Name: Lythean Sem
ID: 6511925

Instruction Set Architecture (ISA)

Overview:

This code is a simple Instruction Set Architecture (ISA) simulation with registers and an instruction set. The processor executes instructions, and the code includes functionalities to print both decoded and encoded forms of the instructions, display the result of the registers, calculate clock cycles and simulate a pipelined execution.

The number of registers that are used in this simulation includes r0, r1, r2, r3, r4, r5, r6 and r7 is used exclusively for storing remainder.

The operations include "mov" which load number or register to a particular register, "add" is to plus a number to the register or plus register with another register, "mul" is to multiply a number to the register or multiply register with another register, "sub" is to subtract a number to the register or subtract register with another register, lastly "div" is to divide a number to the register or divide register with another register.

Class Structure:

1. Registers Class:
    - Attributes:
        - reg_val: Represents the value stored in the register.
        - reg_adr: Represents the address of the register.
    - Methods:
        - set_reg_adr: Sets the register address.
        - get_reg_adr: Retrieves the register address.
        - set_reg_val: Sets the register value.
        - get_reg_val: Retrieves the register value.
        - to_32bit_val: Converts the register value to a 32-bit binary string.
        - to_3bit_adr: Converts the register address to a 3-bit binary string.
2. InstructionSet class:
    - Attributes:
        - step: Represents the step in the execution.
        - opcode: Represents the operation code of the instruction.
        - register: Represents the register involved in the instruction.
        - clkcyc: Represents the number of clock cycles the instruction takes.
        - operand: Represents an operand for some instructions.
        - value: Represents a value associated with the instruction.
    - Methods:
        - five_bit_opcode: Converts the opcode to a 5-bit binary string.
        - encode_instruction: Encodes the instruction in a specific format.
        - to_32bit_val: Converts the value to a 32-bit binary string.
        - __str__: Provides a string representation of the instruction.

Main Functionality:

