Islamic University

Department of
Information and Communication Technology

Course Code: ICT-3206

Course: Computer Architecture and Organization Laboratory

Lab Report

Submitted to:
Dr. Md Zahidul Islam
Professor
Department of Information and Communication Technology
Islamic University, Bangladesh

Submitted by:
Name: S. M. Sajjad Hossain Soykot
Roll: 2018040
Reg. No: 1315
Session: 2020-21

INDEX

Experiment 1: Register Transfer Operations (4-bit)

Objectives:

- Simulate basic register transfer operations using a 4-bit register model.
- Perform data transfer, increment, bitwise AND, and logical left shift micro-operations.
- Understand how control signals direct specific operations between registers.

Theory:

A register is a small, high-speed storage element inside the CPU used to hold binary data temporarily. Micro-operations are the simplest operations performed on the data in registers, such as:

- **Data Transfer (A → B)**: Copies the contents from one register to another.
- **Increment**: Adds 1 to the register content, useful in counters and loops.
- **Logical AND**: Bitwise comparison producing `1` only where both bits are `1`.
- **Logical Left Shift**: Moves all bits one position left, inserting `0` at the LSB; used for multiplication by 2 in unsigned binary arithmetic.

In this experiment, a **Control Unit** coordinates these operations by activating the right control signals for each step.

Source Code:

```
class Register4:
    def __init__(self, value=0):
        self.value = value & 0b1111  # keep 4 bits

    def load(self, value):
        self.value = value & 0b1111

    def inc(self):
        self.value = (self.value + 1) & 0b1111

    def and_with(self, other):
        return Register4(self.value & other.value)

    def shift_left(self):
        self.value = ((self.value << 1) & 0b1111)

    def bits(self):
        return format(self.value, "04b")

class ControlUnit:
    def __init__(self):
        self.A = Register4()
        self.B = Register4()
        self.C = Register4()
```

```
    def transfer_A_to_B(self):
        self.B.load(self.A.value)

    def increment_A(self):
        self.A.inc()

    def and_A_B_to_C(self):
        self.C = self.A.and_with(self.B)

    def shift_A_left(self):
        self.A.shift_left()

    def state(self):
        return dict(A=self.A.bits(), B=self.B.bits(), C=self.C.bits())

def demo():
    cu = ControlUnit()
    cu.A.load(0b1010)
    print("Initial:", cu.state())
    cu.transfer_A_to_B()
    print("After transfer A→B:", cu.state())
    cu.increment_A()
    print("After INC A:", cu.state())
    cu.and_A_B_to_C()
    print("After AND A&B → C:", cu.state())
    cu.shift_A_left()
    print("After SHIFT LEFT A:", cu.state())

if __name__ == "__main__":
    demo()
```

Output:

```
Initial: {'A': '1010', 'B': '0000', 'C': '0000'}
After transfer A→B: {'A': '1010', 'B': '1010', 'C': '0000'}
After INC A: {'A': '1011', 'B': '1010', 'C': '0000'}
After AND A&B → C: {'A': '1011', 'B': '1010', 'C': '1010'}
After SHIFT LEFT A: {'A': '0110', 'B': '1010', 'C': '1010'}
```

Experiment 2: 4-bit ALU with Status Flags + 1-bit shifter

Objectives:

- Design and simulate a 4-bit Arithmetic Logic Unit (ALU) that performs **ADD**, **SUB**, **AND**, and **OR** operations.
- Implement **Carry (C)** and **Zero (Z)** status flags.
- Integrate a **1-bit logical left shifter** for the ALU output.

Theory:

An **ALU** is the core computational unit of the CPU, capable of executing arithmetic (e.g., addition, subtraction) and logic (e.g., AND, OR) operations.

