ADVANCED COMPUTER ARCHITECTURE
LAB RECORD

Roll No	**25MSCSCKK0001**
Name	**JIDHUN.P.P**
Class	**MSC. COMPUTER SCIENCE**

Department of Computer Science
School of Engineering & Technology
Pondicherry University
Karaikal -609605, Puducherry, India

Department of Compute Science School
of Engineering &Technology
Pondicherry University
Karaikal-609 605,
Puducherry, India

BONAFIDE CERTIFICATE

Name: _____

Reg. No: _____**Class:** _____

Course Title: _____

Course Code: _____

Certified that this is the bonafide record of work done by me during **Odd / Even** Semester of **2025 – 2027** and submitted for the Practical Examination on:

Date: _____

Staff In-Charge **Head of the Department**

Examiners:

1. _____
2. _____

Index of Lab Exercises

Exercise 1: Simulation of Computer Components

Lab Program 1: Implement Basic Logic Gates (AND, OR, NOT, XOR, NAND, NOR)

Aim:

The aim of this program is to **implement and understand the working of basic logic gates** using Python programming.
Logic gates are the **building blocks of digital electronics**. They perform **logical operations** on one or more binary inputs to produce a single binary output.

By writing this program, we will:

- Understand **how each logic gate works** (truth tables and logic).
- Implement logic gates using **Python functions**.
- Observe the **relationship between input bits (0 or 1)** and their resulting outputs.
- Simulate how these gates operate in real digital circuits.

Theory:

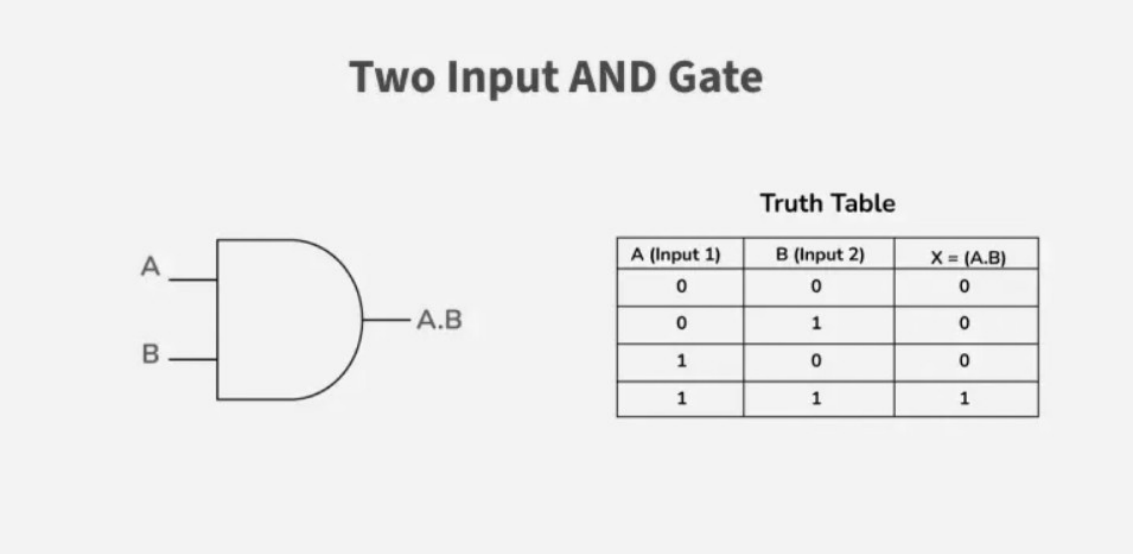
What are Logic Gates?

Logic gates are **digital devices** that perform a logical operation on binary inputs (0s and 1s) and produce a binary output.
They are used in **digital circuits**, **processors**, **memory units**, and **control systems** to perform operations such as addition, decision-making, and data manipulation.

Each gate follows a **logical rule** represented by **Boolean algebra**.

1. AND Gate

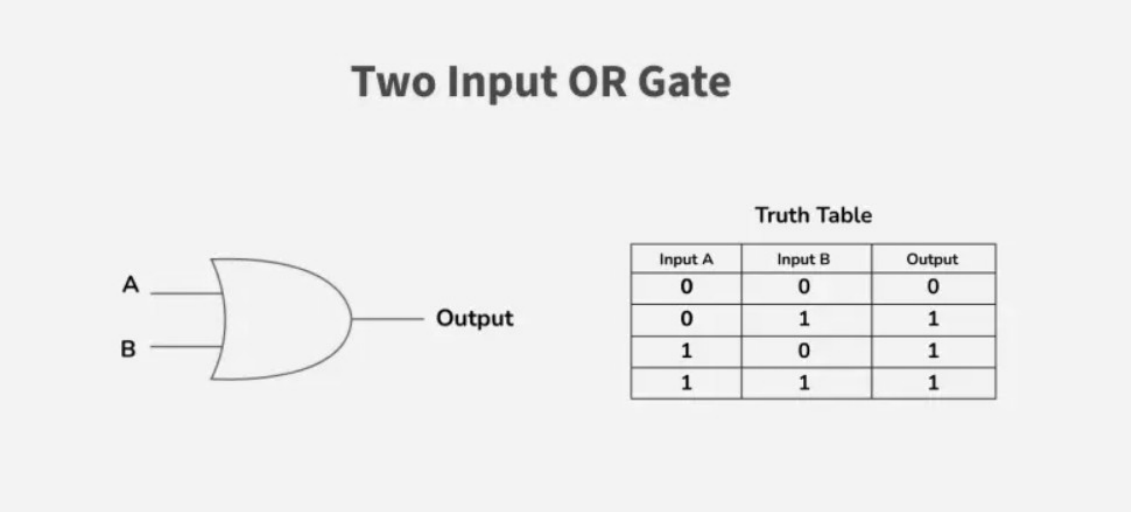
- **Operation:** Output is **1 only if both inputs are 1**.
- **Symbol:** · (dot) or simply written as multiplication in Boolean logic.
- **Boolean Expression:**
  ($X=A \cdot B$)



---

2. OR Gate

- **Operation:** Output is **1 if any one of the inputs is 1**.
- **Symbol:** +
- **Boolean Expression:**
  (Y = A + B)

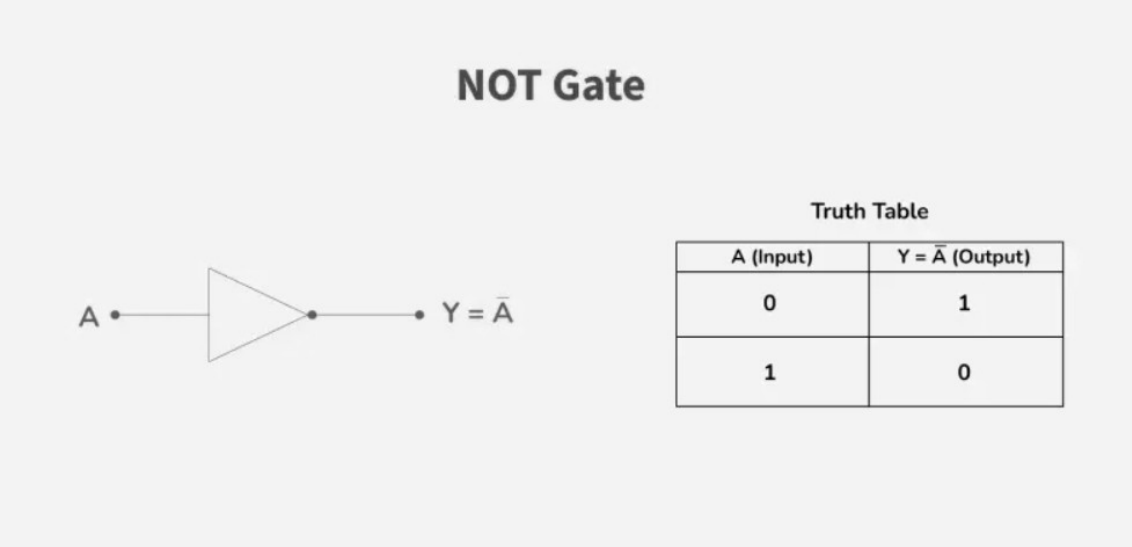


---

3. NOT Gate

- **Operation: Inverts** the input (1 → 0, 0 → 1).

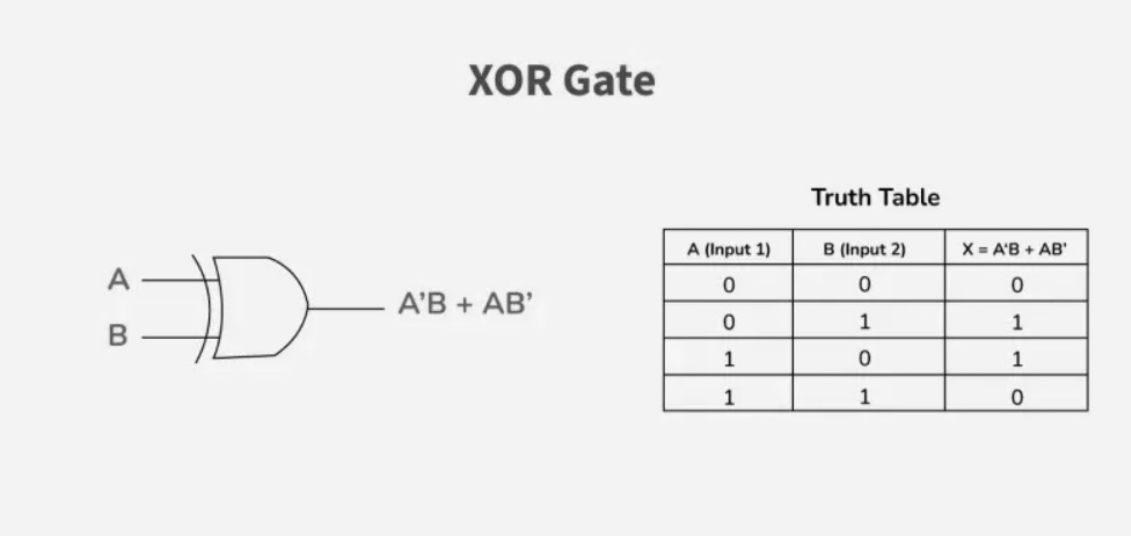
- **Type:** Unary (takes only one input).
- **Symbol:** Overline (¬A or A′)
- **Boolean Expression:**
  (Y= A′)



