Report 8

Sadrodin Fayazi (401102242)

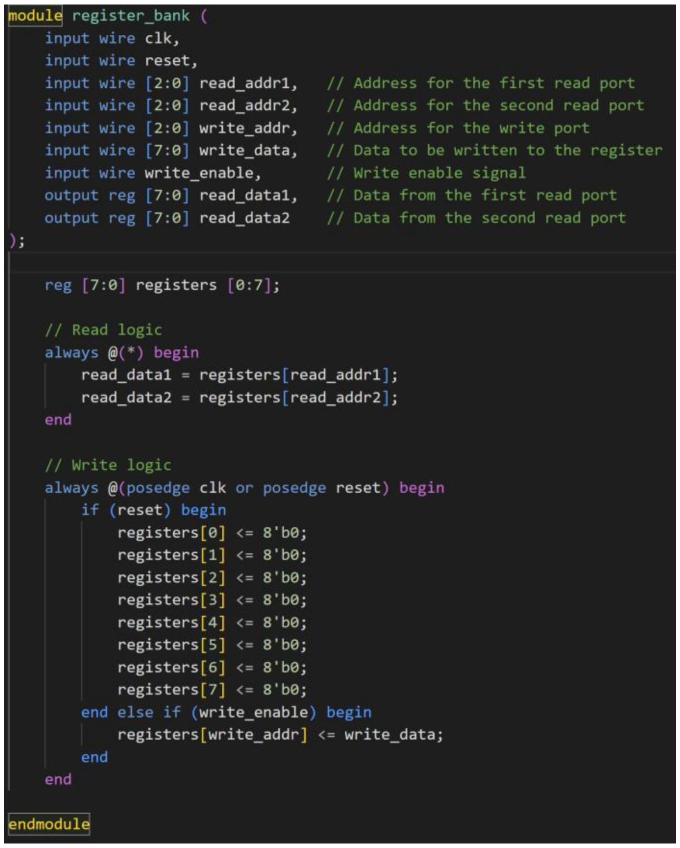
Abbas Naimi (401102723)

At first we will explain modules that we used in the code:

Module: `register_bank`

This module defines a register bank with 8 registers, each 8 bits wide. It supports reading from two ports and writing to one port.

- **Inputs**:
  - `clk`: Clock signal.
  - `reset`: Reset signal.
  - `read_addr1`, `read_addr2`: 3-bit addresses for reading.
  - `write_addr`: 3-bit address for writing.
  - `write_data`: 8-bit data to be written.
  - `write_enable`: Write enable signal.
- **Outputs**:
  - `read_data1`, `read_data2`: 8-bit data outputs for the read ports.
- **Functionality**:
  - **Read Logic**: Combinational logic that continuously outputs the data from the addresses specified by `read_addr1` and `read_addr2`.
  - **Write Logic**: Sequential logic that writes `write_data` to `registers[write_addr]` on the rising edge of `clk` if `write_enable` is high, or resets all registers if `reset` is high.

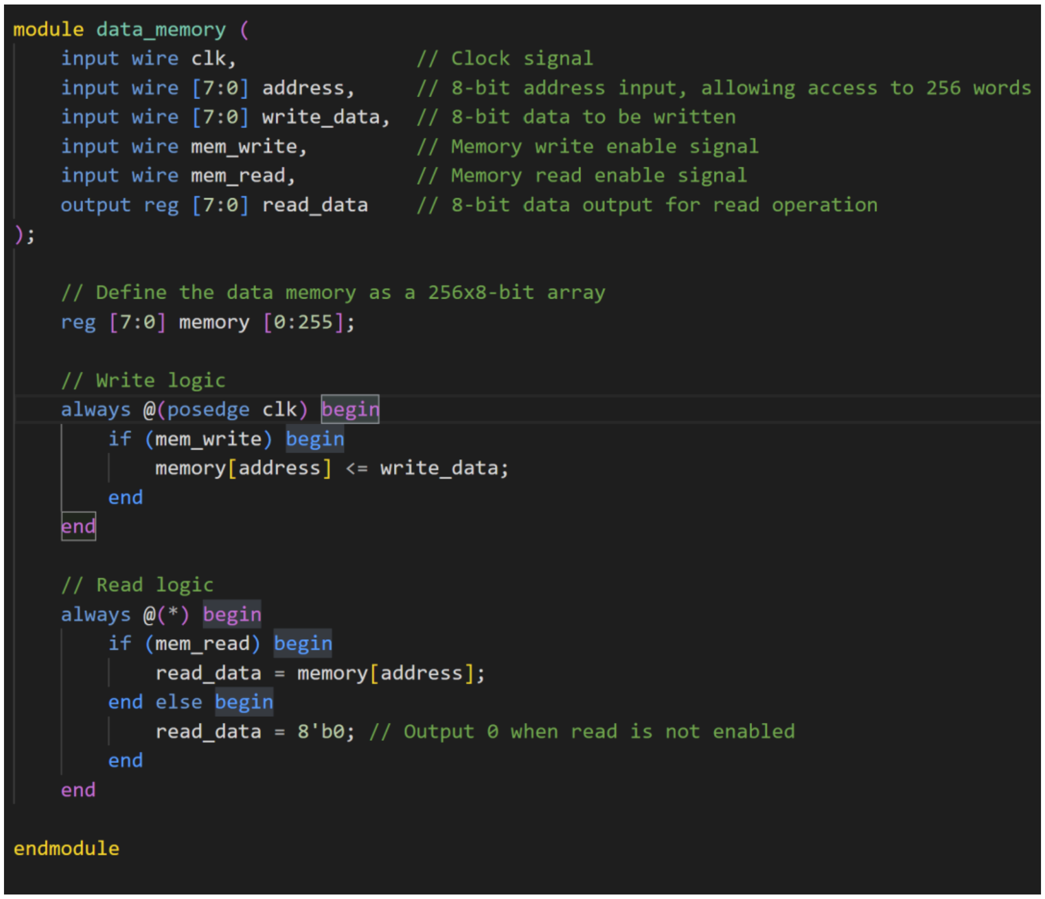


```
module register_bank (
    input wire clk,
    input wire reset,
    input wire [2:0] read_addr1,   // Address for the first read port
    input wire [2:0] read_addr2,   // Address for the second read port
    input wire [2:0] write_addr,   // Address for the write port
    input wire [7:0] write_data,   // Data to be written to the register
    input wire write_enable,       // Write enable signal
    output reg [7:0] read_data1,   // Data from the first read port
    output reg [7:0] read_data2    // Data from the second read port
);

    reg [7:0] registers [0:7];

    // Read logic
    always @(*) begin
        read_data1 = registers[read_addr1];
        read_data2 = registers[read_addr2];
    end

    // Write logic
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            registers[0] <= 8'b0;
            registers[1] <= 8'b0;
            registers[2] <= 8'b0;
            registers[3] <= 8'b0;
            registers[4] <= 8'b0;
            registers[5] <= 8'b0;
            registers[6] <= 8'b0;
            registers[7] <= 8'b0;
        end else if (write_enable) begin
            registers[write_addr] <= write_data;
        end
    end

endmodule
```

Module: `data_memory`

This module defines a data memory with 256 words, each 8 bits wide.

- **Inputs**:
  - `clk`: Clock signal.
  - `address`: 8-bit address input.
  - `write_data`: 8-bit data to be written.
  - `mem_write`: Memory write enable signal.
  - `mem_read`: Memory read enable signal.
- **Outputs**:
  - `read_data`: 8-bit data output for read operations.
- **Functionality**:
  - **Write Logic**: Sequential logic that writes `write_data` to `memory[address]` on the rising edge of `clk` if `mem_write` is high.
  - **Read Logic**: Combinational logic that outputs `memory[address]` if `mem_read` is high, or outputs `8'b0` otherwise.

