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, advanced features, it also 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-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.

Week 7: Advanced Features

Objective: Add advanced CPU features.

Tasks:

• Implement branching and control flow with JUMP and JUMPZ instructions to allow conditional and unconditional program flow.

• Add support for subroutines using CALL and RET instructions to manage reusable code blocks and nested calls with a call stack.

• Introduce an interrupt handling mechanism to manage special tasks or events.

• Create a simple instruction pipeline to improve CPU efficiency by overlapping fetch, decode, and execute stages.

Week 8: Performance Optimization

Objective: Optimize the emulator for better performance.

Tasks:

• Profile the CPU emulator to measure execution time and identify performance bottlenecks.

• Optimize critical code paths, such as arithmetic operations, memory access, and branching logic, to enhance efficiency.

• Refine the assembler to produce compact and efficient machine code by optimizing instruction encoding.

A sample program to run on this emulator:

This program calculates the sum of the first `N` natural numbers using a loop and stores the result in memory.

Program Explanation

1. Take an input `N` from the user.
2. Initialize a loop counter (`i = 1`) and a sum register (`sum = 0`).
3. Use a loop to repeatedly add `i` to `sum` until `i > N`.
4. Store the final sum in memory at address `150`.
5. Output the result.

Assembly Code for the Sum of First N Natural Numbers:

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   # Check if i > N (R4 = N - i)

JUMPZ END   # If R4 == 0, exit loop

JUMP LOOP   # Otherwise, continue looping

END:

STORE R3 150   # Store sum in memory at address 150

OUTPUT R3   # Display the result

HALT   # Stop execution

Python Code to Run This Program:

```
if __name__ == "__main__":
    assembly_code = [
        "INPUT R1",      # Read N from user
        "LOAD R2 1",     # Initialize i = 1
        "LOAD R3 0",     # Initialize sum = 0

        "ADD R3 R3 R2", # sum = sum + i
        "ADD R2 R2 1",  # i = i + 1
        "SUB R4 R1 R2", # R4 = N - i
        "JUMPZ 8",      # If R4 == 0, exit loop
        "JUMP 3",       # Otherwise, repeat loop

        "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 the sum stored in memory
    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 & advanced features such as branching, subroutines, and interrupt handling . 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.