CPU Instruction Set (ISA) Simulator Report

Introduction

This CPU ISA simulator is a Java program designed to emulate the functionality of a simple CPU (Central Processing Unit). It provides a command-line interface for users to input and execute instructions. This report provides an overview of the CPU simulator, its functionalities, limitations, and possible improvements.

Functionality

The CPU simulator includes the following key features:

1. Register Management

- The CPU simulator includes eight general-purpose registers, labeled **r0** to **r7**, each capable of storing integer values.

2. Instruction Execution

- Users can input instructions using a command-line interface.
- Supported instructions include:
    - **mov**: Move an integer value or the contents of one register to another.
    - **add**: Add two values (either registers or constants) and store the result in a register.
    - **sub**: Subtract two values (either registers or constants) and store the result in a register.
    - **mul**: Multiply two values (either registers or constants) and store the result in a register.
    - **div**: Divide two values (either registers or constants) and store the result in a register.
    - **end**: Terminate the program execution.

  All the Instructions have been assigned a unique 6-bit Binary Operation Code.

Mov	000000
Add	000001
Sub	000010
Mul	000011
Div	000100
End	000101

3. Instruction Encoding and Decoding

- The simulator translates human-readable instructions into binary machine code and vice versa.
- It displays the decoded form, instruction meaning, and encoded binary form for each instruction executed.

Below, we provide a breakdown of how each instruction is encoded:

mov Instruction

- Opcode (6 bits): Represents the **mov** instruction (binary: **000000**).
- Destination Register (5 bits): Indicates the destination register where the value will be moved.
- Immediate Value (21 bits): Represents the immediate value to be moved.

Example:

- **mov r1 1**
    - Decoded Form: Move 1 to R1
    - Encoded Form (Binary): **000000 00001 000000000000000000001**

add Instruction

- Opcode (6 bits): Represents the **add** instruction (binary: **000001**).
- Destination Register (5 bits): Indicates the destination register where the result will be stored.
- Source Register 1 (5 bits): Represents the first source register for addition.
- Source Value 2 (16 bits): Indicates either the second source register or an immediate value for addition.

Example:

- **add r4 r3 r1**
    - Decoded Form: Add R3 and R1 to R4
    - Encoded Form (Binary): **000001 00100 00011 0000100000000000**

sub Instruction

- Opcode (6 bits): Represents the **sub** instruction (binary: **000010**).
- Destination Register (5 bits): Indicates the destination register where the result will be stored.
- Source Register 1 (5 bits): Represents the first source register for subtraction.
- Source Value 2 (16 bits): Indicates either the second source register or an immediate value for subtraction.

Example:

- **sub r4 r4 r2**

- Decoded Form: Subtract R2 from R4 and store in R4
- Encoded Form (Binary): **000010 00100 00100 0001000000000000**

mul Instruction

- Opcode (6 bits): Represents the **mul** instruction (binary: **000011**).
- Destination Register (5 bits): Indicates the destination register where the result will be stored.
- Source Register 1 (5 bits): Represents the first source register for multiplication.
- Source Value 2 (16 bits): Indicates either the second source register or an immediate value for multiplication.

Example:

- **mul r5 r1 9**
  - Decoded Form: Multiply R1 by 9 and store in R5
  - Encoded Form (Binary): **000011 00101 00001 0000000000001001**

div Instruction

- Opcode (6 bits): Represents the **div** instruction (binary: **000100**).
- Destination Register (5 bits): Indicates the destination register where the result will be stored.
- Source Register 1 (5 bits): Represents the first source register for division.
- Source Value 2 (16 bits): Indicates either the second source register or an immediate value for division.

Example:

- **div r6 r5 r3**
  - Decoded Form: Divide R5 by R3 and store in R6
  - Encoded Form (Binary): **000100 00110 00101 0001100000000000**

These encoding schemes provide a clear representation of each instruction in binary format, allowing the CPU simulator to execute them effectively.

This breakdown will help readers understand how each instruction is represented in binary form and how the encoding process works in your CPU simulator.

4. Cycle Counting

- The simulator calculates the total clock cycles consumed during the program's execution.

