USN □□□□□□□□□□

Internal Assessment Test 2 – December 2024

Sub:	**Digital Design and Computer Organization**				Sub Code:	**BCS302**	Branch:	**AIML**
Date:	**13/12/24**	Duration:	**90 minutes**	Max Marks:	**50**	Sem/Sec:	**III -A, B, C**	**OBE**

Answer any FIVE FULL Questions			**MARKS**	**CO**	**RBT**
1	a	Explain Direct Memory Access with neat diagram	[10]	4	L2
2	a	With neat sketches, explain various methods for handling multiple Interrupts requests raised by Multiple devices.	[10]	4	L1
3	a	What is Cache memory? Analyze the three mapping functions of Cache memory.	[10]	4	L2
4	a	Illustrate in detail the ALU operation in a processor with example.	[10]	5	L3
5	a	Analyze how does execution of a complete instruction carry out	[10]	5	L3
6	a	What is pipelining? Explain the performance of the pipeline with an example.	[10]	5	L2

CI **CCI** **HoD**

USN □□□□□□□□□□

Internal Assessment Test 2 – December 2024

Sub:	**Digital Design and Computer Organization**				Sub Code:	**BCS302**	Branch:	**AIML**
Date:	**13/12/24**	Duration:	**90 minutes**	Max Marks:	**50**	Sem/Sec:	**III -A, B, C**	**OBE**

Answer any FIVE FULL Questions			**MARKS**	**CO**	**RBT**
1	a	Explain Direct Memory Access with neat diagram	[10]	4	L2
2	a	With neat sketches, explain various methods for handling multiple Interrupts requests raised by Multiple devices.	[10]	4	L1
3	a	What is Cache memory? Analyze the three mapping functions of Cache memory.	[10]	4	L2
4	a	Illustrate in detail the ALU operation in a processor with example.	[10]	5	L3
5	a	Analyze how does execution of a complete instruction carry out	[10]	5	L3
6	a	What is pipelining? Explain the performance of the pipeline with an example.	[10]	5	L2

CI **CCI** **HoD**

USN

Internal Assessment Test 2 – December 2024

Sub:	**Digital Design and Computer Organization**			Sub Code:	**BCS302**	Branch:	**AIML**	
Date:	**13/12/24**	Duration:	**90 minutes**	Max Marks:	**50**	Sem/Sec:	**III**	**OBE**

		Answer any FIVE FULL Questions	MARKS	CO	RBT
1	a	Explain Direct Memory Access with neat diagram	[10]	4	L2

Direct Memory Access (DMA) is a feature that allows peripheral devices (such as hard drives, sound cards, or network interfaces) to access the system's main memory (RAM) directly, without involving the CPU. DMA improves system performance by offloading data transfer tasks from the CPU, enabling faster and more efficient data movement.

How DMA Works:

1. **Peripheral Device Request**: A peripheral device (e.g., a disk controller) needs to transfer data to or from memory.
2. **DMA Controller (DMAC)**: The peripheral sends a request to the DMA controller. The DMA controller manages the data transfer process.
3. **DMA Controller Actions**:
   - It sends a request to the CPU to grant access to memory.
   - Once the CPU acknowledges, the DMA controller takes control of the system's memory bus.
4. **Data Transfer**: The DMA controller moves the data directly between the peripheral and the memory without involving the CPU.
5. **Completion**: Once the transfer is complete, the DMA controller sends an interrupt to the CPU, notifying it that the transfer is done.

Types of DMA:

1. **Burst Mode DMA**: The DMA controller transfers data in bursts, and the CPU is locked out of the memory during the transfer.
2. **Cycle Stealing DMA**: The DMA controller steals a cycle of the CPU's time to transfer one byte of data at a time, allowing the CPU to continue processing while data is being moved.
3. **Block Mode DMA**: The DMA controller transfers data in blocks without needing the CPU's intervention in between, but the CPU is locked out during the transfer.

Advantages of DMA:

- **Reduced CPU Load**: DMA offloads data transfer tasks, freeing the CPU for other operations.
- **Faster Data Transfers**: DMA provides direct communication between devices and memory, which is often faster than involving the CPU.
- **Efficient Resource Usage**: Enables high-speed data transfer, which is essential for applications requiring large amounts of data movement, such as video and audio processing.

