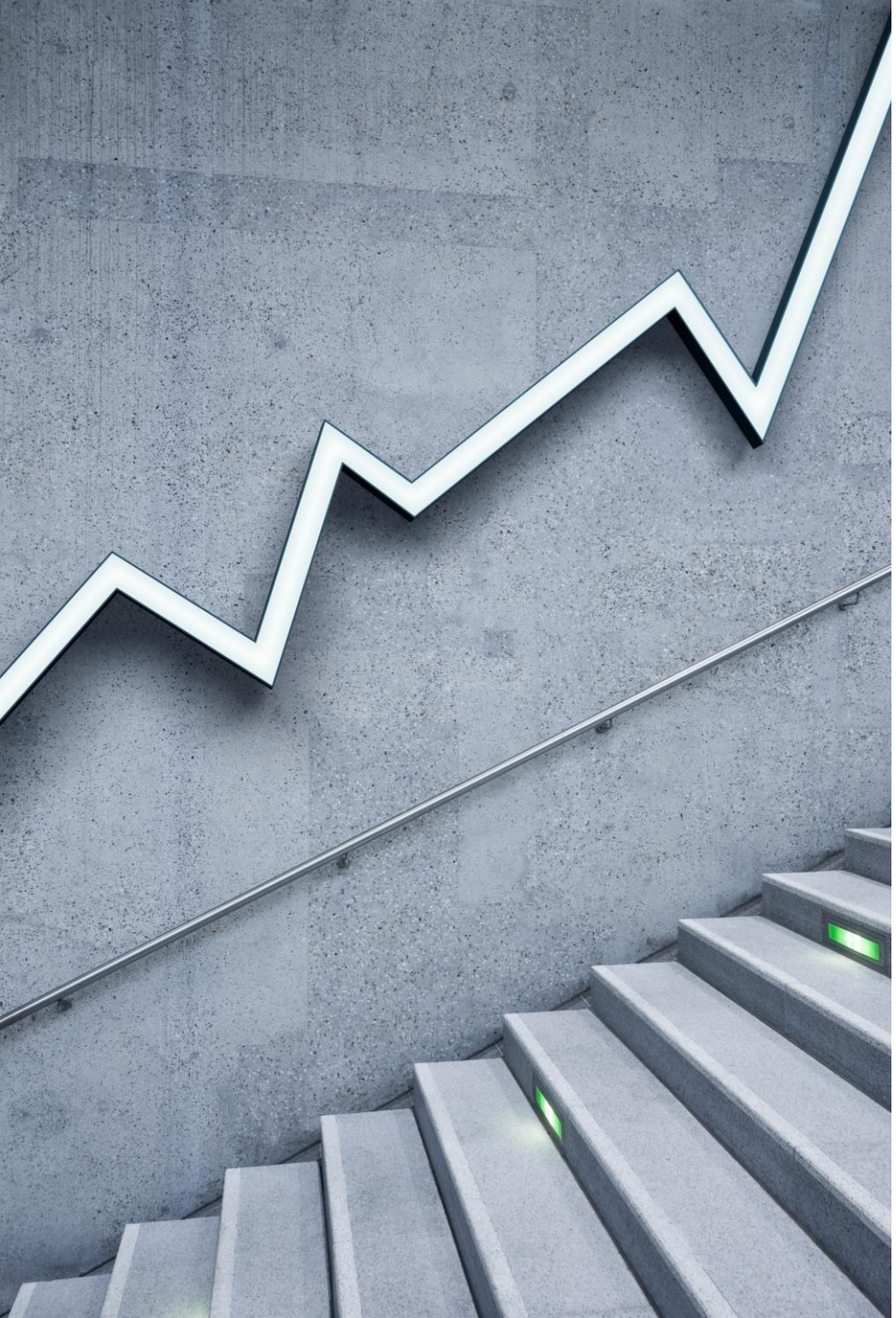


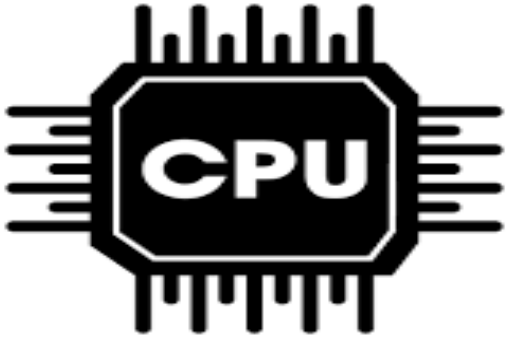
welcome

Virtual CPU Emulator

Presented by: Tisha Chakroborty, Farhana Rahaman Adiba , Arpita Biswas
ID: 11220320954, 11220320976, 11220320978
Department: Computer Science and Engineering
University: Northern University of Business & Technology, Khulna