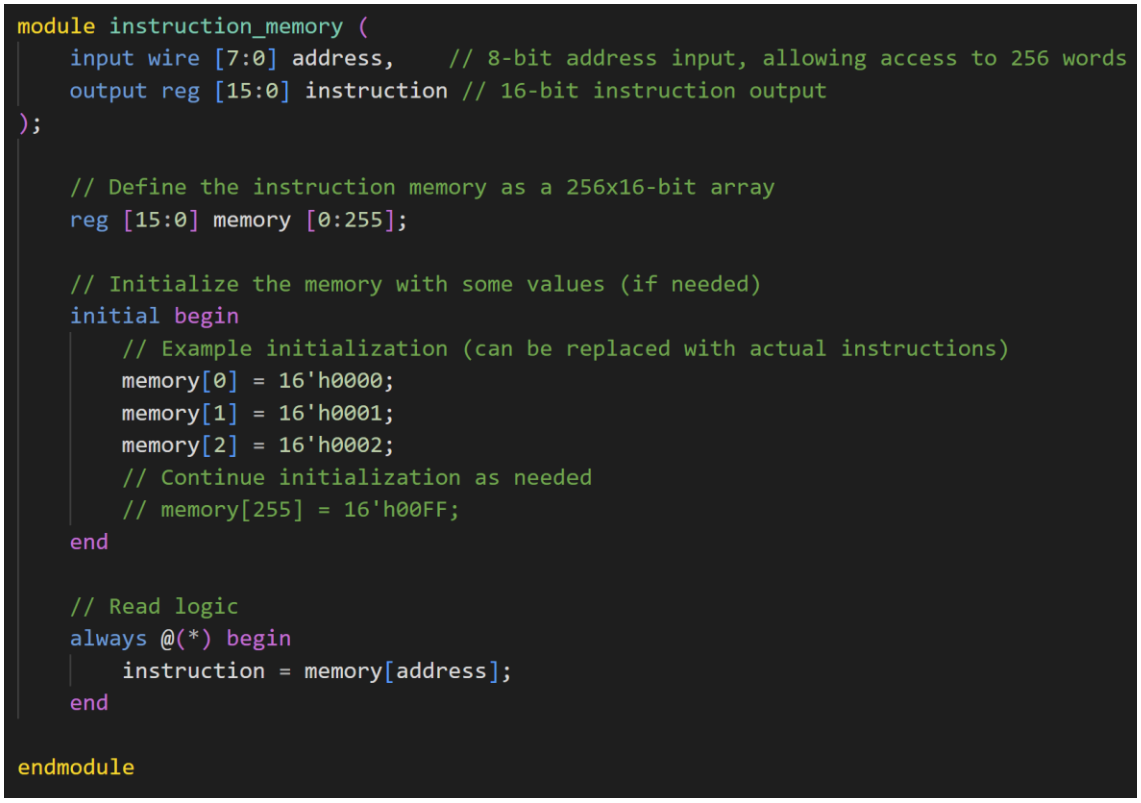


```
module data_memory (
    input wire clk,                 // Clock signal
    input wire [7:0] address,       // 8-bit address input, allowing access to 256 words
    input wire [7:0] write_data,    // 8-bit data to be written
    input wire mem_write,           // Memory write enable signal
    input wire mem_read,            // Memory read enable signal
    output reg [7:0] read_data      // 8-bit data output for read operation
);

    // Define the data memory as a 256x8-bit array
    reg [7:0] memory [0:255];

    // Write logic
    always @(posedge clk) begin
        if (mem_write) begin
            memory[address] <= write_data;
        end
    end

    // Read logic
    always @(*) begin
        if (mem_read) begin
            read_data = memory[address];
        end else begin
            read_data = 8'b0; // Output 0 when read is not enabled
        end
    end

endmodule
```

Module: `instruction_memory`

This module defines an instruction memory with 256 words, each 16 bits wide.

- **Inputs**:
  - `address`: 8-bit address input.
- **Outputs**:
  - `instruction`: 16-bit instruction output.
- **Functionality**:
  - **Initialization**: Initializes the memory with predefined instructions.
  - **Read Logic**: Combinational logic that outputs `memory[address]`.

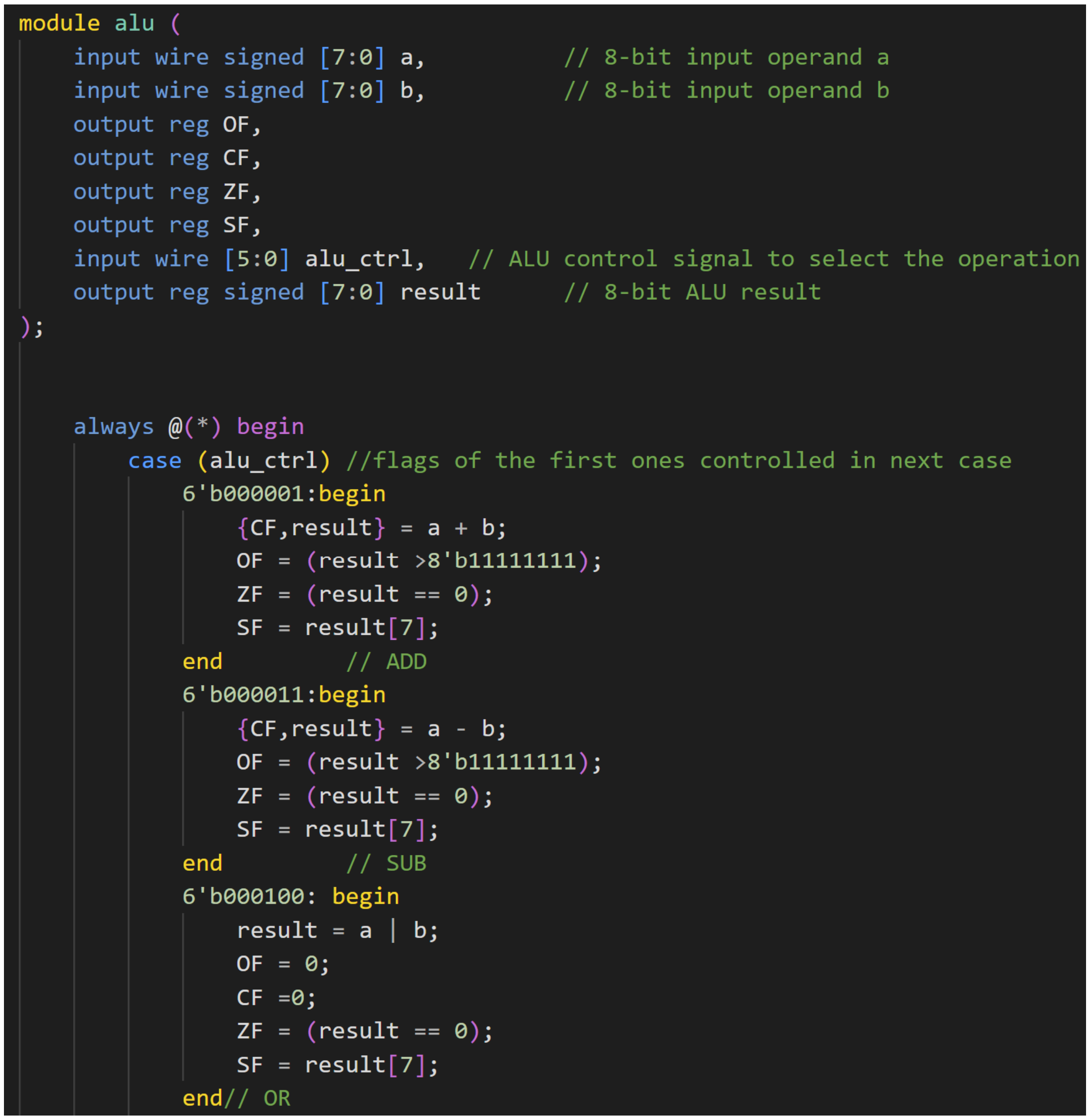


```
module instruction_memory (
    input wire [7:0] address,    // 8-bit address input, allowing access to 256 words
    output reg [15:0] instruction // 16-bit instruction output
);

    // Define the instruction memory as a 256x16-bit array
    reg [15:0] memory [0:255];

    // Initialize the memory with some values (if needed)
    initial begin
        // Example initialization (can be replaced with actual instructions)
        memory[0] = 16'h0000;
        memory[1] = 16'h0001;
        memory[2] = 16'h0002;
        // Continue initialization as needed
        // memory[255] = 16'h00FF;
    end

    // Read logic
    always @(*) begin
        instruction = memory[address];
    end

endmodule
```