- **Addition & Subtraction**: Subtraction is implemented using **two's complement** (`A + (~B + 1)`), with the **Carry flag** indicating overflow in addition or borrow in subtraction.
- **Bitwise AND / OR**: Operate on individual bits; AND outputs `1` only when both inputs have `1`, OR outputs `1` if at least one input has `1`.
- **Zero flag (Z)**: Set when the result is all zeros, useful for branching decisions.
- **Logical Left Shift**: Shifts bits to the left, inserting `0` at the LSB; effectively multiplies unsigned numbers by 2.

The ALU combines these functions under control signals, while the status flags provide feedback for program flow control.

Source Code:

```
# Lab 02 – 4-bit ALU with Status Flags + 1-bit shifter

def full_add_4(a, b, sub=False):
    # a,b are 4-bit ints. If sub=True, compute a + (~b + 1)
    if sub:
        b = ((~b) & 0b1111) + 1
    s = a + b
    carry = 1 if s > 0b1111 else 0
    out = s & 0b1111
    return out, carry

def alu_4(a, b, ctrl):
    # ctrl: 0=ADD, 1=SUB, 2=AND, 3=OR
    if ctrl == 0:
        out, c = full_add_4(a, b, sub=False)
    elif ctrl == 1:
        out, c = full_add_4(a, b, sub=True)
    elif ctrl == 2:
        out, c = (a & b), 0
```

```
    else:
        out, c = (a | b), 0
    z = 1 if out == 0 else 0
    return out, c, z

def shift_left_1(x):
    return (x << 1) & 0b1111

def demo():
    A, B = 0b1010, 0b0101
    labels = ["ADD","SUB","AND","OR"]
    for ctrl in range(4):
        out, c, z = alu_4(A,B,ctrl)
        print(f"{labels[ctrl]}: A={A:04b} B={B:04b} -> OUT={out:04b} C={c} Z={z}
SHIFTED={shift_left_1(out):04b}")

if __name__ == "__main__":
    demo()
```

Output:

```
ADD: A=1010 B=0101 -> OUT=1111 C=0 Z=0 SHIFTED=1110
SUB: A=1010 B=0101 -> OUT=0101 C=1 Z=0 SHIFTED=1010
AND: A=1010 B=0101 -> OUT=0000 C=0 Z=1 SHIFTED=0000
OR: A=1010 B=0101 -> OUT=1111 C=0 Z=0 SHIFTED=1110
```

Experiment 3: Control Logic (Hardwired + PLA simulation)

Opcodes: 00=LOAD, 01=STORE, 10=ADD, 11=SUB

Control lines: MRD, MWR, RLD, ALU_ADD, ALU_SUB

Objectives:

- Implement control signal generation for basic instructions (**LOAD**, **STORE**, **ADD**, **SUB**) using a **hardwired control unit**.
- Simulate a **Programmable Logic Array (PLA)** to produce the same control signals from opcodes.
- Compare hardwired and PLA-based control designs.

Theory:

The **control unit** in a CPU translates **opcodes** into the necessary control signals to direct data paths, ALU operations, and memory access.

- **Hardwired Control**: Uses fixed combinational logic to directly decode opcodes into control lines. It's fast but less flexible for changes.
- **PLA Control**: Uses an **AND-plane** to generate min-terms (specific input combinations) and an **OR-plane** to combine them into outputs. PLA-based design is more adaptable for new instructions.
- **Control Lines**:
    - **MRD** – Memory Read
    - **MWR** – Memory Write
    - **RLD** – Register Load
    - **ALU_ADD** – Perform addition
    - **ALU_SUB** – Perform subtraction

In this experiment, both methods map the same 2-bit opcodes to corresponding control signals, demonstrating two common approaches to instruction decoding.

