

```

class CPU:
    def __init__(self):
        self.registers = [0] * 8 # 8 general-purpose registers
        self.memory = [0] * 1024 # Main memory with 1024 words
        self.pc = 0 # Program counter
        self.cycles = 0 # Total clock cycles
        self.register_mapping = {'r0': 0, 'r1': 1, 'r2': 2,
                                'r3': 3, 'r4': 4, 'r5': 5, 'r6': 6, 'r7': 7}

        self.opcode_mapping = {
            "mov": "000",
            "add": "001",
            "sub": "010",
            "mul": "011",
            "div": "100",
            "end": "101"
        }

        # Initialize pipeline registers
        self.pipeline_registers = {
            'IF/ID': None,
            'ID/EX': None,
            'EX/MEM': None,
            'MEM/WB': None
        }

    def encode_instruction(self, instruction):
        opcode = self.opcode_mapping[instruction[0]]

        if opcode == self.opcode_mapping["end"]:
            return opcode.ljust(32, '0')

        dest_reg = format(self.register_mapping[instruction[1]], '03b')

        # Determine if operand is a register or immediate value
        if isinstance(instruction[2], int):
            src1 = format(instruction[2], '08b')
        else:
            src1 = format(self.register_mapping[instruction[2]], '03b').rjust(
                8, '0') # Here's the change for padding to the left

        if len(instruction) > 3:
            if isinstance(instruction[3], int):
                src2 = format(instruction[3], '08b')
            else:
                src2 = format(self.register_mapping[instruction[3]], '03b').rjust(
                    8, '0') # And here as well
        else:
            src2 = '0' * 8

        # Combine all parts to get the 32 bit instruction
        encoded_instruction = opcode + dest_reg + src1 + src2
        # Fill remaining bits to complete 32 bits
        encoded_instruction = encoded_instruction.ljust(32, '0')

        return encoded_instruction

```

```
def encode_instruction(self, instruction):
    opcode = self.opcode_mapping[instruction[0]]

    if opcode == self.opcode_mapping["end"]:
        return opcode.ljust(32, '0')

    dest_reg = format(self.register_mapping[instruction[1]], '03b')

    # Determine if operand is a register or immediate value
    if isinstance(instruction[2], int):
        src1 = format(instruction[2], '08b')
    else:
        src1 = format(self.register_mapping[instruction[2]], '03b').rjust(8, '0')

    if len(instruction) > 3:
        if isinstance(instruction[3], int):
            src2 = format(instruction[3], '08b')
        else:
            src2 = format(self.register_mapping[instruction[3]], '03b').rjust(8, '0')
    else:
        src2 = '0' * 8

    # Combine all parts to get the 32 bit instruction
    encoded_instruction = opcode + dest_reg + src1 + src2
    # Fill remaining bits to complete 32 bits
    encoded_instruction = encoded_instruction.ljust(32, '0')

    return encoded_instruction
```

```

def execute(self, program):
    self.pc = 0

    print(f"    Decoded \tEncoded Form\t\t\t Updated Registers\t    Cycles")
    while self.pc < len(program):
        instruction = program[self.pc]
        opcode = instruction[0]

        if opcode == "mov":
            dest_reg = self.register_mapping[instruction[1]]
            operand = instruction[2]
            if isinstance(operand, int):
                self.registers[dest_reg] = operand
            else:
                src_reg = self.register_mapping[instruction[2]]
                self.registers[dest_reg] = self.registers[src_reg]
            self.cycles += 0.5

        elif opcode == "add":
            dest_reg = self.register_mapping[instruction[1]]
            src_reg1 = self.register_mapping[instruction[2]]
            if len(instruction) > 3:
                if isinstance(instruction[3], int):
                    src_val2 = instruction[3]
                else:
                    src_reg2 = self.register_mapping[instruction[3]]
                    src_val2 = self.registers[src_reg2]
            else:
                src_val2 = 0

            self.registers[dest_reg] = self.registers[src_reg1] + src_val2
            self.cycles += 0.75

        elif opcode == "sub":
            dest_reg = self.register_mapping[instruction[1]]
            src_reg1 = self.register_mapping[instruction[2]]
            if len(instruction) > 3:
                if isinstance(instruction[3], int):
                    src_val2 = instruction[3]
                else:
                    src_reg2 = self.register_mapping[instruction[3]]
                    src_val2 = self.registers[src_reg2]
            else:
                src_val2 = 0

            self.registers[dest_reg] = self.registers[src_reg1] - src_val2
            self.cycles += 0.75