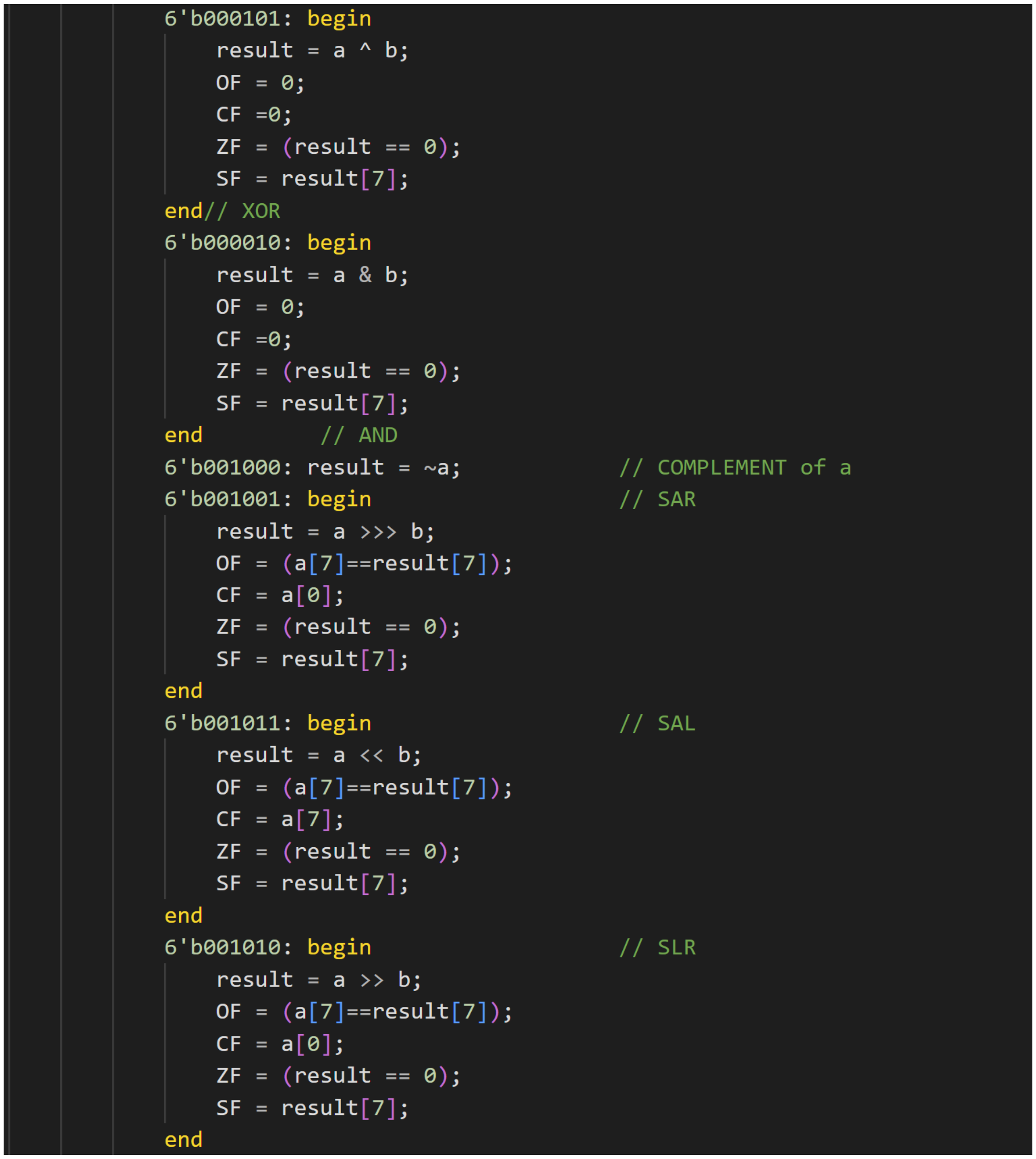
Module: `alu`

This module defines an Arithmetic Logic Unit (ALU) for performing various operations on two 8-bit signed inputs.

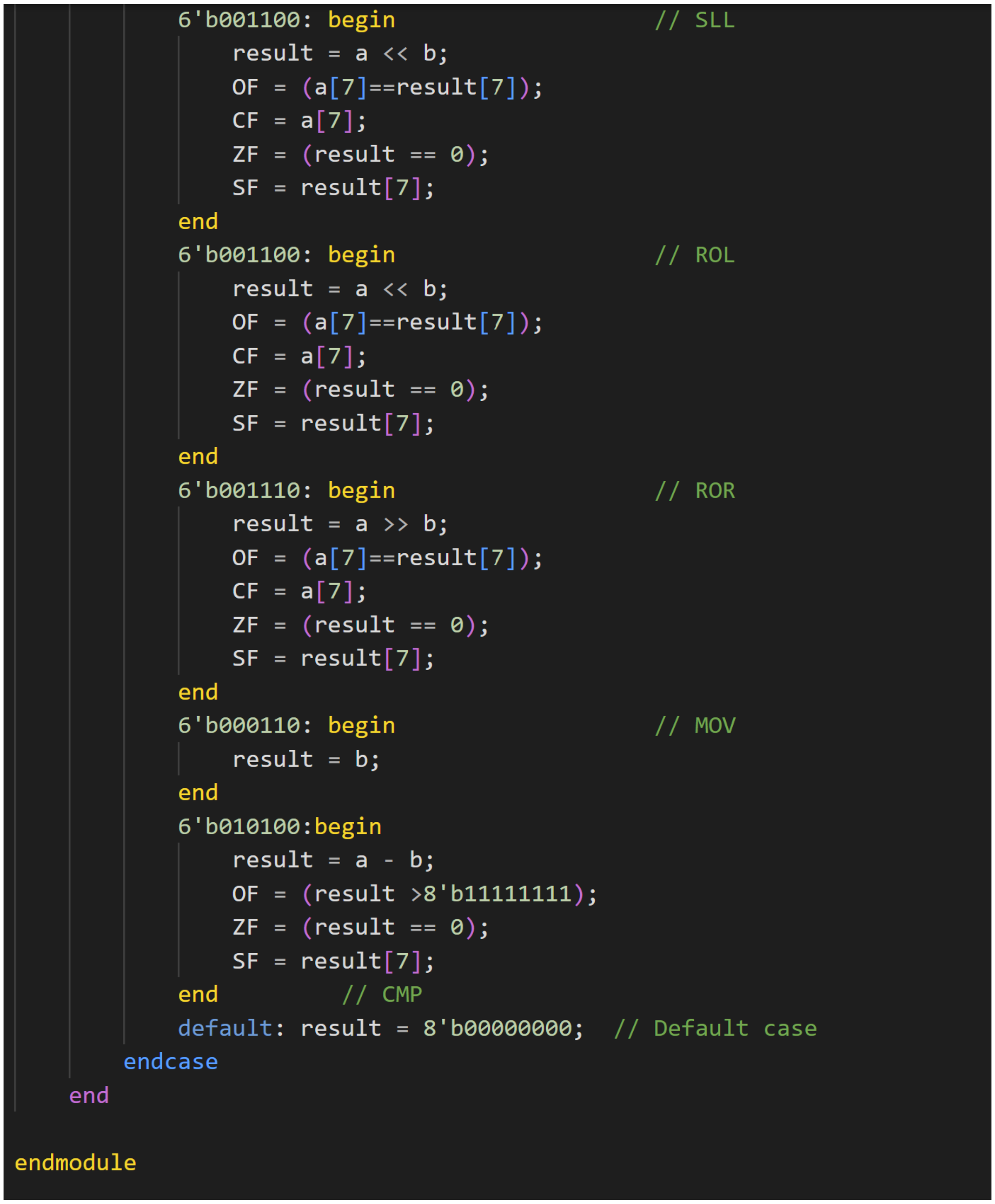
- **Inputs**:
  - `a`, `b`: 8-bit signed input operands.
  - `alu_ctrl`: 6-bit control signal to select the operation.
- **Outputs**:
  - `OF`, `CF`, `ZF`, `SF`: Flags for Overflow, Carry, Zero, and Sign.
  - `result`: 8-bit signed result of the operation.
- **Functionality**:
  - Based on `alu_ctrl`, the ALU performs operations such as addition, subtraction, bitwise operations, shifts, rotates, and comparisons, and sets the corresponding flags.



```
module alu (
    input wire signed [7:0] a,          // 8-bit input operand a
    input wire signed [7:0] b,          // 8-bit input operand b
    output reg OF,
    output reg CF,
    output reg ZF,
    output reg SF,
    input wire [5:0] alu_ctrl,   // ALU control signal to select the operation
    output reg signed [7:0] result      // 8-bit ALU result
);


    always @(*) begin
        case (alu_ctrl) //flags of the first ones controlled in next case
            6'b000001:begin
                {CF,result} = a + b;
                OF = (result >8'b11111111);
                ZF = (result == 0);
                SF = result[7];
            end         // ADD
            6'b000011:begin
                {CF,result} = a - b;
                OF = (result >8'b11111111);
                ZF = (result == 0);
                SF = result[7];
            end         // SUB
            6'b000100: begin
                result = a | b;
                OF = 0;
                CF =0;
                ZF = (result == 0);
                SF = result[7];
            end// OR
```



```
            6'b000101: begin
                result = a ^ b;
                OF = 0;
                CF =0;
                ZF = (result == 0);
                SF = result[7];
            end// XOR
            6'b000010: begin
                result = a & b;
                OF = 0;
                CF =0;
                ZF = (result == 0);
                SF = result[7];
            end          // AND
            6'b001000: result = ~a;             // COMPLEMENT of a
            6'b001001: begin                    // SAR
                result = a >>> b;
                OF = (a[7]==result[7]);
                CF = a[0];
                ZF = (result == 0);
                SF = result[7];
            end
            6'b001011: begin                    // SAL
                result = a << b;
                OF = (a[7]==result[7]);
                CF = a[7];
                ZF = (result == 0);
                SF = result[7];
            end
            6'b001010: begin                    // SLR
                result = a >> b;
                OF = (a[7]==result[7]);
                CF = a[0];
                ZF = (result == 0);
                SF = result[7];
            end
```



```
            6'b001100: begin                   // SLL
                result = a << b;
                OF = (a[7]==result[7]);
                CF = a[7];
                ZF = (result == 0);
                SF = result[7];
            end
            6'b001100: begin                   // ROL
                result = a << b;
                OF = (a[7]==result[7]);
                CF = a[7];
                ZF = (result == 0);
                SF = result[7];
            end
            6'b001110: begin                   // ROR
                result = a >> b;
                OF = (a[7]==result[7]);
                CF = a[7];
                ZF = (result == 0);
                SF = result[7];
            end
            6'b000110: begin                   // MOV
                result = b;
            end
            6'b010100:begin
                result = a - b;
                OF = (result >8'b11111111);
                ZF = (result == 0);
                SF = result[7];
            end           // CMP
            default: result = 8'b00000000;  // Default case
        endcase
    end

endmodule
```