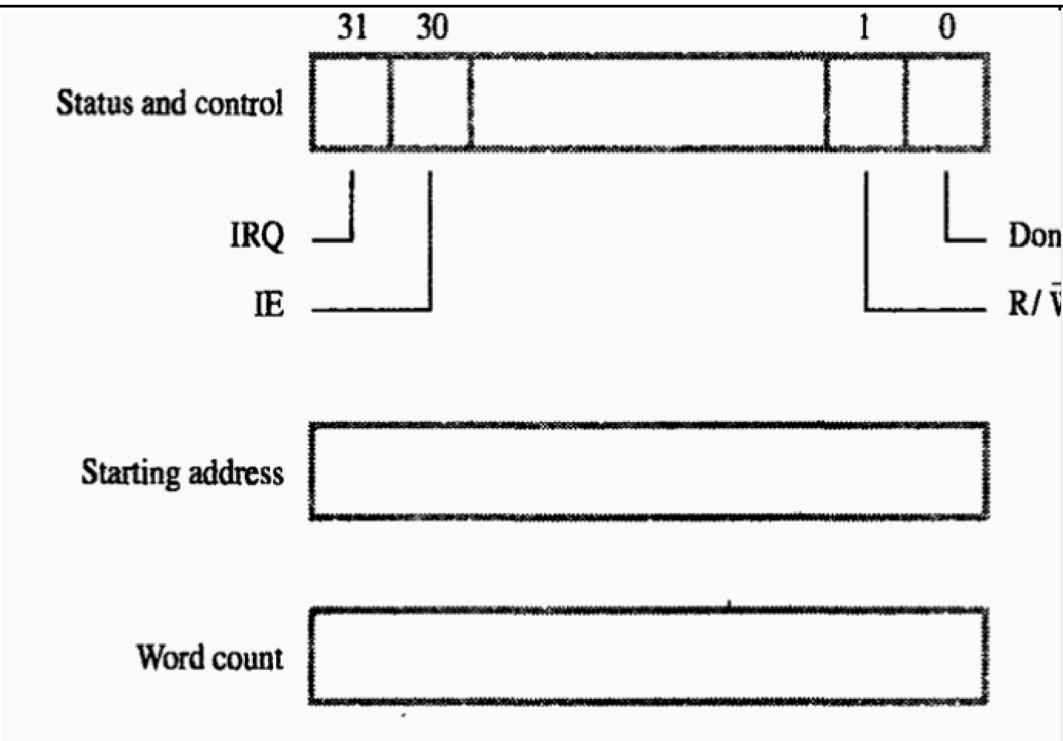

Figure 4.18 Registers in a DMA interface.

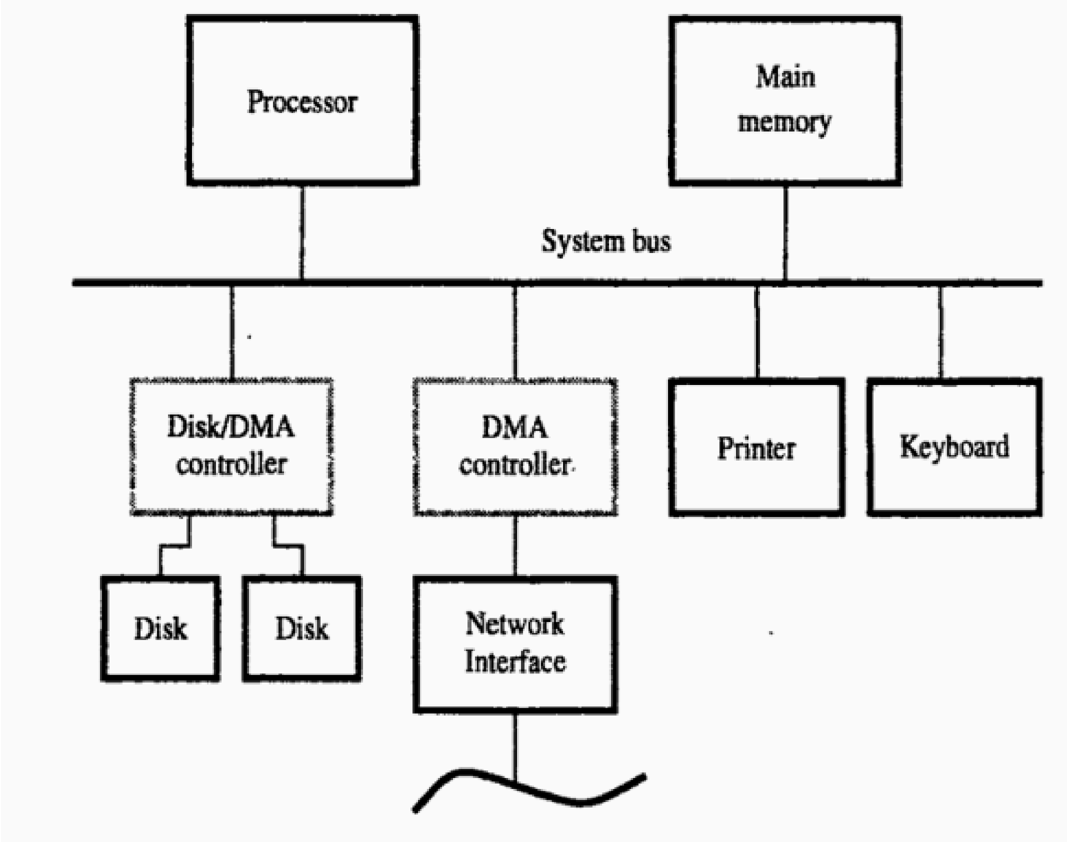

Figure 4.19 Use of DMA controllers in a computer system.

