Mihir Phadke (015745301)
Jai Nayyar (015176655)
Ryan Smith (015792504)
Thurman Dao (015708589)
Haydon Behl
Anson Lee
Lab 3
Mar 6, 2025

1.) CPU module Design Code

```
`timescale 1ns / 1ps

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
);
    // stall register
    reg stall;
    // IF/ID pipeline register
    reg [31:0] IF_ID_insn, IF_ID_pc;
    // ID/EX pipeline register
    reg [31:0] ID_EX_pc, ID_EX_imm;
    reg [4:0] ID_EX_dest, ID_EX_src1;
    reg [2:0] ID_EX_alu_ctrl;
    // EX/MEM pipeline register
    reg [31:0] EX_MEM_alu_result;
    reg [4:0] EX_MEM_dest;
    // MEM/WB pipeline registers
    reg signed [31:0] MEM_WB_result;
    reg [4:0] MEM_WB_dest;
    reg MEM_WB_wen;

    // 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),
        .next_pc(next_pc),
        .pc(pc)
    );

    assign next_pc = pc + 4;
    assign imem_addr = pc;

    // 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 if (stall != 1'b1) begin
            IF_ID_insn <= imem_insn;
            IF_ID_pc <= pc;
        end
    end

    // Decode Stage
    wire [6:0] opcode;
    wire [4:0] destination_reg, source_reg1;
    wire [2:0] funct3;
    wire [11:0] imm;

    instruction_decoder decoder(
        .imem_insn(IF_ID_insn),
        .opcode(opcode),
        .destination_reg(destination_reg),
        .funct3(funct3),
        .source_reg1(source_reg1),
```

```
        .imm(imm)
    );

    // Hazard detection
    always @(*) begin
        // Check if source register of current instruction matches
destination register in pipeline
        if ((ID_EX_dest != 5'b0) && (ID_EX_dest == source_reg1)) begin
            stall = 1'b1;
        end else if ((EX_MEM_dest != 5'b0) && (EX_MEM_dest == source_reg1))
begin
            stall = 1'b1;
        end else if ((MEM_WB_dest != 5'b0) && (MEM_WB_dest == source_reg1))
begin
            stall = 1'b1;
        end else begin
            stall = 1'b0;
        end
    end

    // ALU Control
    wire [2:0] alu_ctrl_temp;

    ALUDecoder alu_decoder(
        .opcode(opcode),
        .funct3(funct3),
        .alu_ctrl(alu_ctrl_temp)
    );

    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_imm <= 32'b0;
            ID_EX_alu_ctrl <= 3'b111;
        end else begin
            ID_EX_pc <= IF_ID_pc;
            ID_EX_dest <= destination_reg;
            ID_EX_src1 <= source_reg1;
            ID_EX_imm <= {{20{imm[11]}}, imm};
            ID_EX_alu_ctrl <= alu_ctrl_temp;
        end
```

```
    end

    // Register File
    wire [31:0] reg_data1;

    register_file reg_file(
        .clk(clk),
        .rst_n(rst_n),
        .wen(MEM_WB_wen),
        .destination_reg(MEM_WB_dest),
        .source_reg1(ID_EX_src1),
        .write_data(MEM_WB_result),
        .read_data1(reg_data1)
    );

    // Execute Stage
    wire [31:0] alu_result;

    ALU alu(
        .op1(reg_data1),
        .op2(ID_EX_imm),
        .alu_ctrl(ID_EX_alu_ctrl),
        .result(alu_result)
    );

    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            EX_MEM_alu_result <= 32'b0;
            EX_MEM_dest <= 5'b0;
        end else begin
            EX_MEM_alu_result <= alu_result;
            EX_MEM_dest <= ID_EX_dest;
        end
    end

    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            MEM_WB_result <= 32'b0;
            MEM_WB_dest <= 5'b0;
            MEM_WB_wen <= 1'b0;
        end else begin
            MEM_WB_result <= EX_MEM_alu_result;
            MEM_WB_dest <= EX_MEM_dest;
```

```
            MEM_WB_wen <= (EX_MEM_dest != 5'b0);
        end
    end


    // open files
    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

    // print to trace files
    always @(posedge clk or negedge rst_n) begin
        if(!rst_n) begin

        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

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

// Program Counter Module
module PC(
```

