32-bit 5-Stage Pipelined RISC-V Processor

Verilog code Report

By

1	Muhammad Yasir	Roll No 37
2	Rehan Ali (Lead)	Roll No 15
3	Shariq Khan	Roll No 39
4	Rayan Badar	Roll No 18

Department of Computer Sciences

Namal University

Mianwali, Pakistan

Submission Date: 13th June, 2025

Verilog Code:

```
//.....Program Counter.....//
module program_counter(
   input clk,
   input rst,
   input PCWrite,          // Control signal from Hazard detection
   input [31:0] pc_in,
   output reg [31:0] pc_out
);
   always @(posedge clk or posedge rst) begin
      if (rst) begin
         pc_out <= 32'b0;
      end else if (PCWrite) begin
         pc_out <= pc_in;
      end
   end
endmodule

//.....PC Adder .....//
module pc_adder(
   input [31:0] pc_in,
   output reg [31:0] pc_next
);
   always @(*) begin
      pc_next = pc_in + 4;
   end
endmodule
//.....PC Mux (2x1) .....//
module pc_mux(
```

```
   input [31:0] pc_in,
   input [31:0] pc_branch,
   input pc_select,
   output reg[31:0] pc_out
);
   always @(*) begin
      pc_out = pc_select ? pc_branch : pc_in;
   end
endmodule

//..... ALU .....//
module ALU(
   input [31:0] A,
   input [31:0] B,
   input [3:0] ALUcontrol_In,
   output reg [31:0] Result,
   output reg Zero
);
   always @(*) begin
      case (ALUcontrol_In)
         4'b0000: Result = A + B;          // ADD
         4'b0001: Result = A - B;          // SUB
         4'b0010: Result = A & B;          // AND
         4'b0011: Result = A | B;          // OR
         4'b0100: Result = A ^ B;          // XOR
         4'b0101: Result = A << B[4:0];    // SLL
         4'b0110: Result = A >> B[4:0];    // SRL
         4'b0111: Result = $signed(A) >>> B[4:0]; // SRA
         4'b1000: Result = ($signed(A) < $signed(B)) ? 32'b1 : 32'b0; // SLT
         default: Result = 32'b0;
```

```
    endcase
    Zero = (Result == 32'b0);
  end
endmodule

//..... ALU control.....//
module SimpleALUControl (
  input [1:0] alu_op,
  input [6:0] func_code,
  input [2:0] op_select,
  output reg [3:0] control_signal
);
  always @(*) begin
    case (alu_op)
      2'b00: begin // I-type or Load/Store
        case (op_select)
          3'b000: control_signal = 4'b0000; // ADDI/LW/SW
          3'b111: control_signal = 4'b0010; // ANDI
          3'b110: control_signal = 4'b0011; // ORI
          3'b100: control_signal = 4'b0100; // XORI
          default: control_signal = 4'b0000;
        endcase
      end
      2'b01: begin // Branch (BEQ, BNE, etc.)
        control_signal = 4'b0001; // SUB for comparison
      end
      2'b10: begin // R-type
        case ({func_code[5], op_select})
          4'b0_000: control_signal = 4'b0000; // ADD
          4'b1_000: control_signal = 4'b0001; // SUB
```

```
            4'b0_111: control_signal = 4'b0010; // AND
            4'b0_110: control_signal = 4'b0011; // OR
            4'b0_100: control_signal = 4'b0100; // XOR
            4'b0_001: control_signal = 4'b0101; // SLL
            4'b0_101: control_signal = 4'b0110; // SRL
            4'b1_101: control_signal = 4'b0111; // SRA
            4'b0_010: control_signal = 4'b1000; // SLT
            default: control_signal = 4'b0000;
          endcase
        end
        default: control_signal = 4'b0000;
      endcase
  end
endmodule
//..... ALU Input A Mux .....//

module ALU_InputA_Mux (
  input [31:0] reg_data,    // From register file
  input [31:0] mem_data,    // From MEM/WB stage (writeback data)
  input [31:0] ex_data,     // From EX/MEM stage (ALU result)
