6530262_MinMyintMohSoe_CA_ISA_Project

1. Registers

ISA design has 8 registers:

- r0 - r6: General use registers
- r7: Special register that holds remainder from division and extra bits that don't fit in the main register during multiplication

Opcodes

Opcode	Value	Description	Cycles
end	0	End program execution	1
mov	1	Move value to register	1
add	2	Add value to register	1
sub	3	Subtract value from register	1
mul	4	Multiply register by value	3
div	5	Divide register by value	5

Instruction Format Design

Field	Bits	Description
Opcode	31-28	Operation code (4 bits)
Destination Register	27-25	Target register (3 bits)
Mode	24	0 = Register mode, 1 = Immediate mode
Source Register	23-21	Source register (when mode = 0)
Immediate Value	23-8	Immediate value (when mode = 1)
Unused	7-0	Not used

2. Bit Representation

- **Opcode**: 4 bits (values 0-15)

- **Registers**: 3 bits (allows for 8 registers, r0-r7)
- **Mode**: 1 bit (0 = register operand, 1 = immediate value)
- **Immediate Value**: 16 bits (when used)
- **Total Instruction Width**: 32 bits

3. Full Source Code

```
import java.util.ArrayList;

import java.util.List;

import java.util.Scanner;



class CPU {

   private int[] registers;

   private int cycleCount;

   private int instructionCount;

   private boolean isRunning;

   public CPU() {

      registers = new int[8];

      cycleCount = 0;

      instructionCount = 0;

      isRunning = true;

   }



   public void execute(List<String> instructionList) {

      for (String instruction : instructionList) {

         System.out.println("\nExecuting: " + instruction);

         executeInstruction(instruction);

         if (!isRunning) {

            break;
```

```
        }
    }
    System.out.println("\nExecution complete.");
}

public String encodeInstruction(String instruction) {
    String[] parts = instruction.split("\\s+");
    String opcode = parts[0].toLowerCase();

    int opcodeValue = 0;
    if (opcode.equals("mov")) {
        opcodeValue = 1;
    } else if (opcode.equals("add")) {
        opcodeValue = 2;
    } else if (opcode.equals("sub")) {
        opcodeValue = 3;
    } else if (opcode.equals("mul")) {
        opcodeValue = 4;
    } else if (opcode.equals("div")) {
        opcodeValue = 5;
    } else if (opcode.equals("end")) {
        opcodeValue = 0;
    }

    int encoding = 0;
    if (opcode.equals("end")) {
        encoding = opcodeValue << 28;
```

```
    } else {
        int destinationRegister = getRegisterNumber(parts[1]);
        encoding |= (opcodeValue & 0xF) << 28;
        encoding |= (destinationRegister & 0x7) << 25;
        int mode = 0;

        if (parts.length >= 3) {
            if (!parts[2].toLowerCase().startsWith("r")) {
                mode = 1;
                int immediateValue = Integer.parseInt(parts[2]);
                encoding |= (mode & 0x1) << 24;
                encoding |= (immediateValue & 0xFFFF) << 8;
            } else {
                int sourceRegister = getRegisterNumber(parts[2]);
                encoding |= (mode & 0x1) << 24;
                encoding |= (sourceRegister & 0x7) << 21;
            }
        }
    }
    String binaryString = String.format("%32s", Integer.toBinaryString(encoding)).replace(' ', '0');
    return "[" + (binaryString) + "]";
}

public void executeInstruction(String instruction) {
    System.out.println("    Decoded: " + instruction);
    String encodedInstruction = encodeInstruction(instruction);
    System.out.println("    Encoded: " + encodedInstruction);
```

```
String[] parts = instruction.split("\\s+");
String opcode = parts[0].toLowerCase();

if (opcode.equals("end")) {
    cycleCount++;
    instructionCount++;
    isRunning = false;
    return;
}
int destinationRegister = getRegisterNumber(parts[1]);
int operandValue = 0;
if (parts.length >= 3) {
    if (parts[2].toLowerCase().startsWith("r")) {
        int sourceRegister = getRegisterNumber(parts[2]);
        operandValue = registers[sourceRegister];
    } else {
        operandValue = Integer.parseInt(parts[2]);
    }
}

if (opcode.equals("mov")) {
    registers[destinationRegister] = operandValue;
    cycleCount++;
} else if (opcode.equals("add")) {
    registers[destinationRegister] = registers[destinationRegister] + operandValue;
    cycleCount++;
```

```
    } else if (opcode.equals("sub")) {
        registers[destinationRegister] = registers[destinationRegister] - operandValue;
        cycleCount++;
    } else if (opcode.equals("mul")) {
        long product = (long) registers[destinationRegister] * operandValue;
        registers[destinationRegister] = (int) product;
        registers[7] = (int) (product >> 32);
        cycleCount += 3;
    } else if (opcode.equals("div")) {
        int dividend = registers[destinationRegister];
        registers[destinationRegister] = dividend / operandValue;
        registers[7] = dividend % operandValue;
        cycleCount += 4;
    }
    instructionCount++;
}

private int getRegisterNumber(String register) {
    register = register.replace(",", "").replace(":", "");
    return Integer.parseInt(register.substring(1));
}

public void printState() {
    System.out.println("\nFinal Register Values:");
    for (int i = 0; i < registers.length; i++) {
        System.out.println("r" + i + " = " + registers[i] + " (" + intTo32BitBinary(registers[i]) + ")");
    }
```

```
        System.out.println("Total cycles: " + cycleCount);
        System.out.println("Instructions executed: " + instructionCount);
        if (instructionCount > 0) {
            System.out.println("CPI (cycles per instruction): " + ((double) cycleCount / instructionCount));
        }
    }

    private String intTo32BitBinary(int value) {
        String binary = Integer.toBinaryString(value);
        return String.format("%32s", binary).replace(' ', '0');
    }
}

public class CPUSimulator {
    public static void main(String[] args) {
        CPU cpu = new CPU();
        Scanner scanner = new Scanner(System.in);
        List<String> instructionList = new ArrayList<>();
        System.out.println("Enter instructions (type 'end' to finish):");
        while (true) {
            System.out.print("Instruction: ");
            String instruction = scanner.nextLine().trim();
            if (instruction.isEmpty()) {
                continue;
            }
            instructionList.add(instruction);
            if (instruction.toLowerCase().startsWith("end")) {
```

```
            break;

        }

    }

    cpu.execute(instructionList);

    System.out.println("\n=== Final CPU State ===");

    cpu.printState();

    scanner.close();

  }

}
```