Module: `cpu`

This module defines a simple CPU using the previously defined modules.

- **Inputs**:
  - `clk`: Clock signal.
  - `reset`: Reset signal.
- **Internal Components**:
  - `pc`: 8-bit Program Counter.
  - `instruction`: 16-bit instruction fetched from `instruction_memory`.
  - `alu_ctrl`: ALU control signal derived from the instruction.
  - `reg_file`: 8-register file.
  - `alu_result`: 8-bit result from the ALU.
  - `alu_a`, `alu_b`: ALU operands.
  - `address`: Address for memory operations.
  - Flags `OF`, `ZF`, `CF`, `SF`: Overflow, Zero, Carry, and Sign flags.
- **Sub-modules**:
  - `instruction_memory`: For fetching instructions.
  - `alu`: For performing arithmetic and logical operations.
  - `register_bank`: For managing registers.
  - `data_memory`: For data storage.
- **Functionality**:
  - **Fetch**: Fetches instructions from `instruction_memory` based on `pc`.
  - **Decode and Execute**: Decodes instructions and executes them using ALU and memory operations.
  - **Update PC**: Updates the Program Counter based on instruction type and condition flags.
  - **Immediate Extraction**: Extracts immediate values from instructions.
  - **ALU Operations**: Uses ALU to perform operations and set flags.
  - **Register Operations**: Reads and writes to the register bank and data memory based on the decoded instructions.

The `cpu` module orchestrates the overall behavior, utilizing the `register_bank`, `data_memory`, `instruction_memory`, and `alu` modules to perform its operations.

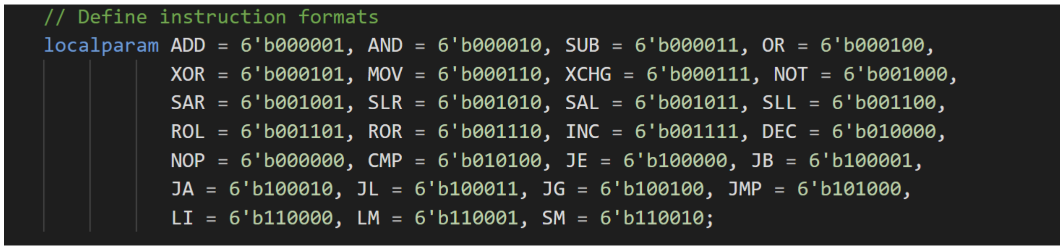
This design represents a simple CPU architecture with a focus on instruction execution, register management, and arithmetic/logical operations.

You can take a look at code in the file, all of lines are explained using comments.

We instantiate all of the following modules in cpu module.

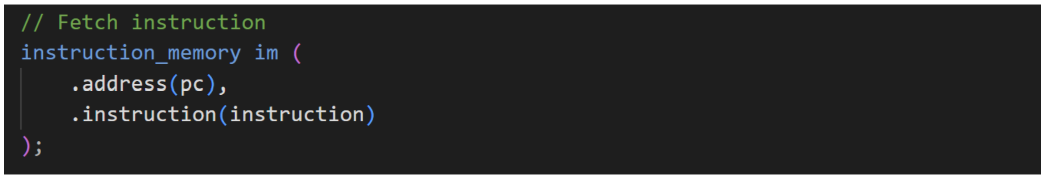
And then we distinguished different input signals using cases.

These are all the instructions, opcodes:



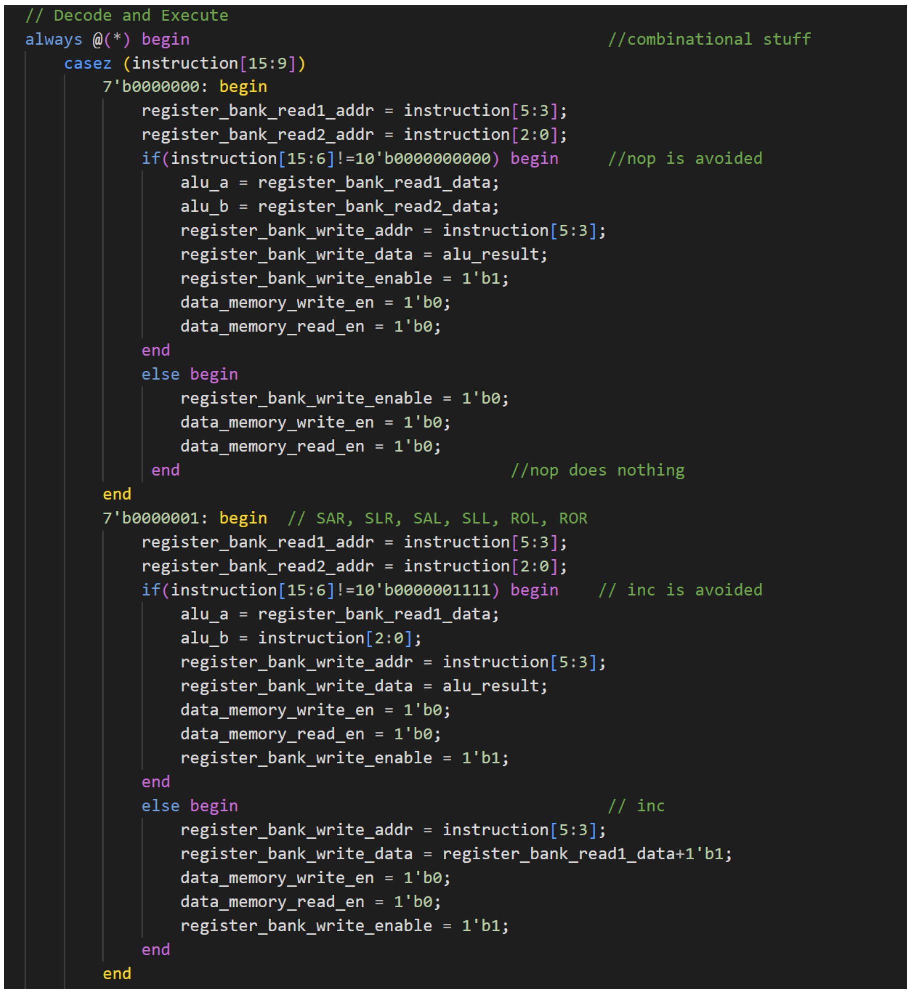
```
// Define instruction formats
localparam ADD = 6'b000001, AND = 6'b000010, SUB = 6'b000011, OR = 6'b000100,
           XOR = 6'b000101, MOV = 6'b000110, XCHG = 6'b000111, NOT = 6'b001000,
           SAR = 6'b001001, SLR = 6'b001010, SAL = 6'b001011, SLL = 6'b001100,
           ROL = 6'b001101, ROR = 6'b001110, INC = 6'b001111, DEC = 6'b010000,
           NOP = 6'b000000, CMP = 6'b010100, JE = 6'b100000, JB = 6'b100001,
           JA = 6'b100010, JL = 6'b100011, JG = 6'b100100, JMP = 6'b101000,
           LI = 6'b110000, LM = 6'b110001, SM = 6'b110010;
```