In this diagram:

- The peripheral device initiates the data transfer via the DMA controller.
- The DMA controller directly transfers data between the peripheral and memory.
- The CPU is minimally involved, which reduces the load on the processor.

This system ensures that large data transfers are handled quickly and efficiently, without overburdening the CPU.

2	a	When multiple devices generate interrupt requests, the system must handle these requests efficiently to ensure that the CPU can respond to each device in an orderly manner. There are several methods for handling multiple interrupt requests:	[10]	4	L1

1. Priority Interrupts

In a system with priority interrupts, each interrupt source (device) is assigned a priority level. When multiple devices generate interrupt requests, the interrupt with the highest priority is serviced first. The CPU or interrupt controller examines the priority of each interrupt request and handles the highest-priority interrupt.

Working:

- Each interrupt source has a priority number (e.g., a number from 0 to 7, where 0 is the highest priority).
- When multiple devices request an interrupt, the interrupt controller ensures that the highest-priority interrupt is processed first.
- Lower-priority interrupts can be serviced after the higher-priority interrupt is handled.

Sketch for Priority Interrupts:

sql

Copy code

```
+---------------------------+
| Interrupt Request from     |
| Device A (Priority 3)      |
+---------------------------+
           |
           v
                             +---------------------------+
+---------------------------+
 | Interrupt Request from       |             | Interrupt
Request from     |
 | Device B (Priority 5)          |----->---| Device C
(Priority 1)      |
                             +---------------------------+
+---------------------------+
           |                                     |
           v                                     v
       CPU checks                           CPU checks
        priority, handles                    highest priority
first
        Device C interrupt                  (Device C in this
case)
       (Highest priority)                    (Priority 1)
```

2. Vectored Interrupts

In a vectored interrupt system, each interrupt source is assigned a specific vector or address in memory. When an interrupt is raised, the CPU or interrupt controller uses the interrupt vector to jump to the correct memory address (interrupt service routine or ISR) to handle the interrupt.

Working:

- Each device has a unique vector or memory address associated with it.
- The interrupt controller sends the vector address to the CPU when an interrupt request is raised.
- The CPU then jumps to the address of the ISR corresponding to the device that raised the interrupt.

Sketch for Vectored Interrupts:

sql
Copy code

```
+---------------------------+
| Interrupt Request from     |
| Device A (Vector 0x10)     |
+---------------------------+
           |
           v
   +----------------------+
   | Interrupt Controller|
   | (Sends Vector 0x10) |
   +----------------------+
           |
           v
+---------------------------+
| CPU Jumps to Vector 0x10  |
| to Handle Device A Interrupt|
+---------------------------+
           |
           v
     Handles Device A ISR
```

3. Polling

In polling, the CPU periodically checks each device (or interrupt controller) to see if any device has raised an interrupt. The CPU does not automatically respond to an interrupt until it checks each device.

Working:

- The CPU regularly checks the status of each device to determine if it needs attention.
- The CPU queries all devices and handles the interrupt for the device that has requested it.
- Polling can waste CPU cycles if no device has raised an interrupt, as the CPU keeps checking.

Sketch for Polling:

mathematica
Copy code

```
+-------------------------+
| CPU Checks Device A       |
| for Interrupt             |
+-------------------------+
           |
           v
+-------------------------+
+-------------------------+
|  No Interrupt from Device |                  | Interrupt
Request from   |
|  A, Checks Device B          |----->---| Device C (Poll
Next)    |
+-------------------------+
+-------------------------+
           |                                            |
           v                                            v
        No Interrupt                                    CPU
Handles Device C Interrupt
       from Device B                                    (If
Device C has raised interrupt)
           |
           v
       CPU Continues Polling Other Devices
```

4. Non-Maskable Interrupt (NMI)

A Non-Maskable Interrupt (NMI) is a type of interrupt that cannot be ignored or disabled by the CPU. It is typically used for critical hardware errors, such as memory failures. While other interrupts may be masked or disabled, NMIs are always processed immediately.