- Cycle counts vary based on the type of instruction (e.g., by **default** it takes 1 cycle, **add** and **sub** take 2 cycles, while **mul** and **div** take 4 cycles).

5. Performance Metrics

- The simulator computes the CPI (Cycles Per Instruction) as a measure of program efficiency. It calculates the CPI by taking the Total Cycle counts and divide it with the total number of instructions.

6. Execution Examples

To illustrate the functionality of the CPU simulator, we provide a series of execution examples. Each example includes the user's input, the decoded form of the instruction, the encoded binary form, and the resulting register values.

Example 1: **mov r1 1**

- User Input: **mov r1 1**
- Decoded Form: Move 1 to R1
- Encoded Form (Binary): **000000 00001 00000000000000000001**
- Registers: **[0, 1, 0, 0, 0, 0, 0, 0]**

Example 2: **mov r2 2**

- User Input: **mov r2 2**
- Decoded Form: Move 2 to R2
- Encoded Form (Binary): **000000 00010 00000000000000000010**
- Registers: **[0, 1, 2, 0, 0, 0, 0, 0]**

Example 3: **add r4 r3 r1**

- User Input: **add r4 r3 r1**
- Decoded Form: Add R3 and R1 to R4
- Encoded Form (Binary): **000001 00100 00011 0000100000000000**
- Registers: **[0, 1, 2, 3, 4, 0, 0, 0]**

Example 4: **sub r4 r4 r2**

- User Input: **sub r4 r4 r2**
- Decoded Form: Subtract R2 from R4 and store in R4
- Encoded Form (Binary): **000010 00100 00100 0001000000000000**
- Registers: **[0, 1, 2, 3, 2, 0, 0, 0]**

Example 5: **mul r5 r1 9**

- User Input: **mul r5 r1 9**
- Decoded Form: Multiply R1 by 9 and store in R5
- Encoded Form (Binary): **000011 00101 00001 0000000000001001**
- Registers: **[0, 1, 2, 3, 2, 9, 0, 0]**

Example 6: **div r6 r5 r3**

- User Input: **div r6 r5 r3**
- Decoded Form: Divide R5 by R3 and store in R6
- Encoded Form (Binary): **000100 00110 00101 0001100000000000**
- Registers: **[0, 1, 2, 3, 2, 9, 3, 0]**

Example 7: **end**

- User Input: **end**

These examples demonstrate the interaction between the user and the CPU simulator, showing how instructions are processed, decoded, and executed, resulting in changes to the register values.

Limitations

While the CPU simulator offers a basic CPU emulation, it has several limitations:

1. Simplified Instruction Set

- The simulator supports a limited set of instructions, which may not cover all real-world CPU operations.

2. Lack of Pipelining

- The simulator does not implement pipelining, which is a key feature in modern CPUs for improved performance.

3. Lack of Memory Management

- The simulator does not have memory management capabilities, which are essential for executing programs that require memory access.

4. No Handling of Decimal Values

- The simulator does not support decimal values; it operates solely with integer arithmetic.

5. Lack of Error Handling

- The simulator lacks robust error handling mechanisms to gracefully handle incorrect or invalid instructions or other runtime errors.

Possible Improvements

We can enhance the CPU simulator by implementing the following improvements:

1. Expanded Instruction Set

- Add support for a wider range of instructions to make the simulator more versatile.

2. Pipelining

- Implement a pipeline architecture to simulate the stages of instruction fetching, decoding, execution, and write-back.

3. Memory Management

- Introduce memory management to enable programs to access and manipulate data in memory.

4. Error Handling

- Implement robust error handling to gracefully handle incorrect or invalid instructions.

Conclusion

The CPU simulator provides a basic platform for understanding the fundamentals of CPU operation, instruction execution, and performance measurement. While it has limitations, further enhancements can turn it into a more comprehensive educational tool or even a simple platform for executing assembly-like programs.