  input [1:0] fwd_A,        // Forwarding control
  output reg [31:0] alu_A   // To ALU input A
);
  always @(*) begin
    case (fwd_A)
      2'b00: alu_A = reg_data;  // Normal operation
      2'b01: alu_A = mem_data;  // Forward from MEM/WB (writeback)
      2'b10: alu_A = ex_data;   // Forward from EX/MEM (ALU result)
      default: alu_A = reg_data;
    endcase
```

```
  end
endmodule

//..... ALU Input B Mux .....//
module ALU_InputB_Mux (
  input [31:0] reg_data,    // From register file
  input [31:0] mem_data,    // From MEM/WB stage (writeback data)
  input [31:0] ex_data,     // From EX/MEM stage (ALU result)
  input [31:0] imm_data,    // Immediate value
  input ALUSrc,             // Control signal
  input [1:0] fwd_B,        // Forwarding control
  output reg [31:0] alu_B   // To ALU input B
);
  reg [31:0] base_data;

  always @(*) begin
    // First handle forwarding
    case (fwd_B)
      2'b00: base_data = reg_data;  // Normal operation
      2'b01: base_data = mem_data;  // Forward from MEM/WB (writeback)
      2'b10: base_data = ex_data;   // Forward from EX/MEM (ALU result)
      default: base_data = reg_data;
    endcase

    // Then select between forwarded data and immediate
    alu_B = ALUSrc ? imm_data : base_data;
  end
endmodule
```

```
//..... register_file .....//
module reg_file(clk, reset, regwrite, rs1, rs2, rd, wd, rd1, rd2);
  input clk, reset, regwrite;
  input [4:0] rs1, rs2, rd;
  input [31:0] wd;
  output [31:0] rd1, rd2;
  reg [31:0] regis[31:0];

  integer i;

  always @(posedge clk or posedge reset) begin
    if (reset) begin
      // Initialize all registers to 0 except x0
      regis[0] <= 32'b0;
      regis[1] <= 32'd3;
      regis[2] <= 32'd2;
      regis[3] <= 32'd12;
      regis[4] <= 32'd20;
      regis[5] <= 32'd3;
      regis[6] <= 32'd44;
      regis[7] <= 32'd4;
      regis[8] <= 32'd2;
      regis[9] <= 32'd1;
      regis[10] <= 32'd23;
      regis[11] <= 32'd4;
      regis[12] <= 32'd90;
      regis[13] <= 32'd10;
      regis[14] <= 32'd20;
      regis[15] <= 32'd30;
      regis[16] <= 32'd40;
```

```
        regis[17] <= 32'd50;
        regis[18] <= 32'd60;
        regis[19] <= 32'd70;
        regis[20] <= 32'd80;
        regis[21] <= 32'd80;
        regis[22] <= 32'd90;
        regis[23] <= 32'd70;
        regis[24] <= 32'd60;
        regis[25] <= 32'd65;
        regis[26] <= 32'd4;
        regis[27] <= 32'd32;
        regis[28] <= 32'd12;
        regis[29] <= 32'd34;
        regis[30] <= 32'd5;
        regis[31] <= 32'd10;
    end
    else if (regwrite && (rd != 0)) begin  // x0 is always 0
        regis[rd] <= wd;
    end
  end

  assign rd1 = (rs1 != 0) ? regis[rs1] : 32'b0;
  assign rd2 = (rs2 != 0) ? regis[rs2] : 32'b0;
endmodule

//.............................instruction memory.....................//
module inst_mem(clk, rst, addr, inst);
  input clk, rst;
  input [31:0] addr;
  output [31:0] inst;
```

```
  reg [31:0] inst_mem[63:0];
  integer i;
  assign inst = inst_mem[addr];

  always @(posedge clk or posedge rst)
  begin
    if (rst) begin
      for (i = 0; i < 64; i = i + 1) begin
        inst_mem[i] = 32'b00;
      end
    end
    else begin
      // R-type
      inst_mem[0]  = 32'b000000000000_00000_000_00000_0010011;  // NOP (addi
x0, x0, 0)
      inst_mem[4]  = 32'b0000000_11001_10000_000_01101_0110011;  // add x13,
x16, x25
      inst_mem[8]  = 32'b0100000_00011_01000_000_00101_0110011;  // sub x5, x8,
x3
      inst_mem[12] = 32'b0000000_00011_00010_111_00001_0110011;  // and x1, x2,