Working:

- NMIs have the highest priority and cannot be masked or delayed.
- When an NMI occurs, the CPU immediately stops processing any other interrupts or instructions and services the NMI.
- These interrupts are used for critical system failures.

Sketch for Non-Maskable Interrupt (NMI):

sql

Copy code

```
+----------------------------+
| Interrupt Request from     |
| Device A (Interrupt Masked)|
+----------------------------+
           |
           v
      CPU checks priority
      of all interrupts
           |
           v
+-----------------------------+
| NMI (Critical Error)        |  <-- Highest Priority
+-----------------------------+
           |
           v
       CPU Handles NMI
       (Interrupt Cannot Be Masked)
```

5. Interrupt Nesting

In interrupt nesting, if a higher-priority interrupt occurs while a lower-priority interrupt is being serviced, the CPU will pause the current interrupt service routine (ISR) and handle the higher-priority interrupt. Once the higher-priority interrupt is handled, the CPU resumes the lower-priority ISR.

Working:

- The CPU suspends the current ISR when a higher-priority interrupt is received.
- After handling the higher-priority interrupt, the CPU resumes the lower-priority ISR where it left off.
- This method is useful in systems where multiple devices may raise interrupts that need to be handled based on their priority.

Sketch for Interrupt Nesting:

css

Copy code

```
+---------------------------+
+---------------------------+
| CPU Handles Device A ISR    |               | Interrupt
Request from     |
| (Low Priority)                |----->-------| Device B
(High Priority)   |
+---------------------------+
+---------------------------+
           |                                          |
           v                                          v
      CPU Suspends Device A ISR                  CPU Handles
Device B ISR
           |                                          |
           v                                          v
   After Device B is Handled, CPU                CPU Resumes
Device A ISR
  Continues Device A ISR Processing
```

6. Programmable Interrupt Controller (PIC)

A PIC manages multiple interrupt sources and determines the priority and order in which the CPU should handle them. It can also mask interrupts (disable certain interrupt requests) to prevent lower-priority interrupts from interrupting critical operations.

Working:

- The PIC receives interrupt requests from multiple devices.
- It then determines the priority of each interrupt and sends the interrupt vector to the CPU.
- The CPU processes the highest-priority interrupt first.
- The PIC can also mask or disable certain interrupts when required.

Sketch for PIC:

sql
Copy code

```
+----------------------------+
| Interrupt Request from     |
| Device A, Device B, etc.   |
+----------------------------+
           |
           v
  +------------------------+
  | Programmable Interrupt |
  | Controller (PIC)       |
  +------------------------+
           |
           v
  +--------------------------+
  | Sends Interrupt Vector   |
  | to CPU Based on Priority |
  +--------------------------+
           |
           v
   CPU Handles Interrupt
   Based on Priority
```

3	a	**Cache memory is a small, high-speed memory located between the CPU and the main memory (RAM). Its primary purpose is to store frequently accessed data and instructions to improve the speed and efficiency of the system. The cache is faster than the main memory, and it helps reduce the time it takes for the CPU to access data, significantly boosting system performance.** **Cache memory works by storing copies of data from the main memory, so when the CPU requests data, it first checks if it's available in the cache. If the data is found (a cache hit), the CPU can access it much more quickly. If the data is not found (a cache miss), it has to be fetched from the slower main memory.** **Three Mapping Functions of Cache Memory** **In cache memory, mapping refers to the technique used to determine where in the cache a particular block of data from main memory will be stored. There are three main types of cache mapping:** **1. Direct-Mapped Cache** **2. Fully Associative Cache** **3. Set-Associative Cache** **Each method has its own approach to placing data into cache and differs in terms of complexity, speed, and efficiency.** --- **1. Direct-Mapped Cache** **In a direct-mapped cache, each block of memory can be mapped to exactly one cache line. This means that for a given block of data in main memory, there is a specific location in the cache where it will always be placed.** **How it works:** ● **The memory address is divided into three parts: Tag, Index, and Block offset.** ○ **Tag: Identifies which block of data in memory is being referenced.** ○ **Index: Determines which cache line (or set) the block of data will be placed in.** ○ **Block offset: Indicates the specific location within the cache line.** **Example:** **If you have a cache with 4 lines and memory with 16 blocks, the index will determine which cache line a block of data from memory should go into.**	[10]	4	L2

Each block of memory will be mapped to exactly one line in the cache.

- **Advantages:**
  - **Simple and fast, since each block of memory is mapped to a specific cache line.**
  - **Low hardware complexity.**
- **Disadvantages:**
  - **Conflict misses: If multiple memory blocks map to the same cache line, they will overwrite each other. This results in cache misses, even if there is enough space in the cache overall.**

---

2. Fully Associative Cache

In a fully associative cache, any block of memory can be placed in any cache line. There is no restriction on which memory block goes into which line of the cache.