Code

```
import java.util.Arrays;
import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;

public class CPU {
    private int[] registers = new int[8]; // 8 general-purpose registers
    private int totalCycles = 0; // Total clock cycles
    private int totalInstructionsProcessed = 0; // Total number of
instructions processed
    private Map<String, Integer> registerMapping = new HashMap<>();
    private Scanner scanner;

    public CPU() {
        Arrays.fill(registers, 0);

        // Initialize register mapping
        registerMapping.put("r0", 0);
        registerMapping.put("r1", 1);
        registerMapping.put("r2", 2);
        registerMapping.put("r3", 3);
        registerMapping.put("r4", 4);
        registerMapping.put("r5", 5);
        registerMapping.put("r6", 6);
        registerMapping.put("r7", 7);

        scanner = new Scanner(System.in);
    }

    public void execute() {
        while (true) {
            System.out.print("Enter instruction, 'end' to stop: ");
            String input = scanner.nextLine().trim();

            if (input.equalsIgnoreCase("end")) {
                break;
            }

            String[] instruction = input.split("\\s+");
            if (isValidInstruction(instruction)) {
                String opcode = instruction[0];

                // Assign cycle counts based on instruction type
                int cycleCount = 1; // Default cycle count

                if (opcode.equals("add") || opcode.equals("sub")) {
                    cycleCount = 2; // Adjust cycle count for add and sub
instructions
                } else if (opcode.equals("mul") || opcode.equals("div")) {
                    cycleCount = 4; // Adjust cycle count for mul and div
instructions
                }
```

```
                // Increment the total cycle count
                totalCycles += cycleCount;

                if (opcode.equals("mov")) {
                    int destReg = registerMapping.get(instruction[1]);
                    String operand = instruction[2];
                    if (operand.matches("\\d+")) {
                        registers[destReg] = Integer.parseInt(operand);
                    } else {
                        int srcReg = registerMapping.get(operand);
                        registers[destReg] = registers[srcReg];
                    }
                } else if (opcode.equals("add")) {
                    int destReg = registerMapping.get(instruction[1]);
                    int srcReg1 = registerMapping.get(instruction[2]);
                    int srcVal2;
                    if (instruction[3].matches("\\d+")) {
                        srcVal2 = Integer.parseInt(instruction[3]);
                    } else {
                        int srcReg2 = registerMapping.get(instruction[3]);
                        srcVal2 = registers[srcReg2];
                    }
                    registers[destReg] = registers[srcReg1] + srcVal2;
                } else if (opcode.equals("sub")) {
                    int destReg = registerMapping.get(instruction[1]);
                    int srcReg1 = registerMapping.get(instruction[2]);
                    int srcVal2;
                    if (instruction[3].matches("\\d+")) {
                        srcVal2 = Integer.parseInt(instruction[3]);
                        registers[destReg] = registers[srcReg1] - srcVal2;
                    } else {
                        int srcReg2 = registerMapping.get(instruction[3]);
                        registers[destReg] = registers[srcReg1] -
registers[srcReg2];
                    }
                } else if (opcode.equals("mul")) {
                    int destReg = registerMapping.get(instruction[1]);
                    int srcReg1 = registerMapping.get(instruction[2]);
                    int srcVal2;
                    if (instruction[3].matches("\\d+")) {
                        srcVal2 = Integer.parseInt(instruction[3]);
                        registers[destReg] = registers[srcReg1] * srcVal2;
                    } else {
                        int srcReg2 = registerMapping.get(instruction[3]);
                        registers[destReg] = registers[srcReg1] *
registers[srcReg2];
                    }
                } else if (opcode.equals("div")) {
                    int destReg = registerMapping.get(instruction[1]);
                    int srcReg1 = registerMapping.get(instruction[2]);
                    int srcVal2;

                    if (instruction[3].matches("\\d+")) {
                        srcVal2 = Integer.parseInt(instruction[3]);
                    } else {
                        int srcReg2 = registerMapping.get(instruction[3]);
