Mihir Phadke
Haydon Behl
Ryan Smith
Jai Nayyar
Thurman Dao
Anson Lee

Group 9
Apr 17, 2025

Lab 6

```
module ALU(
    input [31:0] op1,
    input [31:0] op2,
    input [2:0] funct3,
    input [6:0] funct7,
    input [4:0] shamt,
    input [2:0] insn_type,
    output reg [31:0] result
);
    always @ (*) begin
        case (insn_type)
            3'b000: begin
                case (funct3)
                    // ADDI
                    3'b000: result = op1 + op2;
                    // SLTI
                    3'b010: begin
                        if ($signed(op1) < $signed(op2)) begin
                            result = 1;
                        end
                        else begin
                            result = 0;
                        end
                    end
                    // SLTIU
                    3'b011: begin
                        if (op1 < op2) begin
                            result = 1;
                        end
            else begin
                result = 0;
            end
        end
        // XORI
        3'b100: result = op1 ^ op2;
        // ORI
        3'b110: result = op1 | op2;
        // ANDI
        3'b111: result = op1 & op2;
        // SLLI
        3'b001: result = op1 << shamt;
        // SRLI / SRAI
        3'b101: begin
            if (funct7 == 7'b0100000) begin
                // SRAI
                result = $signed(op1) >>> shamt;
            end
            else if (funct7 == 7'b0000000) begin
                // SRLI
                result = op1 >> shamt;
            end
            else begin
                // Default case for funct3 = 3'b101
                result = 32'b0;
            end
        end
        // Default case for funct3
        default: result = 32'b0;
    endcase
end
3'b001: begin
    case(funct3)
        // add or sub
        3'b000: begin
            if(funct7 == 7'b0000000) begin
                result = op1 + op2;
            end
            else if(funct7 == 7'b0100000) begin
                result = op1 - op2;
                end
            end
            3'b001:
                result = op1 << op2[4:0];
            3'b010: begin
                if ($signed(op1) < $signed(op2)) begin
                    result = 1;
                end
                else begin
                    result = 0;
                end
            end
            3'b011: begin
                if (op1 < op2) begin
                    result = 1;
                end
                else begin
                    result = 0;
                end
            end
            3'b100:
                result = op1 ^ op2;
            3'b101: begin
                if(funct7 == 7'b0100000) begin
                    result = $signed(op1) >>> op2[4:0];
                end
                else if(funct7 == 7'b0000000) begin
                    result = op1 >> op2[4:0];
                end
            end
            3'b110:
                result = op1 | op2;
            3'b111:
                result = op1 & op2;
            default: result = 32'b0;
        endcase
    end
    // Load-Store (Stores): Calculate effective address for store instructions
    3'b010: begin
        result = op1 + op2; // effective address = base address + offset
```

```
            end

            // Load-Store (Loads): Calculate effective address for load instructions
            3'b011: begin
                result = op1 + op2; // effective address = base address + offset
            end

            // Default case for insn_type
            default: result = 32'b0;
        endcase
    end
endmodule
```

```
module PC(
    input rst_n,
    input clk,