Virtual CPU Implementation

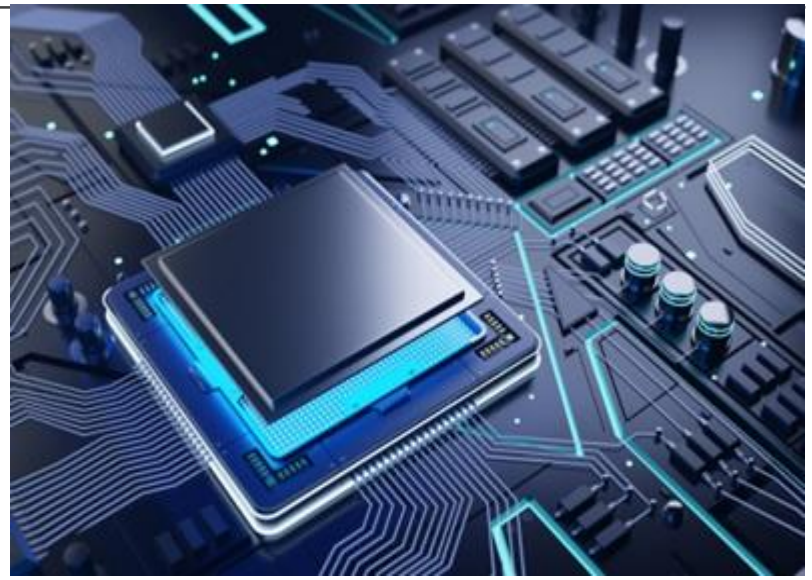
This presentation details the implementation of a virtual CPU in C++, covering key aspects such as registers, memory, instruction execution, and debugging features. The CPU is designed to execute a simple instruction set, demonstrating fundamental computer architecture principles.



Core Virtual CPU Components

- **Basic CPU Components:** Implement ALU, registers, program counter, and instruction register.
- **Memory Management:** Set up simulated memory, implement read/write operations, and manage address mapping.
- **Advanced Features:** Add branching, subroutines, interrupts, and a simple pipeline.

Virtual CPU Features



Registers

16 general-purpose registers

Memory

256-byte memory space

Operations

LOAD, STORE, ADD, SUB, MULT, JUMP, CALL, RETURN, INPUT, OUTPUT

Stack Management

Supports function calls

I/O, Optimization & Testing

- **I/O Operations:** Implement keyboard & display, integrate I/O instructions, test with I/O programs.
- **Performance Optimization:** Profile bottlenecks, optimize code,improve assembler encoding.
- **Final Testing & Debugging:** Test with assembly programs, fix bugs, validate performance.

vertual.cpp Settings

vertual.cpp

```cpp
#include <iostream>                         // Include the input-output stream library
#include <vector>                           // Include the vector library
#include <bitset>                           // Include the bitset library
#include <map>                              // Include the map library
#include <fstream>                          // Include the file stream library
#include <chrono>                           // Include the chrono library for time handling
using namespace std;                        // Use the standard namespace

class CPU {                                 // Define a class named CPU
private:                                    // Access specifier private
    vector<bitset<8>> registers;            // Vector of bitsets to represent 8-bit registers
    map<int, vector<string>> memory;        // Map to represent memory with integer addresses and string instructions
    int pc;                                 // Program counter
    bool running;                           // Flag to indicate if the CPU is running
    ofstream logFile;                       // Output file stream to log CPU activities

public:                                     // Access specifier public
    CPU() : registers(8), pc(0), running(true) // Constructor to initialize registers, program counter and running flag
    {                                       // Constructor body
        logFile.open("cpu_log.txt");        // Open the log file
    }                                       // End of constructor

    ~CPU() {                                // Destructor
        if (logFile.is_open()) {            // Check if the log file is open
            logFile.close();                // Close the log file
        }                                   // End of if statement
    }                                       // End of destructor

    void loadProgram(const vector<vector<string>>& program) { // Function to load the program into memory
        for (size_t i = 0; i < program.size(); i++) { // Iterate over the program
            memory[i] = program[i];         // Load each instruction into memory
        }                                   // End of for loop
    }                                       // End of loadProgram function

    void execute() {                        // Function to execute the program
        auto start_time = chrono::high_resolution_clock::now(); // Record the start time
        while (running && memory.find(pc) != memory.end()) { // While running and program counter in memory
```