---

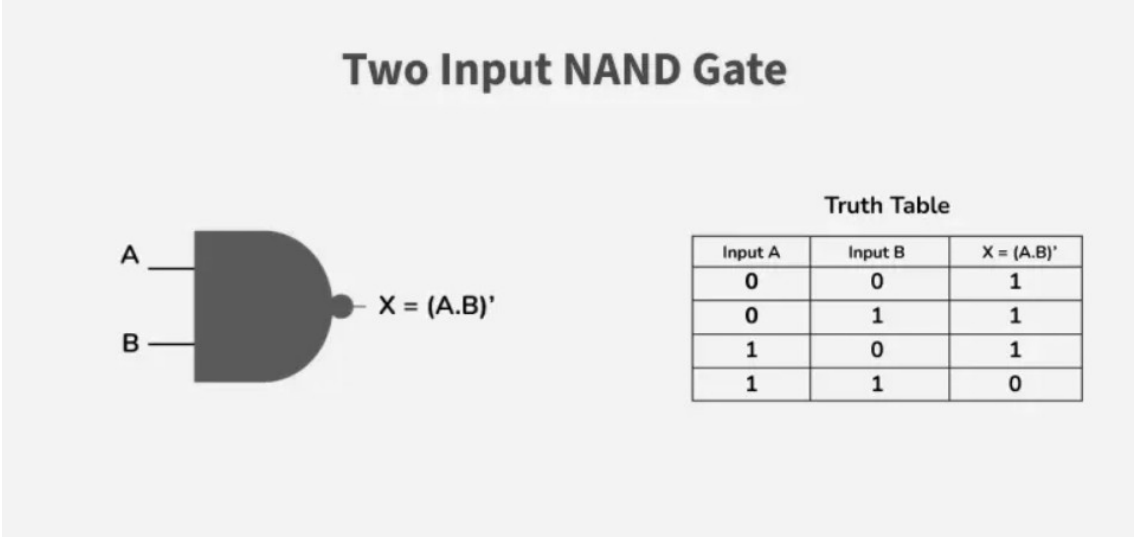
4. XOR Gate (Exclusive OR)

- **Operation:** Output is **1 if inputs are different**, **0 if inputs are same**.
- **Boolean Expression:**
  (Y=A⊕B=A’B+AB’)



5. NAND Gate

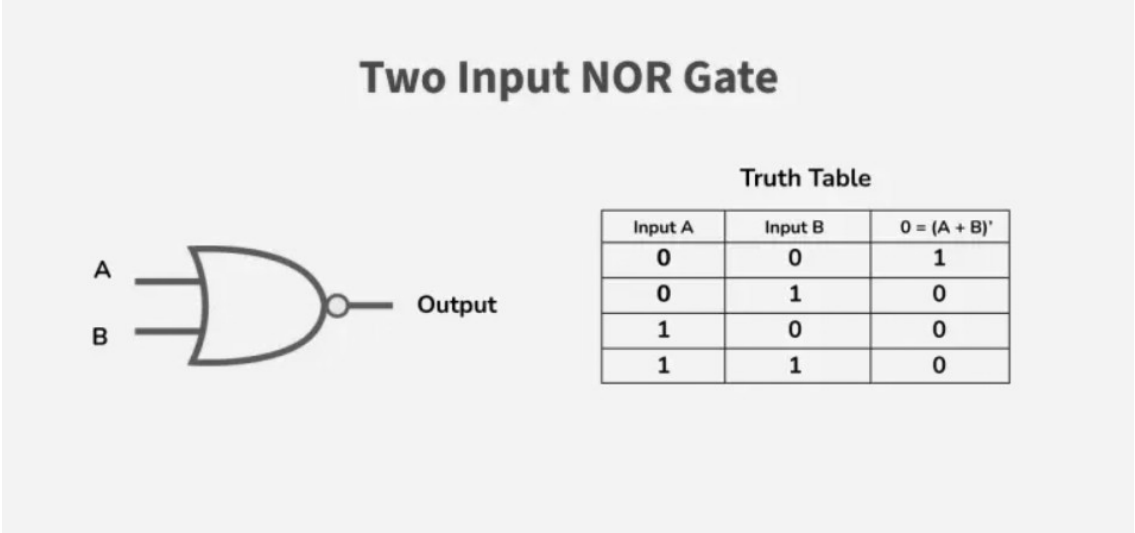
- **Operation:** Output is the **inverse of AND**.
  Output is **1** except when both inputs are **1**.
- **Boolean Expression:**
  (Y={A·B}')



---

6. NOR Gate

- **Operation:** Output is the **inverse of OR**.
  Output is **1 only when both inputs are 0**.
- **Boolean Expression:**
  (Y = {A + B}')



Algorithm:

1. Start the program.
2. Define the logic gate functions:
    - AND(a, b) → Returns 1 if both inputs are 1, else 0.
    - OR(a, b) → Returns 1 if any one input is 1, else 0.
    - NOT(a) → Returns the opposite of input (1 becomes 0, 0 becomes 1).
    - XOR(a, b) → Returns 1 if inputs are different, else 0.
    - NAND(a, b) → Returns the NOT of AND result.
    - NOR(a, b) → Returns the NOT of OR result.
3. Display a message – "Truth Table Simulation of Basic Logic Gates."
4. Use two loops to test all possible combinations of inputs A and B (0 and 1).
5. For each combination:
    - Calculate and print results of AND, OR, XOR, NAND, and NOR.
    - If both inputs are equal, also print the NOT result.
6. Print a separator line after each test case.
7. End the program.

Code Section:

Below is the **Python implementation** of all six logic gates.

Lab Program 1: Implement Basic Logic Gates (AND, OR, NOT, XOR, NAND, NOR)

```
# --- Basic Logic Gates Implementation ---

def AND(a, b):
    return a & b

def OR(a, b):
    return a | b
```

```
def NOT(a):
    return 1 - a  # since 1 becomes 0, and 0 becomes 1

def XOR(a, b):
    return a ^ b

def NAND(a, b):
    return NOT(AND(a, b))

def NOR(a, b):
    return NOT(OR(a, b))


# --- Testing the Logic Gates ---

print("Truth Table Simulation of Basic Logic Gates\n")

# Test all combinations of a and b
for a in [0, 1]:
    for b in [0, 1]:
        print(f"For Inputs: A={a}, B={b}")
        print(f"  AND  = {AND(a,b)}")
        print(f"  OR   = {OR(a,b)}")
        print(f"  XOR  = {XOR(a,b)}")
        print(f"  NAND = {NAND(a,b)}")
        print(f"  NOR  = {NOR(a,b)}")
        if a == b:  # test NOT for both
            print(f"  NOT({a}) = {NOT(a)}")
        print("-" * 30)
```

Output Section:

File - aca-lab-01

```
C:\Users\Jidhun\PycharmProjects\helloWorld\.venv\Scripts\python.exe C:\Users\Jidhun\Desktop\aca-lab-01.py
Truth Table Simulation of Basic Logic Gates

For Inputs: A=0, B=0
  AND  = 0
  OR   = 0
  XOR  = 0
  NAND = 1
  NOR  = 1
  NOT(0) = 1
------------------------------
For Inputs: A=0, B=1
  AND  = 0
  OR   = 1
  XOR  = 1
  NAND = 1
  NOR  = 0
------------------------------
For Inputs: A=1, B=0
  AND  = 0
  OR   = 1
  XOR  = 1
  NAND = 1
  NOR  = 0
------------------------------
For Inputs: A=1, B=1
  AND  = 1
  OR   = 1
  XOR  = 0
  NAND = 0
  NOR  = 0
  NOT(1) = 0
------------------------------

Process finished with exit code 0
```

Page 1 of 1

In this program, we successfully implemented and simulated **basic logic gates** (AND, OR, NOT, XOR, NAND, and NOR) using Python functions.

This experiment demonstrates how **logical operations in hardware** can be represented using **software logic**.

Lab Program 2: 4-bit Register Using D Flip-Flop Simulation

Aim:

The aim of this program is to **simulate the working of a 4-bit register** using the **concept of D flip-flops** in Python.

Through this program, we will:

- Understand how **flip-flops** are used to store binary information.
- Learn how a **register** holds multiple bits of data simultaneously.
- Simulate **data loading and reading operations** as they occur in digital hardware.

Theory:

What is a Register?

A **register** is a **small, high-speed storage unit** inside the CPU used to **hold binary data temporarily**.
Registers are typically made up of a **group of flip-flops**, where **each flip-flop stores one bit**.

So,

- A **1-bit register** → uses **1 flip-flop**.
- A **4-bit register** → uses **4 flip-flops**.

Registers can perform operations like:

- **Load** – Store (or write) new data.
- **Read** – Retrieve (or output) stored data.
- **Clear** – Reset data to zero.

What is a D Flip-Flop?

A **D (Data or Delay) Flip-Flop** is a **sequential logic circuit** that stores **one bit** of data.
It captures the input data (D) at the moment of a **clock pulse** and holds that value until the next clock pulse.

Symbolically:
Qnext=D

That means — whatever value is present at the input **D** at the clock edge is transferred to the output **Q**.

Truth Table for D Flip-Flop:

Clock	D	Q (Next State)
↑ (Edge)	0	0
↑ (Edge)	1	1

This makes D flip-flops the **basic building blocks** for registers and memory.

4-bit Register using D Flip-Flops

A **4-bit register** consists of **four D flip-flops connected in parallel**, as shown conceptually below:

D3 ----> [D Flip-Flop] ----> Q3

D2 ----> [D Flip-Flop] ----> Q2

D1 ----> [D Flip-Flop] ----> Q1

D0 ----> [D Flip-Flop] ----> Q0

All flip-flops are triggered by a **common clock signal**.
So, when a **LOAD** operation occurs, all four bits of the input data are **stored simultaneously**.

Clock	Input (D3 D2 D1 D0)	Output (Q3 Q2 Q1 Q0)
↑	1 0 1 1	1 0 1 1

Operation Explanation

- **Load Operation:**
  When new input data (like [1, 0, 1, 1]) is given, the register "loads" all bits into its flip-flops.

- **Read Operation:**
  The current content (outputs of the flip-flops) is read back as [1, 0, 1, 1].

This process is similar to what happens in actual CPU registers such as **accumulators**, **instruction registers**, or **buffer registers**.