Fetch implementation:

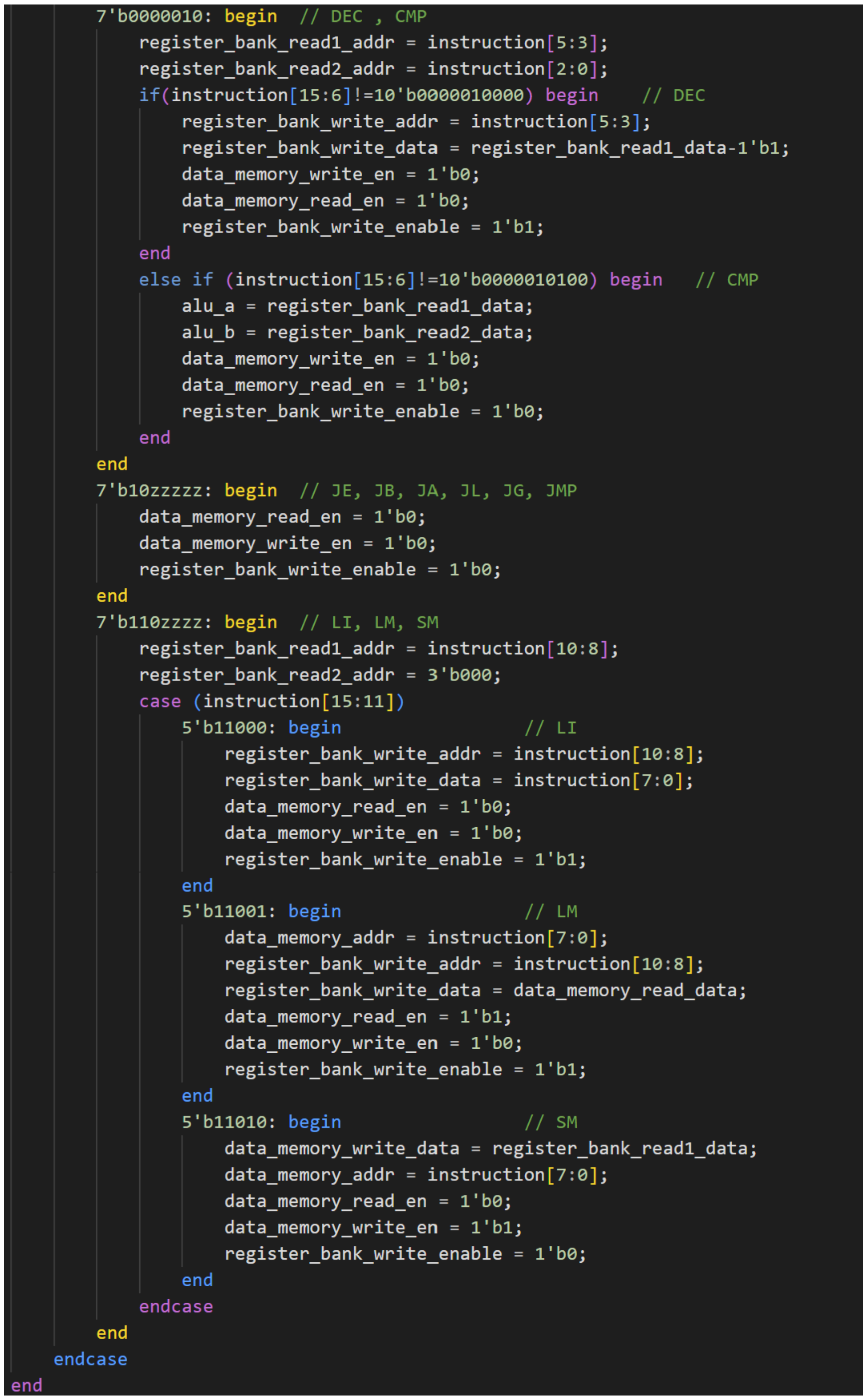


```
// Fetch instruction
instruction_memory im (
    .address(pc),
    .instruction(instruction)
);
```

Decode and execution implementation(the combinational part):

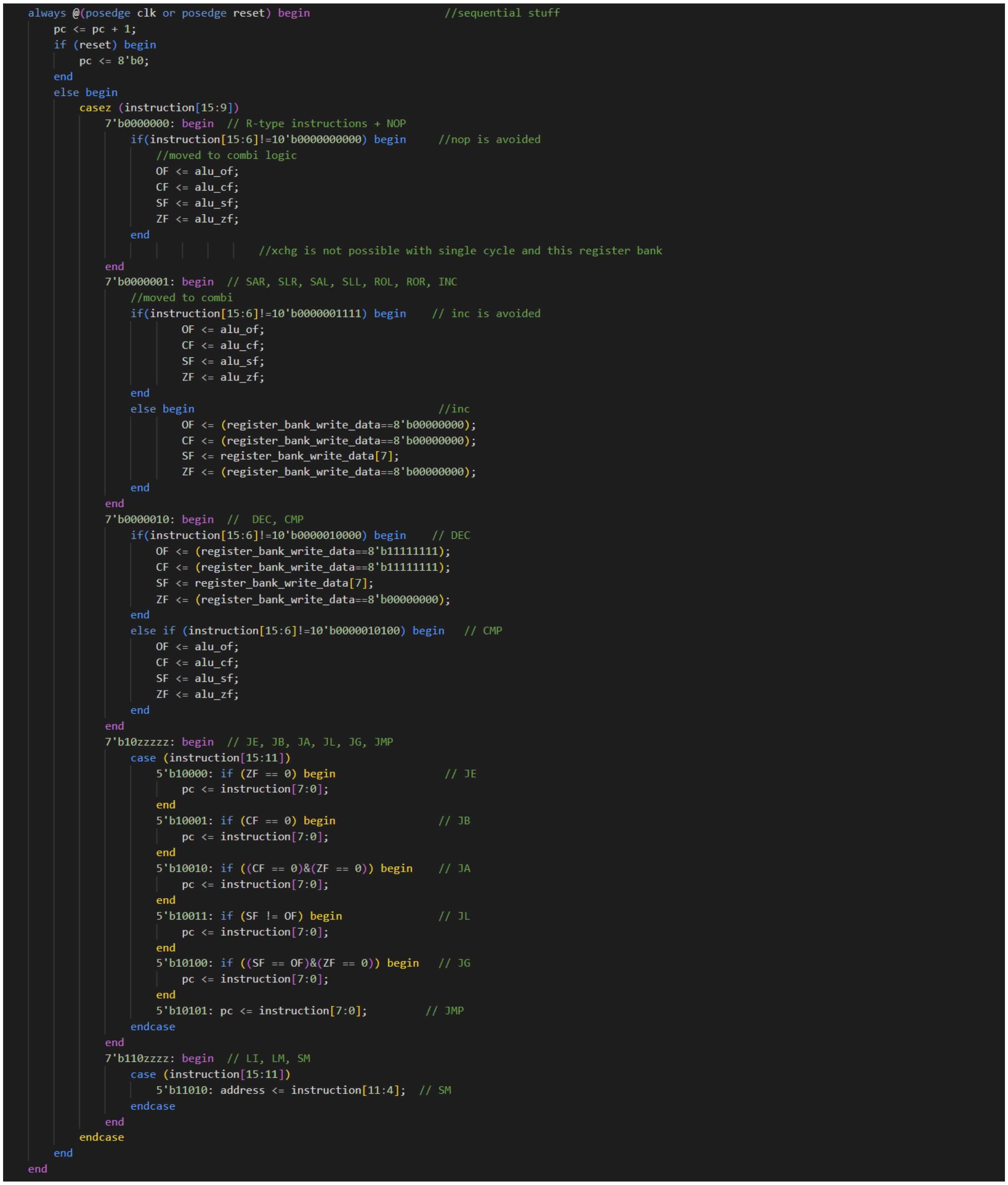


```
// Decode and Execute
always @(*) begin                                           //combinational stuff
    casez (instruction[15:9])
        7'b0000000: begin
            register_bank_read1_addr = instruction[5:3];
            register_bank_read2_addr = instruction[2:0];
            if(instruction[15:6]!=10'b0000000000) begin     //nop is avoided
                alu_a = register_bank_read1_data;
                alu_b = register_bank_read2_data;
                register_bank_write_addr = instruction[5:3];
                register_bank_write_data = alu_result;
                register_bank_write_enable = 1'b1;
                data_memory_write_en = 1'b0;
                data_memory_read_en = 1'b0;
            end
            else begin
                register_bank_write_enable = 1'b0;
                data_memory_write_en = 1'b0;
                data_memory_read_en = 1'b0;
             end                                  //nop does nothing
        end
        7'b0000001: begin  // SAR, SLR, SAL, SLL, ROL, ROR
            register_bank_read1_addr = instruction[5:3];
            register_bank_read2_addr = instruction[2:0];
            if(instruction[15:6]!=10'b0000001111) begin    // inc is avoided
                alu_a = register_bank_read1_data;
                alu_b = instruction[2:0];
                register_bank_write_addr = instruction[5:3];
                register_bank_write_data = alu_result;
                data_memory_write_en = 1'b0;
                data_memory_read_en = 1'b0;
                register_bank_write_enable = 1'b1;
            end
            else begin                                      // inc
                register_bank_write_addr = instruction[5:3];
                register_bank_write_data = register_bank_read1_data+1'b1;
                data_memory_write_en = 1'b0;
                data_memory_read_en = 1'b0;
                register_bank_write_enable = 1'b1;
            end
        end
```



```
            7'b0000010: begin  // DEC , CMP
                register_bank_read1_addr = instruction[5:3];
                register_bank_read2_addr = instruction[2:0];
                if(instruction[15:6]!=10'b0000010000) begin    // DEC
                    register_bank_write_addr = instruction[5:3];
                    register_bank_write_data = register_bank_read1_data-1'b1;
                    data_memory_write_en = 1'b0;
                    data_memory_read_en = 1'b0;
                    register_bank_write_enable = 1'b1;
                end
                else if (instruction[15:6]!=10'b0000010100) begin   // CMP
                    alu_a = register_bank_read1_data;
                    alu_b = register_bank_read2_data;
                    data_memory_write_en = 1'b0;
                    data_memory_read_en = 1'b0;
                    register_bank_write_enable = 1'b0;
                end
            end
            7'b10zzzzz: begin  // JE, JB, JA, JL, JG, JMP
                data_memory_read_en = 1'b0;
                data_memory_write_en = 1'b0;
                register_bank_write_enable = 1'b0;
            end
            7'b110zzzz: begin  // LI, LM, SM
                register_bank_read1_addr = instruction[10:8];
                register_bank_read2_addr = 3'b000;
                case (instruction[15:11])
                    5'b11000: begin                 // LI
                        register_bank_write_addr = instruction[10:8];
                        register_bank_write_data = instruction[7:0];
                        data_memory_read_en = 1'b0;
                        data_memory_write_en = 1'b0;
                        register_bank_write_enable = 1'b1;
                    end
                    5'b11001: begin                 // LM
                        data_memory_addr = instruction[7:0];
                        register_bank_write_addr = instruction[10:8];
                        register_bank_write_data = data_memory_read_data;
                        data_memory_read_en = 1'b1;
                        data_memory_write_en = 1'b0;
                        register_bank_write_enable = 1'b1;
                    end
                    5'b11010: begin                 // SM
                        data_memory_write_data = register_bank_read1_data;
                        data_memory_addr = instruction[7:0];
                        data_memory_read_en = 1'b0;
                        data_memory_write_en = 1'b1;
                        register_bank_write_enable = 1'b0;
                    end
                endcase
            end
        endcase
end
```

