week 2: Instruction Set Architecture (ISA) Design

Objective: Design the Instruction Set Architecture (ISA) for the virtual CPU in a clear, structured manner.

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 that the CPU can execute. The code provided is designed to convert a simple set of assembly language instructions into machine code. It defines opcodes for specific instructions, then parses and converts each instruction into a hexadecimal machine code format.

Objective: Develop a Python-based assembler that converts assembly code into machine code.

- **Input**: Assembly source file (e.g., `program.asm`)
- **Output**: Machine code file (e.g., `program.bin`)

Assembler Workflow:

- **Lexical Analysis**: Read and tokenize the assembly code into opcode and operand tokens.
- **Parsing**: Validate the syntax and convert the instructions to their binary format.
- **Output Generation**: Write the resulting binary code into a machine code file.

Example Python Code for the Assembler:

```
OPCODE_MAP = {'ADD': '0001', 'SUB': '0010', 'LOAD': '0011', 'STORE': '0100', 'JMP': '0101', 'CMP':
'0110'}

DEF ASSEMBLE_INSTRUCTION(INSTRUCTION):

  TOKENS = INSTRUCTION.SPLIT()

  OPCODE = OPCODE_MAP.GET(TOKENS[0])

  OPERANDS = TOKENS[1].REPLACE(',', '').SPLIT()
```

```
    BINARY_INSTR = OPCODE + FORMAT(INT(OPERANDS[0][1:]), '04B') + FORMAT(INT(OPERANDS[1][1:]), '04B')
+ FORMAT(INT(OPERANDS[2][1:]), '04B')

    RETURN BINARY_INSTR

WITH OPEN('PROGRAM.ASM', 'R') AS ASM_FILE, OPEN('PROGRAM.BIN', 'W') AS BIN_FILE:

    FOR LINE IN ASM_FILE:

        MACHINE_CODE = ASSEMBLE_INSTRUCTION(LINE.STRIP())

        BIN_FILE.WRITE(MACHINE_CODE + '\N')
```

Explanation of the Code:

- **`opcode_map`**: A dictionary mapping assembly instruction mnemonics to their corresponding binary opcodes.
- **`assemble_instruction()`**: A function that splits each instruction into its components, looks up the opcode, and formats the operands into a 32-bit binary instruction.
- **File I/O**: Reads the assembly source file line by line, converts each line to machine code, and writes it to an output file.

Summary:

We designed the ISA for our virtual CPU, defining key instructions like `ADD`, `SUB`, `LOAD`, and `STORE`, with corresponding opcodes and operand formats. We documented the 4-byte instruction layout and developed a Python-based assembler to convert assembly code into machine code, ensuring accurate translation for CPU execution.