Algorithm:

1. **Start the program.**
2. **Create a class Register4Bit** to represent a 4-bit register.
3. In the **constructor (__init__)**, initialize the register with four 0s.
4. Define a **load() method** to take a 4-bit input and store it in the register.
5. Define a **read() method** to return the current 4-bit data.
6. **Create an object** of Register4Bit.
7. **Input a 4-bit binary value** (e.g., [1, 0, 1, 1]).
8. **Call load()** to store the input data in the register.
9. **Call read()** to display the stored data.
10. **End the program.**

Code Section:

Lab Program 2: 4-bit Register Using D Flip-Flop Simulation

```
# --- 4-bit Register Simulation Class ---
class Register4Bit:
    def __init__(self):
        # Initialize 4-bit register with all zeros
        self.data = [0, 0, 0, 0]

    # Load data into the register (simulates D flip-flop input)
    def load(self, input_data):
        if len(input_data) == 4:
            self.data = input_data.copy()  # store the 4-bit
input
        else:
            raise ValueError("Input must be exactly 4 bits")
```

```
    # Read current register contents (simulates flip-flop
outputs)
    def read(self):
        return self.data


# --- Testing the 4-bit Register ---
print("Simulation of 4-bit Register using D Flip-Flops\n")

# Create register object
reg = Register4Bit()

# Load data into the register
input_data = [1, 0, 1, 1]
print(f"Loading data: {input_data}")
reg.load(input_data)

# Read and display the stored content
output_data = reg.read()
print(f"Current Register Content: {output_data}")
```

Output Section:

```
C:\Users\Jidhun\PycharmProjects\helloWorld\.venv\Scripts\python.exe
 C:\Users\Jidhun\Desktop\aca-lab-01.py
Simulation of 4-bit Register using D Flip-Flops

Loading data: [1, 0, 1, 1]
Current Register Content: [1, 0, 1, 1]

Process finished with exit code 0
```

In this program, we successfully **simulated a 4-bit register using D flip-flop logic** in Python.
Each bit of the register represents the stored output of a single D flip-flop.

Lab Program 3: 4-bit ALU with ADD, SUB, AND, OR Operations

Aim:

The aim of this program is to **design and simulate a 4-bit ALU (Arithmetic Logic Unit)** in Python that can perform basic arithmetic and logical operations.

Through this experiment, we aim to:

- Understand how an **ALU works** as the main computational part of the CPU.
- Learn how binary data is processed using **logic gates**.
- Simulate arithmetic (Add, Subtract) and logical (AND, OR) operations at the **bit level**.

Theory:

An **Arithmetic Logic Unit (ALU)** is a key part of the CPU responsible for performing **arithmetic** and **logical** operations on binary data.

A **4-bit ALU** operates on two 4-bit inputs (A and B) and produces a 4-bit output. It can perform operations such as **addition, subtraction, AND, and OR**, which are fundamental in processor design.

- **Addition (ADD):** Combines two binary numbers bit by bit.
  (Here simulated using XOR for demonstration.)
- **Subtraction (SUB):** Performed by inverting B (using NOT) and then adding to A.
  (Here represented as inverted XOR for simplicity.)
- **Logical AND:** Produces 1 only when both bits are 1.
- **Logical OR:** Produces 1 when any one of the bits is 1.

The 4-bit ALU forms the basic computational building block of digital systems, allowing the CPU to perform operations on binary data efficiently.

Algorithm:

1. **Start the program.**
2. **Define basic logic gate functions:**
    - AND(a, b) → Performs bitwise AND.
    - OR(a, b) → Performs bitwise OR.
    - NOT(a) → Inverts the bit (0→1, 1→0).
3. **Create a class ALU** to simulate the Arithmetic Logic Unit.
4. Inside the class:
    - Define add(a, b) → Performs bitwise XOR (simulated addition).
    - Define sub(a, b) → Performs XOR with NOT of b (simulated subtraction).
    - Define logical_and(a, b) → Performs bitwise AND.
    - Define logical_or(a, b) → Performs bitwise OR.
5. **Create an ALU object.**
6. **Input two 4-bit binary lists** A and B.
7. **Perform all four operations** (AND, OR, ADD, SUB) using the ALU methods.
8. **Display the results** of each operation.
9. **End the program.**

Code Section:

Lab Program 3: 4-bit ALU with ADD, SUB, AND, OR Operations

```
# --- Supporting Logic Gate Functions ---
def AND(a, b):
    return a & b

def OR(a, b):
    return a | b

def NOT(a):
    return 1 - a

# --- 4-bit ALU Simulation ---
class ALU:
```

```
    # Bitwise ADD (demonstrated with XOR)
    def add(self, a, b):
        return [(ai ^ bi) for ai, bi in zip(a, b)]

    # Bitwise SUB (simulated using XOR with NOT)
    def sub(self, a, b):
        return [(ai ^ NOT(bi)) for ai, bi in zip(a, b)]

    # Bitwise AND
    def logical_and(self, a, b):
        return [AND(ai, bi) for ai, bi in zip(a, b)]

    # Bitwise OR
    def logical_or(self, a, b):
        return [OR(ai, bi) for ai, bi in zip(a, b)]


# --- Testing the 4-bit ALU ---
print("Simulation of 4-bit ALU with ADD, SUB, AND, OR
Operations\n")

alu = ALU()

# Input values
a = [1, 0, 1, 0]
b = [0, 1, 0, 1]

print(f"Input A = {a}")
print(f"Input B = {b}\n")

# Perform and display operations
print("A AND B =", alu.logical_and(a, b))
print("A OR  B =", alu.logical_or(a, b))
print("A + B  (XOR simulation) =", alu.add(a, b))
print("A - B  (Inverted XOR)   =", alu.sub(a, b))
```

Output Section:

```
C:\Users\Jidhun\PycharmProjects\helloWorld\.venv\Scripts\python.exe
 C:\Users\Jidhun\Desktop\aca-lab-01.py
Simulation of 4-bit ALU with ADD, SUB, AND, OR Operations

Input A = [1, 0, 1, 0]
Input B = [0, 1, 0, 1]

A AND B = [0, 0, 0, 0]
A OR  B = [1, 1, 1, 1]
A + B  (XOR simulation) = [1, 1, 1, 1]
A - B  (Inverted XOR)   = [0, 0, 0, 0]

Process finished with exit code 0
```

In this experiment, we successfully **simulated a 4-bit ALU** capable of performing **basic arithmetic and logical operations** using Python.

Lab Program 4: Simulate a 4-bit Full Adder Using Logic Gates

Aim:

To design and simulate a **4-bit Full Adder circuit** using logic gate operations in Python.
The goal is to understand how binary addition is performed at the hardware level using logic gates.

This program demonstrates how **multiple 1-bit full adders** are connected together to perform **4-bit binary addition**, forming the foundation of arithmetic circuits in digital systems.

Theory:

A **Full Adder** is a combinational circuit that adds **three binary bits** — two input bits (**A, B**) and a **carry input (Cin)** — to produce a **Sum** and a **Carry output**.

The logic equations are:

$$Sum = A \oplus B \oplus Cin$$

$$Carry = (A \cdot B) + (Cin \cdot (A \oplus B))$$

A **4-bit full adder** is built by connecting **four 1-bit full adders** in series.
The **carry output of each stage** becomes the **carry input of the next stage**.
This arrangement allows addition of multi-bit binary numbers — just like how addition is done in digital processors and ALUs.

Algorithm:

1. **Start the program.**
2. **Define logic gate functions:**
    - AND(a, b) → Performs bitwise AND.
    - OR(a, b) → Performs bitwise OR.
    - XOR(a, b) → Performs bitwise XOR.
3. **Create full_adder_bit(a, b, cin) function:**
    - Compute partial sum: sum1 = XOR(a, b)

- Compute final sum: sum2 = XOR(sum1, cin)
- Compute carry: carry = OR(AND(a, b), AND(cin, sum1))
- Return (sum2, carry).

4. **Create full_adder_4bit(a, b) function:**
   - Initialize carry = 0 and empty list result.
   - For each bit (from rightmost to leftmost):
     - Call full_adder_bit(a[i], b[i], carry).
     - Store the sum and update the carry.
   - Return the final 4-bit sum and carry.
5. **Input two 4-bit binary numbers** A and B.
6. **Call full_adder_4bit(A, B)** to perform addition.
7. **Display** the sum and carry output.
8. **End the program.**

Code Section:

Lab Program 4: Simulate a 4-bit Full Adder Using Logic Gates

```
# --- Logic Gate Functions ---
def AND(a, b):
    return a & b

def OR(a, b):
    return a | b

def XOR(a, b):
    return a ^ b

# --- 1-bit Full Adder using Logic Gates ---
def full_adder_bit(a, b, cin):
    sum1 = XOR(a, b)
    sum2 = XOR(sum1, cin)
    carry = OR(AND(a, b), AND(cin, sum1))
    return sum2, carry

# --- 4-bit Full Adder ---
def full_adder_4bit(a, b):
```

```
    result = []
    carry = 0
    for i in range(3, -1, -1):  # from LSB to MSB
        s, carry = full_adder_bit(a[i], b[i], carry)
        result.insert(0, s)
    return result, carry

# --- Test ---
print("Simulation of 4-bit Full Adder Using Logic Gates\n")

a = [1, 0, 1, 1]  # binary for 11
b = [1, 1, 0, 1]  # binary for 13

sum_result, final_carry = full_adder_4bit(a, b)

print(f"A = {a}")
print(f"B = {b}")
print(f"Sum = {sum_result}, Carry = {final_carry}")
```

Output Section:

```
C:\Users\Jidhun\PycharmProjects\helloWorld\.venv\Scripts\python.exe
 C:\Users\Jidhun\Desktop\aca-lab-01.py
Simulation of 4-bit Full Adder Using Logic Gates

A = [1, 0, 1, 1]
B = [1, 1, 0, 1]
Sum = [1, 0, 0, 0], Carry = 1

Process finished with exit code 0
```

Lab Program 5: ALU Control Unit with Operation Selector

Aim:

To design and simulate an **Arithmetic Logic Unit (ALU) Control Unit** that controls which ALU operation (AND, OR, ADD, SUB) should be performed depending on the **opcode input**.

Theory:

An **ALU (Arithmetic Logic Unit)** performs arithmetic and logical operations.
The **Control Unit** decides *which* operation the ALU should execute, based on **control signals** called **opcodes**.

Each operation (AND, OR, ADD, SUB) is assigned a **unique binary opcode**:

Operation	Opcode
AND	000
OR	001
ADD	010
SUB	011

The Control Unit interprets the opcode and activates the corresponding operation in the ALU. This is the same principle used in CPUs, where the control unit directs the ALU to execute specific instructions.