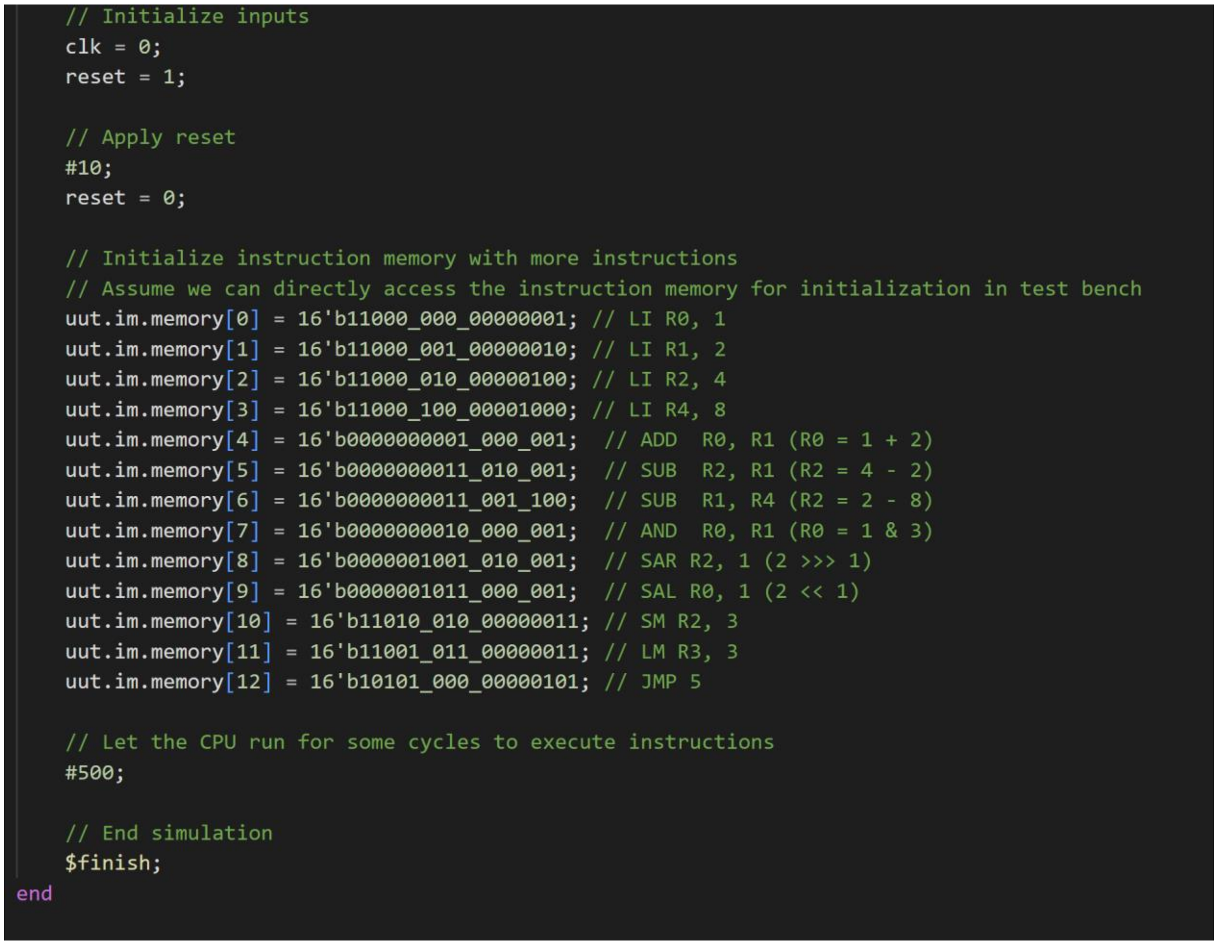
And these are the implementation of the sequential parts:



```
always @(posedge clk or posedge reset) begin                      //sequential stuff
    pc <= pc + 1;
    if (reset) begin
        pc <= 8'b0;
    end
    else begin
        casez (instruction[15:9])
            7'b0000000: begin  // R-type instructions + NOP
                if(instruction[15:6]!=10'b0000000000) begin     //nop is avoided
                    //moved to combi logic
                    OF <= alu_of;
                    CF <= alu_cf;
                    SF <= alu_sf;
                    ZF <= alu_zf;
                end
                                    //xchg is not possible with single cycle and this register bank
            end
            7'b0000001: begin  // SAR, SLR, SAL, SLL, ROL, ROR, INC
                //moved to combi
                if(instruction[15:6]!=10'b0000001111) begin    // inc is avoided
                        OF <= alu_of;
                        CF <= alu_cf;
                        SF <= alu_sf;
                        ZF <= alu_zf;
                end
                else begin                                      //inc
                        OF <= (register_bank_write_data==8'b00000000);
                        CF <= (register_bank_write_data==8'b00000000);
                        SF <= register_bank_write_data[7];
                        ZF <= (register_bank_write_data==8'b00000000);
                end
            end
            7'b0000010: begin  //  DEC, CMP
                if(instruction[15:6]!=10'b0000010000) begin    // DEC
                    OF <= (register_bank_write_data==8'b11111111);
                    CF <= (register_bank_write_data==8'b11111111);
                    SF <= register_bank_write_data[7];
                    ZF <= (register_bank_write_data==8'b00000000);
                end
                else if (instruction[15:6]!=10'b0000010100) begin   // CMP
                    OF <= alu_of;
                    CF <= alu_cf;
                    SF <= alu_sf;
                    ZF <= alu_zf;
                end
            end
            7'b10zzzzz: begin  // JE, JB, JA, JL, JG, JMP
                case (instruction[15:11])
                    5'b10000: if (ZF == 0) begin                 // JE
                        pc <= instruction[7:0];
                    end
                    5'b10001: if (CF == 0) begin                // JB
                        pc <= instruction[7:0];
                    end
                    5'b10010: if ((CF == 0)&(ZF == 0)) begin    // JA
                        pc <= instruction[7:0];
                    end
                    5'b10011: if (SF != OF) begin               // JL
                        pc <= instruction[7:0];
                    end
                    5'b10100: if ((SF == OF)&(ZF == 0)) begin   // JG
                        pc <= instruction[7:0];
                    end
                    5'b10101: pc <= instruction[7:0];         // JMP
                endcase
            end
            7'b110zzzz: begin  // LI, LM, SM
                case (instruction[15:11])
                    5'b11010: address <= instruction[11:4];  // SM
                endcase
            end
        endcase
    end
end
```

