# Project Name : Build a Virtual CPU emulator

**Group name : Nazia Neha**

**ID: 1039**

**Naimul Islam**

**ID: 981**

**Al.Mahim Saikot**

**ID:960**

**Submitted To :**

**Vashkar Kar**

Lecturer,Computer Science Department

Northern University Business & Technology,Khulna

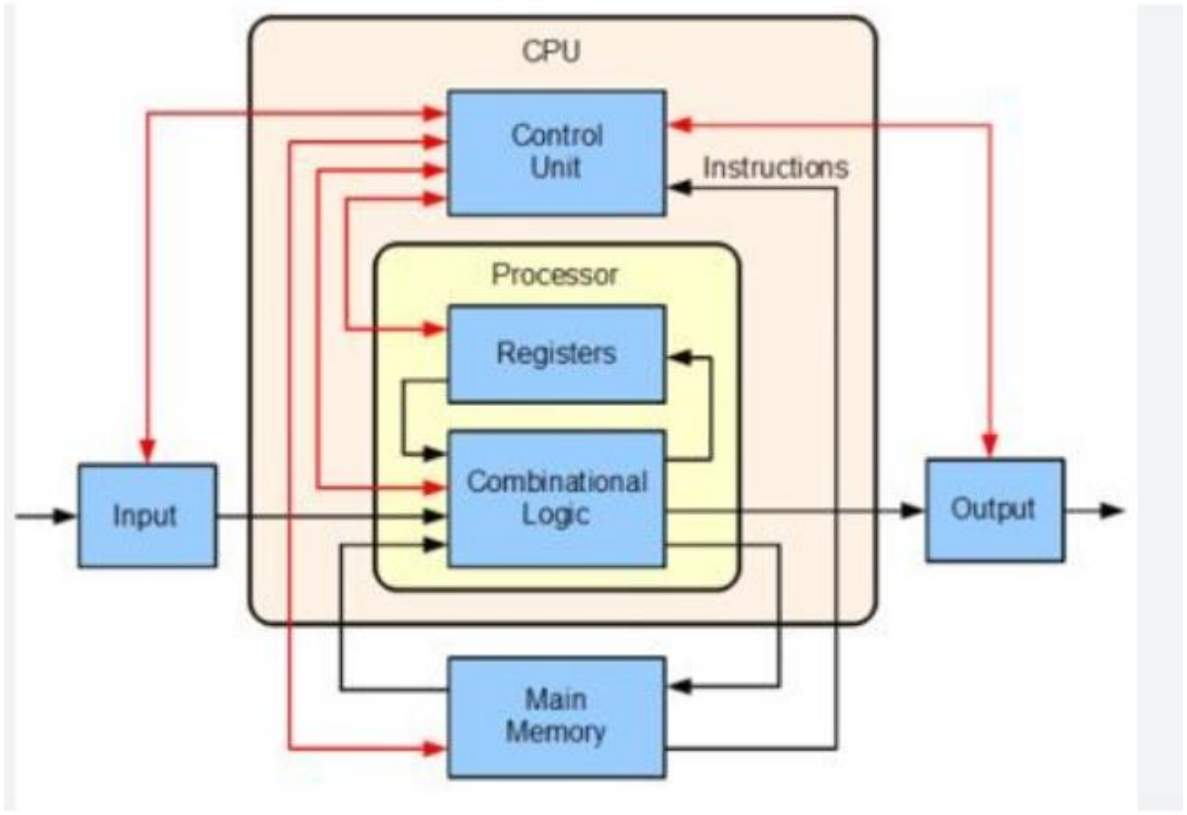

**Fig : The ISA Design for the Virtual CPU**

In the context of a virtual CPU emulator project, the basic instruments of the Instruction Set Architecture (ISA) include the following core instructions:

1. Arithmetic Instructions:

Perform basic mathematical operations:

ADD R1, R2: Adds the values in registers R1 and R2 and stores the result in R1.

SUB R1, R2: Subtracts the value in R2 from R1 and stores the result in R1.

