NATIONAL UNIVERSITY OF SCIENCE AND TECHNALOGY

DEPARTMENT OF COMPUTER AND SOFTWARE ENGINEERING

Lab#13: RISC-V Hazard Detection and Forwarding

Lab Report # 13

Course Instructor: *Dr Shahid Ismail* **Lab Instructor:** *Usama Shoukat*

Sr. No.	Student's Name	CMS ID
01	SHAZIL	532263

Due Date: 16-Dec-2025
Submission Date: 16-Dec-2025

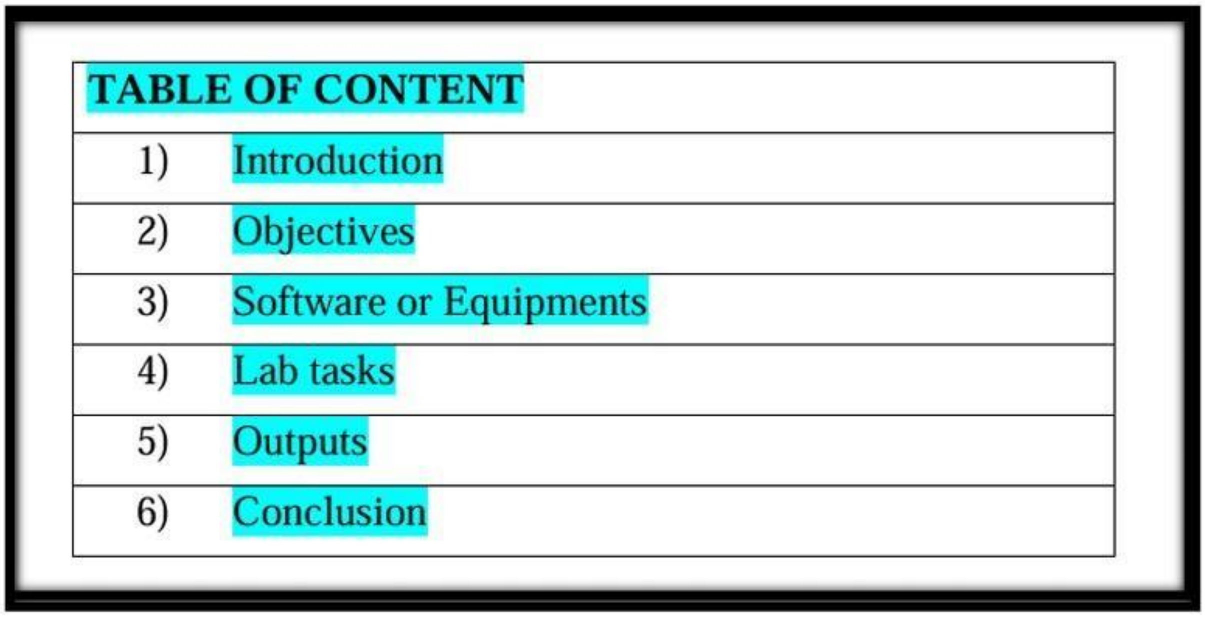


TABLE OF CONTENT

INTRODUCTION:

Pipelining: Control Hazards

A control hazard may occur when the flow of execution of a processor becomes unexpected. In the pipeline designed previously, a control hazard may occur in a branch instruction as shown in the figure below.
Branches cause control hazards because branch decision is delayed, and the processor continues execution until the decision has been made. In other words, a branch predictor exists with the assumption that branch is not taken.

In the case that branch needs to be taken, the processor must flush all the instructions following the branch instruction. To flush, you must disable register and memory writes, however, you may not disable writing of pipeline registers. This can be done in a similar manner to disabling RegWrite in Data Hazards.

The Branch Problem (The Hazard)

The crucial instruction here is the first one: 40 beq x1, x0, 16.

- **beq** stands for "**B**ranch if **eq**ual." It's a *conditional* instruction.

  o **Condition:** Check if the values in registers x1 and x0 are equal. o **Outcome:**

  1. **If they ARE NOT equal (Branch NOT Taken):** The program just continues to the next instruction (44 and...).
  2. **If they ARE equal (Branch IS Taken):** The program jumps to a new instruction at a different address (72 lw x4, 100(x7)).

3. OBJECTIVE:

The objective of this lab is to add hazard detection unit and forwarding unit to already created datapath of Pipeline the Design to Enhance the Number Instructions Executed Per cycle.

SOFTWARE\TOOL USED:

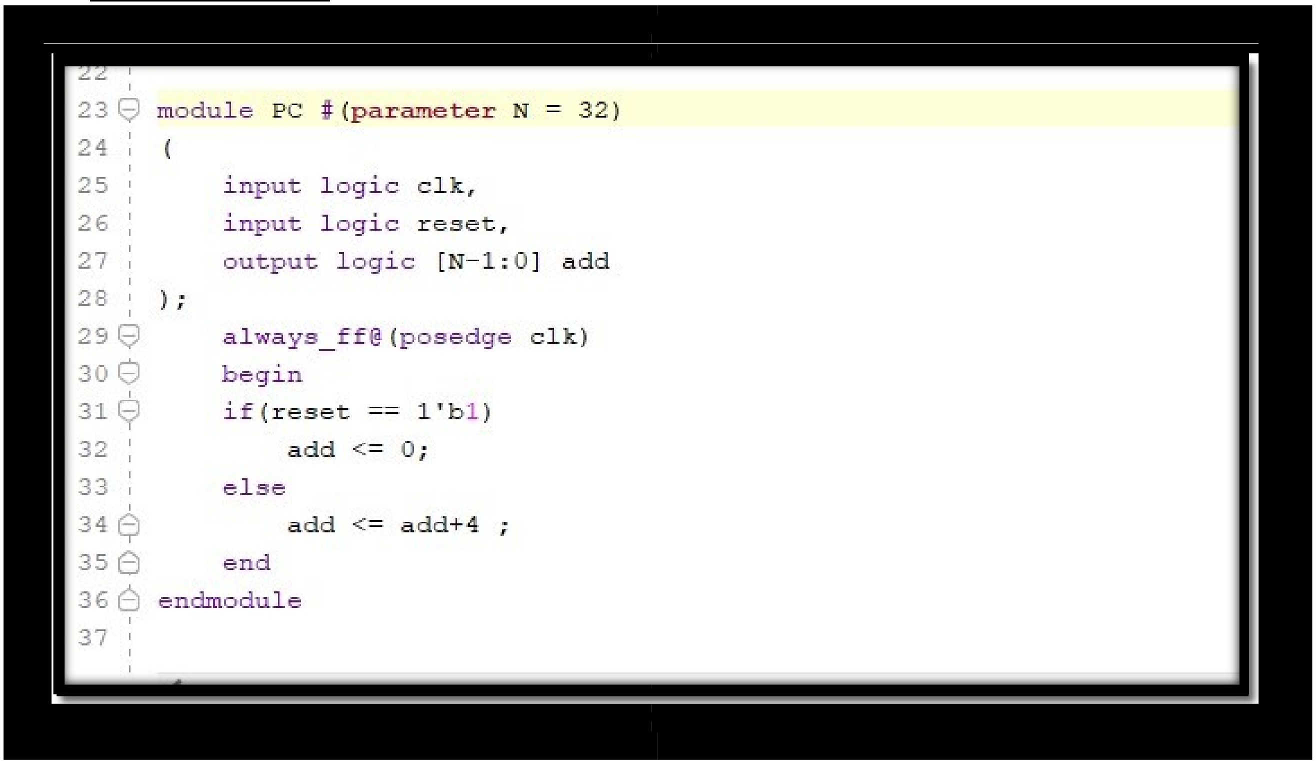
Xilinx Vivado

Task

Implement additional logic to cater control hazards, so that branches could be taken correctly, without unexpected changes in state elements. You have to add two more modules named as Hazard detection unit and forwarding Unit.

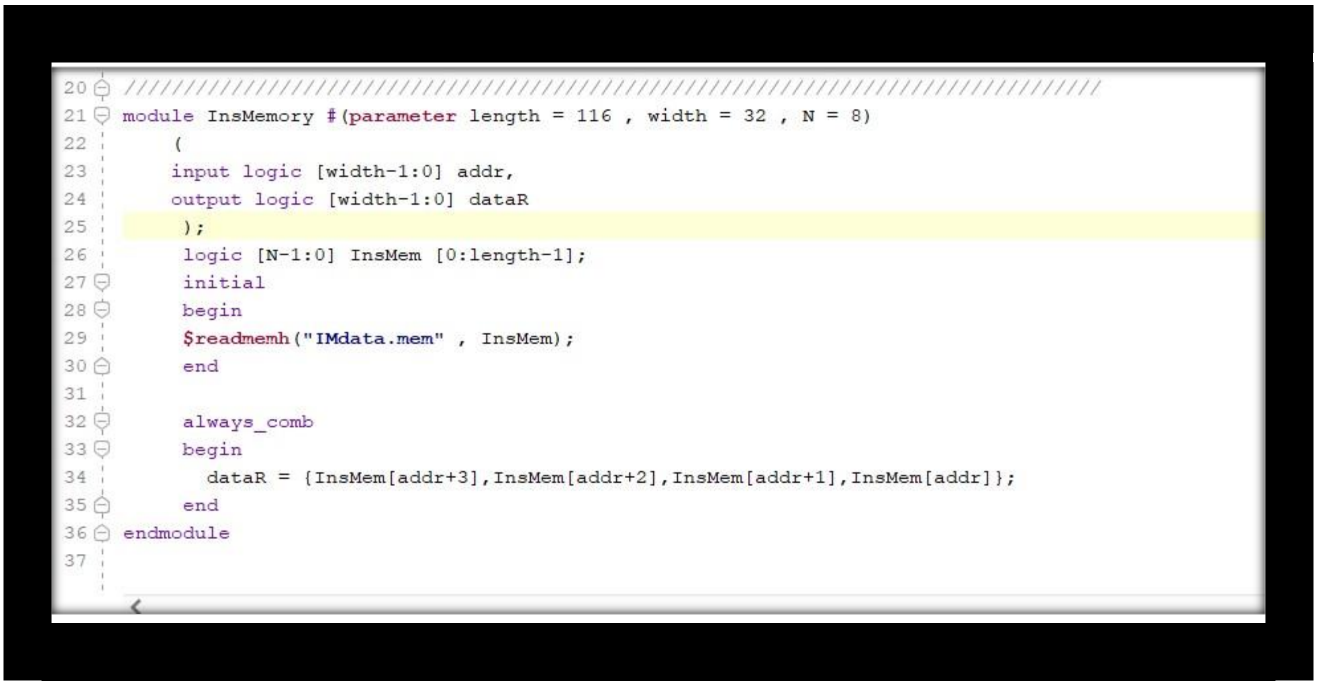
OUTPUT:

PC MODULE:



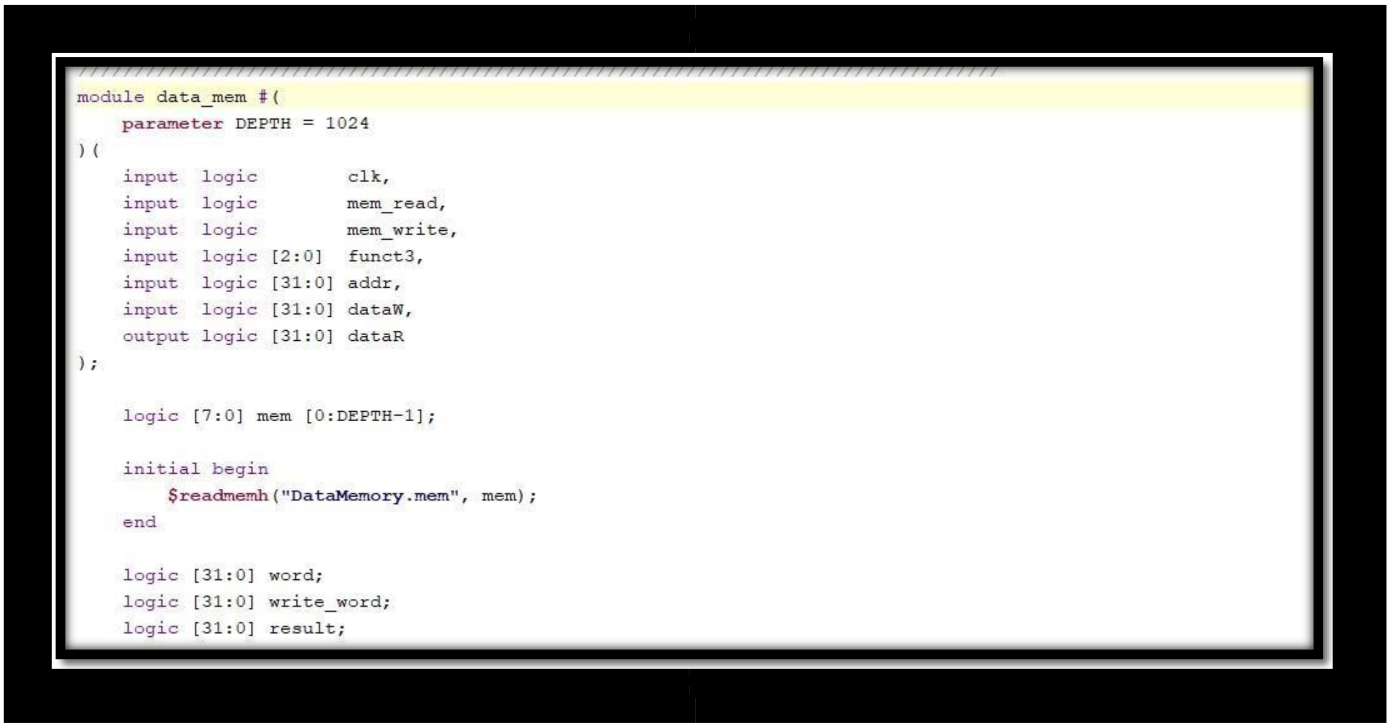
```
module PC #(parameter N = 32)
(
    input logic clk,
    input logic reset,
    output logic [N-1:0] add
);
    always_ff@(posedge clk)
    begin
    if(reset == 1'b1)
        add <= 0;
    else
        add <= add+4 ;
    end
endmodule
```

INSTRUCTION MEMORY

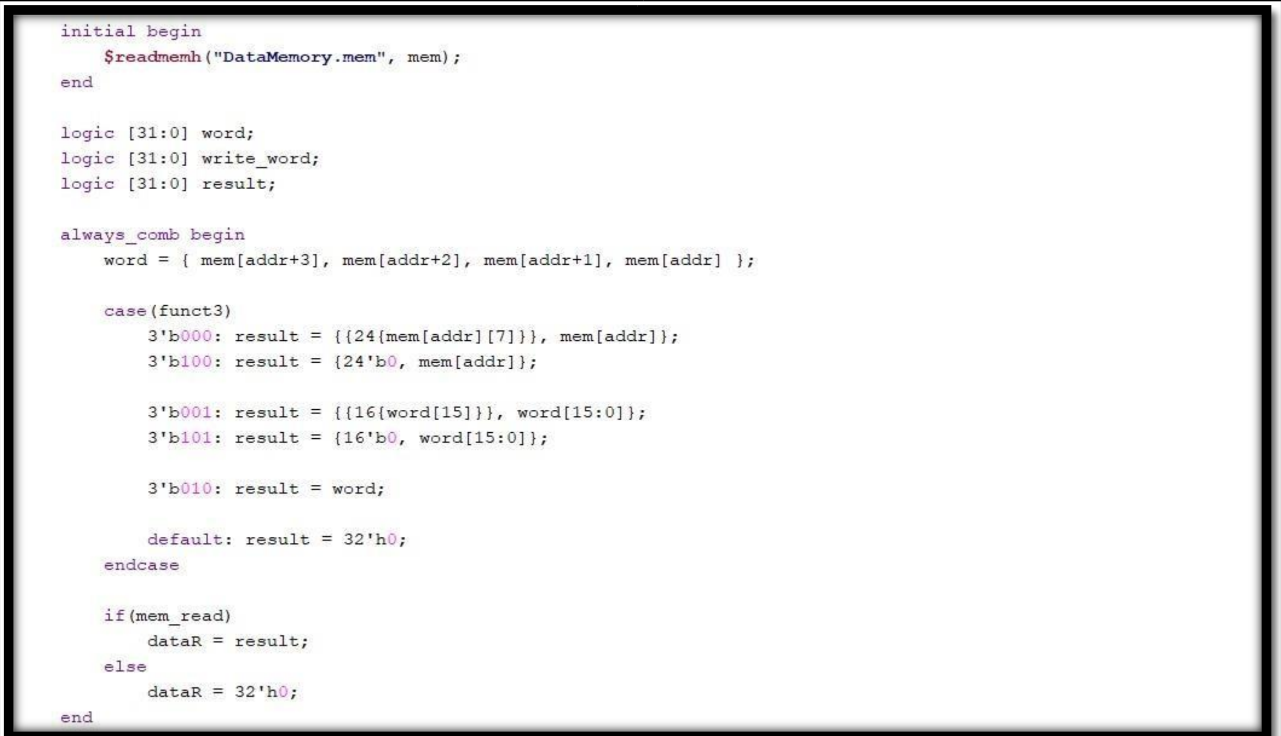


```
//////////////////////////////////////////////////////////////////////////////////////
module InsMemory #(parameter length = 116 , width = 32 , N = 8)
    (
    input logic [width-1:0] addr,
    output logic [width-1:0] dataR
     );
     logic [N-1:0] InsMem [0:length-1];
     initial
     begin
     $readmemh("IMdata.mem" , InsMem);
     end

     always_comb
     begin
       dataR = {InsMem[addr+3],InsMem[addr+2],InsMem[addr+1],InsMem[addr]};
     end
endmodule
```

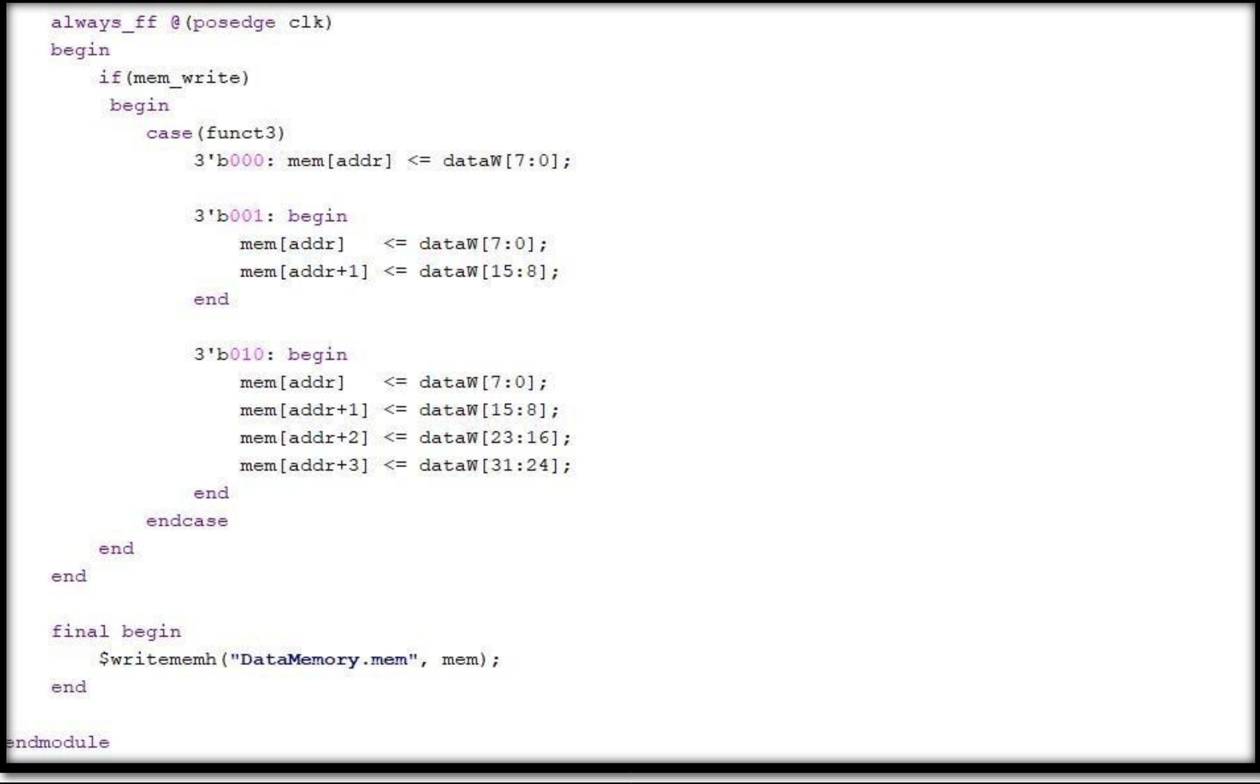
DATA MEMORY:



```
module data_mem #(
    parameter DEPTH = 1024
)(
    input  logic        clk,
    input  logic        mem_read,
    input  logic        mem_write,
    input  logic [2:0]  funct3,
    input  logic [31:0] addr,
    input  logic [31:0] dataW,
    output logic [31:0] dataR
);

    logic [7:0] mem [0:DEPTH-1];

    initial begin
        $readmemh("DataMemory.mem", mem);
    end

    logic [31:0] word;
    logic [31:0] write_word;
    logic [31:0] result;
```