Source Code:

```
# Lab 03 – Control Logic (Hardwired + PLA simulation)

# Opcodes: 00=LOAD, 01=STORE, 10=ADD, 11=SUB
# Control lines: MRD, MWR, RLD, ALU_ADD, ALU_SUB

def hardwired_control(opcode):
    MRD = int(opcode == 0b00)
    MWR = int(opcode == 0b01)
    RLD = int(opcode in (0b00,0b10,0b11))
    ALU_ADD = int(opcode == 0b10)
    ALU_SUB = int(opcode == 0b11)
    return dict(MRD=MRD, MWR=MWR, RLD=RLD, ALU_ADD=ALU_ADD, ALU_SUB=ALU_SUB)
```

```
def pla_control(opcode):
    # AND-plane minterms m0..m3 for inputs [op1 op0]
    m = [0,0,0,0]
    m[opcode] = 1
    # OR-plane to form outputs
    MRD = m[0]
    MWR = m[1]
    RLD = m[0] or m[2] or m[3]
    ALU_ADD = m[2]
    ALU_SUB = m[3]
    return dict(MRD=MRD, MWR=MWR, RLD=RLD, ALU_ADD=ALU_ADD, ALU_SUB=ALU_SUB)

def demo():
    names = ["LOAD","STORE","ADD","SUB"]
    for op in range(4):
        hw = hardwired_control(op)
        pla = pla_control(op)
        print(f"Opcode {op:02b} ({names[op]}) -> Hardwired {hw} | PLA {pla}")

if __name__ == "__main__":
    demo()
```

Output:

```
Opcode 00 (LOAD) -> Hardwired {'MRD': 1, 'MWR': 0, 'RLD': 1, 'ALU_ADD': 0, 'ALU_SUB': 0} | PLA {'MRD': 1, 'MWR': 0, 'RLD': 1, 'ALU_ADD': 0, 'ALU_SUB': 0}
Opcode 01 (STORE) -> Hardwired {'MRD': 0, 'MWR': 1, 'RLD': 0, 'ALU_ADD': 0, 'ALU_SUB': 0} | PLA {'MRD': 0, 'MWR': 1, 'RLD': 0, 'ALU_ADD': 0, 'ALU_SUB': 0}
Opcode 10 (ADD) -> Hardwired {'MRD': 0, 'MWR': 0, 'RLD': 1, 'ALU_ADD': 1, 'ALU_SUB': 0} | PLA {'MRD': 0, 'MWR': 0, 'RLD': 1, 'ALU_ADD': 1, 'ALU_SUB': 0}
Opcode 11 (SUB) -> Hardwired {'MRD': 0, 'MWR': 0, 'RLD': 1, 'ALU_ADD': 0, 'ALU_SUB': 1} | PLA {'MRD': 0, 'MWR': 0, 'RLD': 1, 'ALU_ADD': 0, 'ALU_SUB': 1}
```

Experiment 4: Interrupt-Driven I/O (Simulation)

Simulate a CPU doing work; a keyboard 'event' interrupts processing.

Objectives:

- Simulate **asynchronous event handling** using an **Interrupt Service Routine (ISR)**.
- Observe how the CPU pauses main program execution to service an interrupt.
- Demonstrate the benefit of interrupts over polling for I/O operations.

Theory:

An **interrupt** is a hardware or software signal that temporarily halts the CPU's current execution so that it can handle an urgent task.

- **Main Program Flow**: Executes normal tasks until an interrupt occurs.
- **Interrupt Service Routine (ISR)**: Special function triggered automatically on interrupt to handle the event (e.g., reading input, toggling an LED).
- **Interrupt-Driven I/O**: More efficient than **polling**, as the CPU only responds when events occur, saving processing time.
- **Context Saving/Restoring**: Before executing the ISR, the CPU saves its current state; after servicing, it restores the state and resumes where it left off.

In this simulation, a **keyboard event** is emulated by a timed trigger, which sets a flag to invoke the ISR, illustrating real-world interrupt behavior in embedded systems.