MUL R1, R2: Multiplies the values in R1 and R2.

DIV R1, R2: Divides the value in R1 by R2.

2. Data Transfer Instructions:

Handle moving data between memory and registers:

LOAD R1, [0x10]: Loads the value from memory address 0x10 into register R1.

STORE R1, [0x20]: Stores the value from R1 into memory address 0x20.

3. Logical Instructions:

Perform bitwise operations:

AND R1, R2: Performs a bitwise AND between R1 and R2.

OR R1, R2: Performs a bitwise OR between R1 and R2

# Instruction Format for ISA - Virtual CPU Emulator

**General Structure**

Each instruction consists of the following fields:

1. Opcode (8 bits): Specifies the operation (e.g., ADD, LOAD).

2. Operands:

Register/Immediate values or Memory addresses.

Up to 3 operands depending on the instruction type.

**Format Example (Fixed 32-bit Instruction):**

| Opcode (8 bits) | Operand1 (8 bits) | Operand2 (8 bits) | Operand3 (8 bits) |

**Instruction Categories**

1. Arithmetic/Logic Instructions:

ADD R1, R2, R3 → Add R2 and R3, store in R1.

SUB R1, R2, R3 → Subtract R3 from R2, store in R1.

2. Data Movement Instructions:

LOAD R1, [ADDR] → Load memory content at ADDR into R1.

STORE R1, [ADDR] → Store R1 content into memory at ADDR.

**Addressing Modes**

1. Immediate: Operand is a constant value.
2. Register: Operand is a register value.
3. Direct: Operand is a memory address.
4. Indirect: Operand address is in a register.

**Example Encoding**

ADD R1, R2, R3:

Binary: 0001 0001 0010 0011

- Opcode: 0001 (ADD)
- Operand1: 0001 (R1)
- Operand2: 0010 (R2)
- Operand3: 0011 (R3)

Here is the C++ code for a simple assembler that converts assembly code into machine code:

```cpp
#include <iostream>

#include <sstream>

#include <unordered_map>

#include <bitset>


// Define opcode and register maps

std::unordered_map<std::string, std::string> opcode_map = {

    {"ADD", "0001"},

    {"SUB", "0010"},

    {"LOAD", "0011"},

    {"STORE", "0100"},

    {"HALT", "1111"}

};


std::unordered_map<std::string, std::string> register_map = {

    {"R0", "000"},

    {"R1", "001"},

    {"R2", "010"},

    {"R3", "011"},

    {"R4", "100"},

    {"R5", "101"},
    {"R6", "110"},
    {"R7", "111"}
};

std::string assemble(const std::string& instruction) {
    std::istringstream iss(instruction);
    std::string opcode, operand1, operand2, operand3;
    iss >> opcode;

    if (opcode == "HALT") {
        return opcode_map[opcode] + "00000000";
       // HALT instruction (no operands)
    }

    iss >> operand1 >> operand2;
    if (opcode == "LOAD" || opcode == "STORE") {
        // For LOAD/STORE instructions, the second operand is a memory
address
        iss >> operand3;
        return opcode_map[opcode] + register_map[operand1] + "0000" +
std::bitset<4>(std::stoi(operand3, nullptr, 16)).to_string();
    }
    iss >> operand3;
 // Get the third operand for ADD, SUB, etc.
 return opcode_map[opcode] + register_map[operand1] +
register_map[operand2] + register_map[operand3];
}

int main() {
    // Sample assembly code
    std::string assembly_code[] = {
        "ADD R1 R2 R3",        // ADD R1, R2, R3
        "SUB R1 R4 R5",        // SUB R1, R4, R5
        "LOAD R1 0x1000",      // LOAD R1, [0x1000]
        "STORE R2 0x1004",     // STORE R2, [0x1004]
        "HALT"                 // HALT
    };

    for (const std::string& line : assembly_code) {
     std::cout << "Assembly: " << line << " --> Machine Code: " <<
assemble(line) << std::endl;
    }

    return 0;
}
```