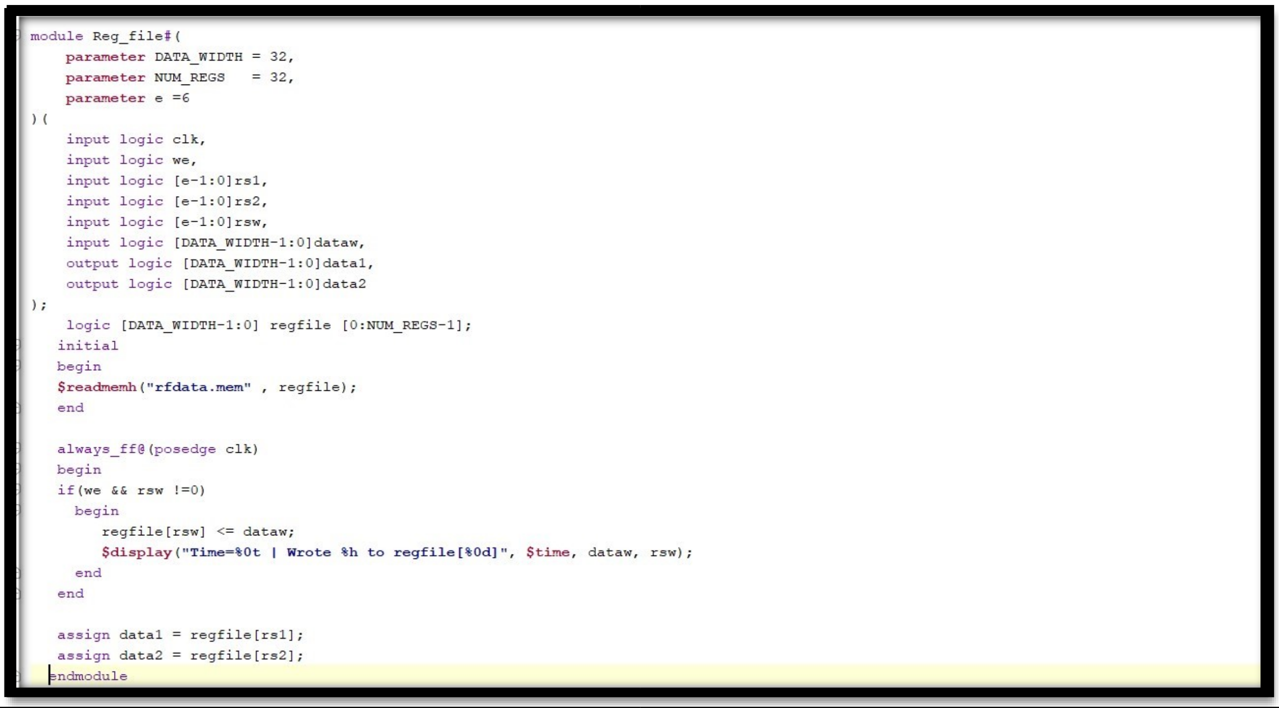


```
initial begin
    $readmemh("DataMemory.mem", mem);
end

logic [31:0] word;
logic [31:0] write_word;
logic [31:0] result;

always_comb begin
    word = { mem[addr+3], mem[addr+2], mem[addr+1], mem[addr] };

    case(funct3)
        3'b000: result = {{24{mem[addr][7]}}, mem[addr]};
        3'b100: result = {24'b0, mem[addr]};

        3'b001: result = {{16{word[15]}}, word[15:0]};
        3'b101: result = {16'b0, word[15:0]};

        3'b010: result = word;

        default: result = 32'h0;
    endcase

    if(mem_read)
        dataR = result;
    else
        dataR = 32'h0;
end
```



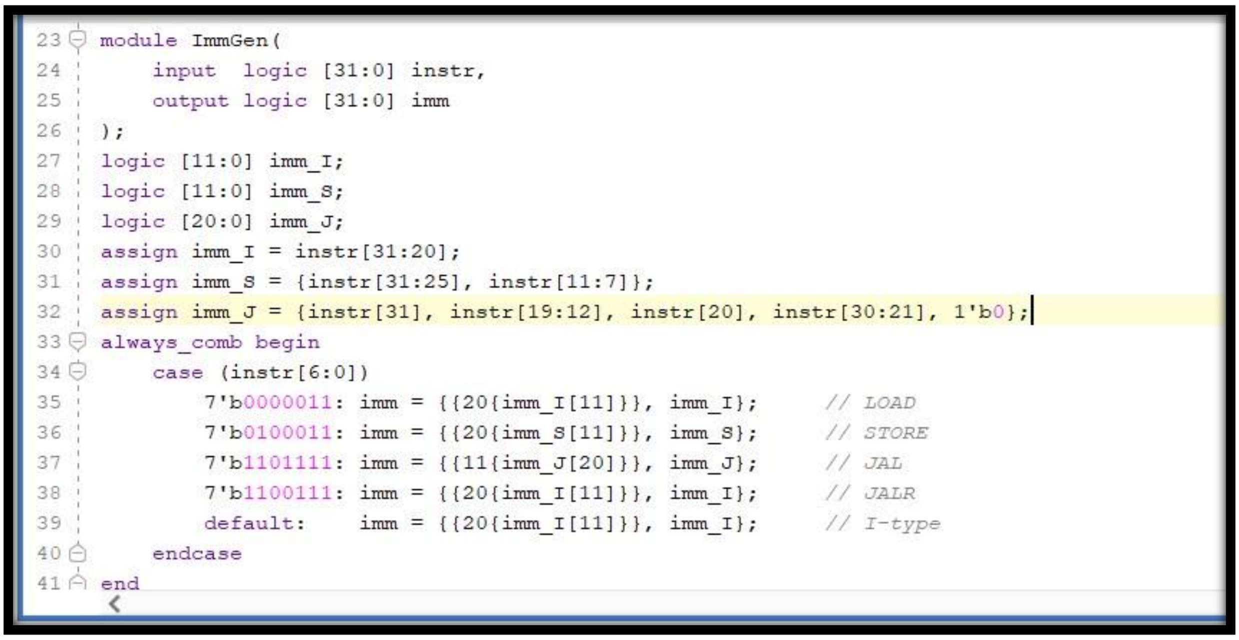
```
always_ff @(posedge clk)
begin
    if(mem_write)
     begin
        case(funct3)
            3'b000: mem[addr] <= dataW[7:0];

            3'b001: begin
                mem[addr]   <= dataW[7:0];
                mem[addr+1] <= dataW[15:8];
            end

            3'b010: begin
                mem[addr]   <= dataW[7:0];
                mem[addr+1] <= dataW[15:8];
                mem[addr+2] <= dataW[23:16];
                mem[addr+3] <= dataW[31:24];
            end
        endcase
    end
end

final begin
    $writememh("DataMemory.mem", mem);
end

endmodule
```

REGISTER FILE:



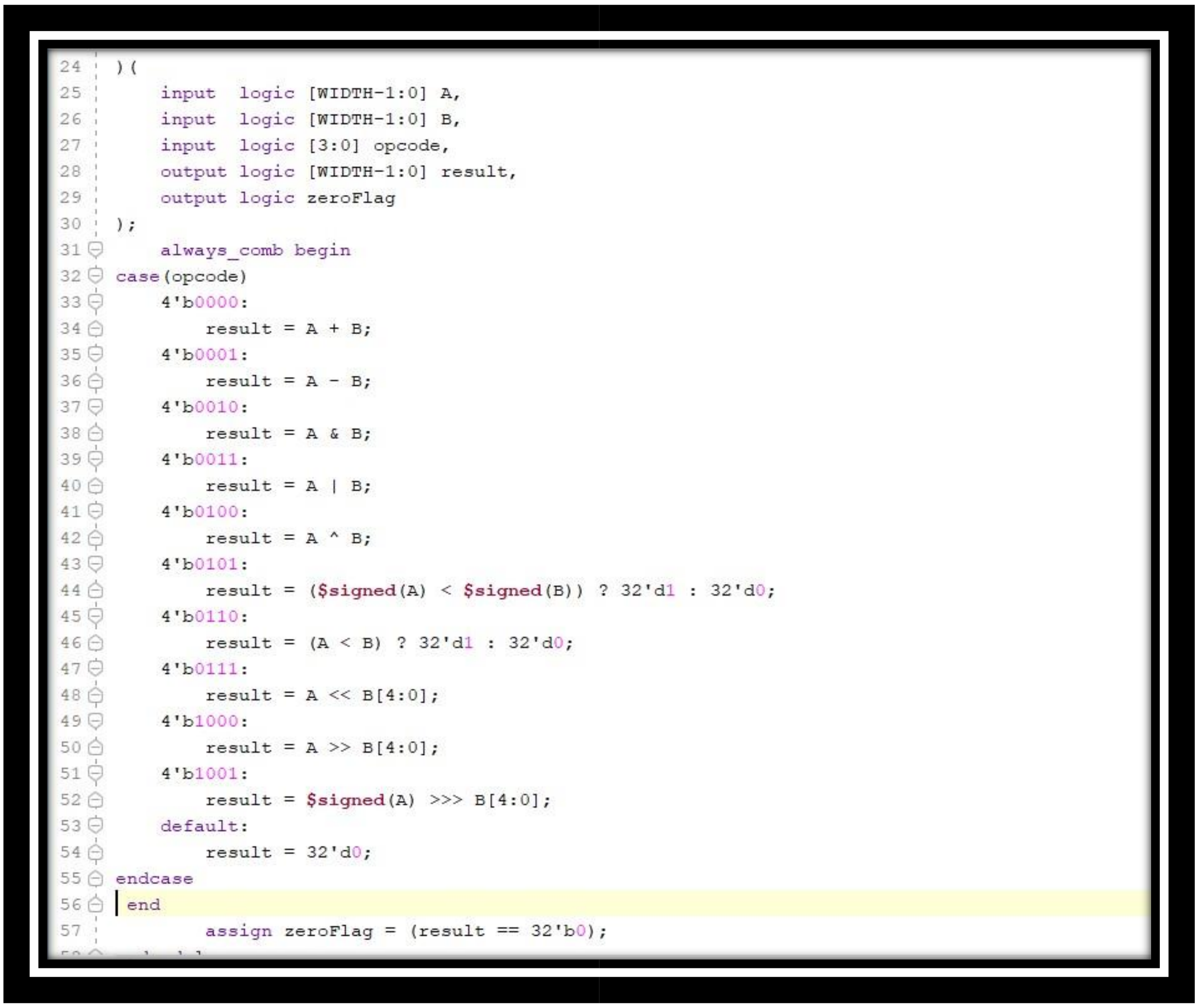
```
module Reg_file#(
    parameter DATA_WIDTH = 32,
    parameter NUM_REGS   = 32,
    parameter e =6
)(
    input logic clk,
    input logic we,
    input logic [e-1:0]rs1,
    input logic [e-1:0]rs2,
    input logic [e-1:0]rsw,
    input logic [DATA_WIDTH-1:0]dataw,
    output logic [DATA_WIDTH-1:0]data1,
    output logic [DATA_WIDTH-1:0]data2
);
    logic [DATA_WIDTH-1:0] regfile [0:NUM_REGS-1];
   initial
   begin
   $readmemh("rfdata.mem" , regfile);
   end

   always_ff@(posedge clk)
   begin
   if(we && rsw !=0)
     begin
        regfile[rsw] <= dataw;
        $display("Time=%0t | Wrote %h to regfile[%0d]", $time, dataw, rsw);
     end
   end

   assign data1 = regfile[rs1];
   assign data2 = regfile[rs2];
  endmodule
```

IMMEDIATE GENERATOR:



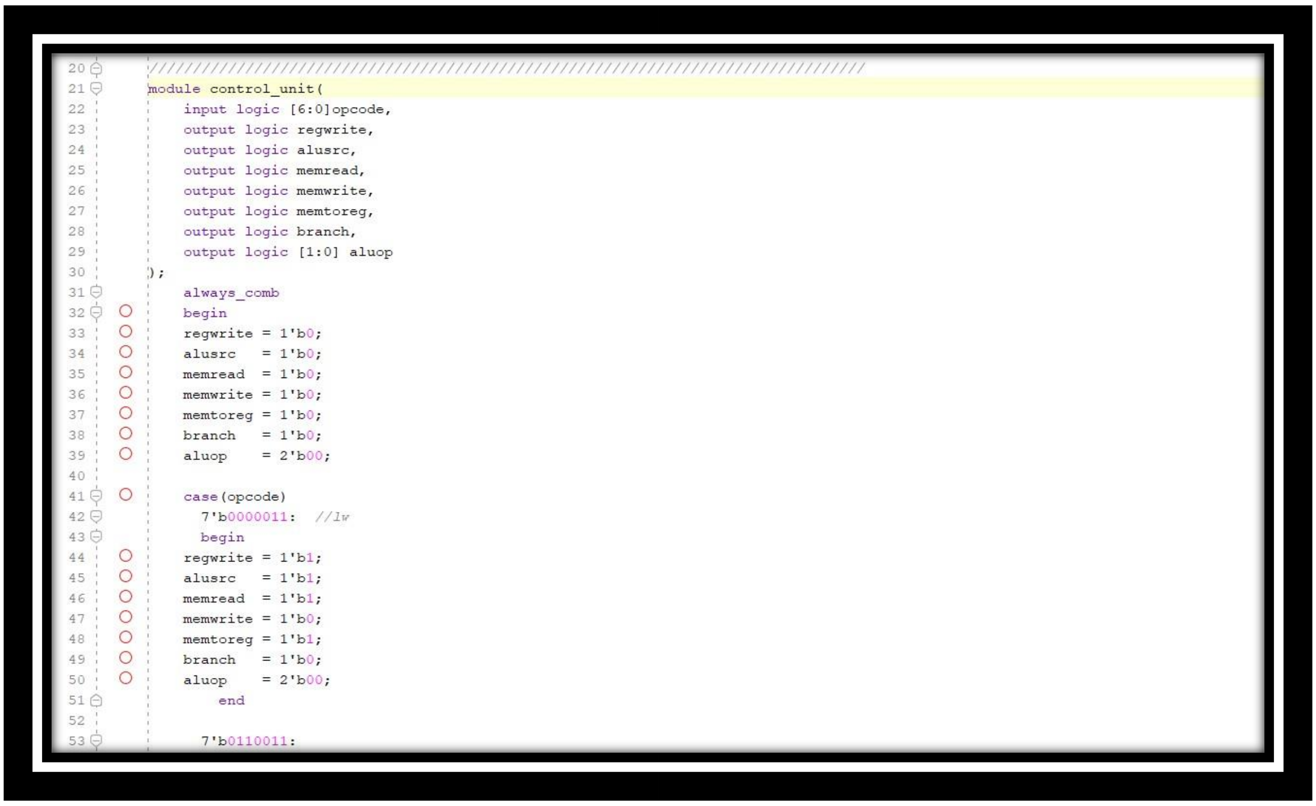
```
module ImmGen(
    input  logic [31:0] instr,
    output logic [31:0] imm
);
logic [11:0] imm_I;
logic [11:0] imm_S;
logic [20:0] imm_J;
assign imm_I = instr[31:20];
assign imm_S = {instr[31:25], instr[11:7]};
assign imm_J = {instr[31], instr[19:12], instr[20], instr[30:21], 1'b0};
always_comb begin
    case (instr[6:0])
        7'b0000011: imm = {{20{imm_I[11]}}, imm_I};     // LOAD
        7'b0100011: imm = {{20{imm_S[11]}}, imm_S};     // STORE
        7'b1101111: imm = {{11{imm_J[20]}}, imm_J};     // JAL
        7'b1100111: imm = {{20{imm_I[11]}}, imm_I};     // JALR
        default:    imm = {{20{imm_I[11]}}, imm_I};     // I-type
    endcase
end
```

ALU MODULE:

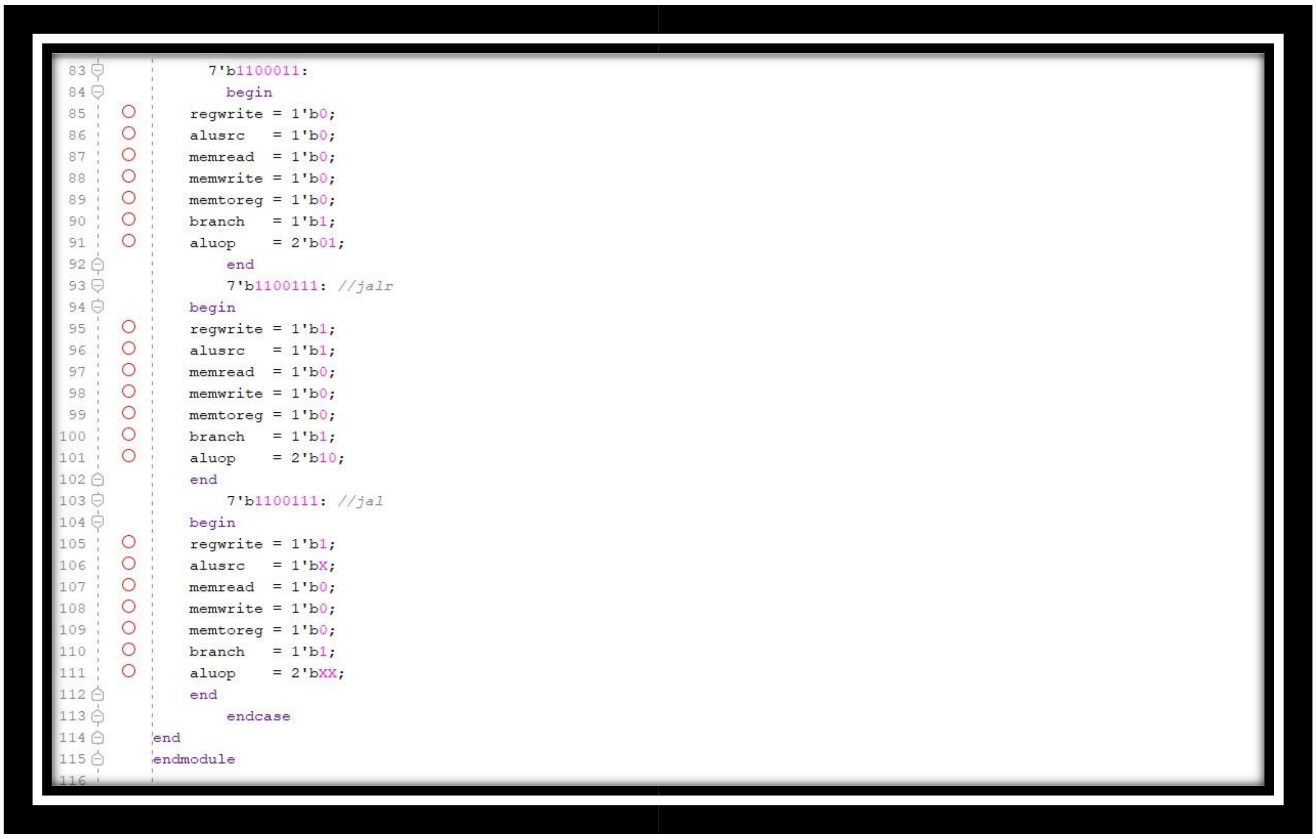


```
)(
    input  logic [WIDTH-1:0] A,
    input  logic [WIDTH-1:0] B,
    input  logic [3:0] opcode,
    output logic [WIDTH-1:0] result,
    output logic zeroFlag
);
    always_comb begin
case(opcode)
    4'b0000:
        result = A + B;
    4'b0001:
        result = A - B;
    4'b0010:
        result = A & B;
    4'b0011:
        result = A | B;
    4'b0100:
        result = A ^ B;
    4'b0101:
        result = ($signed(A) < $signed(B)) ? 32'd1 : 32'd0;
    4'b0110:
        result = (A < B) ? 32'd1 : 32'd0;
    4'b0111:
        result = A << B[4:0];
    4'b1000:
        result = A >> B[4:0];
    4'b1001:
        result = $signed(A) >>> B[4:0];
    default:
        result = 32'd0;
endcase
 end
        assign zeroFlag = (result == 32'b0);
```

CONTROL UNIT:

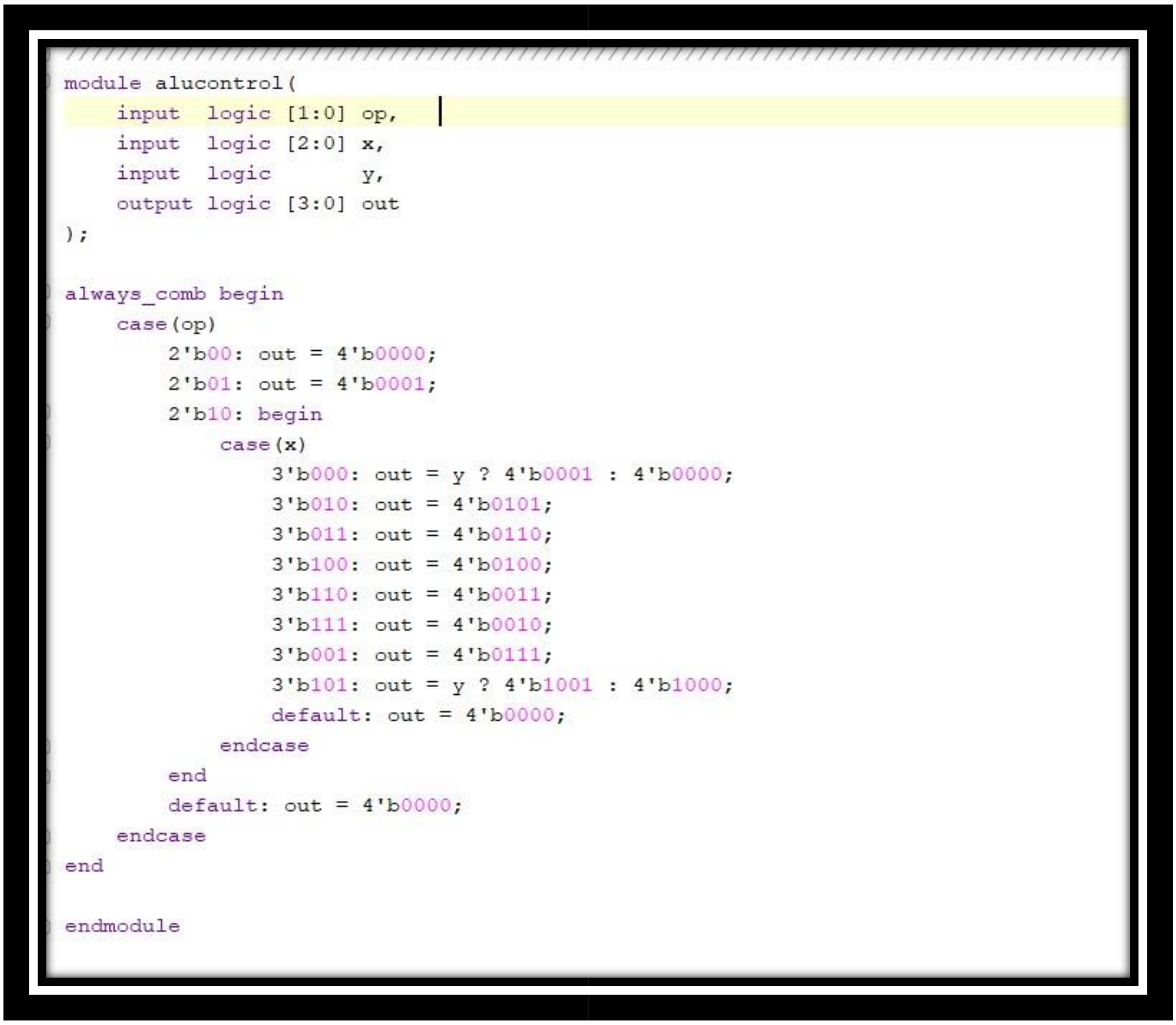


```
////////////////////////////////////////////////////////////////////////////////
module control_unit(
    input logic [6:0]opcode,
    output logic regwrite,
    output logic alusrc,
    output logic memread,
    output logic memwrite,
    output logic memtoreg,
    output logic branch,
    output logic [1:0] aluop
);
    always_comb
    begin
    regwrite = 1'b0;
    alusrc   = 1'b0;
    memread  = 1'b0;
    memwrite = 1'b0;
    memtoreg = 1'b0;
    branch   = 1'b0;
    aluop    = 2'b00;

    case(opcode)
      7'b0000011:  //lw
      begin
    regwrite = 1'b1;
    alusrc   = 1'b1;
    memread  = 1'b1;
    memwrite = 1'b0;
    memtoreg = 1'b1;
    branch   = 1'b0;
    aluop    = 2'b00;
        end

      7'b0110011:
```