x3
      inst_mem[16] = 32'b0000000_00101_00011_110_00100_0110011;  // or x4, x3,
x5
      inst_mem[20] = 32'b0000000_00101_00011_100_00100_0110011;  // xor x4, x3,
x5
      inst_mem[24] = 32'b0000000_00101_00011_001_00100_0110011;  // sll x4, x3,
x5
      inst_mem[28] = 32'b0000000_00101_00011_101_00100_0110011;  // srl x4, x3,
x5
      inst_mem[32] = 32'b0100000_00010_00011_101_00101_0110011;  // sra x5, x3,
x2
```

```
        inst_mem[36] = 32'b0000000_00010_00011_010_00101_0110011;  // slt x5, x3, x2


        // I-type
        inst_mem[40] = 32'b000000000010_10101_000_10110_0010011;  // addi x22, x21, 2
        inst_mem[44]  = 32'b000000000011_01000_110_01001_0010011;  // ori x9, x8, 3
        inst_mem[48] = 32'b000000000100_01000_100_01001_0010011;  // xori x9, x8, 4
        inst_mem[52] = 32'b000000000101_00010_111_00001_0010011;  // andi x1, x2, 5
        inst_mem[56] = 32'b000000000110_00011_001_00100_0010011;  // slli x4, x3, 6
        inst_mem[60] = 32'b000000000111_00011_101_00100_0010011;  // srli x4, x3, 7
        inst_mem[64] = 32'b010000001000_00011_101_00101_0010011;  // srai x5, x3, 8
        inst_mem[68] = 32'b000000001001_00011_010_00101_0010011;  // slti x5, x3, 9


        // L-type
        inst_mem[72] = 32'b000000000101_00011_000_01001_0000011;  // lb x9, 5(x3)
        inst_mem[76] = 32'b000000000011_00011_001_01001_0000011;  // lh x9, 3(x3)
        inst_mem[80] = 32'b000000001111_00010_010_01000_0000011;  // lw x8, 15(x2)


        // S-type
        inst_mem[84] = 32'b0000000_01111_00011_000_01000_0100011; // sb x15, 8(x3)
        inst_mem[88] = 32'b0000000_01110_00110_001_01010_0100011; // sh x14, 10(x6)
        inst_mem[92] = 32'b0000000_01110_00110_010_01100_0100011; // sw x14, 12(x6)


        // B-type
```

```
        inst_mem[96] = 32'b0_000000_01001_01001_000_0110_0_1100011; // beq x9, x9, 12
        inst_mem[100] = 32'b0_000000_01001_01001_001_0111_0_1100011; // bne x9, x9, 14


        // U-type
        inst_mem[104] = 32'b00000000000000101000_00011_0110111;    // lui x3, 40
        inst_mem[108] = 32'b00000000000000010100_00101_0010111;     // auipc x5, 20


        // J-type
        inst_mem[112] = 32'b0_00000000_0_0000010100_00001_1101111; // jal x1, 20


      end
   end
endmodule
//..... Data memory .....//
module memory_unit(
   input clk_sig,
   input rst_sig,
   input read_enable,
   input write_enable,
   input [31:0] address,
   input [31:0] input_data,
   output [31:0] output_data
);
   reg [31:0] memory_array [63:0];
   integer idx;


   assign output_data = (read_enable) ? memory_array[address] : 32'b0;


   always @(posedge clk_sig) begin
```

```
    memory_array[17] = 56;
    memory_array[15] = 65;
  end

  always @(posedge clk_sig or posedge rst_sig) begin
    if (rst_sig) begin
      for (idx = 0; idx < 64; idx = idx + 1) begin
        memory_array[idx] = 32'b0;
      end
    end else if (write_enable) begin
      memory_array[address] = input_data;
    end
  end

endmodule

//..... main control unit .....//
module control_unit(instrct, branch, memread, memtoreg, aluop, memwrite, alusrc,
regwrite);
  input [6:0] instrct;
  output reg branch, memread, memtoreg, memwrite, alusrc, regwrite;
  output reg [1:0] aluop;

  always @(*) begin
    // No default values - each case sets all outputs explicitly

    case (instrct)
  7'b0110011: begin // R-type
    alusrc   = 1'b0;
    memtoreg = 1'b0;
```

```
    regwrite = 1'b1;
    memread  = 1'b0;
    memwrite = 1'b0;
    branch   = 1'b0;
