Virtual CPU Emulator Documentation

Overview

This document details the design and implementation of a Virtual CPU Emulator, completed over several weeks. The CPU supports basic arithmetic and logical operations, memory management, and I/O operations. It incorporates performance optimizations, including a basic instruction pipeline and enhanced assembler functionality, simulating a simplified computer architecture.

Objectives

- Design an Instruction Set Architecture (ISA).
- Implement core CPU components like the Arithmetic Logic Unit (ALU), registers, and program counter.
- Develop the instruction fetch-decode-execute cycle.
- Set up memory management and segmentation.
- Enable basic input/output operations.
- Add advanced capabilities to the CPU emulator.
- Improve the performance of the CPU emulator.

Week 2: Instruction Set Architecture (ISA)

Objective: Design the ISA for the virtual CPU.

Tasks:

- Define basic instructions (ADD, SUB, LOAD, STORE, etc.).
- Document the instruction formats.
- Create a simple assembler to convert assembly code into machine code.

Instruction Formats:

1. `LOAD Rx, VALUE`: Load a constant value into register Rx.
2. `ADD Rx, Ry, Rz`: Add values in registers Ry and Rz, store the result in Rx.
3. `STORE Rx, ADDRESS`: Store the value in Rx into memory at ADDRESS.
4. `INPUT Rx`: Read a value from the user and store it in Rx.

5. `OUTPUT Rx`: Display the value in Rx.
6. `HALT`: Stop the program execution.

Code:

python
RunCopy

```python
def assemble(instructions):
    """Convert assembly code to machine code."""
    machine_code = []
    for instruction in instructions:
        parts = instruction.split()
        opcode = parts[0]
        if opcode in ["LOAD", "STORE", "INPUT", "OUTPUT", "ADD", "SUB", 
"JUMP", "JUMPZ", "CALL", "RET", "HALT"]:
            machine_code.append(instruction)
        else:
            raise ValueError(f"Unknown instruction: {opcode}")
    return machine_code
```

Week 3: Basic CPU Components

Objective: Implement core components of the CPU.

Tasks:

- Build the Arithmetic Logic Unit (ALU).
- Implement general-purpose registers.
- Create the program counter (PC) and instruction register (IR).

ALU Operations:

- ADD: Add two operands.
- SUB: Subtract the second operand from the first.
- LOAD: Pass a value directly.
- HALT: Stop execution.

Description: The ALU performs arithmetic operations, and registers temporarily hold data. The PC tracks the address of the next instruction to execute.

Code:

python
RunCopy

```
class ALU:
    def __init__(self):
        self.custom_operations = {}

    def execute(self, operation, operand1, operand2=None):
        if operation == "ADD":
            return bin(int(operand1, 2) + int(operand2, 2))[2:]  # Binary
addition
        elif operation == "SUB":
            return bin(int(operand1, 2) - int(operand2, 2))[2:]  # Binary
subtraction
        # Other operations...
```

Week 4: Instruction Execution

Objective: Develop the instruction fetch-decode-execute cycle.

Tasks:

- Implement the instruction fetching mechanism.
- Decode instructions and execute them using the ALU and registers.
- Test with simple programs.

Algorithm Overview:

- Fetch: Load the next instruction from memory into the Instruction Register (IR).
- Decode: Identify the operation and operands.
- Execute: Perform the operation using the ALU and update registers/memory.

Description: The fetch-decode-execute cycle allows the CPU to process instructions sequentially, altering registers and memory as needed.

Code:

python
RunCopy

```
def fetch(self):
    """Fetch the next instruction."""
    if self.PC < len(self.memory):
        self.IR = self.memory[self.PC]
```

```python
        self.PC += 1

def decode_and_execute(self):
    """Decode and execute the fetched instruction."""
    if self.IR:
        parts = self.IR.split()
        opcode = parts[0]
        # Handle different opcodes...
```

Week 5: Memory Management

Objective: Implement memory management for the virtual CPU.

Tasks:

- Set up a simulated memory space.
- Implement memory read/write operations.
- Handle address mapping and memory segmentation.

Description: Memory is divided into segments to facilitate organized data storage and retrieval.

