Instruction Fetch-Decode-Execute Cycle

1. Introduction

The fetch-decode-execute cycle is a fundamental concept in computer architecture. It is the process by which a Central Processing Unit (CPU) executes instructions in a program. This document outlines the implementation of the cycle, with detailed explanations and examples.

2. Objectives

The primary objectives of this project are as follows:

- Implement the instruction fetching mechanism.
- Decode instructions and execute them using the Arithmetic Logic Unit (ALU) and registers.
- Test the implementation with simple programs.

3. Components of the Cycle

3.1 Fetch

The fetch phase retrieves the next instruction from memory. The Program Counter (PC) holds the address of the next instruction. This address is sent to memory, and the instruction is retrieved and stored in the Instruction Register (IR).

3.2 Decode

The decode phase interprets the instruction stored in the Instruction Register. The instruction is divided into its components, such as operation code (opcode) and operands.

3.3 Execute

The execute phase performs the operation specified by the instruction. This might involve arithmetic calculations, data transfers, or logical operations, executed using the Arithmetic Logic Unit (ALU) and registers.

4. Implementation

4.1 Algorithm

The following algorithm outlines the fetch-decode-execute cycle:

1. Initialize the Program Counter (PC) to the starting address of the program.
2. Repeat until the program ends:
   a. Fetch the instruction at the address pointed to by PC.
   b. Increment the PC to point to the next instruction.
   c. Decode the fetched instruction.
   d. Execute the decoded instruction.
3. End.

4.2 Sample Code

Below is a Python implementation of the fetch-decode-execute cycle:

```
class ALU:
    """Arithmetic Logic Unit for performing operations."""
    def execute(self, operation, operand1, operand2=None):
        if operation == "ADD":
            return operand1 + operand2
        elif operation == "SUB":
            return operand1 - operand2
        elif operation == "LOAD":
            return operand1  # Simulates loading a value into a register
        elif operation == "HALT":
            return None
        else:
            raise ValueError(f"Unknown operation: {operation}")

class CPU:
    """Central Processing Unit."""
    def __init__(self, program):
        self.PC = 0  # Program Counter
        self.memory = []  # Memory for storing instructions
        self.IR = None  # Instruction Register
        self.registers = [0] * 8  # 8 General-purpose registers (R0 to R7)
        self.ALU = ALU()  # Arithmetic Logic Unit
        self.memory.extend(program)  # Load program into memory

    def fetch(self):
        """Fetch the next instruction."""
        if self.PC < len(self.memory):
            self.IR = self.memory[self.PC]
            self.PC += 1
        else:
            self.IR = None

    def decode_and_execute(self):
        """Decode and execute the fetched instruction."""
        if self.IR:
            parts = self.IR.split()  # Example: "ADD R1 R2 R3"
            opcode = parts[0]
            if opcode == "LOAD":
                reg_index = int(parts[1][1:])  # Extract register index (e.g., R1 -> 1)
                value = int(parts[2])  # Immediate value
                self.registers[reg_index] = self.ALU.execute(opcode, value)
            elif opcode == "ADD":
```

```
                dest_reg = int(parts[1][1:])
                src1 = int(parts[2][1:])
                src2 = int(parts[3][1:])
                self.registers[dest_reg] = self.ALU.execute(opcode, self.registers[src1],
self.registers[src2])
            elif opcode == "STORE":
                reg_index = int(parts[1][1:])
                print(f"Stored value from R{reg_index}: {self.registers[reg_index]}")
            elif opcode == "HALT":
                print("HALT encountered. Stopping execution.")
                return False
            else:
                print(f"Unknown instruction: {self.IR}")
            return True
        return False

    def run(self):
        """Run the fetch-decode-execute cycle."""
        while True:
            self.fetch()
            if not self.decode_and_execute():
                break
        print("Final Register State:", self.registers)

# Example Program
program = [
    "LOAD R1 10",      # Load 10 into R1
    "LOAD R2 20",      # Load 20 into R2
    "ADD R3 R1 R2",    # R3 = R1 + R2
    "STORE R3",        # Output R3's value
    "HALT"             # Stop execution
]

# Create a CPU instance and run the program
cpu = CPU(program)
cpu.run()
```

Detailed Explanation of Input and Output

Input: The Program

The input consists of a list of instructions written as strings. These instructions simulate the operation of a simple CPU.

```
program = [
    "LOAD R1 10",      # Load 10 into R1
    "LOAD R2 20",      # Load 20 into R2
    "ADD R3 R1 R2",    # R3 = R1 + R2
    "STORE R3",        # Output R3's value
    "HALT"             # Stop execution
]
```

Console Output:

```
Stored value from R3: 30
HALT encountered. Stopping execution.
```

Final Register State:

```
Final Register State: [0, 10, 20, 30, 0, 0, 0, 0]
PS C:\Users\User\Desktop\labfinalcpp>
```

`R1`: 10, `R2`: 20, `R3`: 30 (Result of `ADD`), others are unused.

Explanation

The execution proceeds step-by-step as follows:

1. **Instruction 1: `LOAD R1 10`**
   The CPU fetches this instruction, decodes it as a `LOAD` operation, and assigns the value `10` to `R1`.
2. **Instruction 2: `LOAD R2 20`**
   The CPU fetches and decodes the `LOAD` instruction, assigning the value `20` to `R2`.
3. **Instruction 3: `ADD R3 R1 R2`**
   The CPU fetches and decodes the `ADD` instruction, adding the values in `R1` and `R2` (`10 + 20 = 30`) and storing the result in `R3`.
4. **Instruction 4: `STORE R3`**
   The CPU fetches and decodes the `STORE` instruction, outputting the value of `R3` (`30`).
5. **Instruction 5: `HALT`**
   The CPU fetches and decodes the `HALT` instruction, stopping further execution.

Conclusion

The code implements a basic CPU simulator using the fetch-decode-execute cycle. It demonstrates how a CPU processes a series of instructions, including loading values into registers, performing arithmetic operations, storing results, and halting execution. By using classes like `CPU` and `ALU`, the code modularizes the components of a CPU, making it easy to understand and extend for more complex operations. This simple model highlights the core functionality of a CPU in a clear and structured manner.