Source Code:

```
# Lab 04 – Interrupt-Driven I/O (Simulation)
# Simulate a CPU doing work; a keyboard 'event' interrupts processing.

import threading, time

class CPU:
    def __init__(self):
        self.counter = 0
        self.interrupt_flag = False

    def isr(self):
        print("[ISR] Interrupt serviced: toggling LED and logging event.")
        # simulated LED state change
        print("[ISR] LED -> ON")
        time.sleep(0.1)
        print("[ISR] LED -> OFF")
```

```
    def main_loop(self, duration=3.0):
        start = time.time()
        while time.time() - start < duration:
            # simulate work
            self.counter += 1
            if self.interrupt_flag:
                self.interrupt_flag = False
                self.isr()
            time.sleep(0.05)
        print("[CPU] main loop done. counter =", self.counter)

def trigger_interrupt(cpu, after=0.5):
    time.sleep(after)
    cpu.interrupt_flag = True

def demo():
    cpu = CPU()
    t = threading.Thread(target=trigger_interrupt, args=(cpu,1.0))
    t.start()
    cpu.main_loop(2.5)
    t.join()

if __name__ == "__main__":
    demo()
```

Output:

```
[ISR] Interrupt serviced: toggling LED and logging event.
[ISR] LED -> ON
[ISR] LED -> OFF
[CPU] main loop done. counter = 45
```

Experiment 5: Memory Map & Address Decoding (Simulation)

ROM_RANGE = range(0x0000, 0x1000) # 0000-0FFF
RAM_RANGE = range(0x4000, 0x5000) # 4000-4FFF

Objectives:

- Simulate a **memory map** containing ROM and RAM regions.
- Implement **address decoding** to identify target memory devices.
- Demonstrate **read-only ROM** behavior and writable RAM access.

Theory:

In microprocessor systems, **memory mapping** assigns address ranges to physical memory devices like ROM, RAM, or I/O.

- **Address Decoding**: Logic that interprets high-order address lines to generate **chip select** signals for specific devices.
- **ROM (Read-Only Memory)**: Non-volatile storage; contents cannot be modified during normal operation.
- **RAM (Random Access Memory)**: Volatile storage; allows both read and write operations.
- **Memory-Mapped I/O**: Using part of the address space to access I/O devices instead of memory.

In this experiment, address ranges `0000H–0FFFH` map to ROM and `4000H–4FFFH` map to RAM. The decoding logic checks address ranges to route read/write requests and enforces read-only restrictions for ROM.

Source Code:

```
# Lab 05 – Memory Map & Address Decoding (Simulation)
ROM_RANGE = range(0x0000, 0x1000)   # 0000–0FFF
RAM_RANGE = range(0x4000, 0x5000)   # 4000–4FFF

class Bus:
    def __init__(self):
        self.rom = {addr: 0 for addr in ROM_RANGE}
        self.ram = {addr: 0 for addr in RAM_RANGE}
        # preload a ROM location to show read-only
        self.rom[0x0000] = 0xEA

    def decode(self, addr):
        a15 = (addr >> 15) & 1
        a14 = (addr >> 14) & 1
        # Example: RAM when a15=0 and a14=1
        if addr in RAM_RANGE:
            return "RAM"
        elif addr in ROM_RANGE:
            return "ROM"
        else:
            return "NONE"
```

```
    def read(self, addr):
        region = self.decode(addr)
        if region == "ROM":
            return self.rom[addr]
        if region == "RAM":
            return self.ram[addr]
        raise ValueError("Invalid address")

    def write(self, addr, val):
        region = self.decode(addr)
        if region == "ROM":
            raise PermissionError("Cannot write to ROM")
        if region == "RAM":
            self.ram[addr] = val & 0xFF
        else:
            raise ValueError("Invalid address")

def demo():
    bus = Bus()
    print("Decode 0x4000 ->", bus.decode(0x4000))
    print("Write 0x55 to 0x4000 (RAM)")
    bus.write(0x4000, 0x55)
    print("Read 0x4000 =", hex(bus.read(0x4000)))
    print("Read ROM[0x0000] =", hex(bus.read(0x0000)))
    try:
        bus.write(0x0000, 0x99)
    except PermissionError as e:
        print("Expected error:", e)

if __name__ == "__main__":
    demo()
```

Output:

```
Decode 0x4000 -> RAM
Write 0x55 to 0x4000 (RAM)
Read 0x4000 = 0x55
Read ROM[0x0000] = 0xea
Expected error: Cannot write to ROM
```

Experiment 6: 8085-style tasks in Python

Objectives:

- Replicate common **8085 Assembly Language Programming (ALP)** routines using Python.
- Implement 8-bit **addition**, **subtraction with borrow**, **largest-of-5** search, and **sum of array** operations.
- Demonstrate flag concepts such as **carry** and **borrow** in high-level code.