    aluop    = 2'b10;
  end
  7'b0010011: begin // I-type
    alusrc   = 1'b1;
    memtoreg = 1'b0;
    regwrite = 1'b1;
    memread  = 1'b0;
    memwrite = 1'b0;
    branch   = 1'b0;
    aluop    = 2'b10;
  end
  7'b0000011: begin // Load
    alusrc   = 1'b1;
    memtoreg = 1'b1;
    regwrite = 1'b1;
    memread  = 1'b1;
    memwrite = 1'b0;
    branch   = 1'b0;
    aluop    = 2'b00;
  end
  7'b0100011: begin // Store
    alusrc   = 1'b1;
    memtoreg = 1'b0; // Don't care
    regwrite = 1'b0;
    memread  = 1'b0;
    memwrite = 1'b1;
```

```
    branch   = 1'b0;
    aluop    = 2'b00;
  end
  7'b1100011: begin // Branch
    alusrc   = 1'b0;
    memtoreg = 1'b0; // Don't care
    regwrite = 1'b0;
    memread  = 1'b0;
    memwrite = 1'b0;
    branch   = 1'b1;
    aluop    = 2'b11;
  end
  7'b1101111: begin // Jump
    alusrc   = 1'b0;
    memtoreg = 1'b0;
    regwrite = 1'b1;
    memread  = 1'b0;
    memwrite = 1'b0;
    branch   = 1'b0;
    aluop    = 2'b10;
  end
  7'b0110111: begin // LUI
    alusrc   = 1'b0;
    memtoreg = 1'b0;
    regwrite = 1'b1;
    memread  = 1'b0;
    memwrite = 1'b0;
    branch   = 1'b0;
    aluop    = 2'b10;
  end
```

```
   default: begin
      alusrc   = 1'b0;
      memtoreg = 1'b0;
      regwrite = 1'b0;
      memread  = 1'b0;
      memwrite = 1'b0;
      branch   = 1'b0;
      aluop    = 2'b00;
   end
endcase


   end
endmodule


//..... immediate generator .....//
module imm_gen (
   input [31:0] instr,
   input [6:0] opcode,
   output reg [31:0] imm_out
);


   always @(*) begin
      case (opcode)
         7'b0010011: imm_out = {{20{instr[31]}}, instr[31:20]}; // I-type
         7'b0000011: imm_out = {{20{instr[31]}}, instr[31:20]}; // Load-type
         7'b0100011: imm_out = {{20{instr[31]}}, instr[31:25], instr[11:7]}; // Store-type
         7'b1100011: imm_out = {{19{instr[31]}}, instr[7], instr[30:25], instr[11:8], 1'b0};
// B-type
         7'b0110111: imm_out = {instr[31:12], 12'b0}; // U-type (LUI)
         7'b0010111: imm_out = {instr[31:12], 12'b0}; // U-type (AUIPC)
```

```
        7'b1101111: imm_out = {{11{instr[31]}}, instr[19:12], instr[20], instr[30:21], 1'b0}; // J-type (JAL)
        default:    imm_out = 32'b0; // Default case
      endcase
   end

endmodule

//.....Branch Adder.....//
module Branch_Adder(
   input [31:0] PC,
   input [31:0] offset,
   output reg [31:0] branch_target
);
   always @(*) begin
      branch_target = PC + offset;  // Fixed: No left shift needed
   end
endmodule

//.....IF/ID.....//
module IFID_reg(
   input clk, rst,
   input IFID_Write,          // Control signal from Hazard detection
   input flush,               // New input for branch flush
   input [31:0] pc_if, instr_if,
   output reg [31:0] pc_id, instr_id
);
   always@ (posedge clk or posedge rst) begin
      if(rst || flush) begin
         pc_id <= 32'b0;
```

```
        instr_id <= 32'b0;
      end else if (IFID_Write) begin
        pc_id <= pc_if;
        instr_id <= instr_if;
      end
   end
endmodule


//.....................................ID/EXE.....................//
module IDEX_reg (
   input clk, rst,
   input [31:0] pc_id, rd1_id, rd2_id, immGen_id,
   input [31:0] inst_id,
   input branch_id, memread_id, memtoreg_id, memwrite_id, alusrc_id, regwrite_id,
   input [1:0] aluop_id,
   output reg [31:0] pc_ex, rd1_ex, rd2_ex, immGen_ex, inst_ex,
   output reg branch_ex, memread_ex, memtoreg_ex, memwrite_ex, alusrc_ex,
regwrite_ex,
   output reg [1:0] aluop_ex,
