Basic CPU Components: A Simulation Project

Objective

This project simulates the fundamental operations of a CPU, providing an interactive interface for users to perform tasks such as loading instructions, executing them, writing data to registers, reading data, and exiting the program. By using an **Arithmetic Logic Unit (ALU)** and a set of **general-purpose registers**, the system executes basic arithmetic and logical instructions, offering a practical demonstration of CPU functionality.

Algorithm Overview

Core Steps

1. **Initialization:**
    - Set up **4 general-purpose registers** (R0 to R3), all initialized to `0`.
    - Initialize the **Program Counter (PC)** to `0`.
    - Create an **Instruction Register (IR)** to hold the currently loaded instruction.
2. **User Actions:**
    - **Write Action:** Allow the user to store a value in a specified register.
    - **Read Action:** Display the value stored in a specified register.
    - **Load Instruction:** Accept an instruction, parse it, and store it in the instruction register.
    - **Execute Instruction:** Fetch and execute the instruction using the ALU and update the relevant registers.
    - **Exit:** Terminate the program.
3. **Supported Operations:**
    - **Arithmetic Operations:** Addition, subtraction.
    - **Logical Operations:** AND, OR, NOT.
    - **Error Handling:** Ensure valid input, including proper register numbers and instruction formats.
4. **Program Flow:**
    - The program loops, continuously prompting the user for actions.
    - Actions are performed sequentially until the user chooses to exit.

Detailed Algorithm

1. Initialization

- **Registers:** Initialize four registers (`R0`, `R1`, `R2`, `R3`) to `0`.
- **Program Counter (PC):** Set the program counter to `0`.
- **Instruction Register (IR):** Prepare to store user-provided instructions.

2. Input and Actions

- **Write Action:**
  1. Prompt the user to select a register (`R0` to `R3`).
  2. Prompt the user to input a value.
  3. Store the value in the selected register.
- **Read Action:**
  1. Prompt the user to select a register (`R0` to `R3`).
  2. Display the value stored in the selected register.
- **Load Instruction:**
  1. Prompt the user to input an instruction (e.g., `ADD 0 1 2`).
  2. Parse the operation (e.g., `ADD`) and the involved registers (`0, 1, 2`).
  3. Store the parsed instruction in the instruction register.
- **Execute Instruction:**
  1. Fetch the instruction from the instruction register.
  2. Perform the operation as per the parsed instruction:
     - **ADD:** Add values from two registers and store the result in a third register.
     - **SUB:** Subtract one register's value from another and store the result.
     - **AND, OR:** Perform bitwise operations on two registers and store the result.
     - **NOT:** Perform a bitwise complement on a register and store the result.
  3. Increment the program counter (`PC`).
- **Error Handling:**
  - Ensure register numbers are valid (0-3).
  - Validate instruction format.
  - Provide feedback for invalid inputs.
- **Exit:** Terminate the program when the user selects the "exit" option.

Sample Implementation

```
class CPU:
    def __init__(self):
        self.registers = [0, 0, 0, 0]
        self.program_counter = 0
        self.instruction_register = None

    def write(self, register, value):
        """Writes a value to a specified register."""
        if 0 <= register <= 3:
            self.registers[register] = value
            print(f"Stored {value} in R{register}")
        else:
            print("Error: Invalid register number")

    def read(self, register):
        """Reads the value of a specified register."""
        if 0 <= register <= 3:
            print(f"Value in R{register}: {self.registers[register]}")
        else:
            print("Error: Invalid register number")

    def load(self, instruction):
        """Loads an instruction into the instruction register."""
        try:
            parts = instruction.split()
            operation = parts[0].upper()
            registers = list(map(int, parts[1:]))
            if operation in {"ADD", "SUB", "AND", "OR", "NOT"} and all(0
<= r <= 3 for r in registers):
                self.instruction_register = (operation, registers)
                print(f"Loaded instruction: {instruction}")
            else:
                print("Error: Invalid instruction format or registers")
        except ValueError:
            print("Error: Instruction parsing failed")

    def execute(self):
        """Executes the loaded instruction."""
        if self.instruction_register is None:
```

```
            print("Error: No instruction loaded")
            return

        operation, registers = self.instruction_register
        try:
            if operation == "ADD":
                self.registers[registers[0]] =
self.registers[registers[1]] + self.registers[registers[2]]
            elif operation == "SUB":
                self.registers[registers[0]] =
self.registers[registers[1]] - self.registers[registers[2]]
            elif operation == "AND":
                self.registers[registers[0]] =
self.registers[registers[1]] & self.registers[registers[2]]
            elif operation == "OR":
                self.registers[registers[0]] =
self.registers[registers[1]] | self.registers[registers[2]]
            elif operation == "NOT":
                self.registers[registers[0]] =
~self.registers[registers[1]]
            else:
                print("Error: Unsupported operation")
                return

            print(f"Executed instruction: {operation}")
            self.program_counter += 1
        except IndexError:
            print("Error: Invalid registers for operation")

    def run(self):
        """Interactive loop to perform CPU actions."""
        while True:
            action = input("Choose action (load, execute, write, read,
exit): ").strip().lower()
            if action == "write":
                try:
                    register = int(input("Enter register number (0-3): "))
                    value = int(input("Enter value to store: "))
                    self.write(register, value)
                except ValueError:
                    print("Error: Invalid input")
            elif action == "read":
                try:
```

```
                    register = int(input("Enter register number (0-3): "))
                    self.read(register)
                except ValueError:
                    print("Error: Invalid input")
            elif action == "load":
                instruction = input("Enter instruction (e.g., ADD 0 1 2):
")
                self.load(instruction)
            elif action == "execute":
                self.execute()
            elif action == "exit":
                print("Exiting program...")
                break
            else:
                print("Error: Invalid action")


# Main Execution
if __name__ == "__main__":
    cpu = CPU()
    cpu.run()
```

Sample Interaction:

Input:

CHOOSE ACTION (LOAD, EXECUTE, WRITE, READ, EXIT): WRITE

ENTER REGISTER NUMBER (0-3): 2

ENTER VALUE TO STORE: 15

CHOOSE ACTION (LOAD, EXECUTE, WRITE, READ, EXIT): WRITE

ENTER REGISTER NUMBER (0-3): 1

ENTER VALUE TO STORE: 10

CHOOSE ACTION (LOAD, EXECUTE, WRITE, READ, EXIT): LOAD

ENTER INSTRUCTION (E.G., ADD 0 1 2): ADD 0 1 2

CHOOSE ACTION (LOAD, EXECUTE, WRITE, READ, EXIT): EXECUTE

CHOOSE ACTION (LOAD, EXECUTE, WRITE, READ, EXIT): READ

ENTER REGISTER NUMBER (0-3): 0

Output:

STORED 15 IN R2

STORED 10 IN R1

LOADED INSTRUCTION: ADD 0 1 2

EXECUTED INSTRUCTION: ADD

VALUE IN R0: 25

Conclusion

This project simulates basic CPU operations, including arithmetic and logical processing, using an ALU and general-purpose registers. The interactive design provides insight into CPU concepts like register management, the fetch-execute cycle, and error handling, offering a hands-on way to understand fundamental computer architecture.