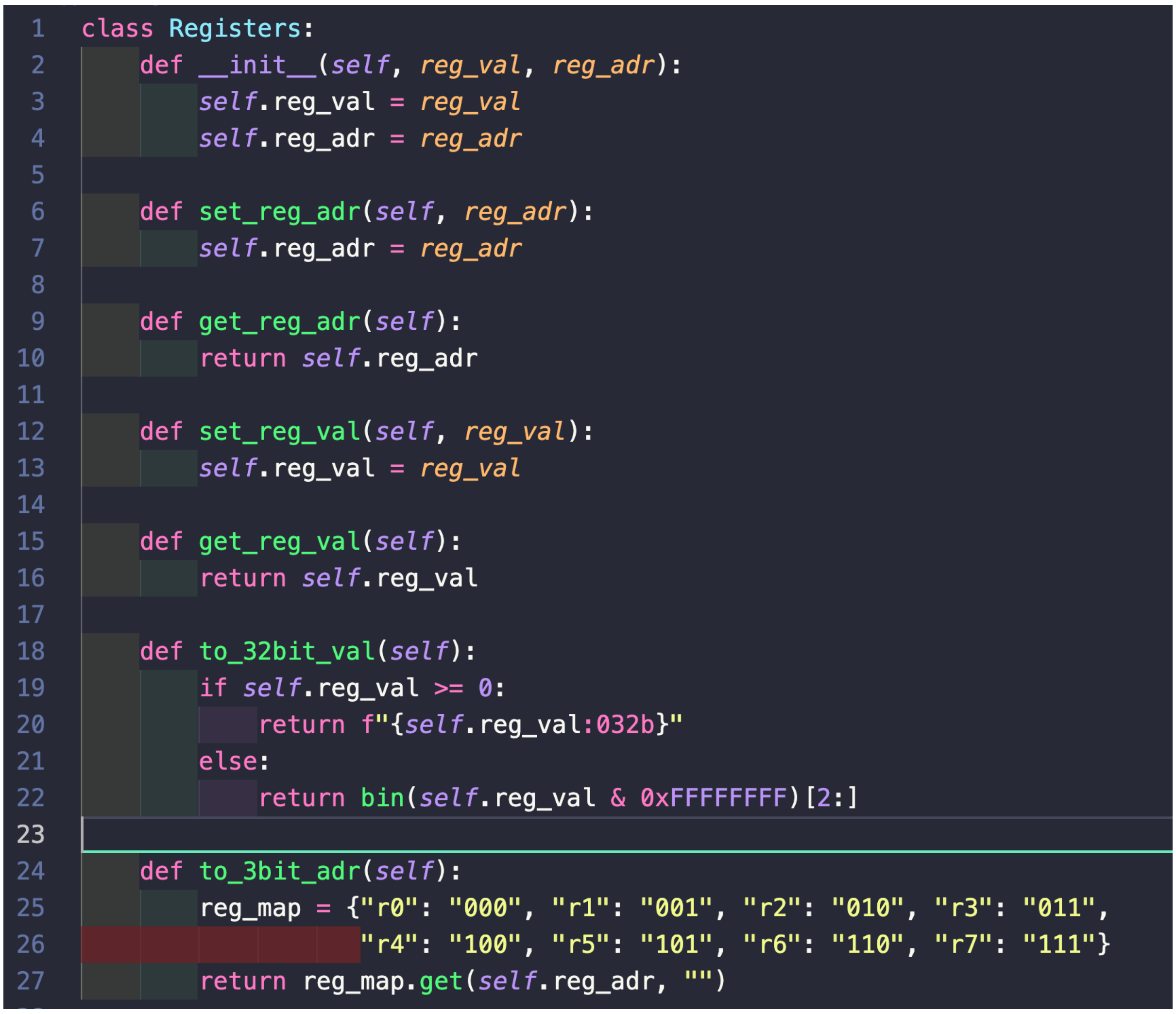
1. main Function:
    - Initializes a list of registers (“regs”) and an empty list of instructions (“instructions”)
    - Accepts input from user until “end 0 0” is entered.
    - Processes instructions, updates register values, and creates “InstructionSet” objects.
    - Calls “print_output” to display the results.
2. print_output Functions:
    - Prints decoded and encoded forms of each instruction along with clock cycles.
    - Displays the state of registers after execution.
    - Calculates and prints Clocks Per Instruction (CPI).
    - Simulates a pipelined execution and prints the result.

