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.

This code is a Python program designed to convert a simple set of assembly language instructions into machine code. The program defines several opcodes for specific instructions, then parses and converts each instruction to a hexadecimal machine code format. Here’s an explanation of each part:

i. Opcode Dictionary

```
OPCODES = {
    'ADD': 0x01,
    'SUB': 0x02,
    'LOAD': 0x03,
    'STORE': 0x04
}
```

- This dictionary defines opcode values for each instruction in hexadecimal format.
- `ADD`, `SUB`, `LOAD`, and `STORE` instructions are assigned specific opcode values (e.g., `ADD` is `0x01`).
- These opcodes are used as the first part of each machine code instruction.

ii. `assemble_instruction()` Function

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

- This function takes a single assembly instruction (e.g., `"ADD R1, R2, R3"`) as input.
- It splits the instruction into individual parts (e.g., `["ADD", "R1,", "R2,", "R3"]`).
- It looks up the opcode from the `OPCODES` dictionary. If the instruction is not found in the dictionary, it raises an error.

Encoding Specific Instructions:

- **`ADD` and `SUB` Instructions**:

```
if parts[0].upper() in ['ADD', 'SUB']:
    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}"
```

  - For `ADD` and `SUB` instructions, the format is assumed to be: `OPCODE DEST SRC1 SRC2`.
  - Each register part (e.g., `R1`, `R2`) is converted to an integer representing the register number (e.g., `R1` becomes `1`).
  - The result is formatted in hexadecimal and returned as a string, e.g., `"0x01 0x01 0x02 0x03"` for `"ADD R1, R2, R3"`.

- `LOAD` **Instruction**:

```
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())
    return f"{opcode:02x} {reg:02x} {address:04x}"
```

  - For `LOAD` instructions, the format is: `OPCODE REG ADDRESS`.
  - The register number is extracted, and the address is converted to an integer.
  - If the address is in hexadecimal format (e.g., `0x10`), it's converted to an integer with base `16`.
  - The result is returned in hexadecimal format.

- `STORE` **Instruction**:

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

  - For `STORE` instructions, the format is `OPCODE ADDRESS REG`.
  - Similar to `LOAD`, it extracts the address and register values.
  - The result is returned in hexadecimal format.

- **Error Handling**:

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

  - If an unsupported instruction is encountered, it raises an error.

3. Sample Assembly Code

```
assembly_code = [
    "ADD R1, R2, R3",
    "LOAD R1, 0x10",
    "STORE 0x10, R1"
]
```

- This list contains a sample set of assembly instructions to be converted to machine code.

4. Conversion to Machine Code

```
machine_code = [assemble_instruction(instr) for instr in assembly_code]
print("Machine Code:")
print("\n".join(machine_code))
```

- Each instruction in `assembly_code` is passed through the `assemble_instruction()` function.
- The machine code for each instruction is printed as a list of strings.

Example Output

For the sample assembly code:

1. `"ADD R1, R2, R3"` → `0x01 0x01 0x02 0x03`
2. `"LOAD R1, 0x10"` → `0x03 0x01 0x0010`
3. `"STORE 0x10, R1"` → `0x04 0x0010 0x01`

The output would look like:

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

Summary

This program translates assembly language instructions into hexadecimal machine code by parsing instruction components, mapping them to opcodes, and formatting them into a specific structure. The program currently supports `ADD`, `SUB`, `LOAD`, and `STORE` instructions and provides error handling for unknown instructions.