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


// Instruction Decoder Module
module instruction_decoder(
    input [31:0] imem_insn,
    output reg [6:0] opcode,
    output reg [4:0] destination_reg,
    output reg [2:0] funct3,
    output reg [4:0] source_reg1,
    output reg [11:0] imm
);
    always @(*) begin
        opcode = imem_insn[6:0];
        destination_reg = imem_insn[11:7];
        funct3 = imem_insn[14:12];
        source_reg1 = imem_insn[19:15];
        imm = imem_insn[31:20];
    end
endmodule


// ALU Decoder Module
module ALUDecoder(
    input [6:0] opcode,
    input [2:0] funct3,
    output reg [2:0] alu_ctrl
);
    always @(*) begin
        case (opcode)
            7'b0010011: begin
```

```
                case (funct3)
                    3'b000: alu_ctrl = 3'b000; // ADDI
                    default: alu_ctrl = 3'b111;
                endcase
            end
            default: alu_ctrl = 3'b111;
        endcase
    end
endmodule


// ALU Module
module ALU(
    input [31:0] op1,
    input [31:0] op2,
    input [2:0] alu_ctrl,
    output reg [31:0] result
);
    always @(*) begin
        case (alu_ctrl)
            3'b000: result = op1 + op2; // ADDI
            default: result = 32'b0;
        endcase
    end
endmodule


// Register File Module
module register_file(
    input clk,
    input rst_n,
    input wen,
    input [4:0] destination_reg,
    input [4:0] source_reg1,
    input [31:0] write_data,
    output reg [31:0] read_data1
);
    reg [31:0] registers [0:31];

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

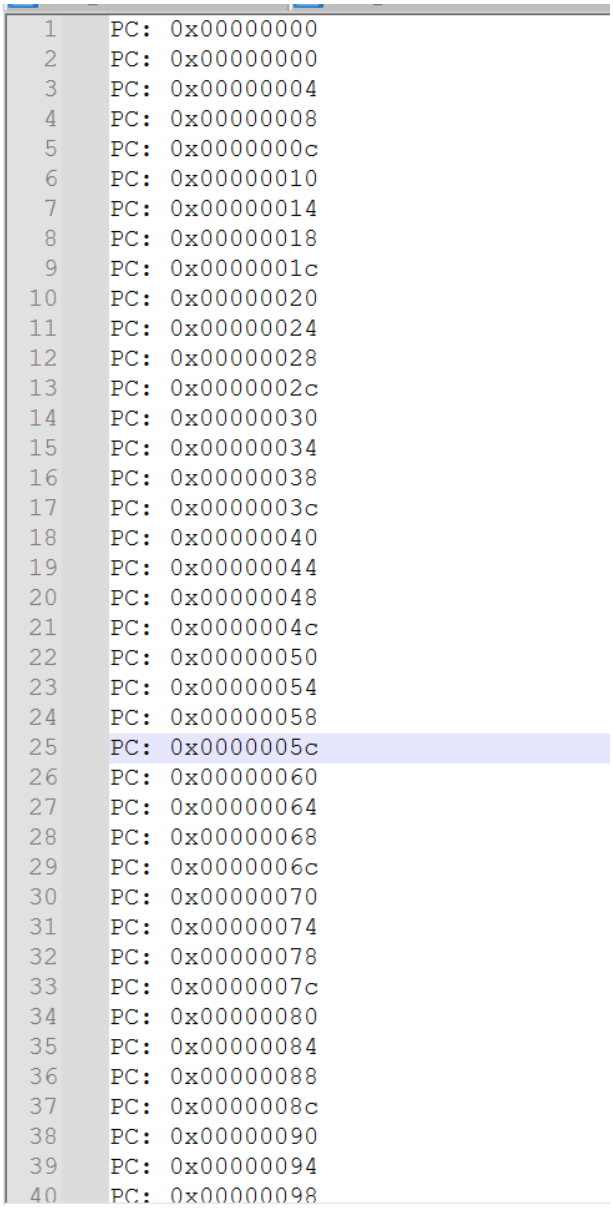
```
    end

    always @(*) begin
        read_data1 = registers[source_reg1];
    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;
        end
    end
endmodule
```

2.) The Two Trace Files (Screenshots within the files or attached when turned in)