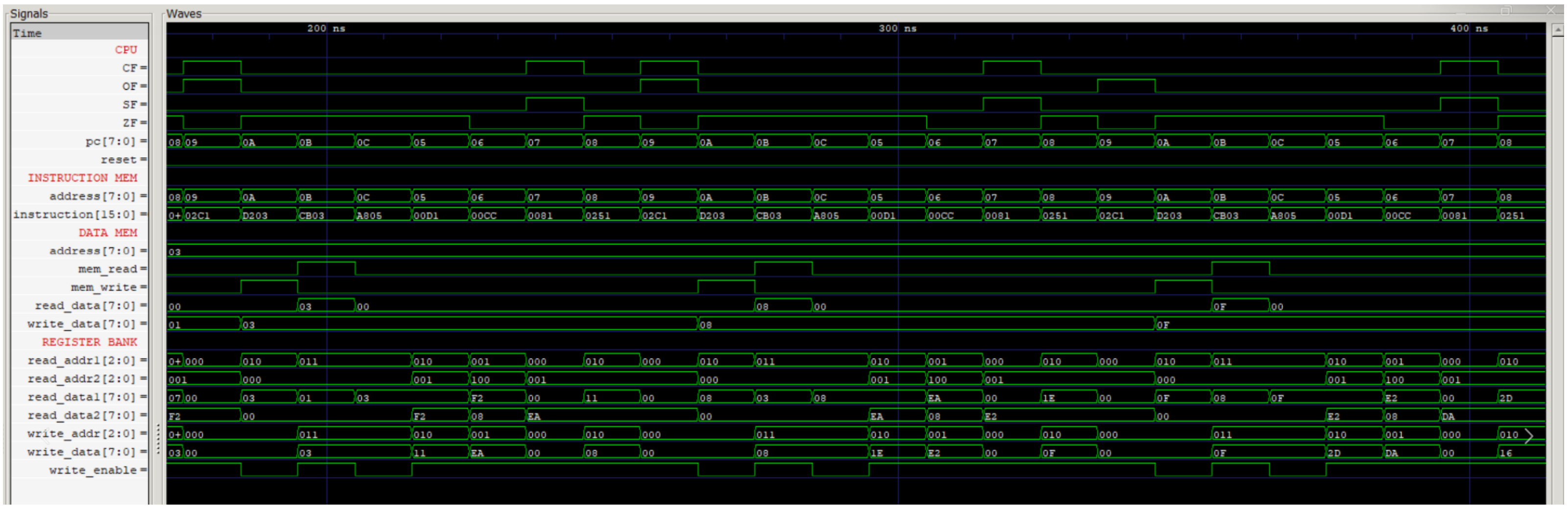
Test Bench

We have all of instructions in our CPU and in test bench we checked one instruction of each group. Here's the instructions we checked in test bench:



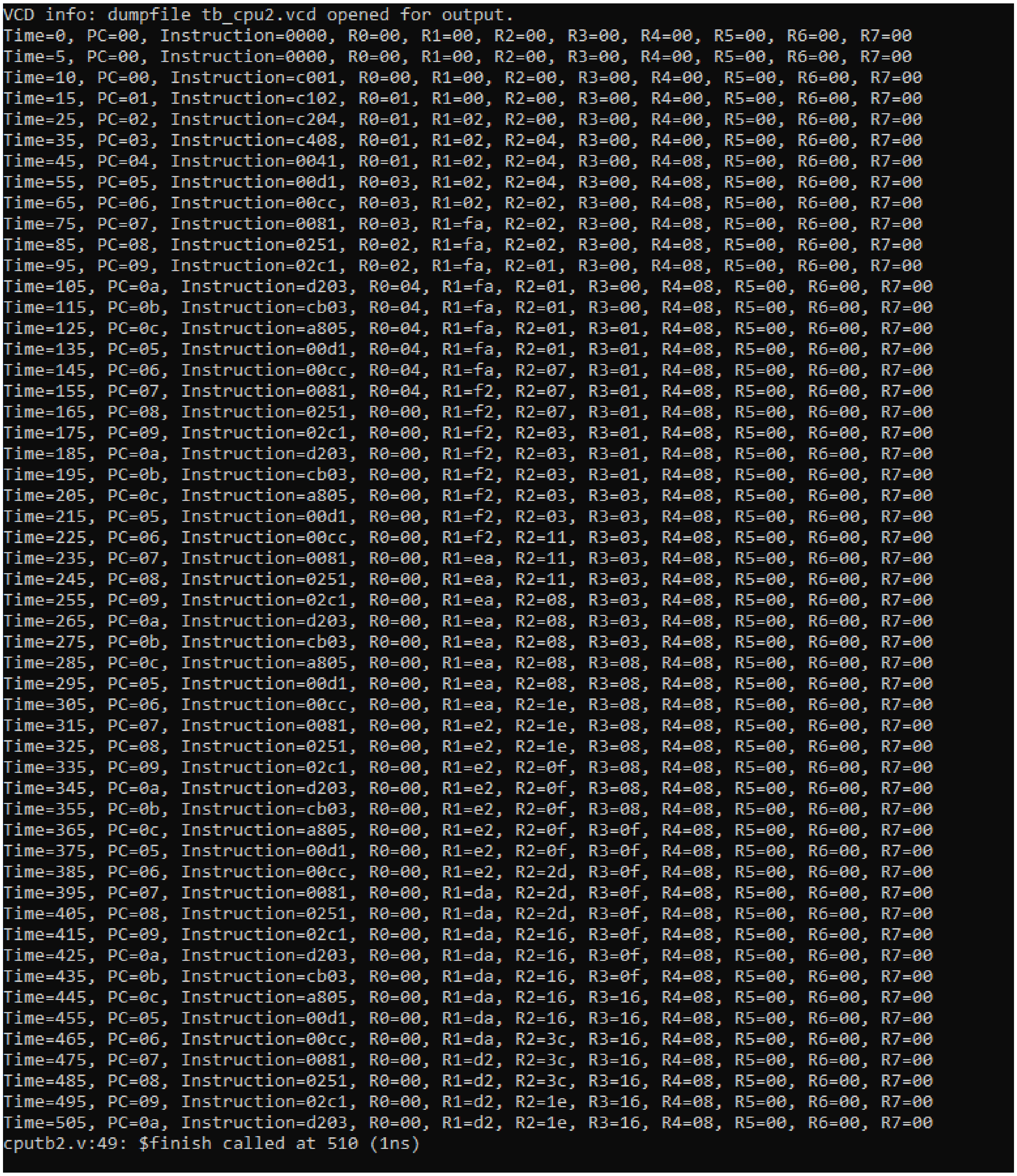
```
    // Initialize inputs
    clk = 0;
    reset = 1;

    // Apply reset
    #10;
    reset = 0;

    // Initialize instruction memory with more instructions
    // Assume we can directly access the instruction memory for initialization in test bench
    uut.im.memory[0] = 16'b11000_000_00000001; // LI R0, 1
    uut.im.memory[1] = 16'b11000_001_00000010; // LI R1, 2
    uut.im.memory[2] = 16'b11000_010_00000100; // LI R2, 4
    uut.im.memory[3] = 16'b11000_100_00001000; // LI R4, 8
    uut.im.memory[4] = 16'b0000000001_000_001;  // ADD  R0, R1 (R0 = 1 + 2)
    uut.im.memory[5] = 16'b0000000011_010_001;  // SUB  R2, R1 (R2 = 4 - 2)
    uut.im.memory[6] = 16'b0000000011_001_100;  // SUB  R1, R4 (R2 = 2 - 8)
    uut.im.memory[7] = 16'b0000000010_000_001;  // AND  R0, R1 (R0 = 1 & 3)
    uut.im.memory[8] = 16'b0000001001_010_001;  // SAR R2, 1 (2 >>> 1)
    uut.im.memory[9] = 16'b0000001011_000_001;  // SAL R0, 1 (2 << 1)
    uut.im.memory[10] = 16'b11010_010_00000011; // SM R2, 3
    uut.im.memory[11] = 16'b11001_011_00000011; // LM R3, 3
    uut.im.memory[12] = 16'b10101_000_00000101; // JMP 5

    // Let the CPU run for some cycles to execute instructions
    #500;

    // End simulation
    $finish;
end
```

And these are the results:



And as you can see this CPU is working correctly.