vertual.cpp

```
17    public:                                    // Access specifier public
35        void execute() {                       // Function to execute the program
37            while (running && memory.find(pc) != memory.end()) { // While running and program counter in memory
38                vector<string> instruction = memory[pc++]; // Fetch the current instruction and increment the program counter
39                processInstruction(instruction);  // Process the fetched instruction
40            }                                  // End of while loop
41            auto end_time = chrono::high_resolution_clock::now(); // Record the end time
42            chrono::duration<double> execution_time = end_time - start_time; // Calculate execution time
43            logFile << "Execution Time: " << execution_time.count() << " seconds" << endl; // Log execution time
44            cout << "Execution Time: " << execution_time.count() << " seconds" << endl; // Print execution time
45        }                                      // End of execute function
46
47        void processInstruction(const vector<string>& instr) { // Function to process each instruction
48            string opcode = instr[0];          // Get the opcode from the instruction
49
50            if (opcode == "INPUT") {           // If opcode is INPUT
51                int reg = stoi(instr[1]);      // Get the register index
52                string binary_value;           // Define a string to hold the binary value
53                cout << "Enter binary value (8-bit): "; // Prompt user for input
54                cin >> binary_value;           // Get the binary value from the user
55                registers[reg] = bitset<8>(binary_value); // Store the binary value in the register
56                displayBinary("INPUT", reg);   // Display the binary value
57            }
58            else if (opcode == "OUTPUT") {     // If opcode is OUTPUT
59                int reg = stoi(instr[1]);      // Get the register index
60                cout << "OUTPUT R" << reg << " = " << registers[reg] << endl; // Print the register value
61                logFile << "OUTPUT R" << reg << " = " << registers[reg] << endl; // Log the register value
62            }
63            else if (opcode == "ADD") {        // If opcode is ADD
64                int r1 = stoi(instr[1]), r2 = stoi(instr[2]), r3 = stoi(instr[3]); // Get the register indices
65                registers[r1] = bitset<8>(registers[r2].to_ulong() + registers[r3].to_ulong()); // Perform addition
66                displayBinary("ADD", r1);      // Display the result
67            }
68            else if (opcode == "SUB") {        // If opcode is SUB
69                int r1 = stoi(instr[1]), r2 = stoi(instr[2]), r3 = stoi(instr[3]); // Get the register indices
70                registers[r1] = bitset<8>(registers[r2].to_ulong() - registers[r3].to_ulong()); // Perform subtraction
71                displayBinary("SUB", r1);      // Display the result
```