How it works:

- **The memory address is divided into two parts: Tag and Block offset.**
  - **Tag: Identifies the block of data being referenced.**
  - **Block offset: Indicates the location within the block.**
- **There is no "index" part because any cache line can hold any block of data.**

Example:

If you have a cache with 4 lines and memory with 16 blocks, any block of memory can be placed in any of the 4 cache lines.

- **Advantages:**
  - **No conflict misses since there are no restrictions on which cache line a block can occupy.**
  - **Better for scenarios where data access patterns are unpredictable.**
- **Disadvantages:**
  - **Slower lookup: Since any block can be stored in any line, checking if the block is in the cache involves searching all cache lines. This requires more time and complexity, making it slower than direct-mapped caches.**
  - **Higher hardware complexity: Requires extra logic to manage the mapping and searching of cache lines.**

---

3. Set-Associative Cache

A set-associative cache is a compromise between direct-mapped and fully associative caches. In this type of cache, the cache is divided into multiple

sets, and each set contains multiple lines. A block of memory can be placed in any line within a set, but it cannot be placed in just any line across the entire cache.

How it works:

- **The memory address is divided into three parts: Tag, Set index, and Block offset.**
  - **Tag: Identifies the block of data.**
  - **Set index: Determines which set in the cache the block will go to.**
  - **Block offset: Specifies the location within the block.**
- **The cache is divided into NNN sets, and each set contains MMM cache lines. A block can be mapped to any line within its assigned set.**

Example:

If you have a cache with 4 sets and 2 lines per set, a block of memory will be placed in one of the two lines within the set determined by the set index. If a block from memory maps to a set with available space, it is stored there.

- **Advantages:**
  - **Fewer conflict misses than direct-mapped cache because each set contains multiple lines. This allows for some flexibility in where blocks of data can be stored.**
  - **Faster lookups compared to fully associative caches since only a subset of cache lines needs to be searched.**
- **Disadvantages:**
  - **More complex than direct-mapped cache but less complex than fully associative cache.**
  - **Performance depends on the number of sets and lines in each set. If there are too many sets or too few lines per set, cache performance may degrade.**

Comparison of Mapping Techniques

Mapping Type	Complexity	Lookup Speed	Conflict Misses	Cache Miss Rate	Hardware Cost
Direct-Mapped	Low	Fast	High (if collisions occur)	High (in some cases)	Low

Fully Associative	**High**	**Slow**	**None**	**Low**	**High**
Set-Associative	**Medium**	**Moderate**	**Reduced (less than direct-mapped)**	**Moderate (balanced)**	**Medium**

4 a

Arithmetic Logic Unit (ALU) Operation in a Processor

The **Arithmetic Logic Unit (ALU)** is a fundamental component of the CPU that performs arithmetic and logical operations. It operates on data that it retrieves from registers or memory and outputs the result back to registers or memory. The operations performed by the ALU include addition, subtraction, multiplication, division (arithmetic operations), as well as logical operations like AND, OR, XOR, NOT, and comparison operations (logical operations).

[10] 5 L3

Basic ALU Operations

1. **Arithmetic Operations**:
   - **Addition**: Adds two numbers.
   - **Subtraction**: Subtracts one number from another.
   - **Multiplication**: Multiplies two numbers (in some processors).
   - **Division**: Divides one number by another (in some processors).
   - **Increment/Decrement**: Adds or subtracts 1 from a number.
2. **Logical Operations**:
   - **AND**: Performs a bitwise AND operation between two numbers.
   - **OR**: Performs a bitwise OR operation between two numbers.
   - **XOR**: Performs a bitwise XOR (exclusive OR) operation.
   - **NOT**: Performs a bitwise NOT operation (inverts each bit).
3. **Shift Operations**:
   - **Shift Left (<<)**: Shifts all bits to the left, which generally multiplies the value by 2.
   - **Shift Right (>>) or Arithmetic Right Shift**: Shifts all bits to the right, dividing the value by 2.
4. **Comparison Operations**:
   - **Equality**: Compares two numbers for equality.
   - **Less Than / Greater Than**: Compares two numbers to check if one is less than or greater than the other.

Structure of the ALU

The ALU typically consists of:

- **Inputs**: The ALU receives two inputs (A and B), which are usually numbers stored in registers or provided by memory.
- **Control Unit**: The control unit of the CPU sends control signals to the ALU, specifying which operation the ALU should perform.
- **ALU Operation Control Lines**: The ALU has control lines that select the operation, such as:
  - Add, Subtract, AND, OR, etc.
- **Output**: The result of the operation is returned as output (often to a register or memory).