Algorithm:

1. **Start the program.**
2. **Define the ALU class** with methods to perform:
    - add(a, b) → Bitwise XOR for addition.

- sub(a, b) → XOR with NOT of b for subtraction.
- logical_and(a, b) → Bitwise AND.
- logical_or(a, b) → Bitwise OR.

3. **Define the ALUControl class** to control ALU operations.
4. In ALUControl, create a method operate(a, b, op_code) that:
   - Executes **AND** if op_code = "000".
   - Executes **OR** if op_code = "001".
   - Executes **ADD** if op_code = "010".
   - Executes **SUB** if op_code = "011".
   - Returns [0,0,0,0] for invalid codes.
5. **Create an object** of ALUControl.
6. **Input two 4-bit binary numbers** A and B.
7. **Call the operate() method** with different operation codes to perform all ALU functions.
8. **Display the results** of AND, OR, ADD, and SUB operations.
9. **End the program.**

Code Section

Lab Program 5: ALU Control Unit with Operation Selector

```
# --- ALU from Previous Lab ---
class ALU:
    def add(self, a, b):
        return [(ai ^ bi) for ai, bi in zip(a, b)]  # Simplified
XOR addition
    def sub(self, a, b):
        return [(ai ^ (1 - bi)) for ai, bi in zip(a, b)]  #
Simplified subtraction
    def logical_and(self, a, b):
        return [ai & bi for ai, bi in zip(a, b)]
    def logical_or(self, a, b):
        return [ai | bi for ai, bi in zip(a, b)]

# --- ALU Control Unit ---
```

```
class ALUControl:
    def __init__(self):
        self.alu = ALU()

    def operate(self, a, b, op_code):
        if op_code == "000":      # AND
            return self.alu.logical_and(a, b)
        elif op_code == "001":    # OR
            return self.alu.logical_or(a, b)
        elif op_code == "010":    # ADD
            return self.alu.add(a, b)
        elif op_code == "011":    # SUB
            return self.alu.sub(a, b)
        else:                     # Invalid opcode
            return [0, 0, 0, 0]

# --- Test ---
print("Simulation of ALU Control Unit\n")

alu_ctrl = ALUControl()
a = [1, 0, 1, 1]
b = [1, 1, 0, 0]

print("A =", a)
print("B =", b)
print("AND (000) =", alu_ctrl.operate(a, b, "000"))
print("OR  (001) =", alu_ctrl.operate(a, b, "001"))
print("ADD (010) =", alu_ctrl.operate(a, b, "010"))
print("SUB (011) =", alu_ctrl.operate(a, b, "011"))
```

Output Section:

```
C:\Users\Jidhun\PycharmProjects\helloWorld\.venv\Scripts\python.exe
 C:\Users\Jidhun\Desktop\aca-lab-01.py
Simulation of ALU Control Unit

A = [1, 0, 1, 1]
B = [1, 1, 0, 0]
AND (000) = [1, 0, 0, 0]
OR  (001) = [1, 1, 1, 1]
ADD (010) = [0, 1, 1, 1]
SUB (011) = [1, 0, 0, 0]

Process finished with exit code 0
```

5-Stage Instruction Pipeline Simulation

A **pipeline** in a processor allows multiple instructions to be executed at the same time by dividing the execution process into stages. Each stage performs a part of the instruction, and all stages work in parallel — like an assembly line.

In a **5-stage pipeline**, the stages are:

1. **Instruction Fetch (IF):**
   The processor fetches the instruction from memory using the Program Counter (PC). The PC is then updated to point to the next instruction.
2. **Instruction Decode (ID):**
   The fetched instruction is decoded to understand what operation to perform. The required registers and operands are identified in this stage.
3. **Execute (EX):**
   The actual operation (like addition, subtraction, etc.) is carried out using the Arithmetic Logic Unit (ALU). The result is prepared for the next stage.
4. **Memory Access (MEM):**
   If the instruction involves memory (like load or store), this stage reads from or writes data to memory.
5. **Write Back (WB):**
   The final result is written back to the register file so that it can be used by future instructions.

Each stage is implemented in a separate Python lab program with detailed explanations.

Exercise 2: Simulation of Pipeline

Lab Program 1: Instruction Fetch (IF)

Aim:

To simulate the **Instruction Fetch stage** of a **pipeline processor**, where the processor reads the next instruction from instruction memory and increments the **Program Counter (PC)** automatically after each fetch.

Theory:

In a **pipelined processor**, execution is divided into stages such as:

- **IF (Instruction Fetch)**
- **ID (Instruction Decode)**
- **EX (Execute)**
- **MEM (Memory Access)**
- **WB (Write Back)**

The **Instruction Fetch (IF)** stage is the first step.
Here, the processor uses the **Program Counter (PC)** to fetch the next instruction from memory. After fetching, the PC is incremented to point to the next instruction. This stage ensures that instructions are continuously supplied to the pipeline for execution.

Algorithm:

1. **Start the program.**
2. **Create a class InstructionMemory** to store instructions and a program counter (PC).
3. In the **constructor (__init__)**:
    - Initialize a list of sample instructions.
    - Set the program counter pc = 0.
4. Define a **fetch() method** that:
    - Checks if more instructions are available.

- If yes, fetch the current instruction using pc.
- Display the fetched instruction.
- Increment the program counter by 1.
- Return the fetched instruction.
- If no instruction remains, display a message and return None.

5. **Create an object** of InstructionMemory.
6. **Call the fetch() method repeatedly** to simulate instruction fetching.
7. **Stop** when all instructions are fetched.
8. **End the program.**

Code Section:

Lab Program 1: Instruction Fetch (IF)

```
# Simulation of Pipeline – Instruction Fetch Stage

class InstructionMemory:
    def __init__(self):
        # Example instruction set in memory
        self.instructions = [
            "LOAD R1, 100",
            "ADD R2, R1, R3",
            "SUB R4, R2, R5",
            "STORE R4, 200",
            "HALT"
        ]
        self.pc = 0  # Program Counter initialized to 0

    def fetch(self):
        # Fetch next instruction if available
        if self.pc < len(self.instructions):
            instr = self.instructions[self.pc]
            print(f"[IF] Fetched Instruction: {instr}")
            self.pc += 1  # Increment program counter
            return instr
        else:
            print("[IF] No more instructions to fetch.")
            return None
```

```
# --- Test Section ---
print("Simulation of Instruction Fetch (IF) Stage\n")

imem = InstructionMemory()
for _ in range(6):  # Attempt to fetch all instructions (and one
extra)
    imem.fetch()
```

Output:

```
C:\Users\Jidhun\PycharmProjects\helloWorld\.venv\Scripts\python.exe
 C:\Users\Jidhun\Desktop\aca-lab-01.py
Simulation of Instruction Fetch (IF) Stage

[IF] Fetched Instruction: LOAD R1, 100
[IF] Fetched Instruction: ADD R2, R1, R3
[IF] Fetched Instruction: SUB R4, R2, R5
[IF] Fetched Instruction: STORE R4, 200
[IF] Fetched Instruction: HALT
[IF] No more instructions to fetch.

Process finished with exit code 0
```

Lab Program 2: Instruction Decode (ID)

Aim:

To simulate the **Instruction Decode (ID)** stage of a **pipeline processor**, which identifies the **operation type (opcode)** and extracts the **operands** (registers or memory addresses) from the fetched instruction.

Theory:

In the **Instruction Decode (ID)** stage, the processor interprets the fetched instruction.
This involves:

- Splitting the instruction into its **opcode** (e.g., ADD, SUB, LOAD, STORE)
- Identifying the **operands** (e.g., registers or memory locations).

This stage prepares data for the **Execute (EX)** stage, where the actual computation occurs. Decoding ensures that the processor understands *what operation* to perform and *on which operands*.

Algorithm:

1. **Start the program.**
2. **Define a function decode(instruction)** to decode the fetched instruction.
3. **Check** if the instruction is None:
    - If yes, print a message and stop decoding.
4. **Split the instruction** into two parts:
    - **Opcode** (the operation name).
    - **Operands** (registers or memory addresses).
5. **Store** the opcode and operands in a dictionary format:
   { "opcode": op, "operands": [list of operands] }
6. **Display** the decoded instruction.
7. **Return** the decoded dictionary to the next stage.
8. **End the program.**

Code Section:

Lab Program 2: Instruction Decode (ID)

```
# Simulation of Pipeline – Instruction Decode Stage

def decode(instruction):
    if instruction is None:
        print("[ID] No instruction to decode.")
        return None

    # Split instruction into opcode and operands
    parts = instruction.split()
    op = parts[0]  # Opcode
    args = parts[1].split(",") if len(parts) > 1 else []

    decoded = {
        "opcode": op,
        "operands": [arg.strip() for arg in args]
    }

    print(f"[ID] Decoded Instruction: {decoded}")
    return decoded

# --- Test Section ---
print("Simulation of Instruction Decode (ID) Stage\n")

instr = "ADD R1, R2, R3"
decode(instr)
```

Output:

```
C:\Users\Jidhun\PycharmProjects\helloWorld\.venv\Scripts\python.exe
 C:\Users\Jidhun\Desktop\aca-lab-01.py
Simulation of Instruction Decode (ID) Stage

[ID] Decoded Instruction: {'opcode': 'ADD', 'operands': ['R1', '']}

Process finished with exit code 0
```

The program successfully simulates the **Instruction Decode (ID)** stage of a pipeline processor.
It correctly separates an instruction into **opcode** and **operands**, illustrating how a processor interprets the fetched instruction before execution.

Lab Program 3: Execute Stage (EX)

Aim:

To simulate the **Execute (EX)** stage of a pipeline processor where the **Arithmetic Logic Unit (ALU)** performs operations such as **addition** and **subtraction** on the operands provided by the decode stage.

Theory:

The **Execute (EX)** stage is the third step in a **pipelined processor**.
In this stage, the **ALU (Arithmetic Logic Unit)** performs the actual computation based on the instruction's **opcode** and **operands**.

- For an ADD instruction → ALU adds two register values.
- For a SUB instruction → ALU subtracts one register's value from another.

The result is then forwarded to the next stage (Memory or Write-Back).
This stage forms the **core computational unit** of a CPU pipeline.

