Project Name: Build a Virtual CPU Emulator

1. Defining Basic Instructions

We’ve started with essential instructions like `ADD`, `SUB`, `LOAD`, and `STORE`. Here’s an example of how we’re structuring the instructions:

Instruction	Opcode	Operands	Description
ADD	0x01	Reg1, Reg2, Reg3	Adds Reg2 and Reg3, stores in Reg1
SUB	0x02	Reg1, Reg2, Reg3	Subtracts Reg3 from Reg2, stores in Reg1
LOAD	0x03	Reg, Address	Loads value from Address into Reg
STORE	0x04	Address, Reg	Stores value from Reg into Address

These opcodes and formats will guide the assembler in translating assembly into machine code.

2. Documenting Instruction Formats

For each instruction, we’re using a format with a **4-byte (32-bit) layout**:

- **1st byte**: Opcode
- **2nd byte**: Destination Register
- **3rd byte**: Source Register 1
- **4th byte**: Source Register 2 or Address (for memory operations)

An example for the `ADD` instruction:

```
ADD R1, R2, R3  ; R1 = R2 + R3
```

In memory, this could translate to:

```
0x01 0x01 0x02 0x03
```

This format allows the virtual CPU to read, decode, and execute each instruction correctly.

3. Creating a Simple Assembler in Python

Our assembler reads assembly code, translates it to machine code, and outputs it in a format the CPU can execute.

Here's a basic Python script for the assembler:

```python
# Define opcodes for instructions
OPCODES = {
    'ADD': 0x01,
    'SUB': 0x02,
    'LOAD': 0x03,
    'STORE': 0x04
}

# Function to assemble an instruction
def assemble_instruction(instruction):
    parts = instruction.split()
    opcode = OPCODES.get(parts[0].upper())
    if opcode is None:
        raise ValueError(f"Unknown instruction: {parts[0]}")

    # Encode the instruction with opcode and register values
    if parts[0].upper() in ['ADD', 'SUB']:
        # Example format: ADD R1, R2, R3 -> 0x01 0x01 0x02 0x03
        dest = int(parts[1][1])  # Destination register, e.g., R1 -> 1
        src1 = int(parts[2][1])
        src2 = int(parts[3][1])
        return f"{opcode:02x} {dest:02x} {src1:02x} {src2:02x}"

    elif parts[0].upper() == 'LOAD':
        reg = int(parts[1][1])  # Convert register
        address = int(parts[2].strip(',').strip(), 16) if
parts[2].startswith('0x') else int(parts[2].strip(',').strip())  # Convert
address
        return f"{opcode:02x} {reg:02x} {address:04x}"

    elif parts[0].upper() == 'STORE':
        address = int(parts[1].strip(',').strip(), 16) if
parts[1].startswith('0x') else int(parts[1].strip(',').strip())  # Convert
address
        reg = int(parts[2][1])  # Convert register
        return f"{opcode:02x} {address:04x} {reg:02x}"

    else:
        raise ValueError("Unsupported instruction format.")

# Sample assembly code
assembly_code = [
    "ADD R1, R2, R3",
    "LOAD R1, 0x10",
    "STORE 0x10, R1"
]

# Convert each instruction to machine code
machine_code = [assemble_instruction(instr) for instr in assembly_code]
print("Machine Code:")
print("\n".join(machine_code))
```

Example Output:

For the assembly instructions provided:

```
ADD R1, R2, R3
LOAD R1, 0x10
STORE 0x10, R1
```

The assembler would output:

```
Machine Code:
01 01 02 03
03 01 0010
04 0010 01
```

This setup provides a foundation for executing simple instructions and testing our virtual CPU’s capabilities as we proceed through the project. This week’s tasks are essential to understanding how the CPU processes instructions and manages memory, setting the stage for later developments.