Here’s a vcd file to understand the test bench better:



```
VCD info: dumpfile tb_cpu2.vcd opened for output.
Time=0, PC=00, Instruction=0000, R0=00, R1=00, R2=00, R3=00, R4=00, R5=00, R6=00, R7=00
Time=5, PC=00, Instruction=0000, R0=00, R1=00, R2=00, R3=00, R4=00, R5=00, R6=00, R7=00
Time=10, PC=00, Instruction=c001, R0=00, R1=00, R2=00, R3=00, R4=00, R5=00, R6=00, R7=00
Time=15, PC=01, Instruction=c102, R0=01, R1=00, R2=00, R3=00, R4=00, R5=00, R6=00, R7=00
Time=25, PC=02, Instruction=c204, R0=01, R1=02, R2=00, R3=00, R4=00, R5=00, R6=00, R7=00
Time=35, PC=03, Instruction=c408, R0=01, R1=02, R2=04, R3=00, R4=00, R5=00, R6=00, R7=00
Time=45, PC=04, Instruction=0041, R0=01, R1=02, R2=04, R3=00, R4=08, R5=00, R6=00, R7=00
Time=55, PC=05, Instruction=00d1, R0=03, R1=02, R2=04, R3=00, R4=08, R5=00, R6=00, R7=00
Time=65, PC=06, Instruction=00cc, R0=03, R1=02, R2=02, R3=00, R4=08, R5=00, R6=00, R7=00
Time=75, PC=07, Instruction=0081, R0=03, R1=fa, R2=02, R3=00, R4=08, R5=00, R6=00, R7=00
Time=85, PC=08, Instruction=0251, R0=02, R1=fa, R2=02, R3=00, R4=08, R5=00, R6=00, R7=00
Time=95, PC=09, Instruction=02c1, R0=02, R1=fa, R2=01, R3=00, R4=08, R5=00, R6=00, R7=00
Time=105, PC=0a, Instruction=d203, R0=04, R1=fa, R2=01, R3=00, R4=08, R5=00, R6=00, R7=00
Time=115, PC=0b, Instruction=cb03, R0=04, R1=fa, R2=01, R3=00, R4=08, R5=00, R6=00, R7=00
Time=125, PC=0c, Instruction=a805, R0=04, R1=fa, R2=01, R3=01, R4=08, R5=00, R6=00, R7=00
Time=135, PC=05, Instruction=00d1, R0=04, R1=fa, R2=01, R3=01, R4=08, R5=00, R6=00, R7=00
Time=145, PC=06, Instruction=00cc, R0=04, R1=fa, R2=07, R3=01, R4=08, R5=00, R6=00, R7=00
Time=155, PC=07, Instruction=0081, R0=04, R1=f2, R2=07, R3=01, R4=08, R5=00, R6=00, R7=00
Time=165, PC=08, Instruction=0251, R0=00, R1=f2, R2=07, R3=01, R4=08, R5=00, R6=00, R7=00
Time=175, PC=09, Instruction=02c1, R0=00, R1=f2, R2=03, R3=01, R4=08, R5=00, R6=00, R7=00
Time=185, PC=0a, Instruction=d203, R0=00, R1=f2, R2=03, R3=01, R4=08, R5=00, R6=00, R7=00
Time=195, PC=0b, Instruction=cb03, R0=00, R1=f2, R2=03, R3=01, R4=08, R5=00, R6=00, R7=00
Time=205, PC=0c, Instruction=a805, R0=00, R1=f2, R2=03, R3=03, R4=08, R5=00, R6=00, R7=00
Time=215, PC=05, Instruction=00d1, R0=00, R1=f2, R2=03, R3=03, R4=08, R5=00, R6=00, R7=00
Time=225, PC=06, Instruction=00cc, R0=00, R1=f2, R2=11, R3=03, R4=08, R5=00, R6=00, R7=00
Time=235, PC=07, Instruction=0081, R0=00, R1=ea, R2=11, R3=03, R4=08, R5=00, R6=00, R7=00
Time=245, PC=08, Instruction=0251, R0=00, R1=ea, R2=11, R3=03, R4=08, R5=00, R6=00, R7=00
Time=255, PC=09, Instruction=02c1, R0=00, R1=ea, R2=08, R3=03, R4=08, R5=00, R6=00, R7=00
Time=265, PC=0a, Instruction=d203, R0=00, R1=ea, R2=08, R3=03, R4=08, R5=00, R6=00, R7=00
Time=275, PC=0b, Instruction=cb03, R0=00, R1=ea, R2=08, R3=03, R4=08, R5=00, R6=00, R7=00
Time=285, PC=0c, Instruction=a805, R0=00, R1=ea, R2=08, R3=08, R4=08, R5=00, R6=00, R7=00
Time=295, PC=05, Instruction=00d1, R0=00, R1=ea, R2=08, R3=08, R4=08, R5=00, R6=00, R7=00
Time=305, PC=06, Instruction=00cc, R0=00, R1=ea, R2=1e, R3=08, R4=08, R5=00, R6=00, R7=00
Time=315, PC=07, Instruction=0081, R0=00, R1=e2, R2=1e, R3=08, R4=08, R5=00, R6=00, R7=00
Time=325, PC=08, Instruction=0251, R0=00, R1=e2, R2=1e, R3=08, R4=08, R5=00, R6=00, R7=00
Time=335, PC=09, Instruction=02c1, R0=00, R1=e2, R2=0f, R3=08, R4=08, R5=00, R6=00, R7=00
Time=345, PC=0a, Instruction=d203, R0=00, R1=e2, R2=0f, R3=08, R4=08, R5=00, R6=00, R7=00
Time=355, PC=0b, Instruction=cb03, R0=00, R1=e2, R2=0f, R3=08, R4=08, R5=00, R6=00, R7=00
Time=365, PC=0c, Instruction=a805, R0=00, R1=e2, R2=0f, R3=0f, R4=08, R5=00, R6=00, R7=00
Time=375, PC=05, Instruction=00d1, R0=00, R1=e2, R2=0f, R3=0f, R4=08, R5=00, R6=00, R7=00
Time=385, PC=06, Instruction=00cc, R0=00, R1=e2, R2=2d, R3=0f, R4=08, R5=00, R6=00, R7=00
Time=395, PC=07, Instruction=0081, R0=00, R1=da, R2=2d, R3=0f, R4=08, R5=00, R6=00, R7=00
Time=405, PC=08, Instruction=0251, R0=00, R1=da, R2=2d, R3=0f, R4=08, R5=00, R6=00, R7=00
Time=415, PC=09, Instruction=02c1, R0=00, R1=da, R2=16, R3=0f, R4=08, R5=00, R6=00, R7=00
Time=425, PC=0a, Instruction=d203, R0=00, R1=da, R2=16, R3=0f, R4=08, R5=00, R6=00, R7=00
Time=435, PC=0b, Instruction=cb03, R0=00, R1=da, R2=16, R3=0f, R4=08, R5=00, R6=00, R7=00
Time=445, PC=0c, Instruction=a805, R0=00, R1=da, R2=16, R3=16, R4=08, R5=00, R6=00, R7=00
Time=455, PC=05, Instruction=00d1, R0=00, R1=da, R2=16, R3=16, R4=08, R5=00, R6=00, R7=00
Time=465, PC=06, Instruction=00cc, R0=00, R1=da, R2=3c, R3=16, R4=08, R5=00, R6=00, R7=00
Time=475, PC=07, Instruction=0081, R0=00, R1=d2, R2=3c, R3=16, R4=08, R5=00, R6=00, R7=00
Time=485, PC=08, Instruction=0251, R0=00, R1=d2, R2=3c, R3=16, R4=08, R5=00, R6=00, R7=00
Time=495, PC=09, Instruction=02c1, R0=00, R1=d2, R2=1e, R3=16, R4=08, R5=00, R6=00, R7=00
Time=505, PC=0a, Instruction=d203, R0=00, R1=d2, R2=1e, R3=16, R4=08, R5=00, R6=00, R7=00
cputb2.v:49: $finish called at 510 (1ns)
```