ALU Example: Addition and Logical AND Operations

Let's consider two 4-bit numbers, **A = 1011 (binary)** and **B = 0110 (binary)**, and examine the ALU operations that could be performed on them.

1. Addition Operation (A + B)

In this case, the ALU will perform a binary addition operation on the two numbers **A** and **B**.

A = 1011
B = 0110

The ALU performs the following operation (from least significant bit to most significant bit):

sql
Copy code

```
 1011
+ 0110
------
  10001  (Result: 10001 in binary, which is 17 in decimal)
```

- The carry bit is 1, so the final result is **10001** (which is 17 in decimal).

2. Logical AND Operation (A AND B)

The ALU can also perform logical operations. In this case, we will perform a **bitwise AND** operation on the two numbers **A** and **B**.

A = 1011
B = 0110

The bitwise AND operation compares each bit of **A** and **B** and outputs 1 if both bits are 1, otherwise, it outputs 0.

less
Copy code

```
  A = 1011
   B = 0110
   ----------
A AND B = 0010   (Result: 0010 in binary, which is 2 in decimal)
```

- The **AND** operation between these two binary numbers results in **0010** (which is 2 in decimal).

Step-by-Step ALU Operation Example

Let's break down the addition and logical AND operations in detail, assuming that the ALU is controlled by a set of control signals that select the operation (such as addition or AND).

Step 1: ALU Setup

- The processor has the two input operands, **A** and **B**, ready in the registers (or fetched from memory).
- The **Control Unit** sends control signals to the ALU, telling it which operation to perform (e.g., addition or AND).

Step 2: ALU Operation – Addition

- For addition, the control signals tell the ALU to use an **add** operation.
- The ALU performs a **binary addition** of **A** and **B**.

First, the bits are aligned:
css
Copy code

```
A = 1011
B = 0110
```

  -
  - Then, each bit is added starting from the least significant bit, handling any carry-over from the previous addition.
  - The result is stored in a temporary register and eventually written back to a result register or memory.

Step 3: ALU Operation – Logical AND

- For the **AND** operation, the control signals instruct the ALU to perform the logical AND operation between **A** and **B**.
- Each bit of **A** is compared with the corresponding bit of **B**, and the result is stored.
  - **A**: 1011
  - **B**: 0110
  - **A AND B**: 0010 (binary result)

Step 4: ALU Output

- After performing the operation (addition or AND), the ALU outputs the result. In the case of addition, it would be **10001**, and for the AND operation, it would be **0010**.

- The result is then stored in a register, or it could be written back to memory.

Control Signals in ALU

The ALU is controlled by a set of signals from the **Control Unit** (CU). These signals specify which operation the ALU should perform. The control lines typically include:

- **Opcode**: Specifies the operation (e.g., ADD, AND, OR, etc.).
- **Shift control**: Determines whether a shift operation is required (left or right shift).
- **Comparison flags**: These include flags like Zero, Carry, Sign, and Overflow, which are updated based on the result of the operation.

For example:

- **ADD operation** might be selected by sending an opcode like **0001**.
- **AND operation** might be selected by sending an opcode like **0010**.

Example: Control Signals for Addition and AND

Operation	Opcode (Control Signal)
Addition (A + B)	0001
AND (A AND B)	0010

In this way, the control unit communicates with the ALU to specify the operation.

ALU Flags

The ALU typically updates certain flags after performing an operation. These flags provide important information about the result of the operation:

- **Zero Flag (Z)**: Set if the result of the operation is zero.
- **Carry Flag (C)**: Set if there is a carry-out from the most significant bit (useful for addition and subtraction).
- **Overflow Flag (O)**: Set if the result of an operation exceeds the range of the result (e.g., in signed addition).
- **Negative Flag (N)**: Set if the result of the operation is negative (useful in signed arithmetic).

Example of ALU Operation: Flags for Addition

Let's take the example where **A = 1011** and **B = 0110** (binary) for addition:

markdown
Copy code

```
 1011 (A)
+ 0110 (B)
---------
  10001 (Result: 17 in decimal)
```

- **Carry Flag**: Set (because there was a carry-out from the most significant bit).

		● **Zero Flag**: Not set (the result is not zero). ● **Overflow Flag**: Not set (no overflow in a 4-bit result). ● **Negative Flag**: Not set (the result is positive).			