    input stall,
    output reg [31:0] pc
);
    always @(posedge clk or negedge rst_n) begin
      if (!rst_n) begin
            pc <= 32'b0;
      end
      else begin
            pc <= pc + 4;
      end
    end
endmodule
```

```
module cpu(
    input         rst_n,
    input         clk,
    output reg [31:0] imem_addr,
    input  [31:0] imem_insn,
    output reg [31:0] dmem_addr,
    inout  [31:0] dmem_data,
    output reg        dmem_wen,
    output      [3:0] byte_en
```

```verilog
);
    // Stall register
    reg stall;

    // IF/ID pipeline registers
    reg [31:0] IF_ID_insn, IF_ID_pc;
    reg ID_wen;
    reg ID_dmem_wen;

    // ID/EX pipeline registers
    reg [31:0] ID_EX_pc, ID_EX_imm;
    reg [4:0]  ID_EX_dest, ID_EX_src1, ID_EX_src2;
    reg [2:0]  ID_EX_insn_type;
    reg [2:0]  ID_EX_funct3;
    reg [4:0]  ID_EX_shamt;
    reg [6:0]  ID_EX_funct7;
    reg        ID_EX_alu_op_mux;
    reg        ID_EX_wen;
    reg        ID_EX_dmem_wen;
    reg [31:0] ID_EX_store_data;

    // EX/MEM pipeline registers
    reg [31:0] EX_MEM_alu_result;
    reg [4:0]  EX_MEM_dest;
    reg        EX_MEM_wen;
    reg        EX_MEM_dmem_wen;
    reg [2:0]  EX_MEM_insn_type;  // distinguishes load (3'b011) vs. store (3'b010)
    reg [2:0]  EX_MEM_funct3;
    reg [31:0] EX_MEM_store_data; // Propagated store data
    reg        ram_forward_flag;

    // MEM/WB pipeline registers
    reg signed [31:0] MEM_WB_result;
    reg [4:0]         MEM_WB_dest;
    reg               MEM_WB_wen;
    reg [2:0]         MEM_WB_insn_type;
    reg [2:0]         MEM_WB_funct3;
    reg [31:0]        MEM_WB_alu_result;
    reg [31:0]        MEM_WB_ram_result;
```

```
// dmem_data tri-state control:
// dmem_data is driven by dmem_data_out when writing;
// otherwise it is tri-stated so that RAM can drive it during loads.
reg [31:0] dmem_data_out;
assign dmem_data = (dmem_wen) ? dmem_data_out : 32'hz;

// byte_en output and its internal register. For RAM
reg [3:0] byte_en_dmem_reg;
assign byte_en = byte_en_dmem_reg;
// byte_en internal register. For regfile
reg [3:0] byte_en_reg;

// Clock Cycle Counter
reg [15:0] cycle_counter;
always @(posedge clk or negedge rst_n) begin
    if (!rst_n)
        cycle_counter <= 16'b0;
    else
        cycle_counter <= cycle_counter + 1;
end

// Program Counter Module
wire [31:0] pc, next_pc;
PC pc_module(
    .rst_n(rst_n),
    .clk(clk),
    .stall(stall),
    .pc(pc)
);
always @(*) begin
    imem_addr = pc;
end

// Instruction Fetch Stage
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        IF_ID_insn <= 32'b0;
        IF_ID_pc   <= 32'b0;
    end else begin
        IF_ID_insn <= imem_insn;
        IF_ID_pc   <= pc;
    end
end

// Decode Stage: Instruction Decoder instantiation.
wire [4:0] destination_reg;
wire [2:0] funct3;
wire [4:0] source_reg1;
wire [4:0] source_reg2;
wire [11:0] imm;
wire [6:0] funct7;
wire [4:0] shamt;
wire [2:0] insn_type;
wire alu_op_mux;
wire wen;

instruction_decoder decoder(
    .imem_insn(IF_ID_insn),
    .destination_reg(destination_reg),
    .funct3(funct3),
    .source_reg1(source_reg1),
    .source_reg2(source_reg2),
    .imm(imm),
    .funct7(funct7),
    .shamt(shamt),
    .insn_type(insn_type),
    .alu_op_mux(alu_op_mux),
    .wen(ID_wen),
    .dmem_wen(ID_dmem_wen)
);

// ID/EX Pipeline Register Update (including store data capture)
// For store instructions, the store data comes from reg_data2.
// (If needed, you might add forwarding for store data as well.)
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        ID_EX_pc        <= 32'b0;
        ID_EX_dest      <= 5'b0;
        ID_EX_src1      <= 5'b0;