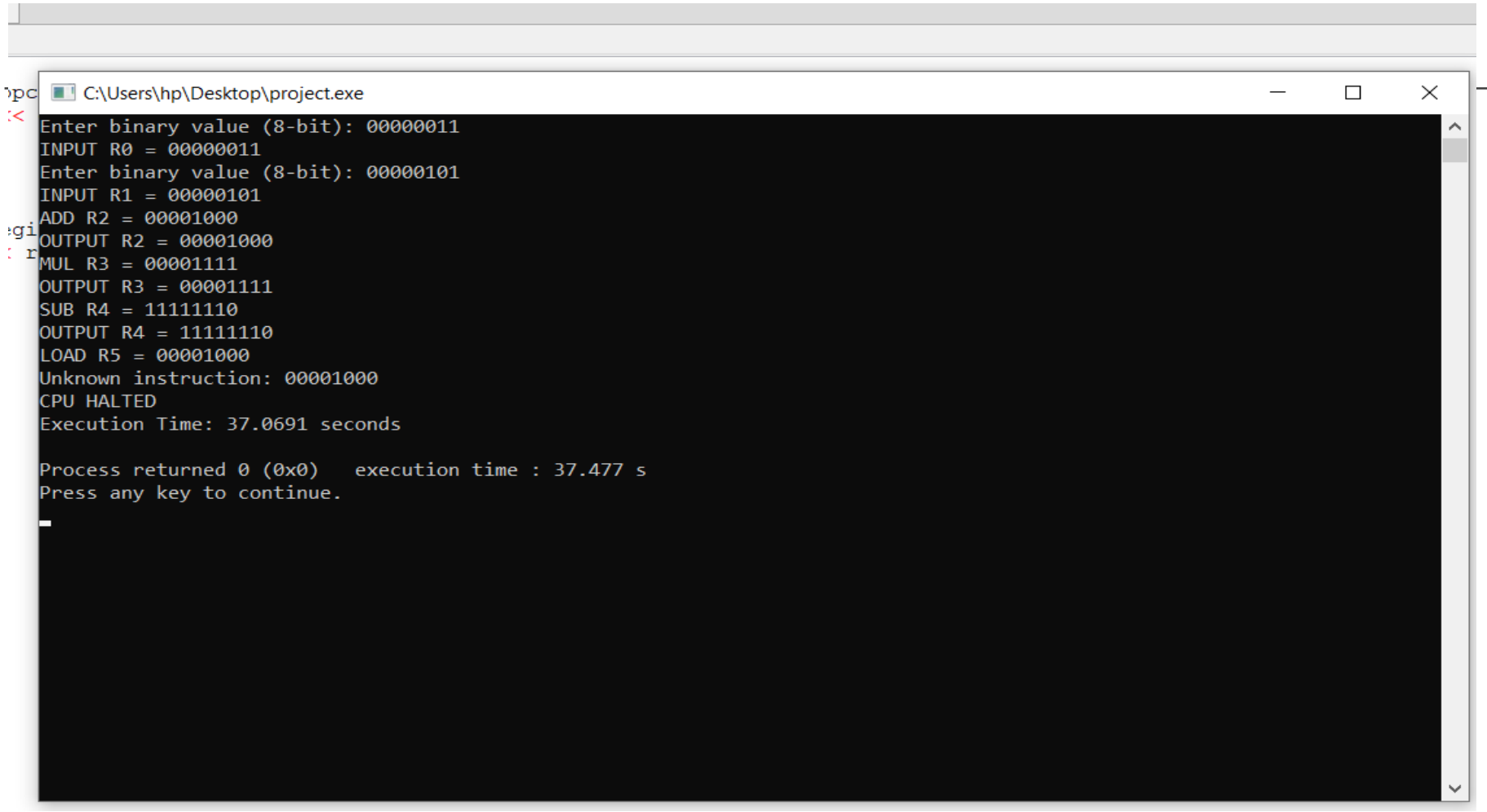
```
    }
    else if (opcode == "MUL") {          // If opcode is MUL
        int r1 = stoi(instr[1]), r2 = stoi(instr[2]), r3 = stoi(instr[3]); // Get the register indices
        registers[r1] = bitset<8>(registers[r2].to_ulong() * registers[r3].to_ulong()); // Perform multiplication
        displayBinary("MUL", r1);        // Display the result
    }
    else if (opcode == "JUMP") {         // If opcode is JUMP
        int address = stoi(instr[1]);    // Get the address
        pc = address;                    // Set the program counter to the address
    }
    else if (opcode == "BEQ") {          // If opcode is BEQ
        int r1 = stoi(instr[1]), r2 = stoi(instr[2]), address = stoi(instr[3]); // Get the register indices and address
        if (registers[r1] == registers[r2]) { // If the values in the registers are equal
            pc = address;                // Set the program counter to the address
        }
    }
    else if (opcode == "LOAD") {         // If opcode is LOAD
        int r1 = stoi(instr[1]), address = stoi(instr[2]); // Get the register index and address
        registers[r1] = bitset<8>(memory[address][0]); // Load the value from memory into the register
        displayBinary("LOAD", r1);       // Display the loaded value
    }
    else if (opcode == "STORE") {        // If opcode is STORE
        int r1 = stoi(instr[1]), address = stoi(instr[2]); // Get the register index and address
        memory[address] = {registers[r1].to_string()}; // Store the value from the register into memory
    }
    else if (opcode == "HALT") {         // If opcode is HALT
        running = false;                 // Set running to false to halt the CPU
        cout << "CPU HALTED\n";          // Print CPU halted message
        logFile << "CPU HALTED\n";       // Log CPU halted message
    }
    else {                               // If opcode is unknown
        cout << "Unknown instruction: " << opcode << endl; // Print unknown instruction message
        logFile << "Unknown instruction: " << opcode << endl; // Log unknown instruction message
    }
```