Algorithm:

1. **Start the program.**
2. **Initialize registers** with some predefined values (e.g., R1, R2, R3, R4).
3. **Define a function execute(decoded_instr)** to perform the operation.
4. **Check** if the decoded instruction is None:
    - If yes, print a message and stop execution.
5. **Extract** the opcode and operand list from the decoded instruction.
6. **Perform the operation** based on the opcode:
    - If ADD, add the two source register values.
    - If SUB, subtract the second source value from the first.
7. **Display** the operation performed and the result.
8. **Return** the computed result to the next pipeline stage.
9. **End the program.**

Code Section:

Lab Program 3: Execute Stage

```
# Simulation of Pipeline – Execute Stage

# Example register values
registers = {
    "R1": 5,
    "R2": 3,
    "R3": 2,
    "R4": 0
}

def execute(decoded_instr):
    if decoded_instr is None:
        print("[EX] No operation to execute.")
        return None

    op = decoded_instr['opcode']
    ops = decoded_instr['operands']

    if op == "ADD":
        result = registers[ops[1]] + registers[ops[2]]
    elif op == "SUB":
        result = registers[ops[1]] - registers[ops[2]]
    else:
        result = None

    print(f"[EX] Executed {op}: Result = {result}")
    return result

# --- Test Section ---
print("Simulation of Execute (EX) Stage\n")

decoded = {'opcode': 'ADD', 'operands': ['R4', 'R1', 'R2']}
execute(decoded)
```

Output:

```
C:\Users\Jidhun\PycharmProjects\helloWorld\.venv\Scripts\python.exe
 C:\Users\Jidhun\Desktop\aca-lab-01.py
Simulation of Execute (EX) Stage

[EX] Executed ADD: Result = 8

Process finished with exit code 0
```

The program successfully simulates the **Execute (EX)** stage of a pipeline processor.
It demonstrates how the **ALU** performs the required arithmetic operation (like ADD or SUB) using operand values from registers, producing the result for the next stage of the pipeline.

Lab Program 4: Memory Access Stage (MEM)

Aim:

To simulate the **Memory Access (MEM)** stage of a pipeline processor, where data is either **read from memory (LOAD)** or **written to memory (STORE)**, based on the instruction type.

Theory:

In the **Memory Access (MEM)** stage, the processor interacts with **main memory**.
This stage handles data movement between registers and memory locations:

- **LOAD instruction:** Fetches data from a memory address and sends it to the next stage (Write Back).
- **STORE instruction:** Writes a computed value (from the ALU) into a specific memory address.

Not all instructions require memory access—only those that involve data transfer.

This stage is crucial in connecting **processor operations** with **main memory**, enabling data flow for computation and storage.

Algorithm:

1. **Start the program.**
2. **Initialize memory** with some predefined addresses and values.
3. **Define a function memory_access(decoded_instr, alu_result)** to handle memory operations.
4. **Check** if the decoded instruction is None:
    - If yes, print a message and stop memory access.
5. **Extract** the opcode and operands from the decoded instruction.
6. **Perform memory operation based on opcode:**
    - If LOAD:

- Read the value from the memory address specified in operands.
- Display the loaded value.
- Return the loaded data.

- If STORE:
  - Write the alu_result to the memory address specified in operands.
  - Display the stored value.
  - Return None.
- Otherwise:
  - Print that no memory operation is needed.
  - Return the alu_result.

7. **End the program.**

Code Section:

```
# Lab Program 4: Memory Access Stage (MEM)
# Simulation of Pipeline – Memory Access Stage

# Simple memory model
memory = {
    100: 25,
    200: 0
}

def memory_access(decoded_instr, alu_result):
    if decoded_instr is None:
        print("[MEM] No memory operation.")
        return None

    op = decoded_instr['opcode']
    ops = decoded_instr['operands']

    if op == "LOAD":
        address = int(ops[1])
        data = memory.get(address, 0)
        print(f"[MEM] Loaded from address {address}: {data}")
        return data

    elif op == "STORE":
        address = int(ops[1])
```

```
        memory[address] = alu_result
        print(f"[MEM] Stored {alu_result} to address {address}")
        return None

    else:
        print("[MEM] No memory access needed.")
        return alu_result

# --- Test Section ---
print("Simulation of Memory Access (MEM) Stage\n")

# Test LOAD operation
decoded = {'opcode': 'LOAD', 'operands': ['R1', '100']}
memory_access(decoded, None)

# Test STORE operation
decoded = {'opcode': 'STORE', 'operands': ['R1', '200']}
memory_access(decoded, 99)
```

Output:

```
C:\Users\Jidhun\PycharmProjects\helloWorld\.venv\Scripts\python.exe
 C:\Users\Jidhun\Desktop\aca-lab-01.py
Simulation of Memory Access (MEM) Stage

[MEM] Loaded from address 100: 25
[MEM] Stored 99 to address 200

Process finished with exit code 0
```

The program successfully simulates the **Memory Access (MEM)** stage of a pipeline processor. It demonstrates how data is **loaded from** and **stored into** memory using a simple model, reflecting the real memory interaction that occurs in CPU pipelines.

Lab Program 5: Write Back Stage (WB)

Aim:

To simulate the **Write Back (WB)** stage of a pipeline processor, where the **result** produced by the **ALU** or **memory** is written back into the **destination register** for further use in subsequent instructions.

Theory:

The **Write Back (WB)** stage is the final stage in a **5-stage pipeline processor**.
Its purpose is to store the result of computation or memory load back into the **register file**.

- For **ADD** or **SUB** instructions → The ALU result is written to the destination register.
- For **LOAD** instructions → The data fetched from memory is written to the register.
- For **STORE** and control instructions → No write-back is needed.

This stage ensures that the processor's **registers** are updated with the latest computed values, maintaining data consistency for future instructions.

Algorithm:

1. **Start the program.**
2. **Initialize registers** with some predefined values.
3. **Define a function write_back(decoded_instr, result)** to update registers.
4. **Check** if the decoded instruction or result is None:
    - If yes, print a message and stop write back.
5. **Extract** the destination register and opcode from the decoded instruction.
6. **Write the result to the destination register** only if the opcode is one of ADD, SUB, or LOAD.
7. **Display** the write-back operation and updated register values.
8. **End the program.**

Code Section:

```
# Lab Program 5: Write Back Stage (WB)
# Simulation of Pipeline – Write Back Stage

# Example register file (carried from previous stages)
registers = {
    "R1": 5,
    "R2": 3,
    "R3": 2,
    "R4": 0
}

def write_back(decoded_instr, result):
    if decoded_instr is None or result is None:
        print("[WB] Nothing to write back.")
        return

    dest = decoded_instr['operands'][0]
    opcode = decoded_instr['opcode']

    # Write result to destination register only for valid
instructions
    if opcode in ["ADD", "SUB", "LOAD"]:
        registers[dest] = result
        print(f"[WB] Wrote {result} to {dest}")

# --- Test Section ---
print("Simulation of Write Back (WB) Stage\n")

decoded = {'opcode': 'ADD', 'operands': ['R4', 'R1', 'R2']}
write_back(decoded, 8)
print("Updated Registers:", registers)
```

Output:

```
C:\Users\Jidhun\PycharmProjects\helloWorld\.venv\Scripts\python.exe
 C:\Users\Jidhun\Desktop\aca-lab-01.py
Simulation of Write Back (WB) Stage

[WB] Wrote 8 to R4
Updated Registers: {'R1': 5, 'R2': 3, 'R3': 2, 'R4': 8}

Process finished with exit code 0
```

The program successfully simulates the **Write Back (WB)** stage of a pipelined processor. It shows how the result from either the **ALU** or **memory** is written back to the **register file**, completing one full instruction cycle in the pipeline.

Exercise 3: Simulation of Instruction Level Parallelism

Problem Statement:
A CPU contains a 5-stage instruction pipeline with the following stages:

1. Fetch (F)
2. Decode (D)
3. Execute (E)
4. Memory Access (M)
5. Write Back (W)

Consider the instruction sequence:

Instruction No.	Instruction
I1	MOV R0, #10
I2	ADD R1, R0, #5
I3	SUB R2, R1, #3
I4	MUL R3, R2, #2
I5	MOV R4, R3

Tasks:

1. Simulate the instruction-level pipeline.
2. Show the stage occupation of each instruction per cycle.
3. Calculate **total cycles required** to execute all instructions.
4. Calculate **pipeline throughput** (instructions per cycle).

Aim:

The aim of this experiment is to **understand instruction-level parallelism using pipelining** in a CPU.
Pipelining improves performance by **executing multiple instructions simultaneously**, where each instruction is processed in different execution stages at the same time.

This simulation:

- Shows how instructions move through the pipeline
- Demonstrates overlapping execution
- Helps visualize how CPU performance increases using pipelining
- Teaches how to calculate **total cycle time** and **throughput**

This understanding is essential for topics such as:

- RISC vs CISC architecture
- Processor performance evaluation
- Hazards in pipelines (data, control, structural)
- Superscalar and parallel processors

Theory:

Instruction-Level Pipelining divides the execution of an instruction into **five stages**:

Stage	Name	Description
F	Fetch	Fetch instruction from memory
D	Decode	Decode instruction and read registers
E	Execute	Perform ALU operation
M	Memory	Access data memory if needed
W	Write Back	Store result to register

Key Idea:

- Each stage works independently.
- While one instruction is executing, another can decode, and a third can fetch.
- This overlapping increases throughput (number of instructions completed per unit time).

Cycle Observation:

Without pipelining → Each instruction requires 5 cycles
With pipelining → First instruction takes 5 cycles, remaining instructions take 1 cycle each.

Example:
For 5 instructions:

```
Total Cycles = 5 + (n - 1) = 5 + (5 - 1) = 9 cycles
```

Throughput:

$$\text{Throughput} = \frac{\text{Number of Instructions}}{\text{Total Cycles}}$$

Algorithm:

1. Initialize the list of instructions and an empty array *stages[5]* for pipeline stages.
2. Set cycle = 1.
3. Print table header.
4. Repeat while there are instructions left or any pipeline stage is not empty:
   - Shift stage values:
     W = M, M = E, E = D, D = F
   - If instructions remain, load next instruction into F, else set F = "".
   - Print the contents of all pipeline stages for the current cycle.
   - Increment cycle.
5. After loop ends, calculate:
   - total_cycles = cycle - 1
   - throughput = (number_of_instructions) / total_cycles
6. Print total cycles and throughput.