Theory:

Classic **8085 ALP** tasks operate on 8-bit accumulators and use **status flags** to indicate conditions like overflow or borrow.

- **Addition**: Performed in the accumulator; the **carry flag** is set if the result exceeds 8 bits.
- **Subtraction**: Done as `A - B`; the **borrow flag** indicates if the minuend is smaller than the subtrahend.
- **Largest of N elements**: Sequentially compare values, keeping track of the maximum.
- **Sum of array**: Iteratively add elements, wrapping around to 8 bits (mod 256) to simulate 8-bit CPU behavior.

In this Python simulation, the logic mirrors the 8085 routines but avoids assembly syntax, making the behavior easier to observe while preserving the original operational flow.

Source Code:

```python
# Lab 06 – 8085-style tasks in Python

def add_8(a, b):
    s = a + b
    carry = s > 0xFF
    return s & 0xFF, carry

def sub_8(a, b):
    d = a - b
    borrow = d < 0
    return d & 0xFF, borrow

def largest_of_5(arr):
    assert len(arr) == 5
    m = arr[0]
    for x in arr[1:]:
        if x > m: m = x
    return m

def sum_of_arr(arr):
    s = 0
    for x in arr:
        s = (s + x) & 0xFF
    return s
```

```
def demo():
    a,b = 0x25, 0x13
    s,c = add_8(a,b)
    print(f"ADD {a:#04x}+{b:#04x}={s:#04x} C={int(c)}")
    d,br = sub_8(0x50,0x30)
    print(f"SUB 0x50-0x30={d:#04x} Borrow={int(br)}")
    arr = [0x11,0x88,0x34,0x22,0x77]
    print("Largest of", arr, "=", hex(largest_of_5(arr)))
    print("Sum of", arr, "=", hex(sum_of_arr(arr)))

if __name__ == "__main__":
    demo()
```

Output:

```
ADD 0x25+0x13=0x38 C=0
SUB 0x50-0x30=0x20 Borrow=0
Largest of [17, 136, 52, 34, 119] = 0x88
Sum of [17, 136, 52, 34, 119] = 0x66
```

Experiment 7: Threshold + LED Control (Port simulation)

Objectives:

- Simulate controlling an LED through a specific I/O port using Python.
- Implement threshold-based decision logic to switch the LED ON or OFF.
- Demonstrate basic port write and read operations in a high-level simulation.

Theory:

In microprocessor-based systems, I/O ports are used to interface with external devices like LEDs, displays, and sensors.

- A threshold value is compared against an input (e.g., a calculated sum or sensor reading) to determine the output state.
- If the input meets or exceeds the threshold, the LED port is written with `0xFF` to represent LED ON; otherwise, it is written with `0x00` for LED OFF.
- Port write operations send data to an output device, while port read operations retrieve the current state of the device.
- This method is widely used in embedded systems for event-triggered control, such as warning indicators, alarms, and automation tasks.

In this Python simulation, the threshold decision logic is implemented through conditional statements, and LED control is emulated by writing to and reading from a dictionary-based virtual port.