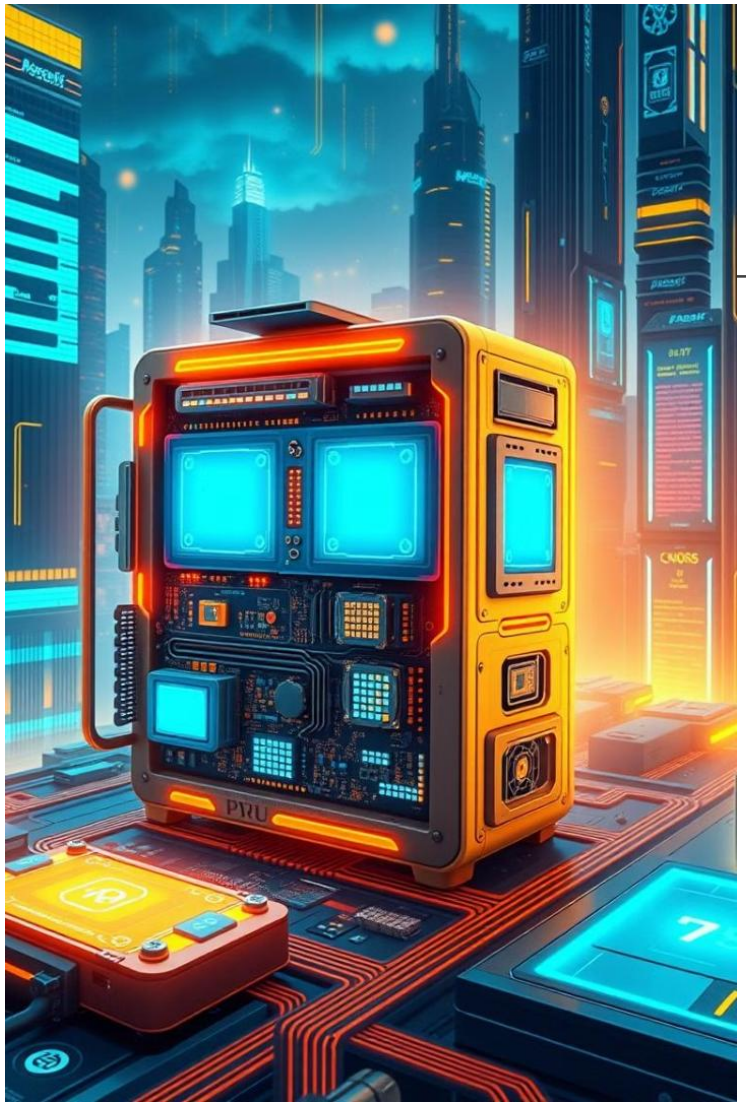
```
public:                                 // Access specifier public
    void processInstruction(const vector<string>& instr) { // Function to process each instruction
        else {                          // If opcode is unknown
            cout << "Unknown instruction: " << opcode << endl; // Print unknown instruction message
            logFile << "Unknown instruction: " << opcode << endl; // Log unknown instruction message
        }
    }                                   // End of processInstruction function

    void displayBinary(string op, int reg) {  // Function to display binary value
        cout << op << " R" << reg << " = " << registers[reg] << endl; // Print the binary value
        logFile << op << " R" << reg << " = " << registers[reg] << endl; // Log the binary value
    }
};                                      // End of class CPU

int main() {                            // Main function
    CPU cpu;                            // Create an instance of CPU
    vector<vector<string>> program = {  // Define a program with a list of instructions
        {"INPUT", "0"},                 // Instruction to input value into register 0
        {"INPUT", "1"},                 // Instruction to input value into register 1
        {"ADD", "2", "0", "1"},         // Instruction to add values in registers 0 and 1, store in register 2
        {"OUTPUT", "2"},                // Instruction to output value in register 2
        {"MUL", "3", "0", "1"},         // Instruction to multiply values in registers 0 and 1, store in register 3
        {"OUTPUT", "3"},                // Instruction to output value in register 3
        {"SUB", "4", "0", "1"},         // Instruction to subtract value in register 1 from value in register 0, store in register 4
        {"OUTPUT", "4"},                // Instruction to output value in register 4
        {"STORE", "2", "10"},           // Instruction to store value in register 2 to memory address 10
        {"LOAD", "5", "10"},            // Instruction to load value from memory address 10 into register 5
        {"OUTPUT", "5"},                // Instruction to output value in register 5
        {"JUMP", "14"},                 // Instruction to jump to address 14
        {"INPUT", "6"},                 // Instruction to input value into register 6
        {"BEQ", "6", "0", "16"},        // Instruction to branch if values in registers 6 and 0 are equal
        {"HALT"}                        // Instruction to halt the CPU
```