   output reg [4:0] rd_ex, rs1_ex, rs2_ex
);
   always @(posedge clk or posedge rst) begin
      if (rst) begin
        pc_ex <= 32'b0;
        rd1_ex <= 32'b0;
        rd2_ex <= 32'b0;
        immGen_ex <= 32'b0;
        inst_ex <= 32'b0;
        aluop_ex <= 2'b0;
        branch_ex <= 1'b0;
```

```
        memread_ex <= 1'b0;
        memtoreg_ex <= 1'b0;
        memwrite_ex <= 1'b0;
        alusrc_ex <= 1'b0;
        regwrite_ex <= 1'b0;
        rd_ex <= 5'b0;
        rs1_ex <= 5'b0;
        rs2_ex <= 5'b0;
      end
      else begin
        pc_ex <= pc_id;
        rd1_ex <= rd1_id;
        rd2_ex <= rd2_id;
        immGen_ex <= immGen_id;
        inst_ex <= inst_id;
        aluop_ex <= aluop_id;
        branch_ex <= branch_id;
        memread_ex <= memread_id;
        memtoreg_ex <= memtoreg_id;
        memwrite_ex <= memwrite_id;
        alusrc_ex <= alusrc_id;
        regwrite_ex <= regwrite_id;
        rd_ex <= inst_id[11:7]; // rd field
        rs1_ex <= inst_id[19:15]; // rs1 field
        rs2_ex <= inst_id[24:20]; // rs2 field
      end
    end
endmodule
```

```
//.....EX/MEM.....//
module EXMEM (
   input clk, rst,
   input branch_ex, memread_ex, memtoreg_ex, memwrite_ex, regwrite_ex, zero_ex,
   input [31:0] jumpSum_ex, aluResult_ex, rd2_ex,
   input [4:0] rd_ex,
   output reg branch_mem, memread_mem, memtoreg_mem, memwrite_mem,
regwrite_mem, zero_mem,
   output reg [31:0] jumpSum_mem, aluResult_mem, rd2_mem,
   output reg [4:0] rd_mem
);
   always@ (posedge clk or posedge rst) begin
      if (rst) begin
         branch_mem <= 1'b0;
         memread_mem <= 1'b0;
         memtoreg_mem <= 1'b0;
         memwrite_mem <= 1'b0;
         regwrite_mem <= 1'b0;
         zero_mem <= 1'b0;
         jumpSum_mem <= 32'b0;
         aluResult_mem <= 32'b0;
         rd2_mem <= 32'b0;
         rd_mem <= 5'b0;
      end
      else begin
         branch_mem <= branch_ex;
         memread_mem <= memread_ex;
         memtoreg_mem <= memtoreg_ex;
         memwrite_mem <= memwrite_ex;
         regwrite_mem <= regwrite_ex;
```

```
        zero_mem <= zero_ex;
        jumpSum_mem <= jumpSum_ex;
        aluResult_mem <= aluResult_ex;
        rd2_mem <= rd2_ex;
        rd_mem <= rd_ex;
      end
   end
endmodule

//.....................................MEM/WB.....................//
module MEMWB (
   input clk, rst,
   input memtoreg_mem, regwrite_mem,
   input [31:0] aluResult_mem, read_data_mem,
   input [4:0] rd_mem,
   output reg memtoreg_wb, regwrite_wb,
   output reg [31:0] aluResult_wb, read_data_wb,
   output reg [4:0] rd_wb
);
   always@ (posedge clk or posedge rst) begin
      if (rst) begin
         memtoreg_wb <= 1'b0;
         regwrite_wb <= 1'b0;
         aluResult_wb <= 32'b0;
         read_data_wb <= 32'b0;
         rd_wb <= 5'b0;
      end
      else begin
         memtoreg_wb <= memtoreg_mem;
         regwrite_wb <= regwrite_mem;
```

```
        aluResult_wb <= aluResult_mem;
        read_data_wb <= read_data_mem;
        rd_wb <= rd_mem;
      end
   end
endmodule

//.....Forwarding Unit .....//
module Forwarding_Unit(
   input [4:0] rs1_IDEX, rs2_IDEX,
   input [4:0] EXMEM_rd, MEMWB_rd,
   input EXMEM_RegWrite, MEMWB_RegWrite,
   output reg [1:0] fwd_A, fwd_B
);
   always @(*) begin
      // Default no forwarding
      fwd_A = 2'b00;
      fwd_B = 2'b00;

      // Forwarding for operand A (rs1)
      if (EXMEM_RegWrite && (EXMEM_rd != 0) && (EXMEM_rd == rs1_IDEX)) begin
         fwd_A = 2'b10; // Forward from EX/MEM (highest priority)
      end