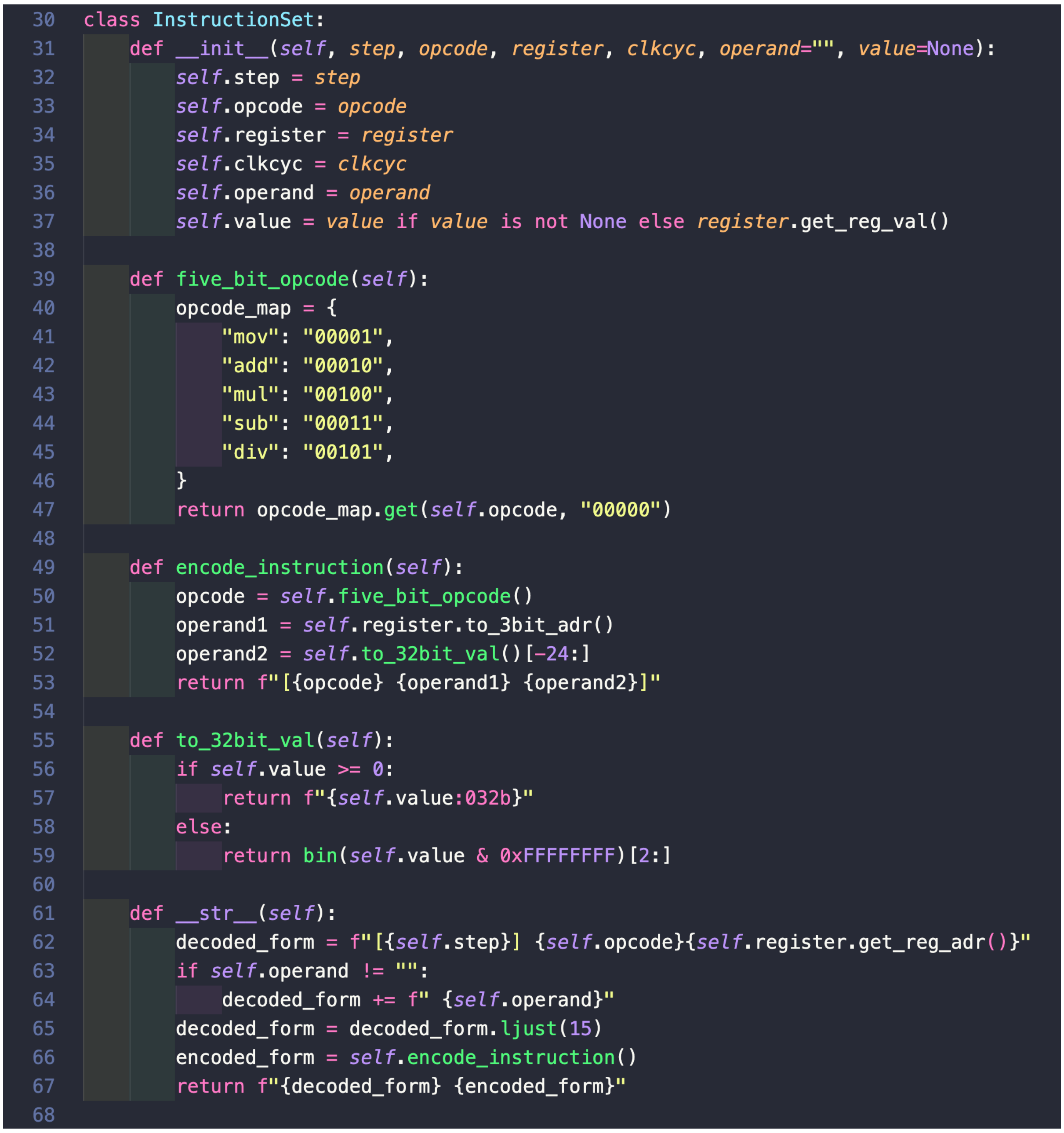
User input:

The user enters (opcode) (operand1) (operand2).