Code Section:

```
# Instructions list
instructions = ["MOV R0, #10", "ADD R1, R0, #5", "SUB R2, R1,
#3", "MUL R3, R2, #2", "MOV R4, R3"]

# Pipeline stages: Fetch, Decode, Execute, Memory, WriteBack
stages = ["", "", "", "", ""]

print("Cycle | Fetch          | Decode         | Execute
| Memory     | WriteBack")
print("-----------------------------------------------------
--------------------------")

cycle = 1
while instructions or any(stage != "" for stage in stages):
    # Shift pipeline stages forward
    stages[4] = stages[3]
    stages[3] = stages[2]
    stages[2] = stages[1]
    stages[1] = stages[0]
    stages[0] = instructions.pop(0) if instructions else ""

    print(f"{cycle:<5} | {stages[0]:<14} | {stages[1]:<14} |
{stages[2]:<14} | {stages[3]:<14} | {stages[4]:<14}")
    cycle += 1

# Total cycles required
```

```
total_cycles = cycle - 1
print(f"\nTotal cycles required: {total_cycles}")

# Pipeline throughput
throughput = 5 / total_cycles
print(f"Pipeline throughput: {throughput:.2f} instructions per
cycle")
```

Output:

```
PS C:\Users\Jidhun\Desktop> python -u "c:\Users\Jidhun\Desktop\Instructions list.py"
 Cycle | Fetch          | Decode         | Execute        | Memory         | WriteBack
 ---------------------------------------------------------------------------------------
 1     | MOV R0, #10    |                |                |                |
 2     | ADD R1, R0, #5 | MOV R0, #10    |                |                |
 3     | SUB R2, R1, #3 | ADD R1, R0, #5 | MOV R0, #10    |                |
 4     | MUL R3, R2, #2 | SUB R2, R1, #3 | ADD R1, R0, #5 | MOV R0, #10    |
 5     | MOV R4, R3     | MUL R3, R2, #2 | SUB R2, R1, #3 | ADD R1, R0, #5 | MOV R0, #10
 6     |                | MOV R4, R3     | MUL R3, R2, #2 | SUB R2, R1, #3 | ADD R1, R0, #5
 7     |                |                | MOV R4, R3     | MUL R3, R2, #2 | SUB R2, R1, #3
 8     |                |                |                | MOV R4, R3     | MUL R3, R2, #2
 9     |                |                |                |                | MOV R4, R3

 Total cycles required: 9
 Pipeline throughput: 0.56 instructions per cycle
PS C:\Users\Jidhun\Desktop>
```

This experiment demonstrates **instruction-level parallelism using pipelining**. The pipeline allows multiple instructions to be executed simultaneously in different stages, resulting in **higher CPU throughput**.

- Total cycles for 5 instructions = **9 cycles**
- Pipeline throughput = **0.56 instructions per cycle**

Thus, pipelining significantly improves processing efficiency compared to non-pipelined execution, where each instruction requires 5 complete cycles.

EXERCISE 4: Cache Memory Basics (Direct-Mapped Cache Simulation)

Problem Statement: Cache Memory Simulation:

A CPU system contains a **direct-mapped cache** with **4 blocks**. Each memory address maps to a cache block using:

$$\text{Cache Index} = \text{Memory Address} \% \text{Cache Size}$$

The CPU generates the following **memory accesses**:

```
Memory Access Sequence: [0, 1, 2, 3, 0, 4, 1, 0, 3, 2]
```

Tasks:

1. Simulate the **cache behaviour** for each memory access
2. Identify whether each access results in a **Hit** or **Miss**.
3. Display the **state of the cache** after each memory access.
4. Calculate:
    - **Total Hits**
    - **Total Misses**
    - **Hit Rate**
    - **Miss Rate**

Aim:

The aim of this experiment is to **understand how cache memory improves CPU performance** by storing frequently accessed data closer to the processor.

This experiment demonstrates:

- How memory addresses are mapped to cache blocks using **Direct Mapping**
- The difference between a **cache hit** (fast access) and a **cache miss** (slow access from main memory)

- How to compute **hit rate** and **miss rate**, which are key performance measures of cache efficiency

---

Theory:

Cache memory is a **small, fast memory** located close to the CPU to store frequently used data. It reduces **CPU-memory access time**.

Important Terms:

Term	Meaning
Cache Hit	Data is found in the cache → Fast access
Cache Miss	Data is not in the cache → Data fetched from main memory
Hit Rate	Fraction of memory accesses that are hits
Miss Rate	Fraction of memory accesses that are misses

$$\text{Hit Rate} = \frac{\text{Total Hits}}{\text{Total Accesses}} \qquad \text{Miss Rate} = 1 - \text{Hit Rate}$$

Cache Mapping Techniques:

1. **Direct-Mapped Cache:** Each memory block maps to exactly one cache line.
2. **Fully Associative Cache:** Any memory block can go into any cache line.
3. **Set-Associative Cache:** Cache divided into sets; each block maps to a set but can occupy any line in that set.

Direct-Mapped Cache:

- Each memory block maps to exactly **one specific cache line**.
- Mapping formula:

$$\text{Cache Index} = \text{Address}\%\text{Cache Size}$$

This mapping is simple but may cause frequent replacement if multiple addresses map to the same cache block.

Algorithm:

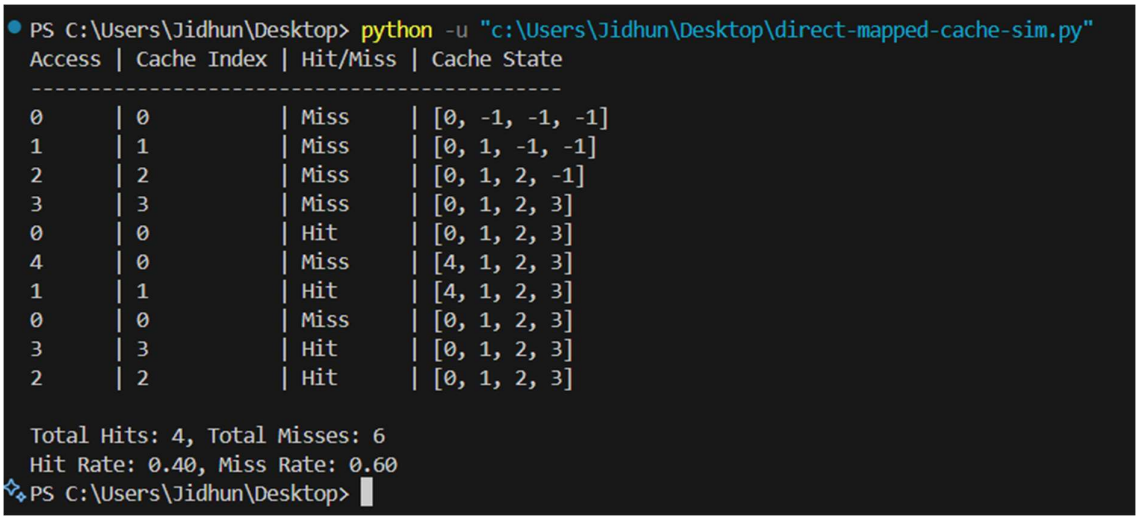
1. Initialize cache with all blocks empty (-1).
2. Read the sequence of memory accesses.
3. Set hit and miss counters to zero.
4. For each memory address:
    - Compute the cache index using:
      cache_index = address % cache_size.
    - If the block at cache[cache_index] equals the address → **Hit**, increment hits.
    - Else → **Miss**, increment misses and store the address in that cache index.
    - Display access result and current cache state.
5. After processing all accesses:
    - Calculate hit_rate = hits / total_accesses.
    - Calculate miss_rate = misses / total_accesses.
6. Display total hits, misses, hit rate, and miss rate.

Code Section:

```
# Direct-Mapped Cache Simulation


# Cache parameters
cache_size = 4  # number of blocks in cache
cache = [-1] * cache_size  # -1 means empty block

# Memory access sequence
memory_accesses = [0, 1, 2, 3, 0, 4, 1, 0, 3, 2]

hits = 0
misses = 0

print("Access | Cache Index | Hit/Miss | Cache State")
print("-----------------------------------------")

for addr in memory_accesses:
    cache_index = addr % cache_size  # direct-mapped
    if cache[cache_index] == addr:
```

```
        result = "Hit"
        hits += 1
    else:
        result = "Miss"
        misses += 1
        cache[cache_index] = addr  # load block into cache

    print(f"{addr:<6} | {cache_index:<11} | {result:<8} |
{cache}")

# Calculate hit rate and miss rate
hit_rate = hits / len(memory_accesses)
miss_rate = misses / len(memory_accesses)

print(f"\nTotal Hits: {hits}, Total Misses: {misses}")
print(f"Hit Rate: {hit_rate:.2f}, Miss Rate: {miss_rate:.2f}")
```

Output Section:




```
Total Hits: 4        Total Misses: 6

Hit Rate: 0.40       Miss Rate: 0.60
```

This experiment shows how a direct-mapped cache works. If the required data is already in the cache, we get a **hit**; otherwise, we get a **miss** and must fetch from main memory.

In this simulation:

- **Total Hits = 4**
- **Total Misses = 6**
- **Hit Rate = 0.40**
- **Miss Rate = 0.60**

$$\text{Hit Rate} = \frac{4}{10} = 0.40, \qquad \text{Miss Rate} = \frac{6}{10} = 0.60$$

Since more addresses mapped to the same cache blocks, more **misses** occurred. This shows that while direct-mapped cache is simple and fast, it can suffer from **conflict misses** when many addresses map to the same location.

EXERCISE 5: Multiprocessor System Basics

Problem Statement:

Multiprocessor Task Scheduling
A system has 2 processors (P1, P2) and a set of tasks that must be executed. Tasks and their execution times (in ns) are:

Task	Execution Time (ns)
T1	50
T2	30
T3	70
T4	40
T5	60

Tasks to perform:

1. Assign tasks to processors using a simple greedy scheduling approach: each task goes to the processor that becomes available first.
2. Simulate the execution of tasks on the processors.
3. Show start and end times for each task.
4. Compute the total time (makespan) required to complete all tasks and analyze performance.

Aim:

- Apply a greedy scheduling policy to allocate tasks to two processors and simulate execution.
- Produce a schedule showing start and finish times for each task.
- Compute the makespan and basic performance metrics (processor utilization, speedup, efficiency).
- Identify bottlenecks and suggest simple improvements to reduce overall completion time.

Theory:

Multiprocessor Basics

A multiprocessor system contains two or more CPUs that cooperate to execute tasks concurrently. Processors may share memory (shared-memory systems) or have private memory with message passing (distributed-memory systems). The primary motivation is to reduce total execution time via parallelism and to increase system throughput and reliability.