      else if (MEMWB_RegWrite && (MEMWB_rd != 0) && (MEMWB_rd == rs1_IDEX))
begin
         fwd_A = 2'b01; // Forward from MEM/WB
      end

      // Forwarding for operand B (rs2)
      if (EXMEM_RegWrite && (EXMEM_rd != 0) && (EXMEM_rd == rs2_IDEX)) begin
```

```
        fwd_B = 2'b10;
      end
      else if (MEMWB_RegWrite && (MEMWB_rd != 0) && (MEMWB_rd == rs2_IDEX))
begin
        fwd_B = 2'b01;
      end
   end
endmodule

//.....Hazard Detection.....//
module Hazard_Detection(
   input [4:0] IDEX_rd, IFID_rs1, IFID_rs2,
   input IDEX_MemRead,
   output reg PCWrite, IFID_Write,
   output reg stall
);
   always @(*) begin
      stall = 1'b0;
      PCWrite = 1'b1;
      IFID_Write = 1'b1;

      if (IDEX_MemRead && (IDEX_rd != 0) &&
        ((IDEX_rd == IFID_rs1) || (IDEX_rd == IFID_rs2))) begin
         stall = 1'b1;
         PCWrite = 1'b0;
         IFID_Write = 1'b0;
      end
   end
endmodule
```

```
//..... Writeback Mux (2x1) .....//
module WBMux(
   input [31:0] alu_result,    // From ALU result
   input [31:0] mem_data,      // From data memory
   input memtoreg,             // Control signal
   output reg [31:0] wb_data   // To register file
);
   always @(*) begin
      wb_data = memtoreg ? mem_data : alu_result;
   end
endmodule


//..... Hazard Control Mux  .....//
module HazardControlMux(
   input branch_in, memread_in, memtoreg_in, memwrite_in, alusrc_in, regwrite_in,
   input [1:0] aluop_in,
   input stall,
   output reg branch_out, memread_out, memtoreg_out, memwrite_out, alusrc_out,
regwrite_out,
   output reg [1:0] aluop_out
);
   always @(*) begin
      if (stall) begin
         // Disable all control signals during a stall
         branch_out    = 1'b0;
         memread_out   = 1'b0;
         memtoreg_out  = 1'b0;
         memwrite_out  = 1'b0;
         alusrc_out    = 1'b0;
         regwrite_out  = 1'b0;
```

```
        aluop_out     = 2'b00;
      end
      else begin
        // Pass through normally
        branch_out    = branch_in;
        memread_out   = memread_in;
        memtoreg_out  = memtoreg_in;
        memwrite_out  = memwrite_in;
        alusrc_out    = alusrc_in;
        regwrite_out  = regwrite_in;
        aluop_out     = aluop_in;
      end
    end
endmodule

//.....RISC-V Top.....//
module RISCV_top(
    input clk,
    input reset
);
    // IF Stage Signals
    wire [31:0] pc_if, pc_next_if, pc_branch_if, inst_if;
    wire PCWrite, IFID_Write;

    // ID Stage Signals
    wire [31:0] pc_id, inst_id;
    wire [31:0] rd1_id, rd2_id, imm_gen_result;
    wire branch_id, mem_read_id, mem_to_reg_id, mem_write_id, alu_src_id, reg_write_id;
    wire [1:0] alu_op_id;
    wire stall;
```

```
  // After Hazard Mux
  wire branch_gated, memread_gated, memtoreg_gated, memwrite_gated, alusrc_gated, regwrite_gated;
  wire [1:0] aluop_gated;

  // EX Stage Signals
  wire [31:0] pc_ex, rd1_ex, rd2_ex, immGen_ex, inst_ex;
  wire [31:0] alu_srcA, alu_srcB, alu_result_ex, jumpSum_ex;
  wire [3:0] alu_control_wire;
  wire branch_ex, memread_ex, memtoreg_ex, memwrite_ex, alusrc_ex, regwrite_ex, zero_ex;
  wire [1:0] aluop_ex;
  wire [1:0] fwd_A, fwd_B;
  wire [4:0] rs1_ex, rs2_ex, rd_ex;

  // MEM Stage Signals
  wire [31:0] aluResult_mem, rd2_mem, dm_result_wire, jumpSum_mem;
  wire [4:0] rd_mem;
  wire branch_mem, memread_mem, memtoreg_mem, memwrite_mem, regwrite_mem, zero_mem;

  // WB Stage Signals
  wire [31:0] aluResult_wb, read_data_wb, writeback_wire;
  wire [4:0] rd_wb;
  wire memtoreg_wb, regwrite_wb;

