- *Address Calculation:* For memory access instructions, the effective address is calculated by the ALU.
- *Branch Evaluation:* If the instruction involves a branch, the branch condition is evaluated, and the Program Counter may be updated accordingly.

The execution stage is critical for the overall performance, as complex instructions can take multiple cycles, potentially causing pipeline stalls.

4. Memory Access (MEM)

After execution, some instructions require access to memory to read or write data. The Memory Access stage handles these operations. Key functions include:

- *Load Operations:* Data is read from memory and placed into a register.
- *Store Operations:* Data from a register is written to memory.
- *Address Translation:* In systems with virtual memory, the effective address calculated during the Execute stage is translated to a physical address.

Efficiency in this stage is achieved through techniques like caching, which reduces latency by storing frequently accessed data closer to the CPU.

5. Write Back (WB)

The final stage in the pipeline is Write Back. Here, the results of the executed instructions are written back to the CPU's register file. This stage involves:

- *Register Write:* The computed data or the data fetched from memory is written into the destination register.

- *Completion Logging:* The completion of the instruction is logged, and any necessary updates to status registers or flags are made.

5. Write the sequence of control steps to execute the instruction ADD (R3), R1 on single bus architecture.

Instruction:

ADD (R3), R1

Meaning of the Instruction

R1←R1+M[R3]

That is:

- The **memory location whose address is in R3** is accessed
- Its contents are **added to R1**
- The result is stored back in **R1**

Sequence of Control Steps (Single Bus Architecture)

Step 1: Place Address on Bus

R3out → MARin

- Contents of **R3** are placed on the bus
- Loaded into **Memory Address Register (MAR)**

Step 2: Read Operand from Memory

Read

- Memory read operation is initiated

Step 3: Load Memory Data

MDRin ← M[MAR]

- Data from memory is loaded into **Memory Data Register (MDR)**

Step 4: Load First Operand into ALU Temporary Register

R1out → Yin

- Contents of **R1** loaded into temporary register **Y**

Step 5: Perform Addition in ALU

MDRout , Add → Zin

- ALU adds **Y + MDR**
- Result stored in **Z register**

Step 6: Store Result Back to R1

Zout → R1in

- Final result written back into **R1**

Final Result

$$R1 \leftarrow R1 + M[R3]$$

6. What is the purpose of a control unit? With neat sketches, explain the organization of the hardwired control unit in detail.

The control unit is the part of the computer's central processing unit (CPU) which directs the operation of the processor. It fetches instructions from memory, decodes them, and generates control signals to manage the ALU, memory, and I/O devices.

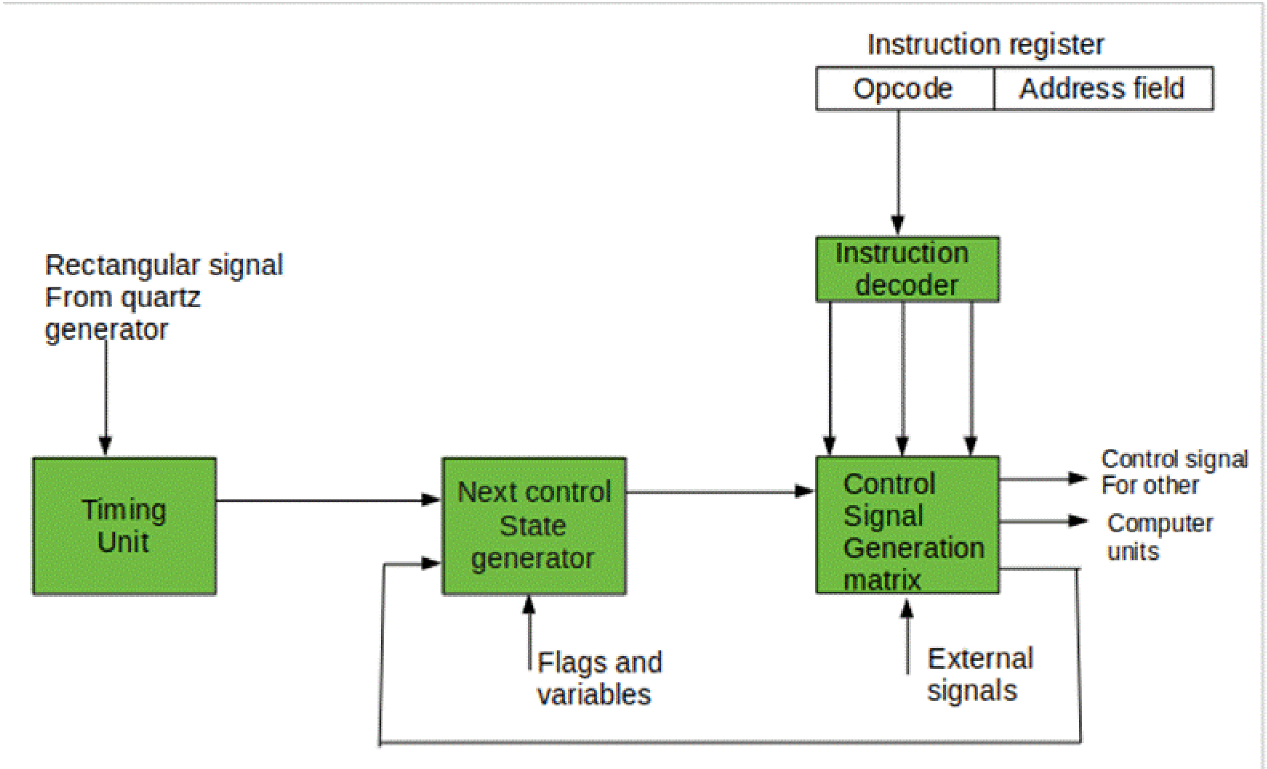
- Its role is to coordinate instruction execution, and its functions vary depending on CPU architecture.
- Devices that need a Control Unit (CU) include CPUs and GPUs.

Purpose of the Control Unit

- It coordinates the sequence of data movements into, out of, and between a processor's many sub-units.
- It interprets instructions.
- It controls data flow inside the processor.
- It receives external instructions or commands, which it converts to a sequence of control signals.
- It controls many execution units (i.e., ALU, data buffers and registers) contained within a CPU.
- It also handles multiple tasks, such as fetching, decoding, execution handling and storing results.

Hardwired Control Unit

In a Hardwired Control Unit, fixed hardware logic generates control signals based on the instruction's opcode. The opcode is decoded, and the decoder activates lines that feed a control signal generator matrix. This matrix, similar to a programmable logic array, combines decoded signals, control states, and external inputs to produce the required execution signals.


Instruction register
Opcode
Address field
Instruction decoder
Rectangular signal From quartz generator
Timing Unit
Next control State generator
Control Signal Generation matrix
Control signal For other Computer units
Flags and variables
External signals

Block diagram of a hardwired control unit of a computer

- Control signals must be generated throughout the instruction execution cycle, not at a single point. Accordingly, the control unit organizes a sequence of states, with some control signals fed back to the next state generator matrix.
- The timing unit (driven by a quartz generator) provides timing signals. When a new instruction arrives, the control unit begins fetching, moves through execution states, and responds to changes in timing, flags, or interrupts by shifting states.
- External signals such as interrupts trigger dedicated states for handling them. Flags and state variables guide the choice of states during execution.
- The final states of the cycle begin fetching the next instruction, and if a stop instruction is encountered, the control unit enters an OS state, waiting for the next command.