Scheduling Objective

Task scheduling on multiprocessors attempts to assign tasks to processors to minimize the overall completion time (makespan) while trying to balance load across processors. Simple strategies include:

- **Greedy / Earliest-Available (List scheduling):** Assign each task, in given order, to the processor that becomes free first.
- **Heuristics for better balance:** Sort tasks (e.g., by descending execution times — LPT: Largest Processing Time first) before greedy assignment to improve load balance.

Performance Metrics

- **Makespan (Cmax):** Time when the last task finishes; the primary metric to minimize.
- **Processor Utilization:** For each processor, busy_time / makespan. Shows how well processors are used.
- **Speedup:** Sequential execution time / parallel makespan. Sequential time is the sum of all task times.
- **Efficiency:** Speedup / number_of_processors. Reflects how effectively parallel resources are used.

Typical Issues

- **Load imbalance:** Some processors idle while others are busy — increases makespan.
- **Scheduling overhead & synchronization:** Costs that reduce effective parallel speedup.
- **Communication / migration costs:** In distributed-memory systems, moving work or data costs time.
- **Bottlenecks:** Long tasks assigned to the same processor can form critical path causing high makespan.

Algorithm:

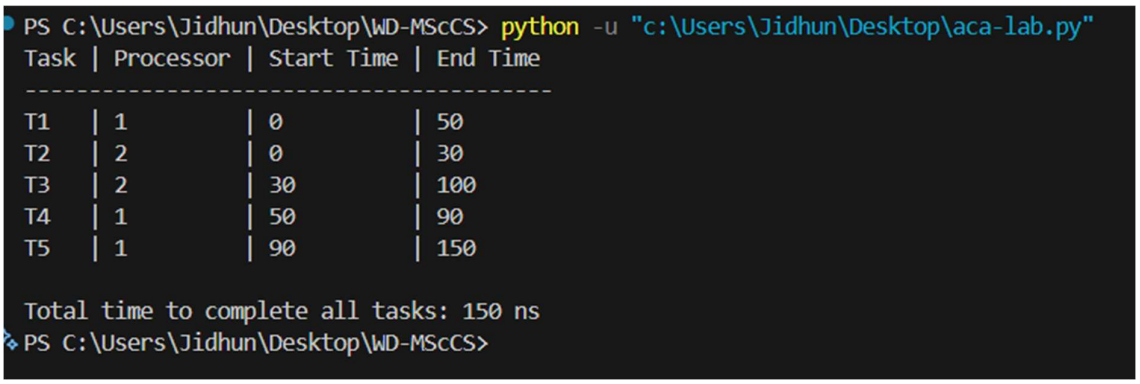
1. List tasks in the order provided: T1, T2, T3, T4, T5.
2. Initialize each processor's available time to 0 (P1_time = 0, P2_time = 0).
3. For each task in list order:
   a. Find the processor with the smallest current available time (the earliest-available).
   b. Assign the task to that processor.
   c. Set the task's start_time = processor's available_time.
   d. Set the task's end_time = start_time + task_execution_time.
   e. Update the processor's available_time = end_time.
4. After all tasks assigned, compute the makespan = max(available_time of all processors).
5. Compute performance metrics:
   - Sequential_time = sum of all task times.
   - Speedup = Sequential_time / makespan.
   - Efficiency = Speedup / number_of_processors.
   - Processor utilizations = busy_time_on_processor / makespan.

Code Section:

```
# Task list (task_id, execution_time in ns)
tasks = [("T1", 50), ("T2", 30), ("T3", 70), ("T4", 40), ("T5",
60)]

# Number of processors
num_processors = 2

# Track next available time for each processor
processor_time = [0] * num_processors

# Store task assignment info
task_allocation = []

# Assign tasks to processors
for task_id, exec_time in tasks:
    # Find the processor which is free the earliest
    next_proc = processor_time.index(min(processor_time))
    start_time = processor_time[next_proc]
    end_time = start_time + exec_time
    processor_time[next_proc] = end_time  # update available
time
```

```
    task_allocation.append((task_id, next_proc+1, start_time,
end_time))

# Display results
print("Task | Processor | Start Time | End Time")
print("-----------------------------------------")
for task_id, proc, start, end in task_allocation:
    print(f"{task_id:<4} | {proc:<9} | {start:<10} | {end}")

# Total time to complete all tasks
total_time = max(processor_time)
print(f"\nTotal time to complete all tasks: {total_time} ns")
```

Output Section:



- Processor finish times: **P1_end = 150 ns**, **P2_end = 100 ns**.
- **Makespan = 150 ns**.

Explanation:

Greedy Scheduling:

- Each task is assigned to the processor that becomes available first.
- T1 & T2 start at 0 → run in parallel on P1 and P2.
- T3 waits for P2 to finish T2, starts at 30 ns.
- T4 waits for P1 to finish T1, starts at 50 ns.
- T5 starts on P2 after T3 completes, at 100 ns.

Total Time:

- Maximum of processor completion times → max(P1_end, P2_end) = 160 ns.

Parallel Execution:

- Multiprocessor system reduces total execution time compared to sequential execution.

EXERCISE 6: Simulation of Vector Processor

Problem Statement:

A vector processor contains vector registers with capacity for **4 elements each**.
Given two vectors:

- **Vector A = [2, 4, 6, 8]**
- **Vector B = [1, 3, 5, 7]**

Tasks to perform:

1. Simulate vector operations: **Addition, Subtraction, Multiplication**
2. Display the results of each operation
3. Simulate storing operands/results in vector registers
4. Extend the simulation to compute the **dot product** of A and B

Aim:

- To understand how vector processors perform element-wise operations on vectors.
- To simulate vector addition, subtraction, and multiplication using vector registers.
- To observe how vector instructions operate in parallel on multiple data elements.
- To compute the dot product and recognize its importance in vector processing.

Theory:

Vector Processing Concept

Vector processors operate on entire sequences of data simultaneously. Each instruction processes all corresponding elements of the vectors stored in **vector registers**. This eliminates repeated instruction fetch/decode cycles seen in scalar CPUs and enables **data-level parallelism**.

2. Vector Registers

- Serve as storage for multiple data elements (here, 4-element registers).
- Support vector instructions that operate on all stored elements simultaneously.

- Improve throughput by minimizing memory access frequency.

3. Vector Operations

Vector instructions perform the same operation on all elements:

- **ADDV**: A[i] + B[i]
- **SUBV**: A[i] − B[i]
- **MULV**: A[i] × B[i]

These operations are typically pipelined, enabling high throughput as multiple elements are processed in stages.

4. Dot Product in Vector Processors

Dot product = Σ (A[i] × B[i])
This operation is a combination of:

1. **Vector multiplication**, and
2. **Reduction** (summing products).

Vector processors support reductions efficiently using specialized pipelines.

5. Parallelism

Although Python simulates the operations **sequentially**, an actual vector processor executes multiple data operations simultaneously using:

- Vector ALUs
- Deep pipelines
- Lane-based parallel execution

Thus, vector processors significantly outperform scalar processors for vector workloads.

Algorithm:

1. **Initialize Vector Registers**
   - Load A = [2, 4, 6, 8] into vector register VR_1
   - Load B = [1, 3, 5, 7] into vector register VR_2
2. **Vector Addition (A + B)**
   - For each index i:
     Result_Add[i] = VR_1[i] + VR_2[i]
   - Store the resulting vector in VR_3

3. **Vector Subtraction (A − B)**

   - For each index i:
     Result_Sub[i] = $VR_1[i] - VR_2[i]$
   - Store the resulting vector in VR_4

4. **Vector Multiplication (A × B)**

   - For each index i:
     Result_Mul[i] = $VR_1[i] \times VR_2[i]$
   - Store the resulting vector in VR_5

5. **Dot Product Computation**

   - Multiply corresponding elements: P[i] = A[i] × B[i]
   - Perform reduction: Dot = Σ P[i]

6. **Display All Vector Results**

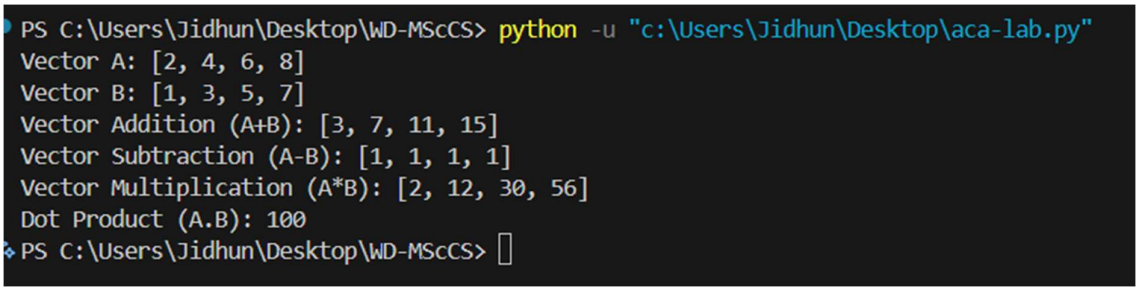
   - Show vector A, vector B, A+B, A−B, A×B, and dot product

Code Section:

```
# Vector registers
vector_register_A = [2, 4, 6, 8]
vector_register_B = [1, 3, 5, 7]

# Function to simulate vector operations
def vector_operation(op, vecA, vecB):
    result = []
    for a, b in zip(vecA, vecB):
        if op == "ADD":
            result.append(a + b)
        elif op == "SUB":
            result.append(a - b)
        elif op == "MUL":
            result.append(a * b)
        else:
            raise ValueError("Unsupported operation")
    return result

# Perform vector operations
vector_add = vector_operation("ADD", vector_register_A,
vector_register_B)
```

```
vector_sub = vector_operation("SUB", vector_register_A,
vector_register_B)
vector_mul = vector_operation("MUL", vector_register_A,
vector_register_B)

# Compute dot product
dot_product = sum(a*b for a, b in zip(vector_register_A,
vector_register_B))

# Display results
print("Vector A:", vector_register_A)
print("Vector B:", vector_register_B)
print("Vector Addition (A+B):", vector_add)
print("Vector Subtraction (A-B):", vector_sub)
print("Vector Multiplication (A*B):", vector_mul)
print("Dot Product (A.B):", dot_product)
```

Output Section:



- Vector registers VR_1 and VR_2 successfully hold and operate on 4-element vectors.
- Element-wise vector operations (ADD, SUB, MUL) were simulated accurately.
- The dot product was computed using vector multiplication followed by reduction.
- The simulation shows how vector processors achieve high performance by applying a single instruction to an entire vector.
- Though Python processes the operations sequentially, real vector processors execute them **in parallel**, demonstrating the core advantage of vector architectures.