OUTPUT:


C:\Users\hp\Desktop\project.exe
Enter binary value (8-bit): 00000011
INPUT R0 = 00000011
Enter binary value (8-bit): 00000101
INPUT R1 = 00000101
ADD R2 = 00001000
OUTPUT R2 = 00001000
MUL R3 = 00001111
OUTPUT R3 = 00001111
SUB R4 = 11111110
OUTPUT R4 = 11111110
LOAD R5 = 00001000
Unknown instruction: 00001000
CPU HALTED
Execution Time: 37.0691 seconds
Process returned 0 (0x0) execution time : 37.477 s
Press any key to continue.



Future Scope

Enhanced ISA: Add complex instructions & floating-point support.

Multithreading:
Implement parallel execution for better performance.

Virtualization Support:
Enable multi-CPU emulation

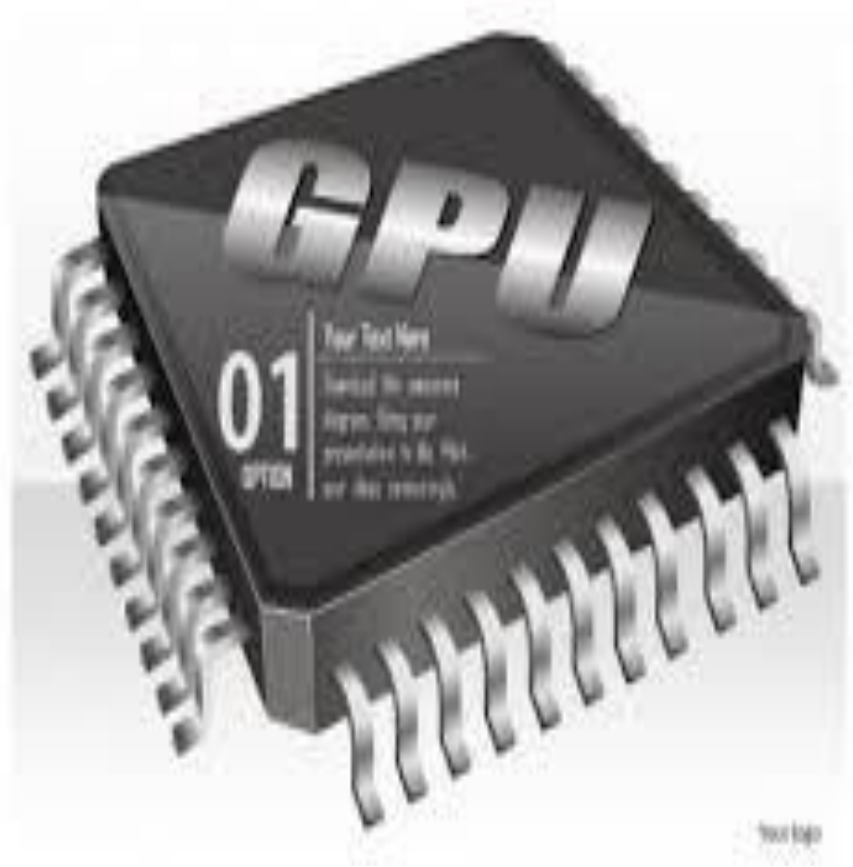
GUI : Develop a visual interface for debugging & execution.

Web-Based: Online testing.

Conclusion

The Virtual CPU project successfully emulates a processor with a custom ISA, memory management, and I/O operations. With advanced features like branching, interrupts, and pipelining, it serves as a foundation for exploring low-level computing. Future enhancements can further improve performance, functionality, and usability.

Graphic Processor Unit



Thank You