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, 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.

Week-by-Week Progress

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.

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.

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:

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

Detailed Algorithm:

1. Initialize the Program Counter (PC) to 0.
2. Repeat until a `HALT` instruction is encountered:

    - Fetch the instruction at the address pointed to by PC.
    - Increment PC.
    - Decode the opcode and operands.
    - Execute the operation.

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.

Memory Segmentation: Memory is divided into two segments:

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

Functions:

- `read_memory(address)`: Reads a value from the specified memory address.
- `write_memory(address, value)`: Writes a value to the specified memory address.
- `read_memory_segmented(segment, offset)`: Reads a value using segment-offset addressing.
- `write_memory_segmented(segment, offset, value)`: Writes a value using segment-offset addressing.

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.

Functions:

- `IODevice.read_input()`: Prompts the user for input.
- `IODevice.display_output(value)`: Displays the output.

Sample Implementation

```
# 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 100",     # Store R3's value into memory address 100
    "INPUT R4",         # Take input and store in R4
    "OUTPUT R4",        # Display the value of R4
    "HALT"              # Stop execution
]

cpu = CPU(program)
cpu.run()
```

Sample Interaction:

```
Input: 15 (user input during INPUT R4)
Output:
Output: 15
HALT encountered. Stopping execution.
Final Register State: [0, 10, 20, 30, 15, 0, 0, 0]
```

Conclusion

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