EXERCISE 7: Simulation of Thread-Level Parallelism (TLP)

Thread-Level Parallelism (TLP) refers to the ability of a processor to execute **multiple independent threads** concurrently. Modern CPUs achieve TLP using multiple cores, simultaneous multithreading (SMT), or hardware-level scheduling. Each thread represents an independent sequence of instructions that may run in parallel with others.

Scenario:

- A program contains **multiple independent tasks**.
- Each task is assigned to a **separate thread**.
- Threads run **in parallel**, simulating CPU-level TLP.
- Track **start time** and **end time** of each task.

Given tasks:

Task	Execution Time (seconds)
Task1	2
Task2	3
Task3	1
Task4	2

All tasks begin execution simultaneously on different thread

Problem Statement:

Simulate Thread-Level Parallelism by executing four independent tasks concurrently using threads. Each task has a fixed execution time. The simulation must:

1. Create a separate thread for each task.
2. Start all threads in parallel.
3. Track the order in which threads start and finish.
4. Show the parallel nature of execution by observing overlap in execution times.
5. Ensure all threads complete before the program terminates.

Aim:

- To understand the concept of Thread-Level Parallelism in modern processors.
- To simulate parallel task execution using software threads.
- To observe thread start/finish timing and compare parallel vs. sequential behavior.
- To demonstrate how TLP improves throughput and reduces execution time.

Theory:

What is Thread-Level Parallelism?

TLP arises when a program is divided into **multiple threads** that can execute simultaneously. Each thread may run on a different CPU core or hardware thread, allowing true parallel execution.

When is TLP useful?

- Independent tasks
- Multi-module operations
- Server workloads
- Web servers handling multiple clients
- Scientific simulations
- Data preprocessing pipelines

How CPUs support TLP

- **Multiple cores** → each core executes a thread simultaneously
- **Simultaneous Multithreading (SMT)** → a core executes more than one thread at once
- **Thread schedulers** → time-sharing when cores are limited

Thread Coordination

Threads may run independently, but programs often require:

- Synchronization
- Waiting for all threads to finish (join)
- Shared memory protection

Real vs Simulated TLP

- Python threading simulates concurrency but may not achieve full parallelism due to the GIL (but sleep() releases GIL).
- Real CPUs execute threads **truly concurrently** on separate cores.

Algorithm:

1. **Define Tasks**
   Each task has a name and execution time (Task1–Task4).
2. **Create Thread Objects**
   For each task, create a thread assigned to a function that simulates work.
3. **Start All Threads**
   Start threads nearly simultaneously to simulate parallel execution.

Each thread prints "started" with its name.

4. **Simulate Execution Time**
   Each thread waits for its specified duration to mimic task execution.
5. **Mark Task Completion**
   Each thread prints "finished" with its execution time.
6. **Wait for All Threads (Join)**
   Ensure main program waits until every thread finishes.
7. **Display Final Output**
   Print “All tasks completed”.

Code Section:

```
import threading
import time

# Define tasks (task_name, execution_time in seconds)
tasks = [("Task1", 2), ("Task2", 3), ("Task3", 1), ("Task4", 2)]

# Function to simulate task execution
def execute_task(task_name, exec_time):
    print(f"{task_name} started")
    time.sleep(exec_time)  # Simulate execution time
    print(f"{task_name} finished after {exec_time} seconds")

# Create threads for each task
threads = []
for task_name, exec_time in tasks:
    t = threading.Thread(target=execute_task, args=(task_name,
exec_time))
    threads.append(t)
    t.start()  # Start thread

# Wait for all threads to complete
for t in threads:
    t.join()

print("All tasks completed")
```

Output Section:

```
PS C:\Users\Jidhun\Desktop\WD-MScCS> python -u "c:\Users\Jidhun\Desktop\aca-lab.py"
Task1 started
Task2 started
Task3 started
Task4 started
Task3 finished after 1 seconds
Task4 finished after 2 seconds
Task1 finished after 2 seconds
Task2 finished after 3 seconds
All tasks completed
PS C:\Users\Jidhun\Desktop\WD-MScCS>
```

Explanation:

- **Parallel Start**

  All tasks start almost simultaneously → representing **Thread-Level Parallelism**.

- **Execution Overlap**

  Despite starting together:

  - **Task3** finishes first (1 sec)
  - **Task1** and **Task4** finish next (2 sec)
  - **Task2** finishes last (3 sec)

  This shows threads execute **independently and concurrently**.

- **Throughput Improvement**

  Sequential execution time would be:
  2 + 3 + 1 + 2 = **8 seconds**

  Parallel execution completed in:
  3 seconds (maximum execution time among threads)

  → This demonstrates TLP improves overall performance drastically.

- **Synchronization**

  .join() ensures the program waits for all threads before printing the final completion message.

EXERCISE 8: Simulation of Data-Level Parallelism (DLP)

Data-Level Parallelism (DLP) is the capability to perform the same operation on multiple data elements simultaneously. DLP is exploited by vector processors, SIMD instructions, and massively parallel accelerators (GPUs) to achieve very high throughput on homogeneous data workloads.

Key Characteristics

- Operates on **vectors, arrays, or large datasets**.
- Each data-element operation is **independent**, enabling parallel processing.
- Common in **vector instructions (SIMD)**, **GPU kernels**, and **vector processors**.
- Typical use cases: vector arithmetic, image filters, and matrix computations.

Example Operations

- Vector addition: C[i] = A[i] + B[i]
- Image processing (per-pixel filters)
- Matrix multiplication (element or block based)

Problem Statement:

Given two arrays:

- **A = [1, 2, 3, 4, 5]**
- **B = [10, 20, 30, 40, 50]**

Perform and simulate, using a parallel-execution model, the following element-wise operations:

1. Addition (A + B)
2. Subtraction (A − B)
3. Multiplication (A × B)

Also demonstrate how to map element-level work to parallel workers (threads/executor) and collect results.

Aim:

- To simulate Data-Level Parallelism by executing identical operations across array elements concurrently.
- To demonstrate element-wise addition, subtraction, and multiplication results.
- To illustrate how a thread pool / executor maps independent element tasks to worker threads and returns results.
- To discuss practical considerations (overhead, scalability, when DLP is most beneficial).

Theory:

DLP vs TLP:

DLP parallelizes *data* (same operation on many elements); TLP parallelizes *control* (multiple threads doing different tasks).

Execution model:

In DLP, an operation (for example, add) is broadcast to many lanes; each lane performs the operation on a distinct piece of data. In software simulation, a thread pool assigns element-pairs (A[i], B[i]) to worker threads which compute results and return them.

Hardware vs software:

Hardware SIMD/vector units operate in lockstep with negligible per-element overhead. Software-level parallelism (threads/executor) introduces task-scheduling and synchronization overhead that can dominate for small arrays.

Scalability:

DLP scales well with data size; larger arrays amortize parallelization overhead and better utilize wide vector units or many GPU cores.

Correctness:

Element independence ensures deterministic results when there are no cross-element dependencies.

Algorithm:

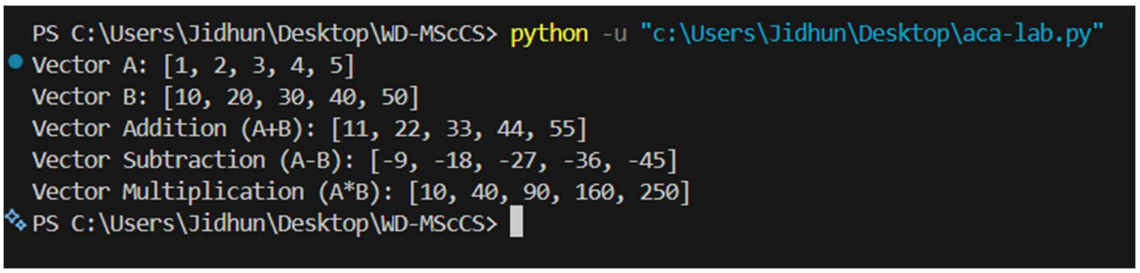
1. **Load data**: Place input arrays `A` and `B` into logical vector registers or memory buffers.
2. **Define element operation**: Choose operation `op ∈ {ADD, SUB, MUL}` that maps `(a,b) → result`.
3. **Create worker pool**: Initialize a set of workers (threads or executor workers).
4. **Map elements to workers**: For each index `i` in 0..n−1, submit the task `(A[i], B[i], op)` to the worker pool.
5. **Execute in parallel**: Workers compute their assigned element operations independently.
6. **Collect results**: Gather results in the original order (or specified order) into a result array.
7. **Aggregate (if required)**: For reductions (e.g., sum of elements), apply a reduction step to combine per-element results.
8. **Shutdown pool**: Wait for all workers to finish and close the pool.
9. **Report outputs**: Display input arrays and result arrays for each operation.

Code Section:

```
from concurrent.futures import ThreadPoolExecutor

# Data arrays
A = [1, 2, 3, 4, 5]
B = [10, 20, 30, 40, 50]

# Function to perform operation on single data element
def process_element(a, b, op):
    if op == "ADD":
        return a + b
    elif op == "SUB":
        return a - b
    elif op == "MUL":
        return a * b
    else:
        raise ValueError("Unsupported operation")

# Function to perform vector operation in parallel
def vector_operation_parallel(vecA, vecB, op):
    result = []
    with ThreadPoolExecutor() as executor:
        # Map each element pair to the executor
        futures = [executor.submit(process_element, a, b, op)
for a, b in zip(vecA, vecB)]
        for f in futures:
            result.append(f.result())
```

```
    return result

# Perform operations
vector_add = vector_operation_parallel(A, B, "ADD")
vector_sub = vector_operation_parallel(A, B, "SUB")
vector_mul = vector_operation_parallel(A, B, "MUL")

# Display results
print("Vector A:", A)
print("Vector B:", B)
print("Vector Addition (A+B):", vector_add)
print("Vector Subtraction (A-B):", vector_sub)
print("Vector Multiplication (A*B):", vector_mul)
```

Output Section:



The simulation demonstrates Data-Level Parallelism by applying identical operations independently across array elements and collecting correct element-wise results.

For production-level performance on large datasets, prefer hardware SIMD, compiler auto-vectorization, or GPU implementations; software threading is useful for teaching and for coarse-grained parallelism where element blocks are large enough to amortize overhead.