        ID_EX_src2      <= 5'b0;
        ID_EX_imm       <= 32'b0;
        ID_EX_funct3    <= 3'b0;
        ID_EX_funct7    <= 7'b0;
        ID_EX_insn_type  <= 3'b111;
        ID_EX_shamt     <= 5'b0;
        ID_EX_alu_op_mux <= 0;
        ID_EX_wen       <= 0;
        ID_EX_dmem_wen   <= 0;
        ID_EX_store_data <= 32'b0;
    end else begin
        ID_EX_pc        <= IF_ID_pc;
        ID_EX_dest      <= destination_reg;
        ID_EX_src1      <= source_reg1;
        ID_EX_src2      <= source_reg2;
        ID_EX_insn_type  <= insn_type;
        ID_EX_funct3    <= funct3;
        ID_EX_funct7    <= funct7;
        ID_EX_shamt     <= shamt;
        ID_EX_alu_op_mux <= alu_op_mux;
        ID_EX_imm       <= {{20{imm[11]}}, imm};
        ID_EX_wen       <= ID_wen;
        ID_EX_dmem_wen   <= ID_dmem_wen;
    end
end

// Register File Instance
wire [31:0] reg_data1;
wire [31:0] reg_data2;
register_file reg_file(
    .clk(clk),
    .rst_n(rst_n),
    .wen(MEM_WB_wen),
    .destination_reg(MEM_WB_dest),
    .source_reg1(ID_EX_src1),
    .source_reg2(ID_EX_src2),
    .write_data(MEM_WB_result),
    .byte_en(byte_en_reg),
    .read_data1(reg_data1),
    .read_data2(reg_data2)
);
```

```
    // Forwarding Logic
    wire [31:0] forwarded_op1;
    wire [31:0] forwarded_op2;

    // Register "skips" for when the requested data is not yet written to reg.
    assign forwarded_op1 = ((EX_MEM_dest != 5'b0) && (EX_MEM_dest == ID_EX_src1)
&& EX_MEM_wen) ? EX_MEM_alu_result :
                           ((MEM_WB_dest != 5'b0) && (MEM_WB_dest == ID_EX_src1) &&
MEM_WB_wen) ? MEM_WB_result :
                           reg_data1;
    assign forwarded_op2 = (ID_EX_alu_op_mux) ? ID_EX_imm :
                           ((EX_MEM_dest != 5'b0) && (EX_MEM_dest == ID_EX_src2) &&
EX_MEM_wen) ? EX_MEM_alu_result :
                           ((MEM_WB_dest != 5'b0) && (MEM_WB_dest == ID_EX_src2) &&
MEM_WB_wen) ? MEM_WB_result :
                           reg_data2;

    // Execute Stage: ALU instance.
    wire [31:0] alu_result;
    ALU alu(
        .op1(forwarded_op1),
        .op2(forwarded_op2),
        .shamt(ID_EX_shamt),
        .funct3(ID_EX_funct3),
        .funct7(ID_EX_funct7),
        .insn_type(ID_EX_insn_type),
        .result(alu_result)
    );

    // EX/MEM Pipeline Register Update (propagating store data)
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            EX_MEM_alu_result <= 32'b0;
            EX_MEM_dest       <= 5'b0;
            EX_MEM_wen        <= 1'b0;
            EX_MEM_dmem_wen   <= 1'b0;
            EX_MEM_funct3     <= 3'b0;
            EX_MEM_insn_type  <= 3'b0;
            EX_MEM_store_data <= 32'b0;
            ram_forward_flag  <= 1'b0;
        end else begin
            EX_MEM_alu_result <= alu_result;
            EX_MEM_dest       <= ID_EX_dest;
            EX_MEM_wen        <= ID_EX_wen;
            EX_MEM_dmem_wen   <= ID_EX_dmem_wen;
            EX_MEM_funct3     <= ID_EX_funct3;    // Propagate store type info
            EX_MEM_insn_type  <= ID_EX_insn_type; // 3'b010: store, 3'b011: load
            EX_MEM_store_data <= ID_EX_store_data;

            // Load / Store, capture store data from forwarding before passing imm offset value
            if (ram_forward_flag == 1'b1) begin
                EX_MEM_store_data = MEM_WB_result;
                ram_forward_flag = 1'b0;
            end
            if ((ID_EX_insn_type == 3'b010) && (EX_MEM_dest != 5'b0) && (EX_MEM_dest
== ID_EX_src2) && EX_MEM_wen) begin
                ID_EX_store_data = EX_MEM_alu_result; // not needed?
                ram_forward_flag = 1'b1;
            end
        end
    end