Non-hazard trace files:

```
12  ----------------------------------
13  MEM_WB_result:           5
14  MEM_WB_dest:  5
15  ----------------------------------
16  MEM_WB_result:         -10
17  MEM_WB_dest:  6
18  ----------------------------------
19  MEM_WB_result:           3
20  MEM_WB_dest:  7
21  ----------------------------------
22  MEM_WB_result:          -1
23  MEM_WB_dest: 28
24  ----------------------------------
25  MEM_WB_result:           5
26  MEM_WB_dest: 29
27  ----------------------------------
28  MEM_WB_result:          11
29  MEM_WB_dest: 30
30  ----------------------------------
31  MEM_WB_result:          -6
32  MEM_WB_dest: 31
33  ----------------------------------
```

The image above shows the trace files for the result written back to the register and the destination register for the non-hazards simulation.



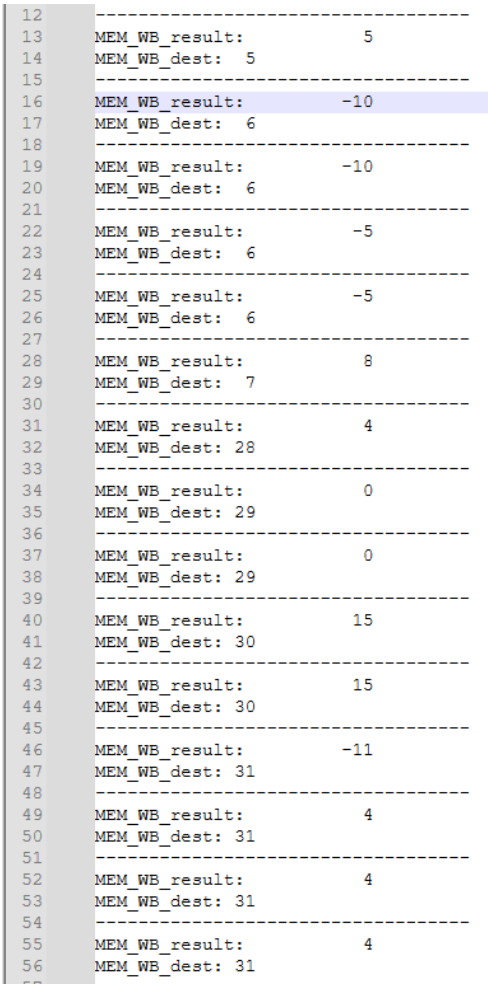
```
PC: 0x00000000
PC: 0x00000000
PC: 0x00000004
PC: 0x00000008
PC: 0x0000000c
PC: 0x00000010
PC: 0x00000014
PC: 0x00000018
PC: 0x0000001c
PC: 0x00000020
PC: 0x00000024
PC: 0x00000028
PC: 0x0000002c
PC: 0x00000030
PC: 0x00000034
PC: 0x00000038
PC: 0x0000003c
PC: 0x00000040
PC: 0x00000044
PC: 0x00000048
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
```

The image above shows the program counter for the non-hazards simulation.

Hazard trace files:

```
PC: 0x00000000
PC: 0x00000000
PC: 0x00000004
PC: 0x00000004
PC: 0x00000004
PC: 0x00000004
PC: 0x00000008
PC: 0x0000000c
PC: 0x00000010
PC: 0x00000010
PC: 0x00000014
PC: 0x00000014
PC: 0x00000018
PC: 0x00000018
PC: 0x00000018
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
```

The image above shows the program counter for the hazards simulation.



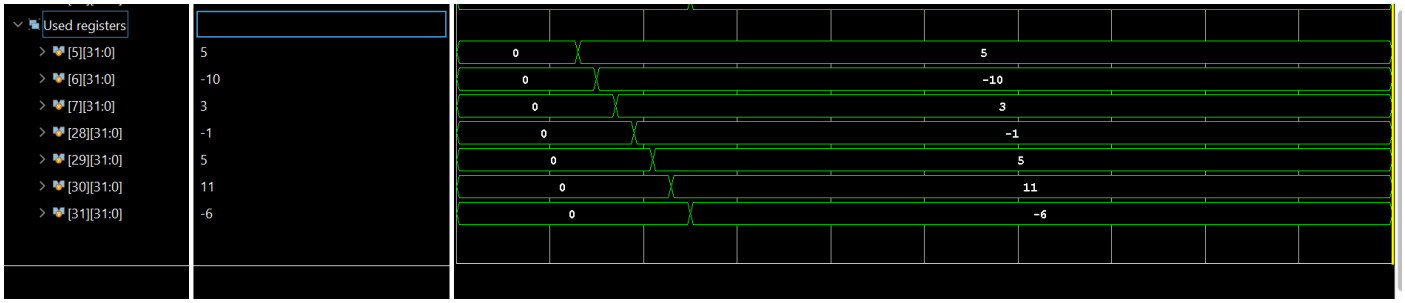
The image above shows the trace file for the result written back to the register and which register it was written to for the hazards simulation.