5 a

The execution of a complete instruction in a processor involves a series of well-defined steps that transform a high-level command (written in machine language or assembly) into an action the processor can execute. This process is carried out through a series of stages, collectively known as the **Instruction Cycle** or **Fetch-Decode-Execute cycle**. Here's an in-depth analysis of each phase involved in the execution of an instruction:

1. Instruction Cycle Overview

The instruction cycle can be broken down into several steps:

1. **Fetch**: Retrieve the instruction from memory.
2. **Decode**: Interpret the instruction to determine what operation to perform.
3. **Execute**: Perform the operation specified by the instruction.
4. **Store (optional)**: Write the result back to memory or a register (if necessary).

The process happens continuously in a CPU, with the processor executing one instruction after another, unless interrupted.

Detailed Steps in Instruction Execution

Step 1: Fetch the Instruction

In the **fetch** phase, the CPU retrieves an instruction from memory (typically from the program counter (PC)). The program counter holds the memory address of the next instruction to be executed.

- The **Instruction Pointer (IP)** or **Program Counter (PC)** contains the address of the next instruction in memory.
- The CPU sends the address in the PC to the **Memory Address Register (MAR)**.
- The instruction located at that address is retrieved from memory and placed in the **Instruction Register (IR)**.

Process of fetching the instruction:

1. The address in the PC is loaded into the MAR.
2. The CPU reads the instruction from memory at the address specified by the MAR.
3. The instruction is placed into the IR.
4. The PC is incremented to point to the next instruction (unless modified by control flow instructions like jumps or branches).

At this point, the instruction is ready for decoding.

Step 2: Decode the Instruction

[10] 5 L3

In the **decode** phase, the instruction in the IR is decoded to determine what action the CPU needs to perform. The instruction is typically broken down into its components, which include:

- **Opcode**: Specifies the operation (e.g., ADD, MOV, SUB).
- **Operands**: Specifies the data or addresses on which the operation should be performed. These could be immediate values, registers, or memory addresses.

The **Control Unit (CU)** is responsible for decoding the instruction. It interprets the opcode and generates the necessary control signals to carry out the operation.

- **Operand Fetching**: If the instruction requires data from memory or a register, the Control Unit fetches the operands (the source data for the operation) from the appropriate registers or memory addresses. These operands could be in the **Register File** or in memory.
- **Address Calculation**: For instructions involving memory access, the effective address is calculated (e.g., adding an offset to a base address).

The decoded instruction is now ready to be executed.

Step 3: Execute the Instruction

In the **execute** phase, the processor performs the actual operation defined by the instruction. This could involve arithmetic or logical operations, data movement, or control flow changes. The ALU (Arithmetic Logic Unit) is typically used for arithmetic or logical operations, while the data path handles data movement.

Possible actions during execution:

- **Arithmetic or Logical Operations**: If the instruction is an arithmetic or logical operation (e.g., ADD, SUB, AND, OR), the operands are processed by the ALU.
- **Data Transfer**: For instructions like MOV (move), data is transferred from one location (e.g., memory to a register, or register to memory).
- **Control Flow Change**: If the instruction is a jump, branch, or call, the **Program Counter (PC)** is updated to the address specified by the instruction (for example, setting the PC to a new location).
- **Memory Access**: For load/store instructions, data is read from or written to memory.

Step 4: Write Back the Result (if needed)

If the instruction involves a result that needs to be stored (e.g., a result from an addition), the result is written back to the destination location, which could be:

- A **register** in the Register File.
- **Memory** (if the instruction involves a store operation).

This step completes the execution of the instruction.

Control Signals and the Role of the Control Unit

The **Control Unit (CU)** plays a crucial role throughout the instruction cycle. It generates control signals that dictate the behavior of various components of the CPU. These signals include:

- **Memory Read/Write**: Indicates whether data should be read from or written to memory.
- **ALU Control**: Specifies the operation to be performed by the ALU (e.g., addition, subtraction, etc.).
- **Register Read/Write**: Determines whether data should be read from or written to a register.
- **PC Update**: Controls the increment or modification of the program counter.

The control signals ensure the correct sequence of events in the instruction cycle.

Example: Executing an Instruction

Let’s walk through an example using a simple instruction in an imaginary CPU:
ADD R1, R2, R3

This instruction adds the contents of registers **R2** and **R3**, then stores the result in **R1**.

Step 1: Fetch the Instruction

1. The program counter (PC) holds the address of the instruction **ADD R1, R2, R3**.
2. The address is loaded into the Memory Address Register (MAR).
3. The CPU reads the instruction from memory and places it in the Instruction Register (IR).
4. The PC is incremented to the next instruction address.

Step 2: Decode the Instruction

1. The Control Unit reads the opcode **ADD** from the instruction in the IR.
2. The Control Unit determines that the ALU should perform an **addition** operation.

3. The operands are identified as registers **R2** and **R3**, which need to be read from the register file.
4. The destination register is identified as **R1**.