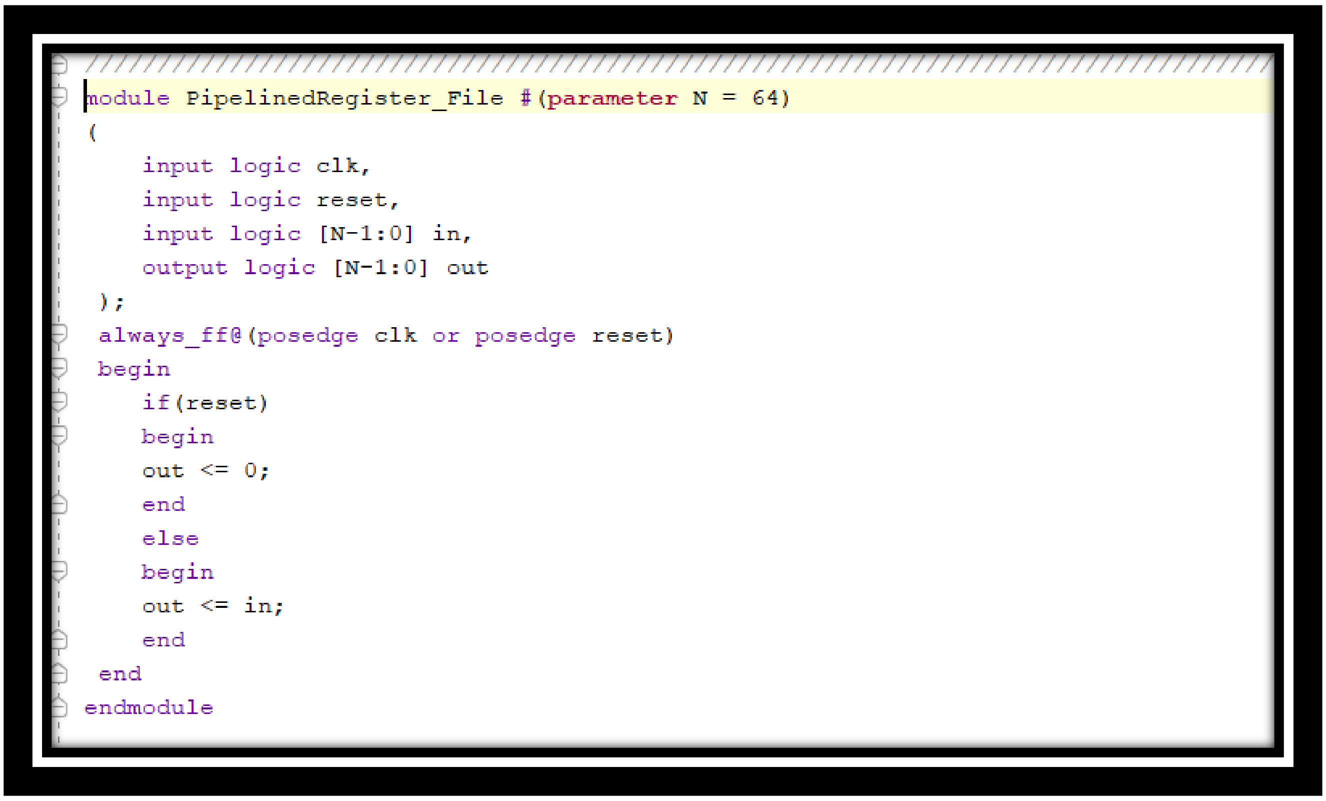
```
      7'b1100011:
        begin
    regwrite = 1'b0;
    alusrc   = 1'b0;
    memread  = 1'b0;
    memwrite = 1'b0;
    memtoreg = 1'b0;
    branch   = 1'b1;
    aluop    = 2'b01;
        end
        7'b1100111: //jalr
    begin
    regwrite = 1'b1;
    alusrc   = 1'b1;
    memread  = 1'b0;
    memwrite = 1'b0;
    memtoreg = 1'b0;
    branch   = 1'b1;
    aluop    = 2'b10;
    end
        7'b1100111: //jal
    begin
    regwrite = 1'b1;
    alusrc   = 1'bX;
    memread  = 1'b0;
    memwrite = 1'b0;
    memtoreg = 1'b0;
    branch   = 1'b1;
    aluop    = 2'bXX;
    end
        endcase
end
endmodule
```

ALU_CONTROL:



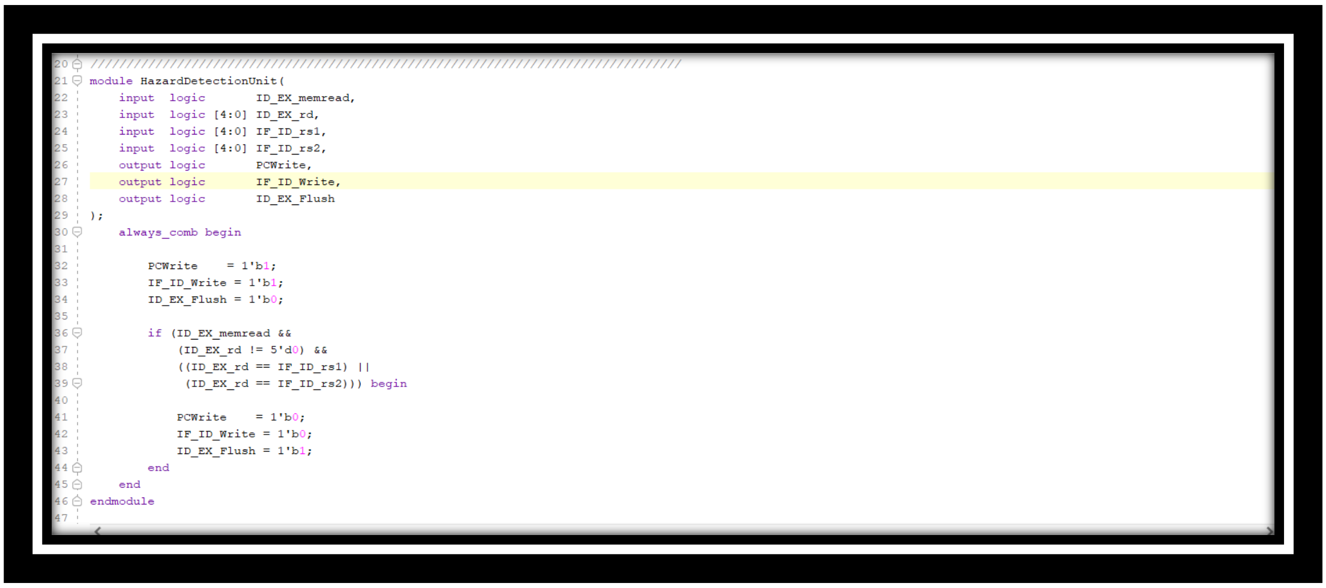
```
module alucontrol(
    input  logic [1:0] op,
    input  logic [2:0] x,
    input  logic       y,
    output logic [3:0] out
);

always_comb begin
    case(op)
        2'b00: out = 4'b0000;
        2'b01: out = 4'b0001;
        2'b10: begin
            case(x)
                3'b000: out = y ? 4'b0001 : 4'b0000;
                3'b010: out = 4'b0101;
                3'b011: out = 4'b0110;
                3'b100: out = 4'b0100;
                3'b110: out = 4'b0011;
                3'b111: out = 4'b0010;
                3'b001: out = 4'b0111;
                3'b101: out = y ? 4'b1001 : 4'b1000;
                default: out = 4'b0000;
            endcase
        end
        default: out = 4'b0000;
    endcase
end

endmodule
```

Pipelined Registers Module:



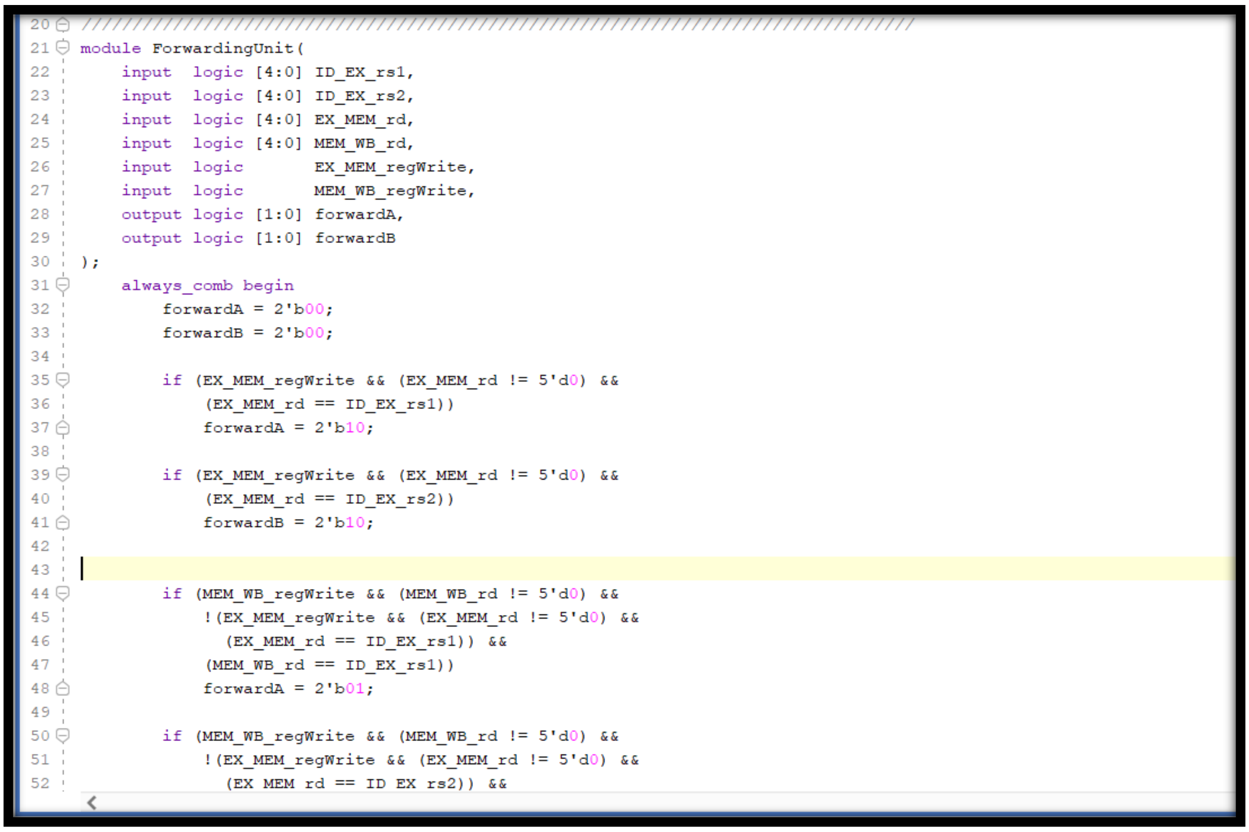
```
/////////////////////////////////////////////////////////////////////////////////////////
module PipelinedRegister_File #(parameter N = 64)
(
    input logic clk,
    input logic reset,
    input logic [N-1:0] in,
    output logic [N-1:0] out
 );
 always_ff@(posedge clk or posedge reset)
 begin
    if(reset)
    begin
    out <= 0;
    end
    else
    begin
    out <= in;
    end
 end
endmodule
```

HAZARD CONTROL UNIT:



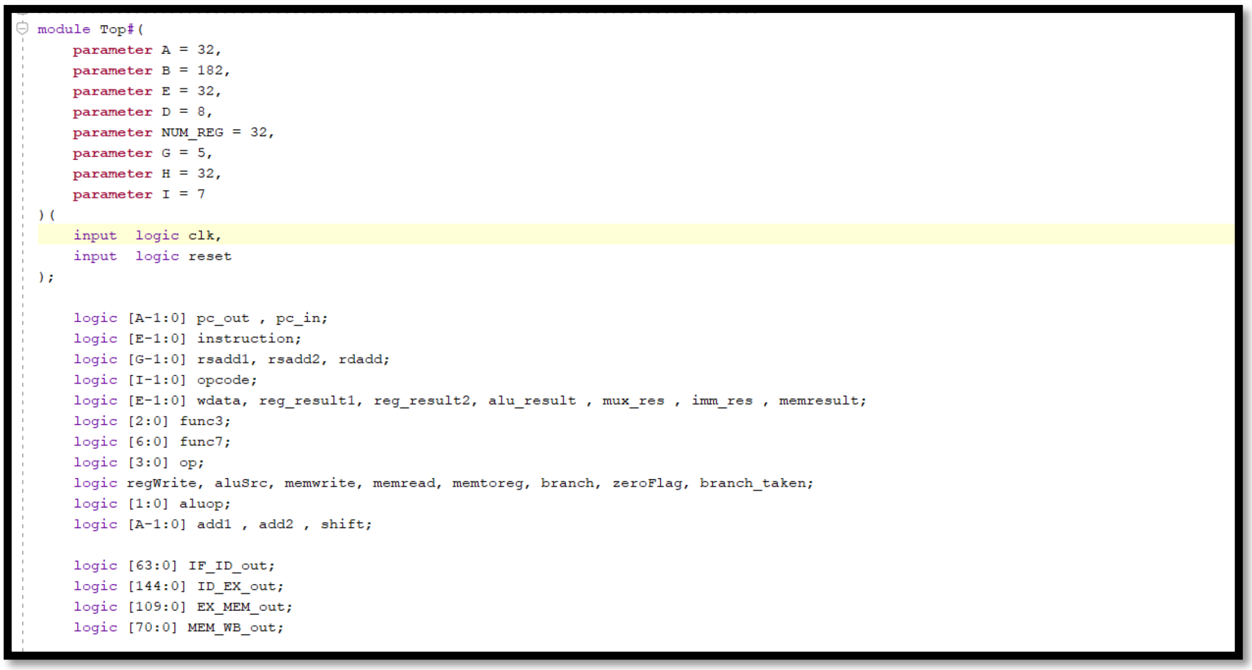
```
////////////////////////////////////////////////////////////////////////////////
module HazardDetectionUnit(
    input  logic       ID_EX_memread,
    input  logic [4:0] ID_EX_rd,
    input  logic [4:0] IF_ID_rs1,
    input  logic [4:0] IF_ID_rs2,
    output logic       PCWrite,
    output logic       IF_ID_Write,
    output logic       ID_EX_Flush
);
    always_comb begin

        PCWrite    = 1'b1;
        IF_ID_Write = 1'b1;
        ID_EX_Flush = 1'b0;

        if (ID_EX_memread &&
            (ID_EX_rd != 5'd0) &&
            ((ID_EX_rd == IF_ID_rs1) ||
             (ID_EX_rd == IF_ID_rs2))) begin

            PCWrite    = 1'b0;
            IF_ID_Write = 1'b0;
            ID_EX_Flush = 1'b1;
        end
    end
endmodule
```

FORWARDING UNIT:

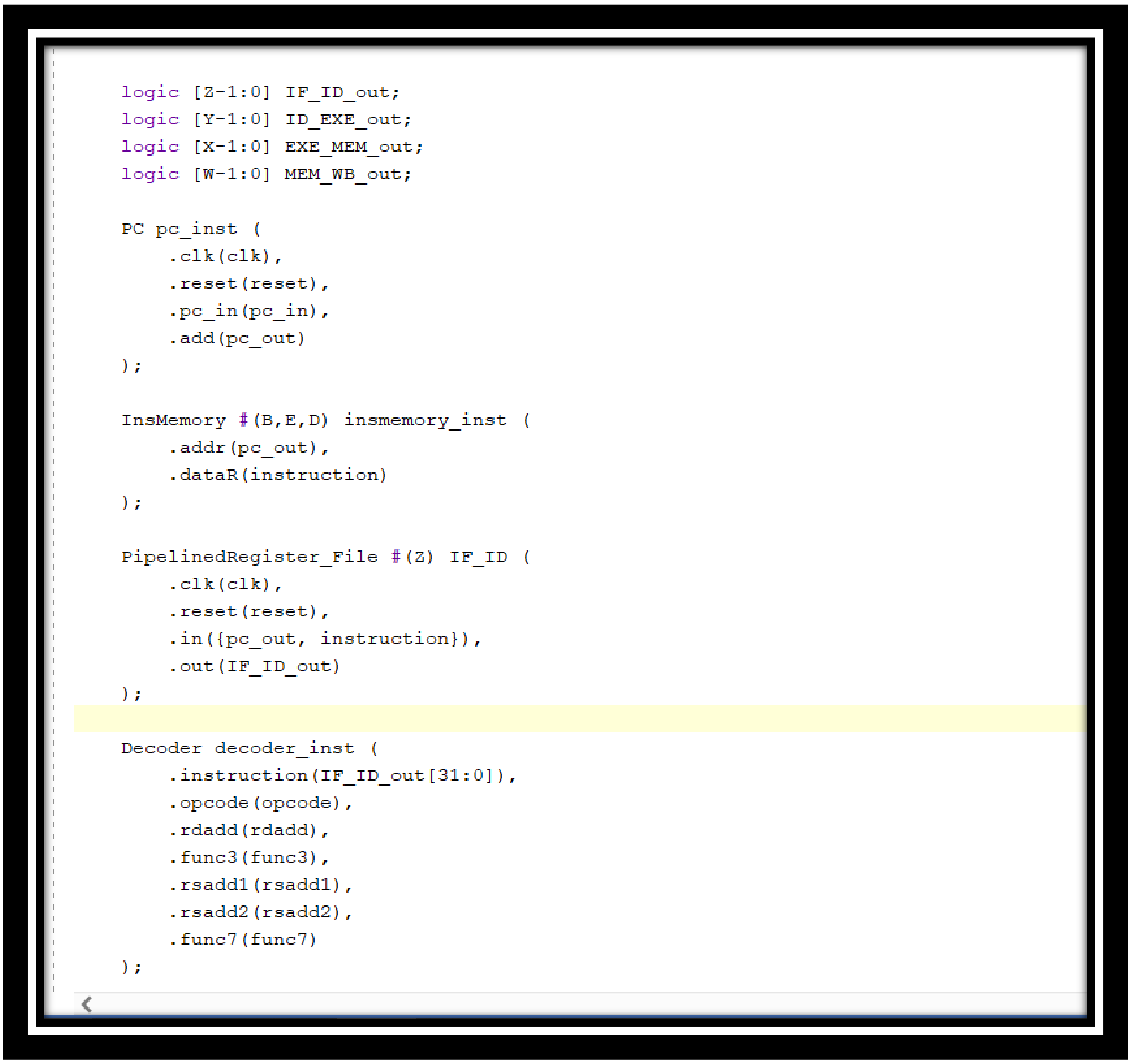


```
///////////////////////////////////////////////////////////////////////////////////
module ForwardingUnit(
    input  logic [4:0] ID_EX_rs1,
    input  logic [4:0] ID_EX_rs2,
    input  logic [4:0] EX_MEM_rd,
    input  logic [4:0] MEM_WB_rd,
    input  logic       EX_MEM_regWrite,
    input  logic       MEM_WB_regWrite,
    output logic [1:0] forwardA,
    output logic [1:0] forwardB
);
    always_comb begin
        forwardA = 2'b00;
        forwardB = 2'b00;

        if (EX_MEM_regWrite && (EX_MEM_rd != 5'd0) &&
            (EX_MEM_rd == ID_EX_rs1))
            forwardA = 2'b10;

        if (EX_MEM_regWrite && (EX_MEM_rd != 5'd0) &&
            (EX_MEM_rd == ID_EX_rs2))
            forwardB = 2'b10;


        if (MEM_WB_regWrite && (MEM_WB_rd != 5'd0) &&
            !(EX_MEM_regWrite && (EX_MEM_rd != 5'd0) &&
              (EX_MEM_rd == ID_EX_rs1)) &&
            (MEM_WB_rd == ID_EX_rs1))
            forwardA = 2'b01;

        if (MEM_WB_regWrite && (MEM_WB_rd != 5'd0) &&
            !(EX_MEM_regWrite && (EX_MEM_rd != 5'd0) &&
              (EX_MEM_rd == ID_EX_rs2)) &&
```

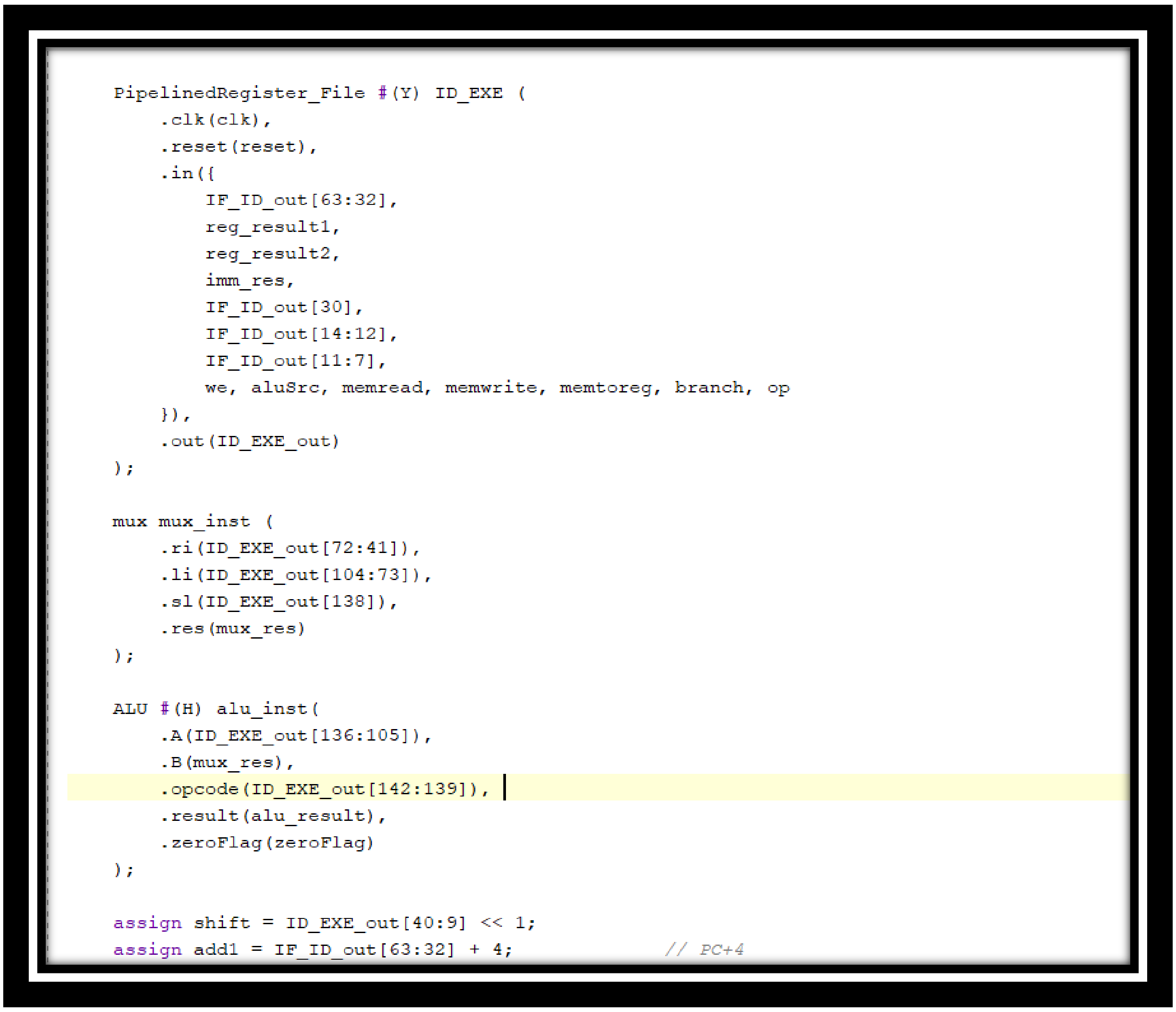
TOP MODULE:

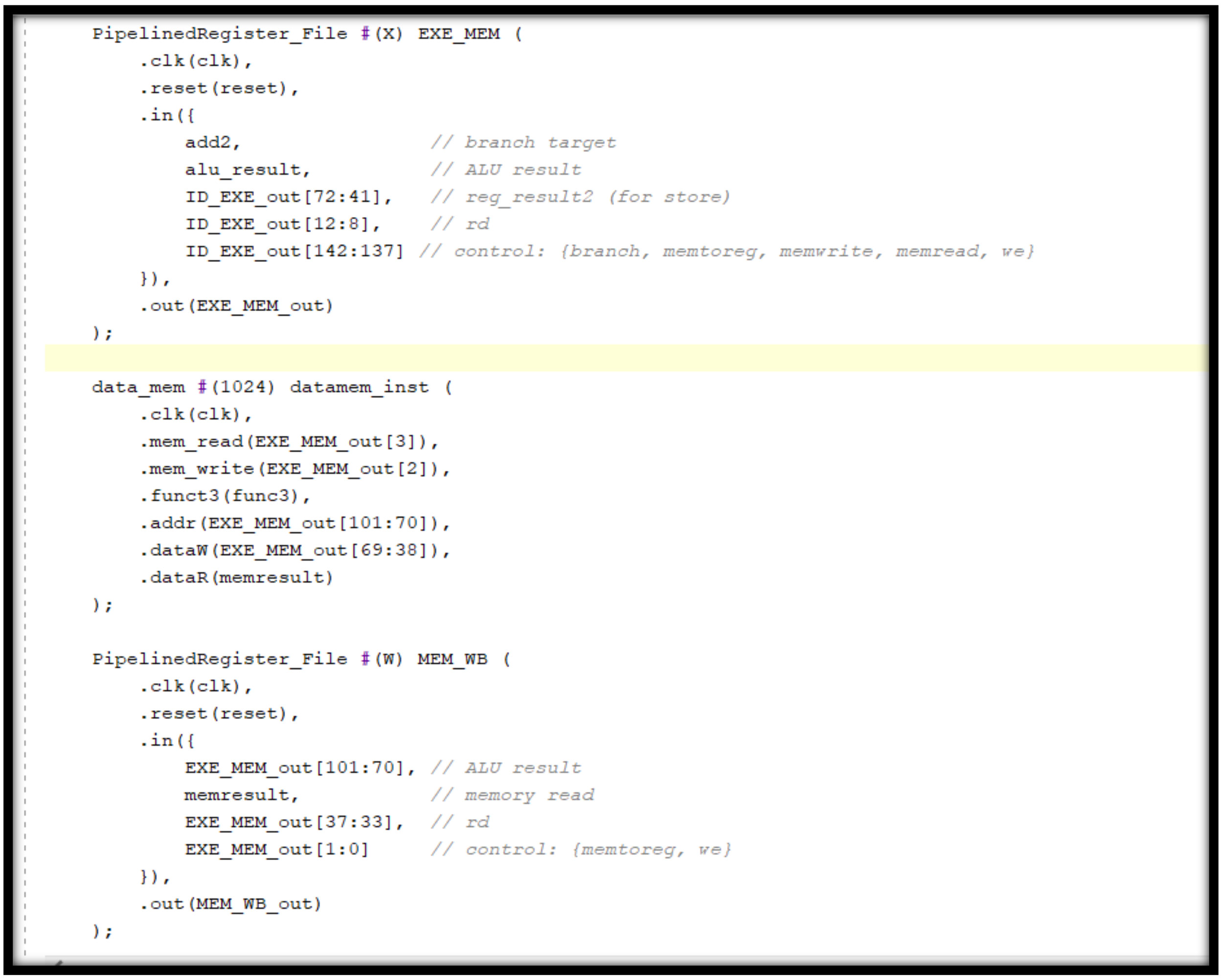


```
module Top#(
    parameter A = 32,
    parameter B = 182,
    parameter E = 32,
    parameter D = 8,
    parameter NUM_REG = 32,
    parameter G = 5,
    parameter H = 32,
    parameter I = 7
)(
    input  logic clk,
    input  logic reset
);

    logic [A-1:0] pc_out , pc_in;
    logic [E-1:0] instruction;
    logic [G-1:0] rsadd1, rsadd2, rdadd;
    logic [I-1:0] opcode;
    logic [E-1:0] wdata, reg_result1, reg_result2, alu_result , mux_res , imm_res , memresult;
    logic [2:0] func3;
    logic [6:0] func7;
    logic [3:0] op;
    logic regWrite, aluSrc, memwrite, memread, memtoreg, branch, zeroFlag, branch_taken;
    logic [1:0] aluop;
    logic [A-1:0] add1 , add2 , shift;

    logic [63:0] IF_ID_out;
    logic [144:0] ID_EX_out;
    logic [109:0] EX_MEM_out;
    logic [70:0] MEM_WB_out;
```

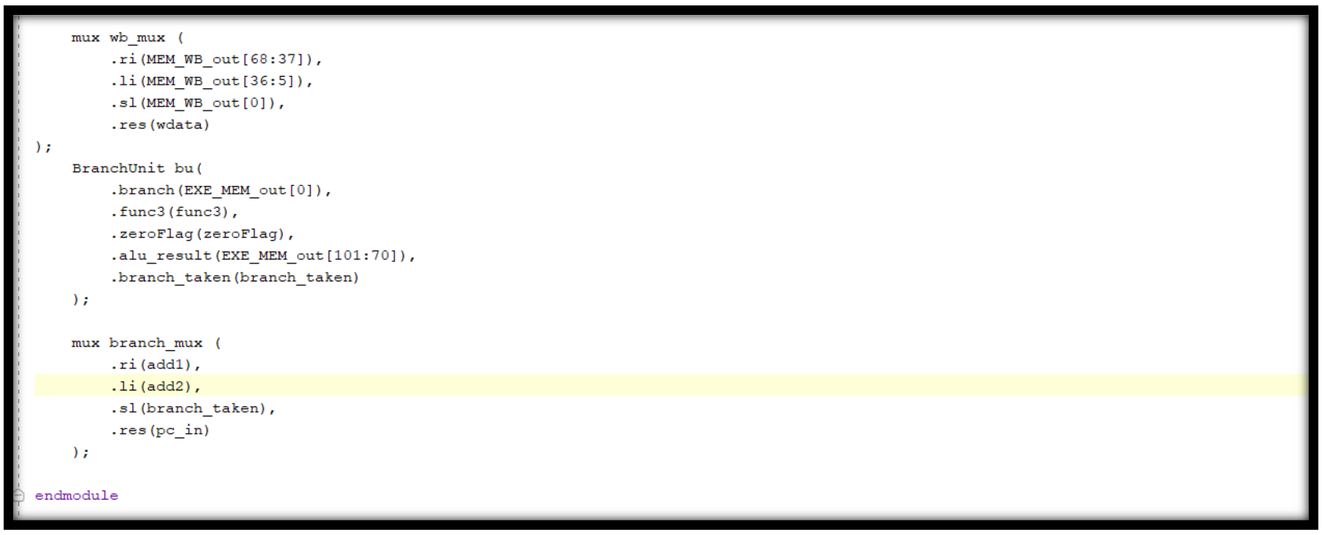


```
logic [Z-1:0] IF_ID_out;
logic [Y-1:0] ID_EXE_out;
logic [X-1:0] EXE_MEM_out;
logic [W-1:0] MEM_WB_out;

PC pc_inst (
    .clk(clk),
    .reset(reset),
    .pc_in(pc_in),
    .add(pc_out)
);

InsMemory #(B,E,D) insmemory_inst (
    .addr(pc_out),
    .dataR(instruction)
);

PipelinedRegister_File #(Z) IF_ID (
    .clk(clk),
    .reset(reset),
    .in({pc_out, instruction}),
    .out(IF_ID_out)
);

Decoder decoder_inst (
    .instruction(IF_ID_out[31:0]),
    .opcode(opcode),
    .rdadd(rdadd),
    .func3(func3),
    .rsadd1(rsadd1),
    .rsadd2(rsadd2),
    .func7(func7)
);
```


```
PipelinedRegister_File #(Y) ID_EXE (
    .clk(clk),
    .reset(reset),
    .in({
        IF_ID_out[63:32],
        reg_result1,
        reg_result2,
        imm_res,
        IF_ID_out[30],
        IF_ID_out[14:12],
        IF_ID_out[11:7],
        we, aluSrc, memread, memwrite, memtoreg, branch, op
    }),
    .out(ID_EXE_out)
);

mux mux_inst (
    .ri(ID_EXE_out[72:41]),
    .li(ID_EXE_out[104:73]),
    .sl(ID_EXE_out[138]),
    .res(mux_res)
);

ALU #(H) alu_inst(
    .A(ID_EXE_out[136:105]),
    .B(mux_res),
    .opcode(ID_EXE_out[142:139]),
    .result(alu_result),
    .zeroFlag(zeroFlag)
);

assign shift = ID_EXE_out[40:9] << 1;
assign add1 = IF_ID_out[63:32] + 4;            // PC+4
```