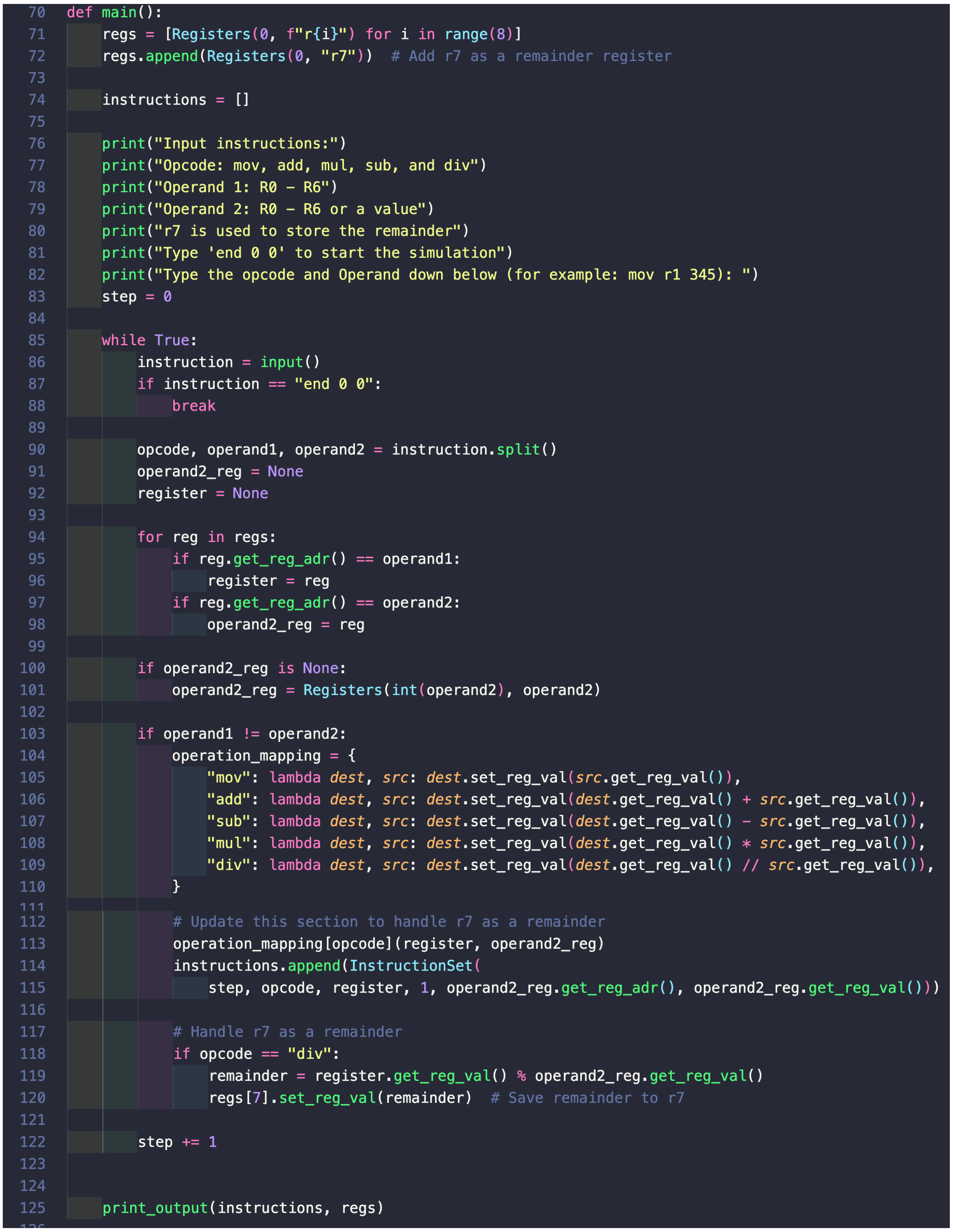
- Opcodes are the operations code.
- Operand1 are registers (r0 to r6).
- Operand2 can be registers or numbers.

Here is the code:

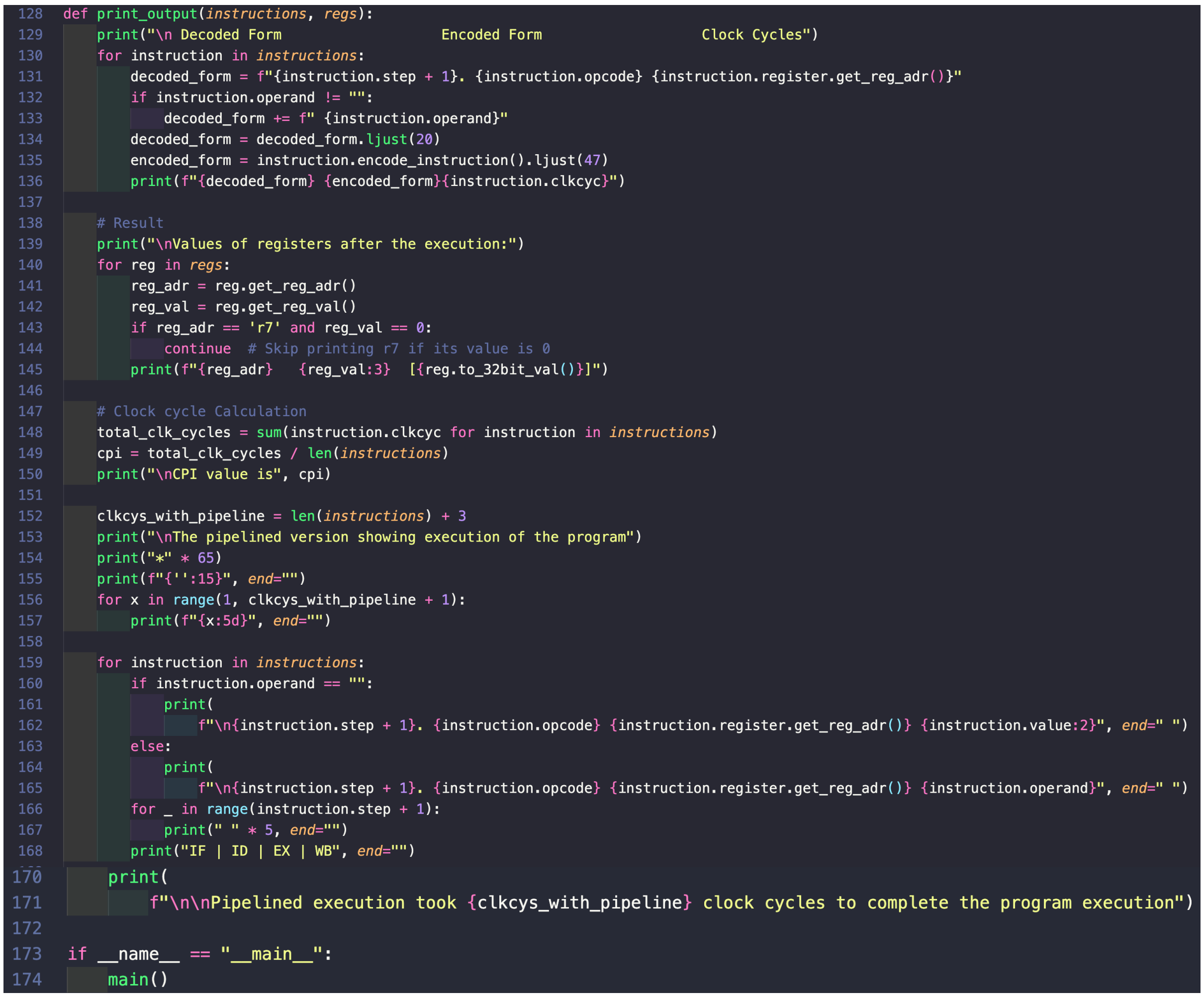

```python
class Registers:
    def __init__(self, reg_val, reg_adr):
        self.reg_val = reg_val
        self.reg_adr = reg_adr

    def set_reg_adr(self, reg_adr):
        self.reg_adr = reg_adr

    def get_reg_adr(self):
        return self.reg_adr

    def set_reg_val(self, reg_val):
        self.reg_val = reg_val

    def get_reg_val(self):
        return self.reg_val

    def to_32bit_val(self):
        if self.reg_val >= 0:
            return f"{self.reg_val:032b}"
        else:
            return bin(self.reg_val & 0xFFFFFFFF)[2:]

    def to_3bit_adr(self):
        reg_map = {"r0": "000", "r1": "001", "r2": "010", "r3": "011",
                           "r4": "100", "r5": "101", "r6": "110", "r7": "111"}
        return reg_map.get(self.reg_adr, "")
```



```python
class InstructionSet:
    def __init__(self, step, opcode, register, clkcyc, operand="", value=None):
        self.step = step
        self.opcode = opcode
        self.register = register
        self.clkcyc = clkcyc
        self.operand = operand
        self.value = value if value is not None else register.get_reg_val()

    def five_bit_opcode(self):
        opcode_map = {
            "mov": "00001",
            "add": "00010",
            "mul": "00100",
            "sub": "00011",
            "div": "00101",
        }
        return opcode_map.get(self.opcode, "00000")

    def encode_instruction(self):
        opcode = self.five_bit_opcode()
        operand1 = self.register.to_3bit_adr()
        operand2 = self.to_32bit_val()[-24:]
        return f"[{opcode} {operand1} {operand2}]"

    def to_32bit_val(self):
        if self.value >= 0:
            return f"{self.value:032b}"
        else:
            return bin(self.value & 0xFFFFFFFF)[2:]

    def __str__(self):
        decoded_form = f"[{self.step}] {self.opcode}{self.register.get_reg_adr()}"
        if self.operand != "":
            decoded_form += f" {self.operand}"
        decoded_form = decoded_form.ljust(15)
        encoded_form = self.encode_instruction()
        return f"{decoded_form} {encoded_form}"