```

```
                        srcVal2 = registers[srcReg2];
                    }

                    if (srcVal2 != 0) {
                        registers[destReg] = registers[srcReg1] / srcVal2;
                    } else {
                        System.out.println("Error: Division by zero!");
                    }
                } else if (opcode.equals("end")) {
                    break; // Exit the program
                }

                // Print the decoded form, instruction meaning, and encoded
form
                String decodedForm = getReadableInstruction(instruction);
                String encodedForm = encodeInstruction(instruction);
                System.out.println("Decoded Form: " + decodedForm);
                System.out.println("Encoded Form (Binary): " + encodedForm);

                // pc++;

                System.out.println("Registers: " +
Arrays.toString(registers));
                System.out.println();
                totalInstructionsProcessed++;
            } else {
                System.out.println("Invalid instruction. Please try
again.\n");
            }
        }
        // Calculate CPI
        double cpi = (double) totalCycles / totalInstructionsProcessed;
        System.out.println("CPI: " + cpi);
        // System.out.println(totalCycles);
        // System.out.println(totalInstructionsProcessed);
        scanner.close();
    }

    private boolean isValidInstruction(String[] instruction) {
        if (instruction.length < 1) {
            return false; // An empty instruction is invalid
        }

        String opcode = instruction[0];

        // Check if the opcode is valid (add, sub, mul, div, mov, or end)
        if (!Arrays.asList("add", "sub", "mul", "div", "mov",
"end").contains(opcode)) {
            return false;
        }

        // Additional validation logic specific to your instructions, if
needed

        return true; // If all checks pass, the instruction is considered
valid
    }
```

```
    private String getReadableInstruction(String[] instruction) {
        String opcode = instruction[0];

        if (opcode.equals("mov")) {
            String destReg = "R" + registerMapping.get(instruction[1]);
            String operand = instruction[2];
            if (operand.matches("\\d+")) {
                // Integer operand
                return "Move " + operand + " to " + destReg;
            } else {
                // Register operand
                String srcReg = "R" + registerMapping.get(operand);
                return "Move " + srcReg + " to " + destReg;
            }
        } else if (opcode.equals("add")) {
            String destReg = "R" + registerMapping.get(instruction[1]);
            String srcReg1 = "R" + registerMapping.get(instruction[2]);
            String srcReg2 = (instruction[3].matches("\\d+")) ?
instruction[3]
                    : "R" + registerMapping.get(instruction[3]);
            if (srcReg2.equals("R" + registerMapping.get(instruction[1]))) {
                return "Add 1 to " + destReg;
            } else {
                return "Add " + srcReg1 + " and " + srcReg2 + " to " +
destReg;
            }
        } else if (opcode.equals("sub")) {
            String destReg = "R" + registerMapping.get(instruction[1]);
            String srcReg1 = "R" + registerMapping.get(instruction[2]);
            String srcReg2 = (instruction[3].matches("\\d+")) ?
instruction[3]
                    : "R" + registerMapping.get(instruction[3]);
            if (srcReg2.equals("R" + registerMapping.get(instruction[1]))) {
                return "Subtract 1 from " + destReg;
            } else {
                return "Subtract " + srcReg2 + " from " + srcReg1 + " and
store in " + destReg;
            }
        } else if (opcode.equals("mul")) {
            String destReg = "R" + registerMapping.get(instruction[1]);
            String srcReg1 = "R" + registerMapping.get(instruction[2]);
            String srcReg2 = (instruction[3].matches("\\d+")) ?
instruction[3]
                    : "R" + registerMapping.get(instruction[3]);
            return "Multiply " + srcReg1 + " by " + srcReg2 + " and store in
" + destReg;
        } else if (opcode.equals("div")) {
            String destReg = "R" + registerMapping.get(instruction[1]);
            String srcReg1 = "R" + registerMapping.get(instruction[2]);
            String srcReg2 = (instruction[3].matches("\\d+")) ?
instruction[3]
                    : "R" + registerMapping.get(instruction[3]);
            return "Divide " + srcReg1 + " by " + srcReg2 + " and store in "
+ destReg;
        } else if (opcode.equals("end")) {