  // Branch control signals
  wire branch_taken = (branch_mem & zero_mem);  // Fixed: Use MEM stage signals
  wire flush = branch_taken;
```

```
//////////////////////////////////////////
// Instruction Fetch Stage
//////////////////////////////////////////
program_counter PC(
   .clk(clk),
   .rst(reset),
   .PCWrite(PCWrite),
   .pc_in(pc_branch_if),
   .pc_out(pc_if)
);

pc_adder PC_adder(
   .pc_in(pc_if),
   .pc_next(pc_next_if)
);

pc_mux PC_mux(
   .pc_in(pc_next_if),
   .pc_branch(jumpSum_mem),  // Fixed: Use MEM stage branch target
   .pc_select(branch_taken),
   .pc_out(pc_branch_if)
);

inst_mem instruction_memory(
   .clk(clk),
   .rst(reset),
   .addr(pc_if),
   .inst(inst_if)
);
```

```
IFID_reg IFID(
    .clk(clk),
    .rst(reset),
    .IFID_Write(IFID_Write),
    .flush(flush),
    .pc_if(pc_if),
    .instr_if(inst_if),
    .pc_id(pc_id),
    .instr_id(inst_id)
);

//////////////////////////////////////////
// Instruction Decode Stage
//////////////////////////////////////////
reg_file register_file(
    .clk(clk),
    .reset(reset),
    .regwrite(regwrite_wb),
    .rs1(inst_id[19:15]),
    .rs2(inst_id[24:20]),
    .rd(rd_wb),
    .wd(writeback_wire),
    .rd1(rd1_id),
    .rd2(rd2_id)
);

control_unit main_control_unit(
    .instrct(inst_id[6:0]),
    .branch(branch_id),
    .memread(mem_read_id),
```

```
    .memtoreg(mem_to_reg_id),
    .aluop(alu_op_id),
    .memwrite(mem_write_id),
    .alusrc(alu_src_id),
    .regwrite(reg_write_id)
);

imm_gen immediate_gen(
    .instr(inst_id),
    .imm_out(imm_gen_result),
    .opcode(inst_id[6:0])
);

Hazard_Detection hazard_detection(
    .IDEX_rd(rd_ex),
    .IFID_rs1(inst_id[19:15]),
    .IFID_rs2(inst_id[24:20]),
    .IDEX_MemRead(memread_ex),
    .PCWrite(PCWrite),
    .IFID_Write(IFID_Write),
    .stall(stall)
);

HazardControlMux hazard_mux (
    .branch_in(branch_id),
    .memread_in(mem_read_id),
    .memtoreg_in(mem_to_reg_id),
    .memwrite_in(mem_write_id),
    .alusrc_in(alu_src_id),
    .regwrite_in(reg_write_id),
```

```
    .aluop_in(alu_op_id),
    .stall(stall),
    .branch_out(branch_gated),
    .memread_out(memread_gated),
    .memtoreg_out(memtoreg_gated),
    .memwrite_out(memwrite_gated),
    .alusrc_out(alusrc_gated),
    .regwrite_out(regwrite_gated),
    .aluop_out(aluop_gated)
);

IDEX_reg IDEX(
    .clk(clk),
    .rst(reset | flush),
    .pc_id(pc_id),
    .rd1_id(rd1_id),          // Fixed: Use correct signals
    .rd2_id(rd2_id),          // Fixed: Use correct signals
    .immGen_id(imm_gen_result), // Fixed: Use correct signal
    .inst_id(inst_id),
    .branch_id(branch_gated),
    .memread_id(memread_gated),
    .memtoreg_id(memtoreg_gated),
    .memwrite_id(memwrite_gated),
    .alusrc_id(alusrc_gated),
    .regwrite_id(regwrite_gated),
    .aluop_id(aluop_gated),
    .pc_ex(pc_ex),
    .rd1_ex(rd1_ex),
    .rd2_ex(rd2_ex),
    .immGen_ex(immGen_ex),
```

```
    .inst_ex(inst_ex),
    .branch_ex(branch_ex),
    .memread_ex(memread_ex),
    .memtoreg_ex(memtoreg_ex),
    .memwrite_ex(memwrite_ex),
    .alusrc_ex(alusrc_ex),
    .regwrite_ex(regwrite_ex),
    .aluop_ex(aluop_ex),
    .rd_ex(rd_ex),
    .rs1_ex(rs1_ex),
    .rs2_ex(rs2_ex)
);

//////////////////////////////////////////
// Execute Stage
//////////////////////////////////////////
ALU_InputA_Mux alu_inputA_mux (
    .reg_data(rd1_ex),
    .mem_data(writeback_wire),  // Fixed: Writeback data from WB stage
    .ex_data(aluResult_mem),    // Fixed: ALU result from MEM stage
    .fwd_A(fwd_A),
