```
import sys

# 1. Declaration/Initialization
# Dictionary to store data values by their memory addresses
memory_ary = {}
# Mapping to associate instructions with their program counter
instruction_mp = {}
# Flag to indicate if data section has started
data_flag = False
# Array to simulate 32 registers
registers = [0] * 32
# Initial program counter starting from address 256
program_counter = 256
# Function to calculate two's complement for negative numbers
def twos_complement(binary_instruction, bits=32):
    return str(int(binary_instruction, 2) - (1 << bits))



# 2. Disassemble functions
# Function to disassemble category 1 instructions
def disassemble_cat1(binary_instruction):
    # Mapping of opcodes to instruction
    opcode_dict = {
        "00000": "beq",
        "00001": "bne",
        "00010": "blt",
        "00011": "sw"
    }

    # Extracting different parts of the instruction
    opcode = binary_instruction[25:30]
    rs1 = int(binary_instruction[12:17], 2)
    rs2 = int(binary_instruction[7:12], 2)
    imm_11_5 = int(binary_instruction[0:7], 2)
    imm_4_0 = int(binary_instruction[20:25], 2)
    # Combine the immediate parts to get the full offset
    offset = (imm_11_5 << 5) | imm_4_0

    # Returning the disassembled instruction based on the opcode
    if opcode in opcode_dict:
        instruction = opcode_dict[opcode]
        if instruction == "sw":
            return f"{instruction} x{rs1}, {offset}(x{rs2})"
        else:
            return f"{instruction} x{rs1}, x{rs2}, #{offset}"

# Function to disassemble category 2 instructions
def disassemble_cat2(instruction):
    opcode = instruction[25:30]
    rd = int(instruction[20:25], 2)
    rs1 = int(instruction[12:17], 2)
    rs2 = int(instruction[7:12], 2)

    # Mapping of opcodes to instruction
    opcode_map = {
        "00000": "add",
        "00001": "sub",
        "00010": "and",
```

```
        "00011": "or",
    }

    # Returning the disassembled instruction
    if opcode in opcode_map:
        mn_inst_code = opcode_map[opcode]
        disassembled = f"{mn_inst_code} x{rd}, x{rs1}, x{rs2}"
        return disassembled

# Function to disassemble category 3 instructions
def disassemble_cat3(binary_instruction):
    opcode_dict = {
        "00000": "addi",
        "00001": "andi",
        "00010": "ori",
        "00011": "sll",
        "00100": "sra",
        "00101": "lw"
    }

    opcode = binary_instruction[25:30]
    rd = int(binary_instruction[20:25], 2)
    rs1 = int(binary_instruction[12:17], 2)
    # Handling immediate values based on the sign bit
    if int(binary_instruction[0]) == 1:
        imm = int(twos_complement(binary_instruction[0:12], 12),12)
    else:
        imm = int(binary_instruction[0:12], 2)

    if opcode in opcode_dict:
        instruction = opcode_dict[opcode]
        if instruction == "sll" or instruction == "sra":
            return f"{instruction} x{rd}, x{rs1}, #{imm}"
        elif instruction == "lw":
            return f"{instruction} x{rd}, {imm}(x{rs1})"
        else:
            return f"{instruction} x{rd}, x{rs1}, #{imm}"

# Function to disassemble category 4 instructions
def disassemble_cat4(binary_instruction):
    opcode_dict = {
        "00000": "jal",
        "11111": "break"
    }

    opcode = binary_instruction[25:30]
    rd = int(binary_instruction[20:25], 2)
    if int(binary_instruction[0]) == 1:
        imm = twos_complement(binary_instruction[0:20], 20)
    else:
        imm = int(binary_instruction[0:20], 2)

    if opcode in opcode_dict:
        instruction = opcode_dict[opcode]
        if instruction == "break":
            return instruction
        return f"{opcode_dict[opcode]} x{rd}, #{imm}"
```

```
# 3. Execution Functions
# Function to execute category 1 instructions
def execute_cat1(instruction, program_counter):
    opcode = instruction[25:30]
    rs1 = int(instruction[12:17], 2)
    rs2 = int(instruction[7:12], 2)
    imm_bits = instruction[0:7] + instruction[20:25]
    # Calculate the offset
    offset = int(imm_bits, 2)

    offset <<= 1
    if offset & 0x1000: # Check for sign bit
        offset |= 0xFFFFE000

    # Update the program counter based on the opcode