```

```

elif opcode == "mul":
    dest_reg = self.register_mapping[instruction[1]]
    src_reg1 = self.register_mapping[instruction[2]]
    if len(instruction) > 3:
        if isinstance(instruction[3], int):
            src_val2 = instruction[3]
        else:
            src_reg2 = self.register_mapping[instruction[3]]
            src_val2 = self.registers[src_reg2]
    else:
        src_val2 = 0

    self.registers[dest_reg] = self.registers[src_reg1] * src_val2
    self.cycles += 1

elif opcode == "div":
    dest_reg = self.register_mapping[instruction[1]]
    src_reg1 = self.register_mapping[instruction[2]]
    if len(instruction) > 3:
        if isinstance(instruction[3], int):
            src_val2 = instruction[3]
        else:
            src_reg2 = self.register_mapping[instruction[3]]
            src_val2 = self.registers[src_reg2]
    else:
        src_val2 = 1 # Default to 1 if src2 is not provided

    # Check for division by zero
    if src_val2 == 0:
        print("Division by zero error! Skipping instruction:", instruction)
        self.pc += 1 # Skip the current instruction
        continue


    self.registers[dest_reg] = self.registers[src_reg1] // src_val2
    self.cycles += 1

elif opcode == "end":
    break # Exit the program

self.pc += 1

# Print the encoded and decoded forms
decoded_instruction = " ".join(map(str, instruction))
encoded_instruction = self.encode_instruction(instruction)
print(
    f"{self.pc}. {decoded_instruction}\t{encoded_instruction} {self.registers}
    {self.cycles}")

```



```
def run(self, program):

    # Execute the program
    self.execute(program)

    # Calculate the number of executed instructions
    num_executed_instructions = sum(
        1 for instr in program if instr[0] != "end")

    # Print the final state of the registers and performance metrics
    print("\nFinal Registers:", self.registers)

    cpi = self.cycles / num_executed_instructions # Corrected CPI calculation
    print(f"Number of Cycles: {self.cycles}")
    print(f"Number of Executed Instructions: {num_executed_instructions}")
    print(f"Cycles per Instruction (CPI): {cpi:.2f}")
```




```
def execute_instruction(self, instruction):
    # run each instruction contrary to execute function
    # this function is specifically designed for pipeline_execute function
    opcode = instruction[0]

    if opcode == "mov":
        dest_reg = self.register_mapping[instruction[1]]
        operand = instruction[2]
        if isinstance(operand, int):
            self.registers[dest_reg] = operand
        else:
            src_reg = self.register_mapping[instruction[2]]
            self.registers[dest_reg] = self.registers[src_reg]
        self.cycles += 0.5

    elif opcode == "add":
        dest_reg = self.register_mapping[instruction[1]]
        src_reg1 = self.register_mapping[instruction[2]]
        if len(instruction) > 3:
            if isinstance(instruction[3], int):
                src_val2 = instruction[3]
            else:
                src_reg2 = self.register_mapping[instruction[3]]
                src_val2 = self.registers[src_reg2]
        else:
            src_val2 = 0

        self.registers[dest_reg] = self.registers[src_reg1] + src_val2
        self.cycles += 0.75

    elif opcode == "sub":
        dest_reg = self.register_mapping[instruction[1]]
        src_reg1 = self.register_mapping[instruction[2]]
        if len(instruction) > 3:
            if isinstance(instruction[3], int):
                src_val2 = instruction[3]
            else:
                src_reg2 = self.register_mapping[instruction[3]]
                src_val2 = self.registers[src_reg2]
        else:
            src_val2 = 0

        self.registers[dest_reg] = self.registers[src_reg1] - src_val2
        self.cycles += 0.75
```

```

elif opcode == "mul":
    dest_reg = self.register_mapping[instruction[1]]
    src_reg1 = self.register_mapping[instruction[2]]
    if len(instruction) > 3:
        if isinstance(instruction[3], int):
            src_val2 = instruction[3]
        else:
            src_reg2 = self.register_mapping[instruction[3]]
            src_val2 = self.registers[src_reg2]
    else:
        src_val2 = 0

    self.registers[dest_reg] = self.registers[src_reg1] * src_val2
    self.cycles += 1

elif opcode == "div":
    dest_reg = self.register_mapping[instruction[1]]
    src_reg1 = self.register_mapping[instruction[2]]
    if len(instruction) > 3:
        if isinstance(instruction[3], int):
            src_val2 = instruction[3]
        else:
            src_reg2 = self.register_mapping[instruction[3]]
            src_val2 = self.registers[src_reg2]
    else:
        src_val2 = 1 # Default to 1 if src2 is not provided

    # Check for division by zero
    if src_val2 == 0:
        print("Division by zero error! Skipping instruction:", instruction)
        self.pc += 1 # Skip the current instruction
        return

    self.registers[dest_reg] = self.registers[src_reg1] // src_val2
    self.cycles += 1

elif opcode == "end":
    pass # No action needed for "end" instruction

else:
    print(f"Unknown opcode: {opcode}")