    // Used when reading from RAM, for when the memory being accessed is not yet populated
    // Outputs 1 instead of X bits
    wire [31:0] fixed_data;
    genvar i;
    generate
        for(i = 0; i < 32; i = i + 1) begin : fix_x_bits
            assign fixed_data[i] = (dmem_data[i] === 1'bx) ? 1'b1 : dmem_data[i];
        end
    endgenerate
    //assign fixed_data = dmem_data;

    // Combinational Load Extraction in the MEM stage:
    // This wire computes the correctly extracted and extended load value based on
    // the effective address (MEM_WB_alu_result) and funct3. Note that it is valid only when
    // a load instruction (MEM_WB_insn_type == 3'b011) is in the MEM stage.
    wire [31:0] load_result;
    assign load_result = (MEM_WB_insn_type == 3'b011) ? (
        (MEM_WB_funct3 == 3'b000) ? // LB: sign-extended byte
            ( {{24{fixed_data[7]}},  fixed_data[7:0]} )
        : (MEM_WB_funct3 == 3'b001) ? // LH: sign-extended halfword
            ( {{16{fixed_data[15]}}, fixed_data[15:0]} )
        : (MEM_WB_funct3 == 3'b010) ? // LW: load word
            fixed_data
        : (MEM_WB_funct3 == 3'b100) ? // LBU: zero-extended byte
            ( {24'b0, fixed_data[7:0]} )
        : (MEM_WB_funct3 == 3'b101) ? // LHU: zero-extended halfword
            ( {16'b0, fixed_data[15:0]} )
        : fixed_data
    ) : 32'b0;

    // Register Load: capture memory data OR alu results
    assign MEM_WB_result = (MEM_WB_insn_type == 3'b011) ? load_result :
MEM_WB_alu_result;

    // MEM/WB Pipeline Register Update:
    // For load instructions (insn_type == 3'b011), capture the data from dmem_data (in
fixed_data).
    // For other instructions, pass the ALU result.
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            MEM_WB_alu_result <= 32'b0;
            MEM_WB_dest   <= 5'b0;
            MEM_WB_wen    <= 1'b0;
            MEM_WB_insn_type <= 1'b0;
            MEM_WB_funct3 <= 3'b0;
            dmem_wen <= 1'b0;
            dmem_addr <= 32'b0;
            byte_en_dmem_reg <= 4'b0;
            byte_en_reg <= 4'b0;
        end else begin
            dmem_data_out = EX_MEM_store_data;
            if (EX_MEM_insn_type == 3'b011) begin
                dmem_addr <= EX_MEM_alu_result; // Use the effective address computed in EX
stage.

                byte_en_dmem_reg <= 4'b0000;
                byte_en_reg <= 4'b1111;
        end else if (EX_MEM_insn_type == 3'b010) begin
            dmem_addr <= EX_MEM_alu_result; // Use the effective address computed in EX
stage.

            // When executing a store (insn_type == 3'b010), drive dmem_wen and set
byte_en_dmem.
            case (EX_MEM_funct3)
                3'b000: begin // SB (Store Byte)
                    byte_en_dmem_reg <= 4'b0001;
                end
                3'b001: begin // SH (Store Halfword)
                    byte_en_dmem_reg <= 4'b0011;
                end
                3'b010: begin // SW (Store Word)
                    byte_en_dmem_reg <= 4'b1111;
                end
                default: byte_en_dmem_reg <= 4'b0000;
            endcase
            byte_en_reg <= 4'b0000;
        end else begin
            dmem_addr <= 32'b0;

            byte_en_dmem_reg <= 4'b0000;
            byte_en_reg <= 4'b1111;
        end

        MEM_WB_funct3 <= EX_MEM_funct3;
        MEM_WB_alu_result <= EX_MEM_alu_result;
        MEM_WB_dest <= EX_MEM_dest;
        MEM_WB_wen  <= EX_MEM_wen;
        MEM_WB_insn_type <= EX_MEM_insn_type;
        dmem_wen <= EX_MEM_dmem_wen;
    end
end

// File Trace Output (for debugging/tracing purposes)
integer fd_pc, fd_data;
initial begin
    fd_pc = $fopen("pc.txt", "w");
```