    if opcode == "00000" and registers[rs1] == registers[rs2]:
        program_counter = program_counter + offset
    elif opcode == "00001" and registers[rs1] != registers[rs2]:
        program_counter = program_counter + offset
    elif opcode == "00010" and registers[rs1] < registers[rs2]:
        program_counter = program_counter + offset
    elif opcode == "00011":
        imm_11_5 = int(instruction[0:7], 2)
        imm_4_0 = int(instruction[20:25], 2)
        offset = (imm_11_5 << 5) | imm_4_0
        memory_address = registers[rs2] + offset
        data_to_store = registers[rs1]
        memory_ary[memory_address] = data_to_store
    return program_counter

# Function to execute category 2 instructions
def execute_cat2(instruction):
    opcode = instruction[25:30]
    rd = int(instruction[20:25], 2)
    rs1 = int(instruction[12:17], 2)
    rs2 = int(instruction[7:12], 2)

    # Perform operations based on the opcode
    if opcode == "00000":
        registers[rd] = registers[rs1] + registers[rs2]
    elif opcode == "00001":
        registers[rd] = registers[rs1] - registers[rs2]
    elif opcode == "00010":
        registers[rd] = registers[rs1] & registers[rs2]
    elif opcode == "00011":
        registers[rd] = registers[rs1] | registers[rs2]

# Function to execute category 3 instructions
def execute_cat3(instruction):
    opcode = instruction[25:30]
    rd = int(instruction[20:25], 2) # Destination register
    rs1 = int(instruction[12:17], 2)  # Source register 1

    # Handle immediate value based on the sign bit
    if int(instruction[0]) == 1:
        immediate = int(twos_complement(instruction[0:12], 12), 12)
    else:
        immediate = int(instruction[0:12], 2)
```

```
    # Perform operations based on the opcode
    if opcode == "00000":
        registers[rd] = registers[rs1] + immediate
    elif opcode == "00001":
        registers[rd] = registers[rs1] & immediate
    elif opcode == "00010":
        registers[rd] = registers[rs1] | immediate
    elif opcode == "00011":
        registers[rd] = registers[rs1] << immediate
    elif opcode == "00100":
        registers[rd] = registers[rs1] >> immediate
    elif opcode == "00101":
        registers[rd] = memory_ary[immediate + registers[rs1]]

# Function to execute Category 4 instructions
def execute_cat4(binary_instruction, program_counter):
    opcode = binary_instruction[25:30]
    rd = int(binary_instruction[20:25], 2)
    # Check the first bit to determine if the immediate value is signed
    if int(binary_instruction[0]) == 1:
        # If the first bit is 1, interpret the next 20 bits as a signed integer
        imm = twos_complement(binary_instruction[0:20], 20)
    else:
        # Otherwise, interpret the immediate value as an unsigned integer
        imm = int(binary_instruction[0:20], 2)

    if opcode == "00000":
        registers[rd] = program_counter + 4
        program_counter = program_counter + 2 * int(imm)
        return program_counter
    else:
        return 0



# 4. Handling Input/Output structure
# Read the input sample file name in my_input from command line arguments
my_input=sys.argv[1]

input_file=open(my_input,"r")
disassembly_output = open("disassembly.txt", "w")


# Main loop to read instructions from the input file
while True:
    string_read2 = input_file.readline()
    instruction = string_read2.strip()
    # Break the loop if no more instructions are found
    if not instruction:
        break
    disassembled_instruction = ""

    # If data_flag is true, we are handling data instructions
    if data_flag:
        # Check if the instruction's first bit indicates it's a data instruction
        if instruction[0] == "1":
            disassembled_instruction = twos_complement(instruction)
            memory_ary[program_counter] = int(disassembled_instruction)
```

```
        else:
            disassembled_instruction = str(int(instruction, 2))
            memory_ary[program_counter] = int(disassembled_instruction)
    else:
        # Handle different instruction categories based on the last two bits
        ins_type_lt2 = instruction[-2:]
        if ins_type_lt2 == "11":
            disassembled_instruction = disassemble_cat1(instruction)