```

```

def pipeline_execute(self):
    while True:
        # Print the current pipeline stage contents
        print("\nCurrent Pipeline State:")
        for stage, instruction in self.pipeline_registers.items():
            print(f"{stage}: {instruction}")

        # Move instructions through the pipeline stages
        self.pipeline_registers['MEM/WB'] = self.pipeline_registers['EX/MEM']
        self.pipeline_registers['EX/MEM'] = self.pipeline_registers['ID/EX']
        self.pipeline_registers['ID/EX'] = self.pipeline_registers['IF/ID']

        # Fetch the next instruction into IF stage
        if self.pc < len(self.program):
            instruction = self.program[self.pc]
            self.pipeline_registers['IF/ID'] = instruction
            self.pc += 1
        else:
            # No more instructions to fetch
            self.pipeline_registers['IF/ID'] = None

        # Execute the instruction in EX stage
        if self.pipeline_registers['EX/MEM']:
            instruction = self.pipeline_registers['EX/MEM']
            self.execute_instruction(instruction)

        # Check for program termination
        if not self.pipeline_registers['IF/ID'] and not self.pipeline_registers['ID/EX'] and
not self.pipeline_registers['EX/MEM']:
            break

        # Increment cycle count
        self.cycles += 1

    # Print the final pipeline state
    print("\nFinal Pipeline State:")
    for stage, instruction in self.pipeline_registers.items():
        print(f"{stage}: {instruction}")

    # Print the total number of clock cycles
    print(f"\nTotal Clock Cycles: {self.cycles}")

```



```

def run_pipeline(self, program):
    # Set the program and reset the CPU state
    self.program = program
    self.pc = 0
    self.cycles = 0
    self.registers = [0] * 8
    self.pipeline_registers = {
        'IF/ID': None,
        'ID/EX': None,
        'EX/MEM': None,
        'MEM/WB': None
    }

    # Perform pipelined execution
    self.pipeline_execute()

    # Print the final state of the registers and performance metrics
    print("Registers:", self.registers)
    print(f"Total Clock Cycles: {self.cycles}")

```

Output without pipeline execution:

Decoded	Encoded Form	Updated Registers	Cycles
1. mov r1 1	00000100000000100000000000000000	[0, 1, 0, 0, 0, 0, 0, 0]	0.5
2. mov r2 0	00001000000000000000000000000000	[0, 1, 0, 0, 0, 0, 0, 0]	1.0
3. mov r3 3	00001100000001100000000000000000	[0, 1, 0, 3, 0, 0, 0, 0]	1.5
4. add r1 r1 1	00100100000000100000000100000000	[0, 2, 0, 3, 0, 0, 0, 0]	2.25
5. add r1 r1 r2	00100100000000100000001000000000	[0, 2, 0, 3, 0, 0, 0, 0]	3.0
6. mul r1 r1 2	01100100000000100000001000000000	[0, 4, 0, 3, 0, 0, 0, 0]	4.0
Division by zero error! Skipping instruction: ['div', 'r4', 'r3', 'r2']			
8. add r2 r1 4	00101000000000100000010000000000	[0, 4, 8, 3, 0, 0, 0, 0]	4.75

Final Registers: [0, 4, 8, 3, 0, 0, 0, 0]
 Number of Cycles: 4.75
 Number of Executed Instructions: 8
 Cycles per Instruction (CPI): 0.59

Output with pipeline execution:

Pipelined execution:

Current Pipeline State:

IF/ID: None
ID/EX: None
EX/MEM: None
MEM/WB: None

Current Pipeline State:

IF/ID: ['mov', 'r3', 3]
ID/EX: ['mov', 'r2', 0]
EX/MEM: ['mov', 'r1', 1]
MEM/WB: None

Current Pipeline State:

IF/ID: ['mov', 'r1', 1]
ID/EX: None
EX/MEM: None
MEM/WB: None

Current Pipeline State:

IF/ID: ['add', 'r1', 'r1', 1]
ID/EX: ['mov', 'r3', 3]
EX/MEM: ['mov', 'r2', 0]
MEM/WB: ['mov', 'r1', 1]

Current Pipeline State:

IF/ID: ['mov', 'r2', 0]
ID/EX: ['mov', 'r1', 1]
EX/MEM: None
MEM/WB: None

Current Pipeline State:

IF/ID: ['add', 'r1', 'r1', 'r2']
ID/EX: ['add', 'r1', 'r1', 1]
EX/MEM: ['mov', 'r3', 3]
MEM/WB: ['mov', 'r2', 0]

Current Pipeline State:

IF/ID: ['mul', 'r1', 'r1', 2]
ID/EX: ['add', 'r1', 'r1', 'r2']
EX/MEM: ['add', 'r1', 'r1', 1]
MEM/WB: ['mov', 'r3', 3]

Current Pipeline State:

IF/ID: ['div', 'r4', 'r3', 'r2']
ID/EX: ['mul', 'r1', 'r1', 2]
EX/MEM: ['add', 'r1', 'r1', 'r2']
MEM/WB: ['add', 'r1', 'r1', 1]

Current Pipeline State:

IF/ID: ['add', 'r2', 'r1', 4]
ID/EX: ['div', 'r4', 'r3', 'r2']
EX/MEM: ['mul', 'r1', 'r1', 2]
MEM/WB: ['add', 'r1', 'r1', 'r2']
Division by zero error! Skipping instruction: ['div', 'r4', 'r3', 'r2']

Current Pipeline State:

IF/ID: None
ID/EX: ['add', 'r2', 'r1', 4]
EX/MEM: ['div', 'r4', 'r3', 'r2']
MEM/WB: ['mul', 'r1', 'r1', 2]

Current Pipeline State:

IF/ID: None
ID/EX: None
EX/MEM: ['add', 'r2', 'r1', 4]
MEM/WB: ['div', 'r4', 'r3', 'r2']

Final Pipeline State:

IF/ID: None
ID/EX: None
EX/MEM: None
MEM/WB: ['add', 'r2', 'r1', 4]

Total Clock Cycles: 14.75

Registers: [0, 4, 8, 3, 0, 0, 0]

Total Clock Cycles: 14.75