            return "End of program";
```

```
        }
        return "";
    }

    public String encodeInstruction(String[] instruction) {
        String opcode = instruction[0];

        if (opcode.equals("mov")) {
            int destReg = registerMapping.get(instruction[1]);
            String operand = instruction[2];
            if (operand.matches("\\d+")) {
                // Integer operand
                int operandValue = Integer.parseInt(operand);
                String operandBinary = intToBinaryString(operandValue, 21);
// 21-bit block for integer
                registers[destReg] = operandValue;
                return "000000 " + intToBinaryString(destReg, 5) + " " +
operandBinary;
            } else {
                // Register operand
                int srcReg = registerMapping.get(operand);
                String operandBinary = intToBinaryString(srcReg, 5) +
"0000000000000000"; // 5 bits for the src register
                registers[destReg] = registers[srcReg];
                return "000000 " + intToBinaryString(destReg, 5) + " " +
operandBinary;
            }
        } else if (opcode.equals("add")) {
            int destReg = registerMapping.get(instruction[1]);
            int srcReg1 = registerMapping.get(instruction[2]);
            String srcVal2;
            if (instruction[3].matches("\\d+")) {
                // Integer operand
                int operandValue = Integer.parseInt(instruction[3]);
                srcVal2 = intToBinaryString(operandValue, 16);
            } else {
                // Register operand
                int srcReg2 = registerMapping.get(instruction[3]);
                srcVal2 = intToBinaryString(srcReg2, 5) + "00000000000"; // 5
bits for the src register
            }
            return "000001 " + intToBinaryString(destReg, 5) + " " +
intToBinaryString(srcReg1, 5) + " " + srcVal2;
        } else if (opcode.equals("sub")) {
            int destReg = registerMapping.get(instruction[1]);
            int srcReg1 = registerMapping.get(instruction[2]);
            String srcVal2;
            if (instruction[3].matches("\\d+")) {
                // Integer operand
                int operandValue = Integer.parseInt(instruction[3]);
                srcVal2 = intToBinaryString(operandValue, 16);
            } else {
                // Register operand
                int srcReg2 = registerMapping.get(instruction[3]);
                srcVal2 = intToBinaryString(srcReg2, 5) + "00000000000"; // 5
bits for the src register
            }
```

```
            return "000010 " + intToBinaryString(destReg, 5) + " " +
intToBinaryString(srcReg1, 5) + " " + srcVal2;
        } else if (opcode.equals("mul")) {
            int destReg = registerMapping.get(instruction[1]);
            int srcReg1 = registerMapping.get(instruction[2]);
            String srcVal2;
            if (instruction[3].matches("\\d+")) {
                // Integer operand
                int operandValue = Integer.parseInt(instruction[3]);
                srcVal2 = intToBinaryString(operandValue, 16);
            } else {
                // Register operand
                int srcReg2 = registerMapping.get(instruction[3]);
                srcVal2 = intToBinaryString(srcReg2, 5) + "00000000000"; // 5
bits for the src register
            }
            return "000011 " + intToBinaryString(destReg, 5) + " " +
intToBinaryString(srcReg1, 5) + " " + srcVal2;
        } else if (opcode.equals("div")) {
            int destReg = registerMapping.get(instruction[1]);
            int srcReg1 = registerMapping.get(instruction[2]);
            String srcVal2;
            if (instruction[3].matches("\\d+")) {
                // Integer operand
                int operandValue = Integer.parseInt(instruction[3]);
                srcVal2 = intToBinaryString(operandValue, 16);
            } else {
                // Register operand
                int srcReg2 = registerMapping.get(instruction[3]);
                srcVal2 = intToBinaryString(srcReg2, 5) + "00000000000"; // 5
bits for the src register
            }
            return "000100 " + intToBinaryString(destReg, 5) + " " +
intToBinaryString(srcReg1, 5) + " " + srcVal2;
        } else if (opcode.equals("end")) {
            return "000101" + "00000000000000000000000"; // Binary encoding
for end
        }
        return "";
    }

    private String intToBinaryString(int value, int numBits) {
        String binary = Integer.toBinaryString(value);
        return String.format("%0" + numBits + "d", Integer.parseInt(binary));
    }

    public static void main(String[] args) {
        CPU cpu = new CPU();
        cpu.execute();
    }
}
```