```



```python
def main():
    regs = [Registers(0, f"r{i}") for i in range(8)]
    regs.append(Registers(0, "r7"))  # Add r7 as a remainder register

    instructions = []

    print("Input instructions:")
    print("Opcode: mov, add, mul, sub, and div")
    print("Operand 1: R0 - R6")
    print("Operand 2: R0 - R6 or a value")
    print("r7 is used to store the remainder")
    print("Type 'end 0 0' to start the simulation")
    print("Type the opcode and Operand down below (for example: mov r1 345): ")
    step = 0

    while True:
        instruction = input()
        if instruction == "end 0 0":
            break

        opcode, operand1, operand2 = instruction.split()
        operand2_reg = None
        register = None

        for reg in regs:
            if reg.get_reg_adr() == operand1:
                register = reg
            if reg.get_reg_adr() == operand2:
                operand2_reg = reg

        if operand2_reg is None:
            operand2_reg = Registers(int(operand2), operand2)

        if operand1 != operand2:
            operation_mapping = {
                "mov": lambda dest, src: dest.set_reg_val(src.get_reg_val()),
                "add": lambda dest, src: dest.set_reg_val(dest.get_reg_val() + src.get_reg_val()),
                "sub": lambda dest, src: dest.set_reg_val(dest.get_reg_val() - src.get_reg_val()),
                "mul": lambda dest, src: dest.set_reg_val(dest.get_reg_val() * src.get_reg_val()),
                "div": lambda dest, src: dest.set_reg_val(dest.get_reg_val() // src.get_reg_val()),
            }

            # Update this section to handle r7 as a remainder
            operation_mapping[opcode](register, operand2_reg)
            instructions.append(InstructionSet(
                step, opcode, register, 1, operand2_reg.get_reg_adr(), operand2_reg.get_reg_val()))

            # Handle r7 as a remainder
            if opcode == "div":
                remainder = register.get_reg_val() % operand2_reg.get_reg_val()
                regs[7].set_reg_val(remainder)  # Save remainder to r7

        step += 1


    print_output(instructions, regs)
```



```python
def print_output(instructions, regs):
    print("\n Decoded Form                   Encoded Form                   Clock Cycles")
    for instruction in instructions:
        decoded_form = f"{instruction.step + 1}. {instruction.opcode} {instruction.register.get_reg_adr()}"
        if instruction.operand != "":
            decoded_form += f" {instruction.operand}"
        decoded_form = decoded_form.ljust(20)
        encoded_form = instruction.encode_instruction().ljust(47)
        print(f"{decoded_form} {encoded_form}{instruction.clkcyc}")

    # Result
    print("\nValues of registers after the execution:")
    for reg in regs:
        reg_adr = reg.get_reg_adr()
        reg_val = reg.get_reg_val()
        if reg_adr == 'r7' and reg_val == 0:
            continue  # Skip printing r7 if its value is 0
        print(f"{reg_adr}   {reg_val:3}  [{reg.to_32bit_val()}]")

    # Clock cycle Calculation
    total_clk_cycles = sum(instruction.clkcyc for instruction in instructions)
    cpi = total_clk_cycles / len(instructions)
    print("\nCPI value is", cpi)

    clkcys_with_pipeline = len(instructions) + 3
    print("\nThe pipelined version showing execution of the program")
    print("*" * 65)
    print(f"{'':15}", end="")
    for x in range(1, clkcys_with_pipeline + 1):
        print(f"{x:5d}", end="")

    for instruction in instructions:
        if instruction.operand == "":
            print(
                f"\n{instruction.step + 1}. {instruction.opcode} {instruction.register.get_reg_adr()} {instruction.value:2}", end=" ")
        else:
            print(
                f"\n{instruction.step + 1}. {instruction.opcode} {instruction.register.get_reg_adr()} {instruction.operand}", end=" ")
        for _ in range(instruction.step + 1):
            print("   " * 5, end="")
        print("IF | ID | EX | WB", end="")
    print(
        f"\n\nPipelined execution took {clkcys_with_pipeline} clock cycles to complete the program execution")

if __name__ == "__main__":
    main()
```