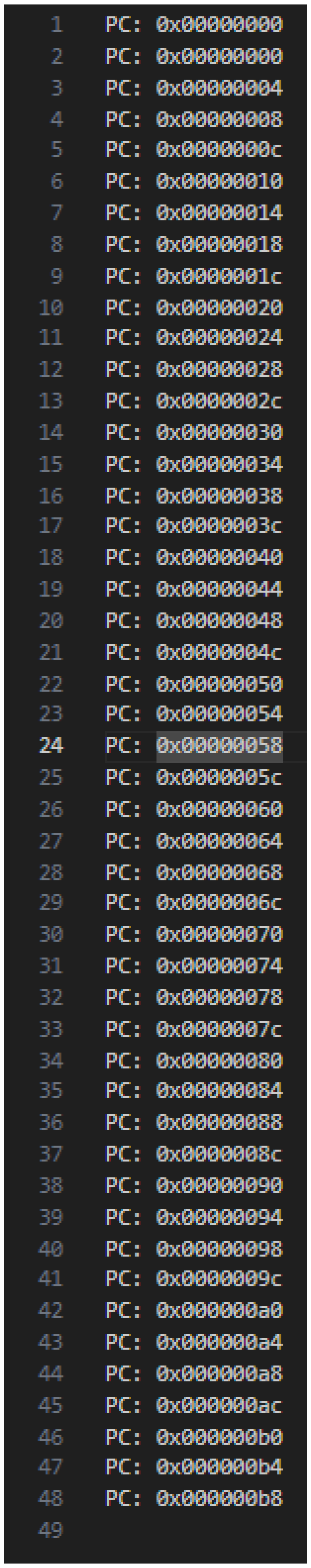
```
        fd_data = $fopen("data.txt", "w");
        if (fd_pc == 0 || fd_data == 0) begin
            $display("Error: Could not open file.");
            $finish;
        end
        $display("File descriptors: fd_pc = %0d, fd_data = %0d", fd_pc, fd_data);
    end

    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            // Do nothing on reset.
        end else begin
            $display("IF_ID_pc: %h, MEM_WB_result: %h, MEM_WB_dest: %h", IF_ID_pc,
MEM_WB_result, MEM_WB_dest);
            $fdisplay(fd_pc, "PC: 0x%h", IF_ID_pc);
            $fdisplay(fd_data, "MEM_WB_result: %d", MEM_WB_result);
            $fdisplay(fd_data, "MEM_WB_dest: %d", MEM_WB_dest);
            $fdisplay(fd_data, "-----------------------------------");
        end
    end

    initial begin
        #1000
        $fclose(fd_pc);
        $fclose(fd_data);
        $display("Files closed successfully.");
    end

endmodule
```

```
module instruction_decoder(
    input [31:0] imem_insn,
    output reg [4:0] destination_reg,
    output reg [2:0] funct3,
    output reg [4:0] source_reg1,
    output reg [4:0] source_reg2,
    output reg [11:0] imm,
            output reg [6:0] funct7,
    output reg [4:0] shamt,
    output reg [2:0] insn_type,
    output reg alu_op_mux,
    output reg wen,
    output reg dmem_wen
);

  always @ (*) begin
      case (imem_insn[6:0])
        7'b0010011: begin //I type instruction
          insn_type = 3'b000;
          destination_reg = imem_insn[11:7];
          funct3 = imem_insn[14:12];
          source_reg1 = imem_insn[19:15];
          source_reg2 = 0;
          imm = imem_insn[31:20];
          funct7 = imem_insn[31:25];
          shamt = imem_insn[24:20];
          alu_op_mux = 1;
          wen = 1;
          dmem_wen = 0;
        end
        7'b0110011: begin //R type instruction
          insn_type = 3'b001;
          destination_reg = imem_insn[11:7];
          funct3 = imem_insn[14:12];
          source_reg1 = imem_insn[19:15];
          source_reg2 = imem_insn[24:20];
          funct7 = imem_insn[31:25];
          imm = 0;
          shamt = 0;
          alu_op_mux = 0;
          wen = 1;
          dmem_wen = 0;
        end
        7'b0100011: begin // Store type Load-Store
          insn_type = 3'b010;
          destination_reg = 0;
          imm[4:0] = imem_insn[11:7];
          funct3 = imem_insn[14:12];
          source_reg1 = imem_insn[19:15];
        source_reg2 = imem_insn[24:20];
        imm[11:5] = imem_insn[31:25];
        funct7 = 0;
        shamt = 0;
        alu_op_mux = 1;
        wen = 0;
        dmem_wen = 1;
      end
      7'b0000011: begin // Load type Load-Store
        insn_type = 3'b011;
        destination_reg = imem_insn[11:7];
        funct3 = imem_insn[14:12];
        source_reg1 = imem_insn[19:15];
        source_reg2 = 5'b0;
        imm[11:0] = imem_insn[31:20];
        funct7 = 0;
        shamt = 0;
        alu_op_mux = 1;
        wen = 1;
        dmem_wen = 0;
      end
      default: begin
        insn_type = 3'b111;
        destination_reg = 0;
        funct3 = 0;
        source_reg1 = 0;
        source_reg2 = 0;
        imm = 0;
        funct7 = 0;
        shamt = 0;
        alu_op_mux = 0;
        wen = 0;
        dmem_wen = 0;
      end
    endcase
  end
endmodule
```

```
module register_file(
    input clk,
    input rst_n,
    input wen,
    input [4:0] destination_reg,
    input [4:0] source_reg1,
    input [4:0] source_reg2,
    input [31:0] write_data,
    input [3:0] byte_en,
    output reg [31:0] read_data1,
    output reg [31:0] read_data2
);
    reg [31:0] registers [0:31];

    initial begin
        integer i;
        for (i = 0; i < 32; i = i + 1)
            registers[i] = 32'b0;
    end

    always @(*) begin
        read_data1 = registers[source_reg1];
        read_data2 = registers[source_reg2];
    end

    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            integer i;
            for (i = 0; i < 32; i = i + 1)
                registers[i] <= 32'b0;
        end else if (wen && (destination_reg != 5'b0)) begin
            //registers[destination_reg] <= write_data;
            if (byte_en[3])
                    registers[destination_reg][31:24] <= #0.1 write_data[31:24];
            if (byte_en[2])
                    registers[destination_reg][23:16] <= #0.1 write_data[23:16];
            if (byte_en[1])
                    registers[destination_reg][15:8] <= #0.1 write_data[15:8];
            if (byte_en[0])
                    registers[destination_reg][7:0] <= #0.1 write_data[7:0];
        end
    end
endmodule
```