Source Code:

```python
# Lab 07 – Threshold + LED Control (Port simulation)

PORT = {"0x80": 0x00}  # LED port

def write_port(addr_hex, val):
    PORT[addr_hex] = val & 0xFF

def read_port(addr_hex):
    return PORT.get(addr_hex, 0x00)

def threshold_led(sum_val, threshold=0x50):
    if sum_val >= threshold:
        write_port("0x80", 0xFF)  # LED ON
    else:
        write_port("0x80", 0x00)  # LED OFF

def demo():
    for s in (0x30, 0x50, 0x70):
        threshold_led(s, 0x50)
        led = read_port("0x80")
        print(f"Sum={s:#04x} -> LED={'ON' if led==0xFF else 'OFF'}")

if __name__ == "__main__":
    demo()
```

Output:

Sum=0x30 -> LED=OFF

Sum=0x50 -> LED=ON

Sum=0x70 -> LED=ON

Experiment 8: Tiny 2-stage Pipeline (Fetch/Execute) toy

Objectives:

- Simulate a simple 2-stage CPU pipeline consisting of Fetch (IF) and Execute (EX) stages.
- Observe instruction flow and overlapping execution between stages.
- Demonstrate how pipelining increases throughput in instruction execution.

Theory:

A pipeline in CPU architecture allows overlapping of instruction stages to improve execution speed. In a 2-stage pipeline:

- **Fetch (IF):** The next instruction is retrieved from memory and placed in the instruction register.
- **Execute (EX):** The previously fetched instruction is decoded and executed, updating registers as needed.
- At each clock cycle, one instruction is executed while the next one is fetched, increasing efficiency compared to sequential execution.
- Pipeline hazards (data, control, structural) may occur when instructions depend on results from previous ones, potentially requiring stalls or forwarding techniques.

In this Python simulation, register operations like `LDI` (load immediate) and `ADD` are processed in the EX stage, while the next instruction is fetched in parallel. The output shows the register states after each execution step, illustrating basic pipelining behavior.

Source Code:

```
# Lab 08 – Tiny 2-stage Pipeline (Fetch/Execute) toy

PROGRAM = [
    ("LDI", "R0", 5),
    ("LDI", "R1", 7),
    ("ADD", "R2", "R0", "R1"),
    ("ADD", "R0", "R0", "R2"),
]

class CPU2Stage:
    def __init__(self, prog):
        self.prog = prog
        self.pc = 0
        self.reg = {f"R{i}":0 for i in range(4)}
        self.IF = None
        self.EX = None

    def step(self):
        # Execute stage
        if self.EX:
```

```
            op = self.EX
            if op[0] == "LDI":
                _, r, val = op
                self.reg[r] = val
            elif op[0] == "ADD":
                _, rd, ra, rb = op
                self.reg[rd] = self.reg[ra] + self.reg[rb]
            print("[EX]", op, "REG:", self.reg)
        # Fetch stage
        self.EX = self.IF
        if self.pc < len(self.prog):
            self.IF = self.prog[self.pc]
            self.pc += 1
        else:
            self.IF = None

    def run(self):
        while self.EX or self.IF or self.pc < len(self.prog):
            self.step()

def demo():
    cpu = CPU2Stage(PROGRAM)
    cpu.run()

if __name__ == "__main__":
    demo()
```

Output:

```
[EX] ('LDI', 'R0', 5) REG: {'R0': 5, 'R1': 0, 'R2': 0, 'R3': 0}
[EX] ('LDI', 'R1', 7) REG: {'R0': 5, 'R1': 7, 'R2': 0, 'R3': 0}
[EX] ('ADD', 'R2', 'R0', 'R1') REG: {'R0': 5, 'R1': 7, 'R2': 12, 'R3': 0}
[EX] ('ADD', 'R0', 'R0', 'R2') REG: {'R0': 17, 'R1': 7, 'R2': 12, 'R3': 0}
```

Experiment 9: Direct-Mapped Cache Simulator (tiny)

Objectives:

- Simulate a simple direct-mapped cache structure using Python.
- Measure cache hit and miss behavior for a given memory access trace.
- Demonstrate the role of cache mapping in memory performance.

Theory:

A cache is a small, high-speed memory that stores frequently accessed data to reduce access time to main memory.