    .alu_A(alu_srcA)
);

ALU_InputB_Mux alu_inputB_mux (
    .reg_data(rd2_ex),
    .mem_data(writeback_wire),  // Fixed: Writeback data from WB stage
    .ex_data(aluResult_mem),    // Fixed: ALU result from MEM stage
    .imm_data(immGen_ex),
    .ALUSrc(alusrc_ex),
```

```
    .fwd_B(fwd_B),
    .alu_B(alu_srcB)
);

ALU alu (
    .A(alu_srcA),
    .B(alu_srcB),
    .ALUcontrol_In(alu_control_wire),
    .Result(alu_result_ex),
    .Zero(zero_ex)
);

SimpleALUControl alu_control (
    .alu_op(aluop_ex),
    .func_code(inst_ex[31:25]),
    .op_select(inst_ex[14:12]),
    .control_signal(alu_control_wire)
);

Branch_Adder branch_adder (
    .PC(pc_ex),
    .offset(immGen_ex),
    .branch_target(jumpSum_ex)
);

Forwarding_Unit forwarding_unit (
    .rs1_IDEX(rs1_ex),
    .rs2_IDEX(rs2_ex),
    .EXMEM_rd(rd_mem),
    .MEMWB_rd(rd_wb),
```

```
    .EXMEM_RegWrite(regwrite_mem),
    .MEMWB_RegWrite(regwrite_wb),
    .fwd_A(fwd_A),
    .fwd_B(fwd_B)
);

EXMEM EXMEM_reg (
    .clk(clk),
    .rst(reset),
    .branch_ex(branch_ex),
    .memread_ex(memread_ex),
    .memtoreg_ex(memtoreg_ex),
    .memwrite_ex(memwrite_ex),
    .regwrite_ex(regwrite_ex),
    .zero_ex(zero_ex),
    .jumpSum_ex(jumpSum_ex),
    .aluResult_ex(alu_result_ex),
    .rd2_ex(rd2_ex),
    .rd_ex(rd_ex),
    .branch_mem(branch_mem),
    .memread_mem(memread_mem),
    .memtoreg_mem(memtoreg_mem),
    .memwrite_mem(memwrite_mem),
    .regwrite_mem(regwrite_mem),
    .zero_mem(zero_mem),
    .jumpSum_mem(jumpSum_mem),
    .aluResult_mem(aluResult_mem),
    .rd2_mem(rd2_mem),
    .rd_mem(rd_mem)
);
```

```
//////////////////////////////////////////
// Memory Stage
//////////////////////////////////////////
memory_unit data_memory (
   .clk_sig(clk),
   .rst_sig(reset),
   .address(aluResult_mem),
   .input_data(rd2_mem),
   .read_enable(memread_mem),
   .write_enable(memwrite_mem),
   .output_data(dm_result_wire)
);

MEMWB MEMWB_reg (
   .clk(clk),
   .rst(reset),
   .memtoreg_mem(memtoreg_mem),
   .regwrite_mem(regwrite_mem),
   .aluResult_mem(aluResult_mem),
   .read_data_mem(dm_result_wire),
   .rd_mem(rd_mem),
   .memtoreg_wb(memtoreg_wb),
   .regwrite_wb(regwrite_wb),
   .aluResult_wb(aluResult_wb),
   .read_data_wb(read_data_wb),
   .rd_wb(rd_wb)
);

//////////////////////////////////////////
```

```
  // Writeback Stage
  ////////////////////////////////////////
  WBMux wb_mux (
     .alu_result(aluResult_wb),
     .mem_data(read_data_wb),
     .memtoreg(memtoreg_wb),
     .wb_data(writeback_wire)
  );

endmodule
```

Testbench:

```
module RISCV_top_tb();

  // Inputs
  reg clk;
  reg reset;

  // Instantiate the Unit Under Test (UUT)
  RISCV_top uut (
     .clk(clk),
     .reset(reset)
  );

  // Clock generation
  initial begin
     clk = 0;
     forever #5 clk = ~clk;
  end
```

```
  // Test procedure
  initial begin
    // Initialize and reset
    reset = 1;
    #10;
    reset = 0;

    // Run for enough cycles to execute all instructions
    #1000;

    // Display selected results
    $display("\nKey Results:");
    $display("x13 (ADD) = %h", uut.register_file.regis[13]);
    $display("x5 (SUB) = %h", uut.register_file.regis[5]);
    $display("x1 (AND) = %h", uut.register_file.regis[1]);
    $display("x22 (ADDI) = %h", uut.register_file.regis[22]);
    $display("Mem[3] (SW) = %h", uut.data_memory.memory_array[3]);

    $finish;
  end

  // Waveform dumping
  initial begin
    $dumpfile("riscv_tb.vcd");
    $dumpvars(0, RISCV_top_tb);
  end

endmodule
```