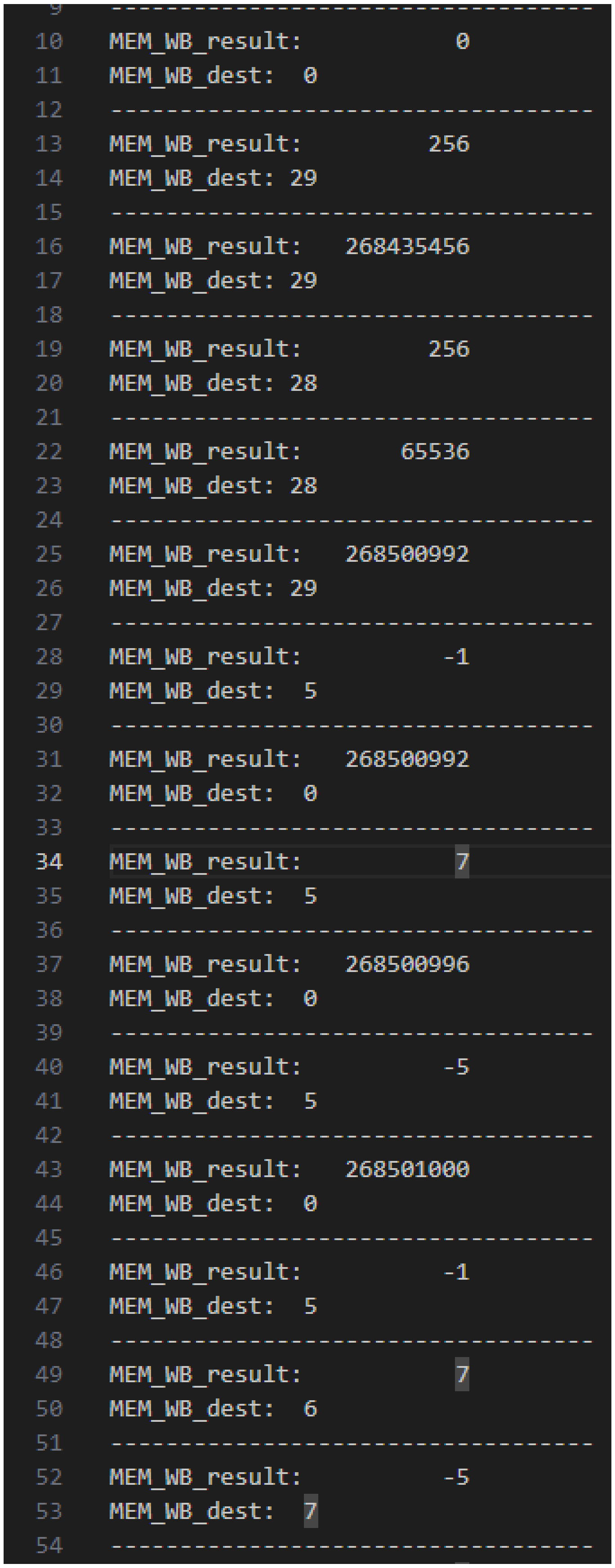
Code Explanation:
This Verilog code implements key components of a pipelined CPU, including instruction decoding, register file access, and debugging output. The instruction_decoder module parses 32 bit instructions into control signals and fields like register indices, immediates, and function codes for I-type, R-type, load, and store instructions. The register_file module supports two simultaneous reads and conditional writes with byte level granularity, resetting all registers on reset. Debugging information, such as the program counter and write back data, is logged to external files each clock cycle for tracing purposes. Pipeline control signals like instruction type and memory write enable are forwarded between stages to support correct execution flow.