Direct-mapped cache:

- –Each memory block maps to exactly one cache line based on its address.
- The **index** selects the cache line, and the **tag** verifies if the correct block is stored.

Cache access process:

- Calculate the index from the address using `(address // line_size) % number_of_lines`.
- Compare stored tag with the address's tag; if matched and valid, it's a *hit*, otherwise a *miss*.

Hit rate:

- Defined as the ratio of cache hits to total accesses.
- Higher hit rates indicate better cache performance.

Miss types:

- *Compulsory miss*: First access to a block.
- *Capacity miss*: Cache is too small to hold all needed blocks.
- *Conflict miss*: Different blocks map to the same cache line, replacing each other.

In this Python simulation, a 4-line cache with 4-byte line size is accessed using a fixed trace. The program counts hits and misses, then calculates the hit rate to illustrate direct-mapped cache behavior.

Source Code:

```
# Lab 09 – Direct-Mapped Cache Simulator (tiny)

class DirectMappedCache:
    def __init__(self, lines=4, line_size=4):
        self.lines = lines
        self.line_size = line_size
        self.tags = [-1]*lines
        self.valid = [0]*lines

    def access(self, addr):
        index = (addr // self.line_size) % self.lines
        tag = addr // (self.line_size * self.lines)
        hit = self.valid[index] and (self.tags[index] == tag)
        if not hit:
            self.valid[index] = 1
            self.tags[index] = tag
        return bool(hit)

def simulate(trace):
    cache = DirectMappedCache(lines=4, line_size=4)
    hits = 0
    for a in trace:
        if cache.access(a): hits += 1
    return hits, len(trace), hits/len(trace)

def demo():
    trace = [0,1,2,3,16,17,18,19,0,1,2,3]
    hits, n, hr = simulate(trace)
    print(f"Trace len={n} hits={hits} hit_rate={hr:.2f}")

if __name__ == "__main__":
    demo()
```

Output:

```
Trace len=12 hits=9 hit_rate=0.75
```

Experiment 10: SIMD vs MIMD mini demo

Objectives:

- Demonstrate the difference between **SIMD** (Single Instruction, Multiple Data) and **MIMD** (Multiple Instruction, Multiple Data) execution models.
- Show how SIMD processes multiple data elements with a single operation.
- Show how MIMD executes independent tasks concurrently.

Theory:

Parallel processing improves performance by executing multiple operations simultaneously.

- **SIMD (Single Instruction, Multiple Data):**
  - Executes the same operation on multiple data elements at once.
  - Ideal for vector/matrix operations, image processing, and other data-parallel workloads.
  - Example: Adding two arrays element-wise in a single step.
- **MIMD (Multiple Instruction, Multiple Data):**
  - Executes different instructions on different data simultaneously.
  - Suitable for task-parallel workloads where each task may perform a unique computation.
  - Example: One task calculates a power, another sums an array, and another performs a different calculation.
- **Performance considerations:**
  - SIMD excels when the same computation must be applied to large datasets.
  - MIMD is better for heterogeneous tasks that cannot be vectorized.

In this Python simulation, `simd_add` performs element-wise addition of two arrays to model SIMD, while `mimd_tasks` executes a list of independent functions with their arguments to model MIMD. The outputs highlight the distinct execution approaches of both models

Source Code:

```
# Lab 10 – SIMD vs MIMD mini demo

def simd_add(a,b):
    return [x+y for x,y in zip(a,b)]

def mimd_tasks(tasks):
    results = []
    for f,args in tasks:
        results.append(f(*args))
    return results

def demo():
    A = [1,2,3,4]
    B = [10,20,30,40]
    print("SIMD add:", simd_add(A,B))
    tasks = [(pow,(2,10)), (pow,(3,5)), (sum,([1,2,3],))]
    print("MIMD tasks:", mimd_tasks(tasks))

if __name__ == "__main__":
    demo()
```

Output:

```
SIMD add: [11, 22, 33, 44]
MIMD tasks: [1024, 243, 6]
```