3.) Post Simulation Convergence with results matching the same results for both hazard and non hazard

Non-hazard waveforms:

Registers | Floating Point | Control and Status

Name	Number	Value
zero	0	0
ra	1	0
sp	2	2147479548
gp	3	268468224
tp	4	0
t0	5	5
t1	6	-10
t2	7	3
s0	8	0
s1	9	0
a0	10	0
a1	11	0
a2	12	0
a3	13	0
a4	14	0
a5	15	0
a6	16	0
a7	17	0
s2	18	0
s3	19	0
s4	20	0
s5	21	0
s6	22	0
s7	23	0
s8	24	0
s9	25	0
s10	26	0
s11	27	0
t3	28	-1
t4	29	5
t5	30	11
t6	31	-6
pc		4194336

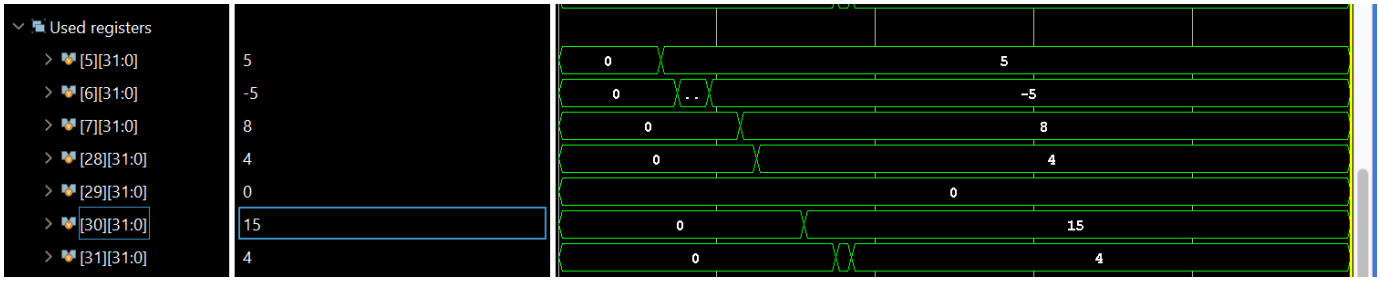


The waveforms above show the values from the no hazards test. As we can see the values match the values in the rars simulation for no hazards.

Hazard waveforms:

Registers | Floating Point | Control and Status

Name	Number	Value
zero	0	0
ra	1	0
sp	2	2147479548
gp	3	268468224
tp	4	0
t0	5	5
t1	6	-5
t2	7	8
s0	8	0
s1	9	0
a0	10	0
a1	11	0
a2	12	0
a3	13	0
a4	14	0
a5	15	0
a6	16	0
a7	17	0
s2	18	0
s3	19	0
s4	20	0
s5	21	0
s6	22	0
s7	23	0
s8	24	0
s9	25	0
s10	26	0
s11	27	0
t3	28	4
t4	29	0
t5	30	15
t6	31	4
pc		4194336



The image above shows the waveform from the hazards test. As we can see the values in the registers match the rars simulation shown above for hazards.

Conclusion/ Lab Learning:

In this lab, we learned how to implement a pipelined RISC-V processor. One of the things that we had planned out was dividing the pipeline stages and understanding how the RISC-V assembly code operates which includes the PC and CPU structure via the gateways to the various modules handling the

data down the pipeline. Understanding how the processor deciphers assembly code into binary code and does calculations was instrumental to understanding and scripting this processor. In this particular lab, we implemented the basic instruction execution “addi” and coded how the instructions are handled, which included how the binary is formatted in I format. Another obstacle in this lab was the hazard handling, as the instruction had to navigate the pipeline using stalls to handle the hazard, which we had to verify using addi_nohazard.dat and addi_hazard.dat respectively.