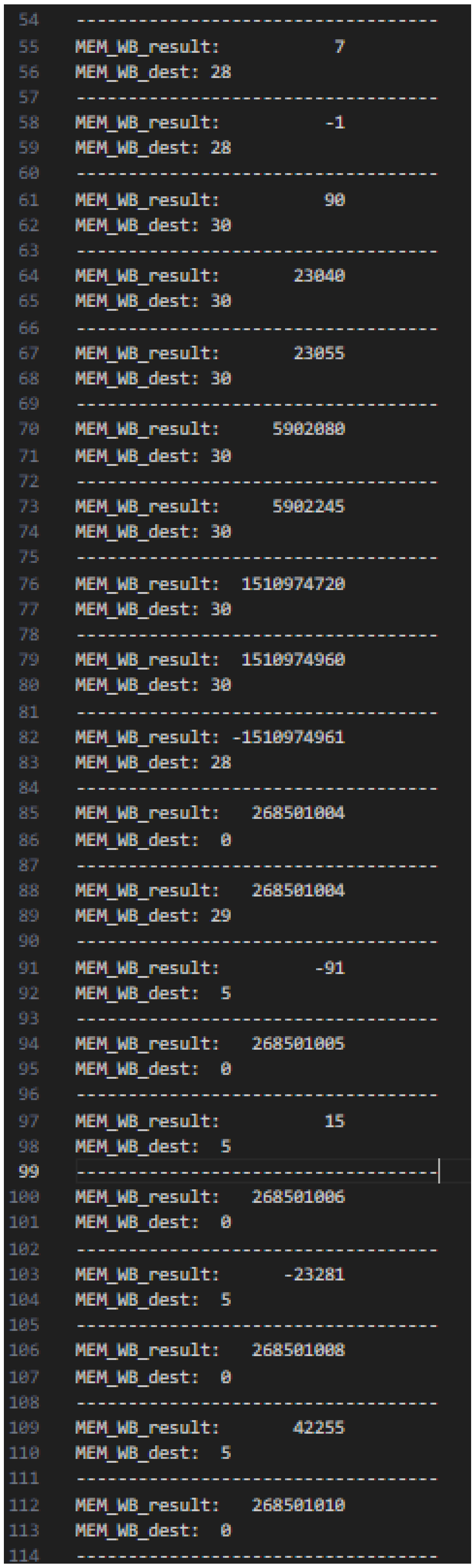
Trace Files:



```
PC: 0x00000000
PC: 0x00000000
PC: 0x00000004
PC: 0x00000008
PC: 0x0000000c
PC: 0x00000010
PC: 0x00000014
PC: 0x00000018
PC: 0x0000001c
PC: 0x00000020
PC: 0x00000024
PC: 0x00000028
PC: 0x0000002c
PC: 0x00000030
PC: 0x00000034
PC: 0x00000038
PC: 0x0000003c
PC: 0x00000040
PC: 0x00000044
PC: 0x00000048
PC: 0x0000004c
PC: 0x00000050
PC: 0x00000054
PC: 0x00000058
PC: 0x0000005c
PC: 0x00000060
PC: 0x00000064
PC: 0x00000068
PC: 0x0000006c
PC: 0x00000070
PC: 0x00000074
PC: 0x00000078
PC: 0x0000007c
PC: 0x00000080
PC: 0x00000084
PC: 0x00000088
PC: 0x0000008c
PC: 0x00000090
PC: 0x00000094
PC: 0x00000098
PC: 0x0000009c
PC: 0x000000a0
PC: 0x000000a4
PC: 0x000000a8
PC: 0x000000ac
PC: 0x000000b0
PC: 0x000000b4
PC: 0x000000b8
```

