Instruction Execution

Here is a Python implementation of the Instruction Fetch-Decode-Execute Cycle for Week 4's tasks, integrating the ALU, registers, and program control:

Code Implementation:

```
class ALU:
  def operate(self, opcode, operand1, operand2=None):
    if opcode == "ADD":
      return operand1 + operand2
    elif opcode == "SUB":
      return operand1 - operand2
    elif opcode == "MUL":
      return operand1 * operand2
    elif opcode == "DIV":
      return operand1 // operand2 if operand2 != 0 else "Error: Division by zero"
    elif opcode == "AND":
      return operand1 & operand2
    elif opcode == "OR":
      return operand1 | operand2
    elif opcode == "XOR":
      return operand1 ^ operand2
    elif opcode == "NOT":
      return ~operand1
```

```
        elif opcode == "CMP":
            return operand1 == operand2
        else:
            raise ValueError(f"Invalid opcode: {opcode}")

class Registers:
    def __init__(self, num_registers=8):
        self.registers = [0] * num_registers

    def read(self, register_index):
        return self.registers[register_index]

    def write(self, register_index, value):
        self.registers[register_index] = value

class ControlUnit:
    def __init__(self, memory_size=256):
        self.program_counter = 0
        self.instruction_register = None
        self.memory = [None] * memory_size

    def fetch(self):
        if self.program_counter < len(self.memory) and
self.memory[self.program_counter]:
```

```
            self.instruction_register = self.memory[self.program_counter]
            self.program_counter += 1
        else:
            self.instruction_register = None

    def load_program(self, program):
        for i, instruction in enumerate(program):
            self.memory[i] = instruction

    def reset(self):
        self.program_counter = 0
        self.instruction_register = None

class CPU:
    def __init__(self):
        self.alu = ALU()
        self.registers = Registers()
        self.control_unit = ControlUnit()

    def decode_and_execute(self, instruction):
        parts = instruction.split()
        opcode = parts[0]

        if opcode in {"ADD", "SUB", "MUL", "DIV", "AND", "OR", "XOR", "CMP"}:
```

```
            reg1 = int(parts[1][1:])
            reg2 = int(parts[2][1:])
            reg3 = int(parts[3][1:])
            operand1 = self.registers.read(reg2)
            operand2 = self.registers.read(reg3)
            result = self.alu.operate(opcode, operand1, operand2)
            self.registers.write(reg1, result)

        elif opcode == "NOT":
            reg1 = int(parts[1][1:])
            reg2 = int(parts[2][1:])
            operand = self.registers.read(reg2)
            result = self.alu.operate(opcode, operand)
            self.registers.write(reg1, result)

        elif opcode == "LOAD":
            reg1 = int(parts[1][1:])
            value = int(parts[2])
            self.registers.write(reg1, value)

        elif opcode == "HALT":
            return False

        else:
```

```
            print(f"Unsupported instruction: {opcode}")
        return True

    def execute_instruction(self):
        self.control_unit.fetch()
        instruction = self.control_unit.instruction_register
        if instruction is None:
            return False
        return self.decode_and_execute(instruction)

    def load_program(self, program):
        self.control_unit.load_program(program)

    def run(self):
        self.control_unit.reset()
        while True:
            if not self.execute_instruction():
                break

# Example program
program = [
    "LOAD R1 5",      # Load 5 into R1
    "LOAD R2 10",     # Load 10 into R2
    "ADD R3 R1 R2",   # Add R1 and R2, store result in R3
```

```
    "SUB R4 R3 R2",    # Subtract R2 from R3, store result in R4
    "MUL R5 R4 R1",    # Multiply R4 and R1, store result in R5
    "NOT R6 R5",       # Perform bitwise NOT on R5, store in R6
    "HALT"             # Stop execution
]

# Initialize and run the CPU
cpu = CPU()
cpu.load_program(program)
cpu.run()

# Print register states after execution
print("Register States:")
for i in range(8):
    print(f"R{i}: {cpu.registers.read(i)}")
```

Explanation:

Objective 1: Instruction Fetching

- The `ControlUnit` fetches instructions from memory using the `fetch` method.
- The fetched instruction is stored in the `instruction_register` for decoding.

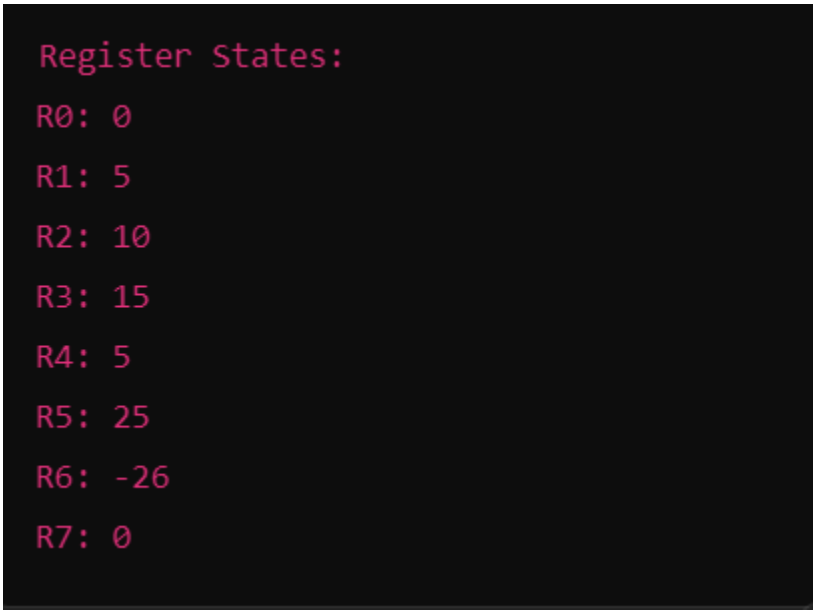
Objective 2: Decode and Execute

- Instructions are decoded in the `decode_and_execute` method by splitting the string into opcode and operands.
- Depending on the opcode, the ALU or register operations are executed.

Objective 3: Testing with Simple Programs

- A simple program is provided to test loading, arithmetic operations (`ADD`, `SUB`, `MUL`), logical operations (`NOT`), and halting the CPU.

Example Output:



This output verifies that the fetch-decode-execute cycle works as intended, correctly executing the instructions and storing the results in registers.