Memory Segmentation: Memory is divided into two segments:

1.Segment 0: Base 0, Limit 512.
2. Segment 1: Base 512, Limit 512.

Code:

python
RunCopy

```python
MEMORY_SIZE = 1024
memory = ['0'] * MEMORY_SIZE  # Initialize memory with binary zeros

def write_memory(address, value):
    if 0 <= address < MEMORY_SIZE:
        memory[address] = value

def read_memory(address):
    if 0 <= address < MEMORY_SIZE:
        return memory[address]
```

Week 6: I/O Operations

Objective: Enable basic input/output operations.

Tasks:

• Implement simulated I/O devices (keyboard, display).

• Create I/O instructions and integrate them with the CPU.

• Test with I/O-intensive programs.

Description: I/O operations allow interaction with the user, enabling data input and output.

Code:

python
RunCopy
```
class IODevice:
    @staticmethod
    def read_input():
        decimal_value = int(input("Enter a value (decimal): "))  # Prompt user for input
        return bin(decimal_value)[2:]

    @staticmethod
    def display_output(value):
        print(f"Output (binary): {value}")
```

Week 7: Advanced Features

Objective: Add advanced CPU features.

Tasks:

- Implement branching and control flow with JUMP and JUMPZ instructions.
- Add support for subroutines using CALL and RET instructions.
- Introduce an interrupt handling mechanism.
- Create a simple instruction pipeline.

Description: These features enhance the emulator's functionality, allowing for more complex programs with control flow and reusable code.

Code:

python
RunCopy
```python
def decode_and_execute(self):
    # Handling JUMP and CALL instructions...
    elif opcode == "CALL":
        self.call_stack.append(self.PC)  # Push return address onto stack
        self.PC = target_address
```

Week 8: Performance Optimization

Objective: Optimize the emulator for better performance.

Tasks:

- Profile the CPU emulator to measure execution time.
- Optimize critical code paths.
- Refine the assembler for efficient machine code.

Description: Performance improvements ensure that the emulator runs efficiently and can handle more complex tasks.

Code:

python
RunCopy
```python
def profile_execution(self):
    """Profile the CPU to identify bottlenecks."""
    import time
    start_time = time.time()
    self.run()
    end_time = time.time()
    print(f"Execution Time: {end_time - start_time:.6f} seconds")
```

Week 9: Final Testing & Debugging

Objective: Thoroughly test and debug the emulator.

Tasks:

- Test with a variety of assembly programs.
- Debug and fix any issues.
- Validate performance against benchmarks.

Description: This final phase ensures that the emulator operates correctly under various conditions and meets performance standards. It involves running multiple test cases, identifying bugs, and optimizing the emulator further.

Code:

python
RunCopy

```
if __name__ == "__main__":
    assembly_code = [
        "INPUT R1",        # Read N from user
        "LOAD R2 1",       # Initialize i = 1
        "LOAD R3 0",       # Initialize sum = 0
        "LOOP:",
        "ADD R3 R3 R2",    # sum = sum + i
        "ADD R2 R2 1",     # i = i + 1
        "SUB R4 R1 R2",    # R4 = N - i
        "JUMPZ END",       # If R4 == 0, exit loop
        "JUMP LOOP",       # Otherwise, repeat loop
        "END:",
        "STORE R3 150",    # Store sum in memory at address 150
        "OUTPUT R3",       # Display the sum
        "HALT"             # Stop execution
    ]

    # Assemble and run the program
    program = assemble(assembly_code)
    cpu = CPU(program)
    cpu.profile_execution()
    print("Sum of first N natural numbers (binary):", read_memory(150))
```

How It Works:

The program repeatedly adds i to sum, increments i, and checks if i > N.

When the loop ends, the sum is stored in memory and printed in binary format.

Conclusion

The Virtual CPU Emulator successfully simulates basic CPU operations, including arithmetic, memory management, and I/O, along with advanced features such as branching, subroutines, and interrupt handling. This project demonstrates how components like the ALU, registers, and memory segmentation integrate to perform computations. The emulator serves as a foundation for more advanced simulations, including pipelining, caching, and multi-core processing.