Step 3: Execute the Instruction

1. The Control Unit generates the appropriate signals to read registers **R2** and **R3**.
2. The values in **R2** and **R3** are passed to the ALU.
3. The ALU performs the addition of **R2** and **R3**.
4. The result of the addition is stored in a temporary register.

Step 4: Write Back the Result

1. The result of the addition is written back to **R1**.
2. The instruction is now complete, and the processor proceeds to fetch the next instruction.

Pipelining and Parallelism

In modern processors, **pipelining** is used to improve instruction throughput. Pipelining involves breaking the instruction cycle into multiple stages, allowing multiple instructions to be processed at the same time in different stages of execution. For example, while one instruction is being fetched, another can be decoded, and a third can be executed.

This improves the efficiency of the CPU, as each part of the instruction cycle is happening in parallel, speeding up the overall process.

Summary of Instruction Execution

The execution of an instruction in a processor involves the following steps:

1. **Fetch**: The instruction is retrieved from memory.
2. **Decode**: The instruction is decoded by the Control Unit, and the required operands are fetched.
3. **Execute**: The CPU executes the operation, typically involving the ALU or data transfer.
4. **Write Back**: The result of the execution is written back to a register or memory.

6	a	This process happens repeatedly for each instruction in the program, ensuring that the program is executed as intended. In more advanced systems, pipelining and parallelism allow for more efficient execution of multiple instructions at once.	[10]	5	L3

Pipelining is a technique used in computer architecture and data processing where multiple stages or tasks are executed simultaneously in a sequence, with each stage processing a different part of the task. It allows for the overlap of operations, meaning while one stage is processing data, the next stage can begin working on new data, resulting in improved overall throughput.

In simpler terms, pipelining allows a system to work on multiple tasks concurrently by breaking them into smaller, independent stages that can be processed in parallel.

Example of Pipelining

Consider a simple **5-stage instruction pipeline** in a CPU. Each instruction goes through 5 stages:

1. **Fetch (IF)**: Retrieve the instruction from memory.
2. **Decode (ID)**: Decode the instruction to understand what operation it needs to perform.
3. **Execute (EX)**: Perform the operation (e.g., arithmetic, logic, etc.).
4. **Memory (MEM)**: Access memory (read/write).
5. **Write-back (WB)**: Write the result back to the register.

These stages work in parallel for multiple instructions. Let’s see how it works with a simple example:

Instructions:

1. **Instruction 1**: Add A and B
2. **Instruction 2**: Subtract C from D
3. **Instruction 3**: Multiply E and F

Pipeline Stages:

Assume that each instruction takes one clock cycle per stage, and each stage is independent. Without pipelining, each instruction would require 5 cycles to complete. However, with pipelining, each stage works concurrently, as shown in the following timeline:

Cycle	Instruction 1	Instruction 2	Instruction 3

1	IF		
2	ID	IF	
3	EX	ID	IF
4	MEM	EX	ID
5	WB	MEM	EX
6		WB	MEM
7			WB

Performance of Pipelining

- **Increased Throughput**: As the stages are overlapped, each instruction begins its next stage in a subsequent cycle, allowing more instructions to complete in a given period of time. In the example above, after the pipeline is filled (i.e., after 5 cycles), one instruction completes every cycle.
- **Throughput = 1 instruction per cycle** (after pipeline is filled). This is an improvement over the non-pipelined approach, which would have taken 15 cycles for 3 instructions (5 cycles per instruction).
- **Latency**: The time it takes for a single instruction to go from start to finish remains the same (5 cycles), but the total time for multiple instructions reduces because of concurrent processing.

Example:

Without pipelining:

- **3 instructions**: 3 × 5 = 15 cycles

With pipelining:

- **3 instructions**: 5 cycles to fill the pipeline, then 1 cycle for each instruction

		thereafter, so the total time = 5 (pipeline filling) + 3 (one per instruction) = 8 cycles. **Pipeline Hazards** While pipelining offers significant performance improvements, there are also potential **hazards** that can reduce its effectiveness: 1. **Data Hazards**: Occur when instructions depend on the results of previous instructions. ○ Example: If Instruction 2 needs the result of Instruction 1 (e.g., `Instruction 1: R1 = A + B`, `Instruction 2: R2 = R1 - C`), there could be a delay in the pipeline. 2. **Control Hazards**: Arise from branch instructions (e.g., conditional jumps), which can disrupt the flow of instructions in the pipeline. 3. **Structural Hazards**: Occur when the hardware cannot support the combination of instructions in the pipeline (e.g., insufficient resources like ALUs or memory ports).			
	a		[10]	5	L3

6	a		[10]	3	L3

CI	**CCI**	**HoD**