Program Counter Trace File

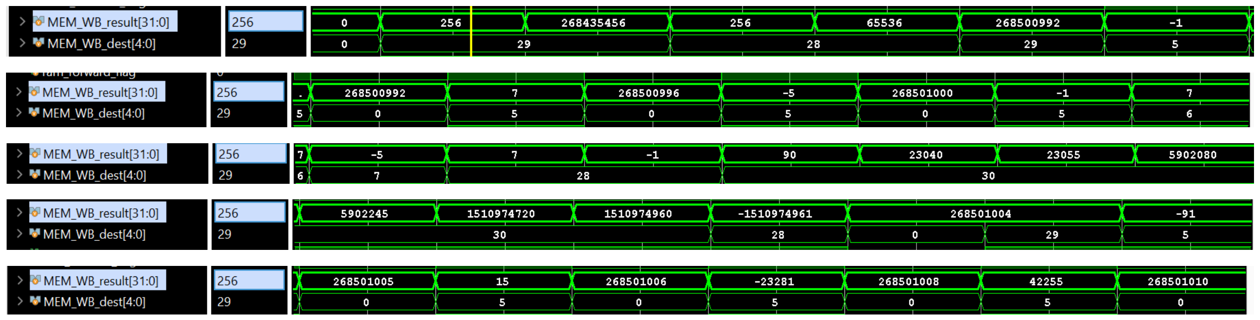

```
-----------------------------------
MEM_WB_result:           0
MEM_WB_dest:  0
-----------------------------------
MEM_WB_result:         256
MEM_WB_dest: 29
-----------------------------------
MEM_WB_result:   268435456
MEM_WB_dest: 29
-----------------------------------
MEM_WB_result:         256
MEM_WB_dest: 28
-----------------------------------
MEM_WB_result:       65536
MEM_WB_dest: 28
-----------------------------------
MEM_WB_result:   268500992
MEM_WB_dest: 29
-----------------------------------
MEM_WB_result:          -1
MEM_WB_dest:  5
-----------------------------------
MEM_WB_result:   268500992
MEM_WB_dest:  0
-----------------------------------
MEM_WB_result:           7
MEM_WB_dest:  5
-----------------------------------
MEM_WB_result:   268500996
MEM_WB_dest:  0
-----------------------------------
MEM_WB_result:          -5
MEM_WB_dest:  5
-----------------------------------
MEM_WB_result:   268501000
MEM_WB_dest:  0
-----------------------------------
MEM_WB_result:          -1
MEM_WB_dest:  5
-----------------------------------
MEM_WB_result:           7
MEM_WB_dest:  6
-----------------------------------
MEM_WB_result:          -5
MEM_WB_dest:  7
-----------------------------------
```



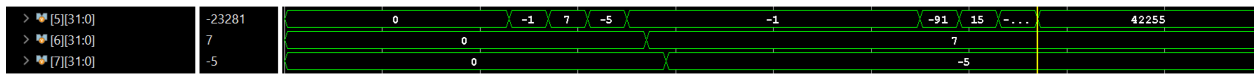
```
-----------------------------------
MEM_WB_result:           7
MEM_WB_dest: 28
-----------------------------------
MEM_WB_result:          -1
MEM_WB_dest: 28
-----------------------------------
MEM_WB_result:          90
MEM_WB_dest: 30
-----------------------------------
MEM_WB_result:       23040
MEM_WB_dest: 30
-----------------------------------
MEM_WB_result:       23055
MEM_WB_dest: 30
-----------------------------------
MEM_WB_result:     5902080
MEM_WB_dest: 30
-----------------------------------
MEM_WB_result:     5902245
MEM_WB_dest: 30
-----------------------------------
MEM_WB_result:  1510974720
MEM_WB_dest: 30
-----------------------------------
MEM_WB_result:  1510974960
MEM_WB_dest: 30
-----------------------------------
MEM_WB_result: -1510974961
MEM_WB_dest: 28
-----------------------------------
MEM_WB_result:   268501004
MEM_WB_dest:  0
-----------------------------------
MEM_WB_result:   268501004
MEM_WB_dest: 29
-----------------------------------
MEM_WB_result:         -91
MEM_WB_dest:  5
-----------------------------------
MEM_WB_result:   268501005
MEM_WB_dest:  0
-----------------------------------
MEM_WB_result:          15
MEM_WB_dest:  5
-----------------------------------
MEM_WB_result:   268501006
MEM_WB_dest:  0
-----------------------------------
MEM_WB_result:      -23281
MEM_WB_dest:  5
-----------------------------------
MEM_WB_result:   268501008
MEM_WB_dest:  0
-----------------------------------
MEM_WB_result:       42255
MEM_WB_dest:  5
-----------------------------------
MEM_WB_result:   268501010
MEM_WB_dest:  0
-----------------------------------
```

Write back result and destination trace file

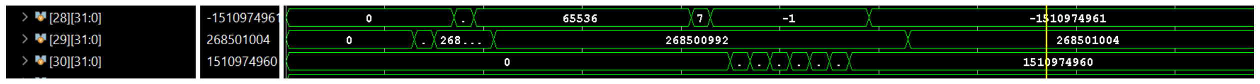
Waveforms:



MEM_WB results and destination register



Results written to registers 5, 6, 7



Results written to registers 28, 29, 30

Lab Learnings:

This lab builds on top of the design we implemented in Lab 4 and 5. This lab in particular would implement load store instructions by CPU design to support byte and half-word store operations. This was implemented by using a byte_en signal, enabling writes to data memory (ram.sv) based on instruction size. This would match RISC V semantics and prepare the CPU design for more complex memory access. One of the things that we have learned from this lab was learning how to rewrite the write back signal and the memory to optimize the error handling. This was important because this would have allowed us to handle error more easier while also allowing the system to be more consistent with the signals.