        elif ins_type_lt2 == "01":
            disassembled_instruction = disassemble_cat2(instruction)
        elif ins_type_lt2 == "10":
            disassembled_instruction = disassemble_cat3(instruction)
        elif ins_type_lt2 == "00":
            break_check = disassemble_cat4(instruction)
            if break_check == "break":
                disassembled_instruction = "break"
                data_flag = True # Set the data flag to true for data instructions
                minimum_daddress = program_counter + 4 # Update minimum data
address
            else:
                disassembled_instruction = break_check

    # Map the current program counter to the instruction and its disassembled
representation
    instruction_mp[program_counter] = [instruction, disassembled_instruction]
    # Write the instruction, program counter, and disassembled instruction to the
disassembly file
    disassembly_output.write(instruction + "\t" + str(program_counter) + "\t" +
disassembled_instruction + "\n")
    # Increment the program counter by 4 (assuming each instruction is 4 bytes)
    program_counter += 4



# Function to print data starting from a minimum data address
def print_data(minimum_daddress):
    data_i = minimum_daddress
    simulation_output.write("Data\n")
    while True:
        if data_i not in memory_ary:
            return
        data_line = str(data_i) + ":"
        for i in range(0, 8):
            if data_i in memory_ary:
                data_line = data_line + "\t" + str(memory_ary[data_i])
                data_i += 4
            else:
                break
        simulation_output.write(data_line + "\n")


# Initialize simulation parameters
i = 256
cycle = 1
simulation_output = open("simulation.txt", "w")


# Main simulation loop
while True:
```

```
    instruction = instruction_mp[i][0]
    ins_type_lt2 = instruction[-2:]
    simulation_output.write("--------------------\n")
    write_to_file = f"Cycle {cycle}:\t{i}\t{instruction_mp[i][1]}\n"
    simulation_output.write(write_to_file)
    register_line1 = "x00:"
    register_line2 = "x08:"
    register_line3 = "x16:"
    register_line4 = "x24:"
    if ins_type_lt2 == "11":
        y = execute_cat1(instruction, i)
        if y == i:
            i = i + 4
        else:
            i = y
    elif ins_type_lt2 == "01":
        execute_cat2(instruction)
        i = i + 4
    elif ins_type_lt2 == "10":
        execute_cat3(instruction)
        i = i + 4
    elif ins_type_lt2 == "00":
        x = execute_cat4(instruction, i)
        if x == 0:
            # If execution ends without errors, write registers to the simulation
output
            for j in range(0, 8):
                register_line1 += "\t" + str(registers[j])
            for j in range(8, 16):
                register_line2 += "\t" + str(registers[j])
            for j in range(16, 24):
                register_line3 += "\t" + str(registers[j])
            for j in range(24, 32):
                register_line4 += "\t" + str(registers[j])
            simulation_output.write("Registers\n" + register_line1 + "\n")
            simulation_output.write(register_line2 + "\n")
            simulation_output.write(register_line3 + "\n")
            simulation_output.write(register_line4 + "\n")
            print_data(minimum_daddress)
            break
        else:
            i = x # Update the instruction pointer with the new address
    cycle = cycle + 1
    # Write register values after each cycle
    for j in range(0, 8):
        register_line1 += "\t" + str(registers[j])
    for j in range(8, 16):
        register_line2 += "\t" + str(registers[j])
    for j in range(16, 24):
        register_line3 += "\t" + str(registers[j])
    for j in range(24, 32):
        register_line4 += "\t" + str(registers[j])
    simulation_output.write("Registers\n" + register_line1 + "\n")
    simulation_output.write(register_line2 + "\n")
    simulation_output.write(register_line3 + "\n")
    simulation_output.write(register_line4 + "\n")

    print_data(minimum_daddress)
```

```
# Close all opened files
input_file.close()
disassembly_output.close()
simulation_output.close()
```