Instruction Set Architecture Design for a Virtual CPU

The following is the detailed design of an Instruction Set Architecture or ISA for a simple virtual CPU. We will be discussing the basic instructions, instruction formats, and a basic assembler.

1. Basic Instructions

There are basic instructions in ISA and they are grouped into the following:

1.1. Data Transfer Instructions

- LOAD R, MEM: Load data from memory location MEM into register R.
- STORE R, MEM: The data stored in the register R are copied to memory location MEM.

1.2. Arithmetic Instructions

- ADD R1, R2, R3: Add contents of R2 and R3, and store the result in R1.
- SUB R1, R2, R3: Subtract contents of R3 from R2, and store the result in R1.
- MUL R1, R2, R3: Multiply contents of R2 and R3, and store the result in R1.
- DIV R1, R2, R3: This instruction will divide the contents of R2 by R3, and store the quotient in R1.

1.3. Logical Instructions

- AND R1, R2, R3: This instruction performs a bitwise AND operation between R2 and R3, and the result is kept in R1.
- OR R1, R2, R3: This instruction performs a bitwise OR operation between R2 and R3, and the result is kept in R1.
- XOR R1, R2, R3: Perform a bitwise XOR between R2 and R3, storing the result in R1.

- NOT R1, R2: Perform a bitwise NOT on R2, storing the result in R1.

1.4. Control Flow Instructions

- JMP ADDR: Jump to the memory address.
- BEQ R1, R2, ADDR: Branch to ADDR if the contents of R1 and R2 are equal.
- BNE R1, R2, ADDR: Branch to ADDR if the contents of R1 and R2 are not equal.

1.5. Immediate Instructions

- ADDI R1, R2, IMM: Add the immediate value IMM to the contents of R2, and store the result in R1.
- LOADI R, IMM: Load the immediate value IMM into register R.

2. Instruction Formats

The virtual CPU utilizes fixed-width 16-bit instructions that differ in their layout depending on the type of instruction being used .

2.1. R-Type (Register Instructions)

- Purpose: Arithmetic and Logical Operations.
- Format:

OPCODE (4 bits) | DEST (4 bits) | SRC1 (4 bits) | SRC2 (4 bits)

Example: ADD R1, R2, R3

0001 | 0001 | 0010 | 0011

2.2. I-Type (Immediate Instructions)

• Purpose: Operations with Immediate Values.

• Format:

OPCODE (4 bits) | DEST (4 bits) | SRC (4 bits) | IMM (4 bits)

Example: ADDI R1, R2, 5

0010 | 0001 | 0010 | 0101

2.3. J-Type (Jump Instructions)

• Purpose: Control flow.

• Format:

OPCODE 4 bits | ADDR 12 bits

Example: JMP 0x3F

0111 | 0000 0011 1111

2.4. M-Type (Memory Instructions)

- Purpose: Load and store operations.
- Format:

OPCODE 4 bits | REG 4 bits | ADDR 8 bits

Example: LOAD R1, 0x10

1000 | 0001 | 0001 0000

3. Simple Assembler

3.1. Assembly to Machine Code Mapping

We'll create a Python-based assembler that maps assembly code to machine code. Here's an example implementation:

```
# 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

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