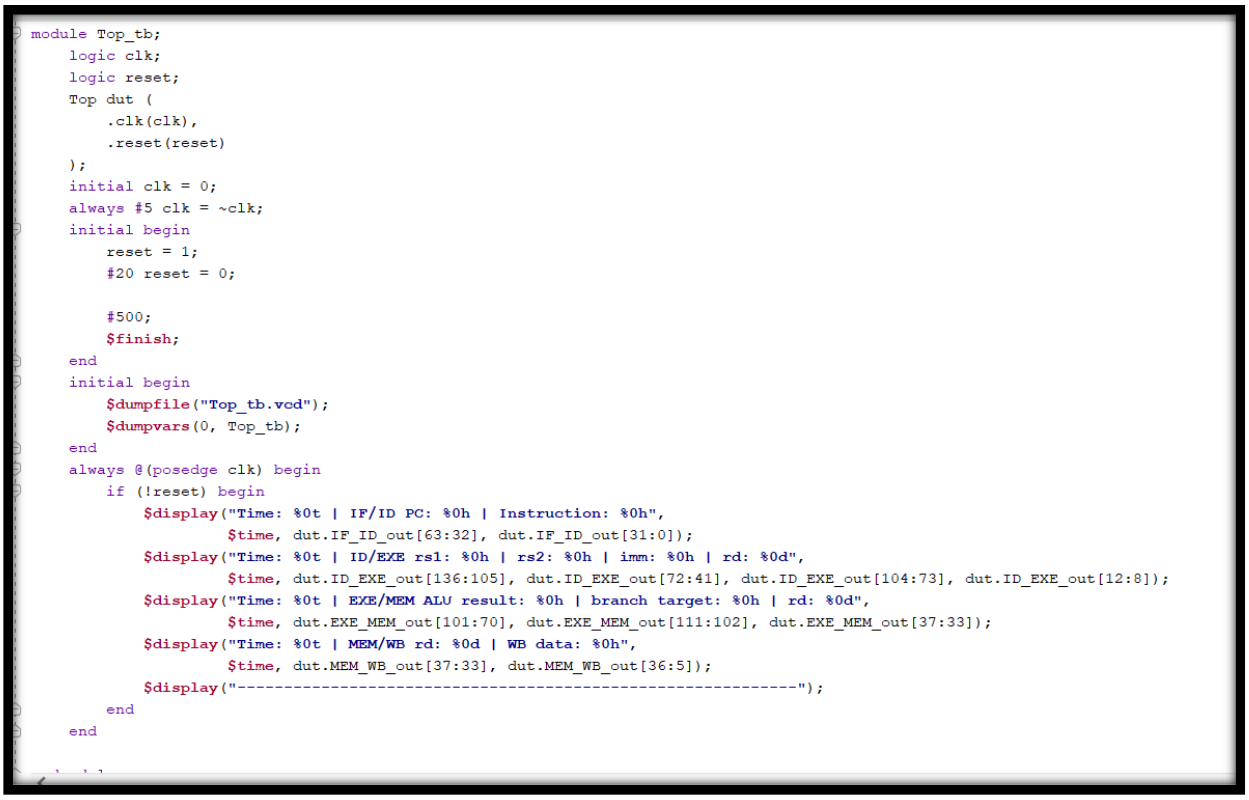


```
    PipelinedRegister_File #(X) EXE_MEM (
        .clk(clk),
        .reset(reset),
        .in({
            add2,                // branch target
            alu_result,          // ALU result
            ID_EXE_out[72:41],   // reg_result2 (for store)
            ID_EXE_out[12:8],    // rd
            ID_EXE_out[142:137] // control: {branch, memtoreg, memwrite, memread, we}
        }),
        .out(EXE_MEM_out)
    );

    data_mem #(1024) datamem_inst (
        .clk(clk),
        .mem_read(EXE_MEM_out[3]),
        .mem_write(EXE_MEM_out[2]),
        .funct3(func3),
        .addr(EXE_MEM_out[101:70]),
        .dataW(EXE_MEM_out[69:38]),
        .dataR(memresult)
    );

    PipelinedRegister_File #(W) MEM_WB (
        .clk(clk),
        .reset(reset),
        .in({
            EXE_MEM_out[101:70], // ALU result
            memresult,           // memory read
            EXE_MEM_out[37:33],  // rd
            EXE_MEM_out[1:0]     // control: {memtoreg, we}
        }),
        .out(MEM_WB_out)
    );
```

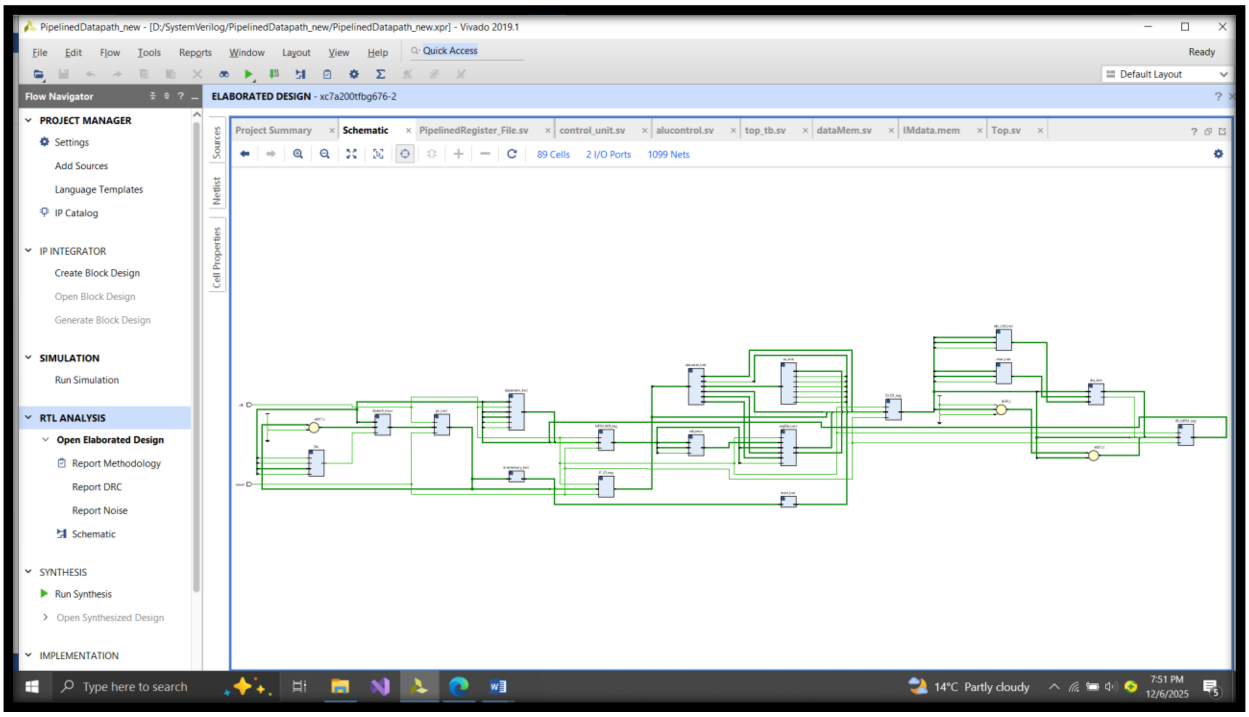


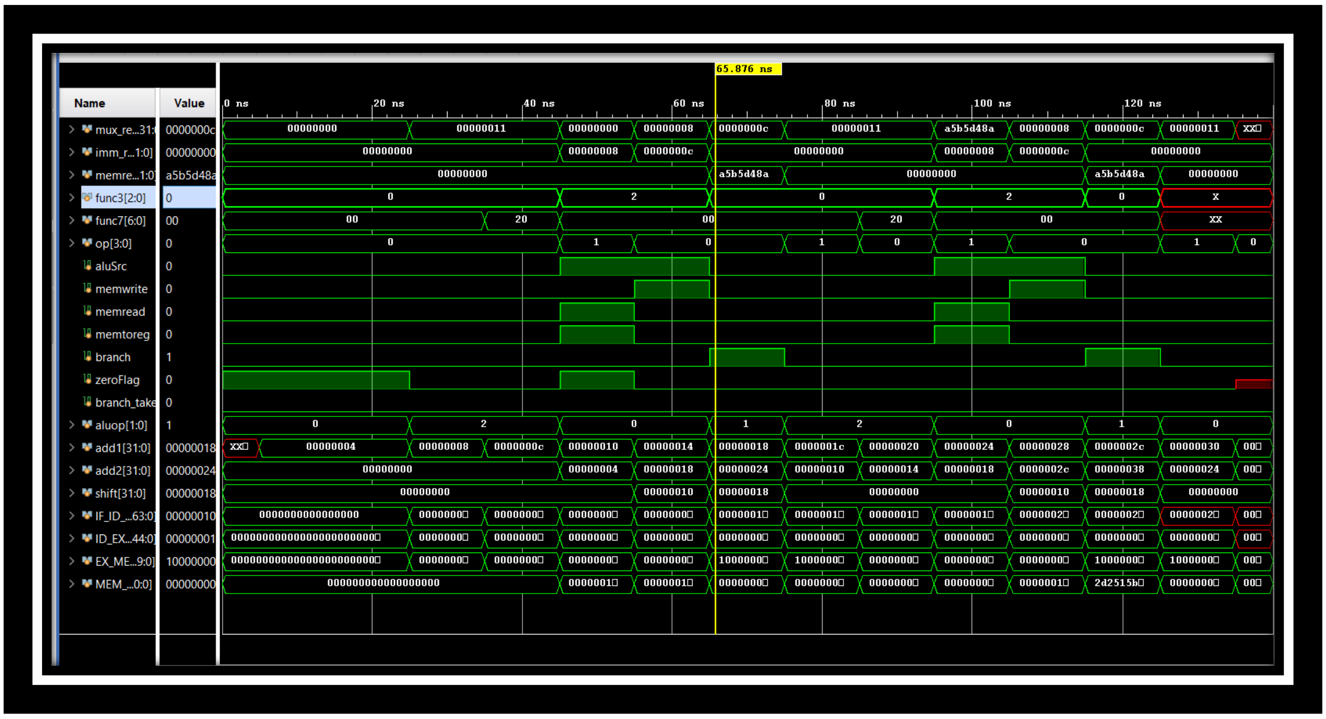
```
    mux wb_mux (
        .ri(MEM_WB_out[68:37]),
        .li(MEM_WB_out[36:5]),
        .sl(MEM_WB_out[0]),
        .res(wdata)
);
    BranchUnit bu(
        .branch(EXE_MEM_out[0]),
        .func3(func3),
        .zeroFlag(zeroFlag),
        .alu_result(EXE_MEM_out[101:70]),
        .branch_taken(branch_taken)
    );

    mux branch_mux (
        .ri(add1),
        .li(add2),
        .sl(branch_taken),
        .res(pc_in)
    );

endmodule
```

TOP TB FILE:



```
module Top_tb;
    logic clk;
    logic reset;
    Top dut (
        .clk(clk),
        .reset(reset)
    );
    initial clk = 0;
    always #5 clk = ~clk;
    initial begin
        reset = 1;
        #20 reset = 0;

        #500;
        $finish;
    end
    initial begin
        $dumpfile("Top_tb.vcd");
        $dumpvars(0, Top_tb);
    end
    always @(posedge clk) begin
        if (!reset) begin
            $display("Time: %0t | IF/ID PC: %0h | Instruction: %0h",
                     $time, dut.IF_ID_out[63:32], dut.IF_ID_out[31:0]);
            $display("Time: %0t | ID/EXE rs1: %0h | rs2: %0h | imm: %0h | rd: %0d",
                     $time, dut.ID_EXE_out[136:105], dut.ID_EXE_out[72:41], dut.ID_EXE_out[104:73], dut.ID_EXE_out[12:8]);
            $display("Time: %0t | EXE/MEM ALU result: %0h | branch target: %0h | rd: %0d",
                     $time, dut.EXE_MEM_out[101:70], dut.EXE_MEM_out[111:102], dut.EXE_MEM_out[37:33]);
            $display("Time: %0t | MEM/WB rd: %0d | WB data: %0h",
                     $time, dut.MEM_WB_out[37:33], dut.MEM_WB_out[36:5]);
            $display("------------------------------------------------------------");
        end
    end
```





CONCLUSION:

Now our processor fetches instructions, decodes them, executes ALU operations, accesses memory, and writes results back. The pipeline moves data across stages every clock. You saw that wrong PC updates and wrong pipeline slicing can stop instructions from advancing. You confirmed that instruction memory must be initialized. You saw that each stage must get correct control bits or the ALU and memory give wrong outputs. The lab showed you how to trace signals, how to check pipeline flow, and how to validate each stage cycle by cycle. You learned how to use a testbench to watch PC, instructions, register values, ALU outputs, and write back results. The main value is that you now understand how a pipelined datapath works, how pipeline registers hold state, and how control signals must align with data.