Lab3 requirements:

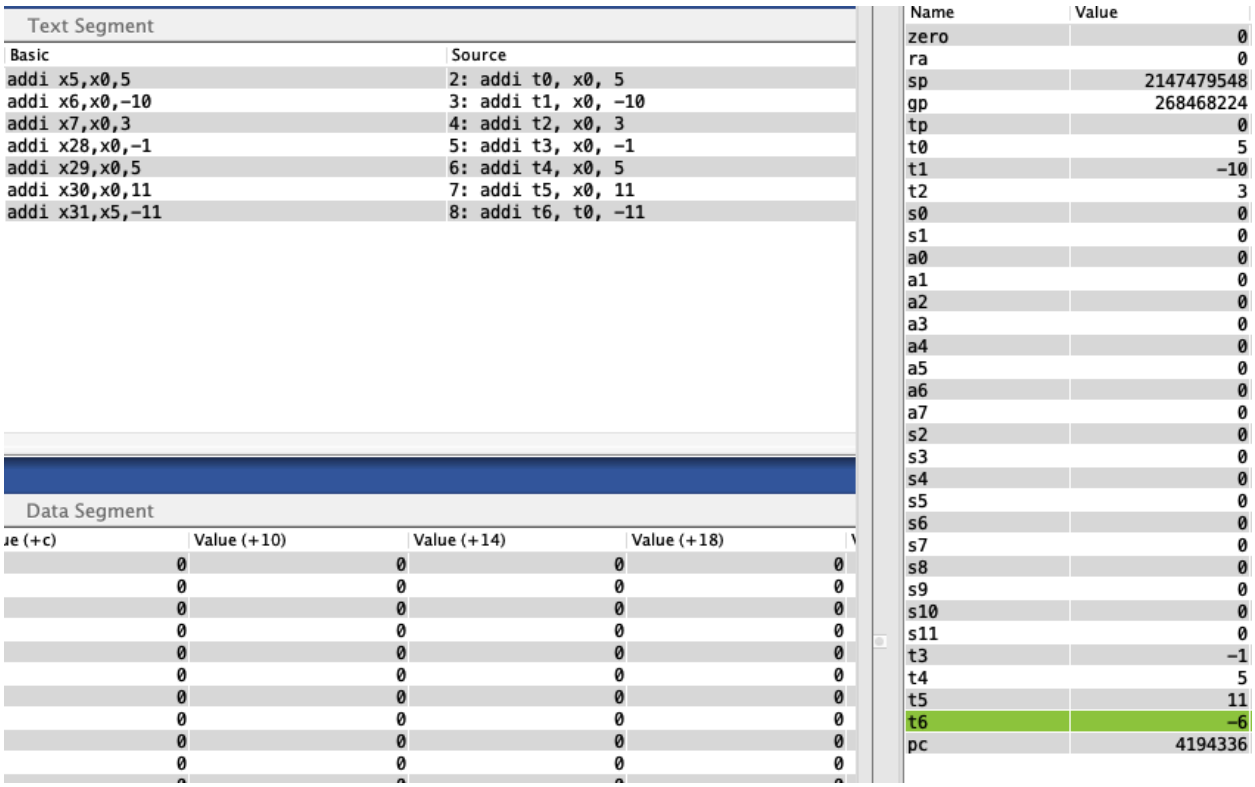
Fetch	1 CC
Decode	1 CC
Execute	1 CC
Memory access	1 CC
Write back	1 CC

Module name - cpu
Ports

Name	Direction	Width	Details
rst_n	Input	0:0	Active low reset
clk	Input	0:0	Clock
imem_addr	Output	31:0	Instruction memory address
imem_insn	Input	31:0	Instruction from Instruction memory
dmem_addr	Output	31:0	Data memory address
dmem_data	InOut	31:0	Data to/ from data memory
dmem_wen	Output	0:0	Write enable for data memory (1 for Store, 0 for Load)

Addi_nohazard.asm:

```
.text
addi t0, x0, 5   #0+5=5
addi t1, x0, -10 #0+(-10)=-10
addi t2, x0, 3   #0+3=3
addi t3, x0, -1  #0+(-1)=-1
addi t4, x0, 5   #0+5=5
addi t5, x0, 11  #0+11=11
addi t6, t0, -11 #5+(-11)=-6
```



Text Segment

Basic	Source
addi x5,x0,5	2: addi t0, x0, 5
addi x6,x0,-10	3: addi t1, x0, -10
addi x7,x0,3	4: addi t2, x0, 3
addi x28,x0,-1	5: addi t3, x0, -1
addi x29,x0,5	6: addi t4, x0, 5
addi x30,x0,11	7: addi t5, x0, 11
addi x31,x5,-11	8: addi t6, t0, -11

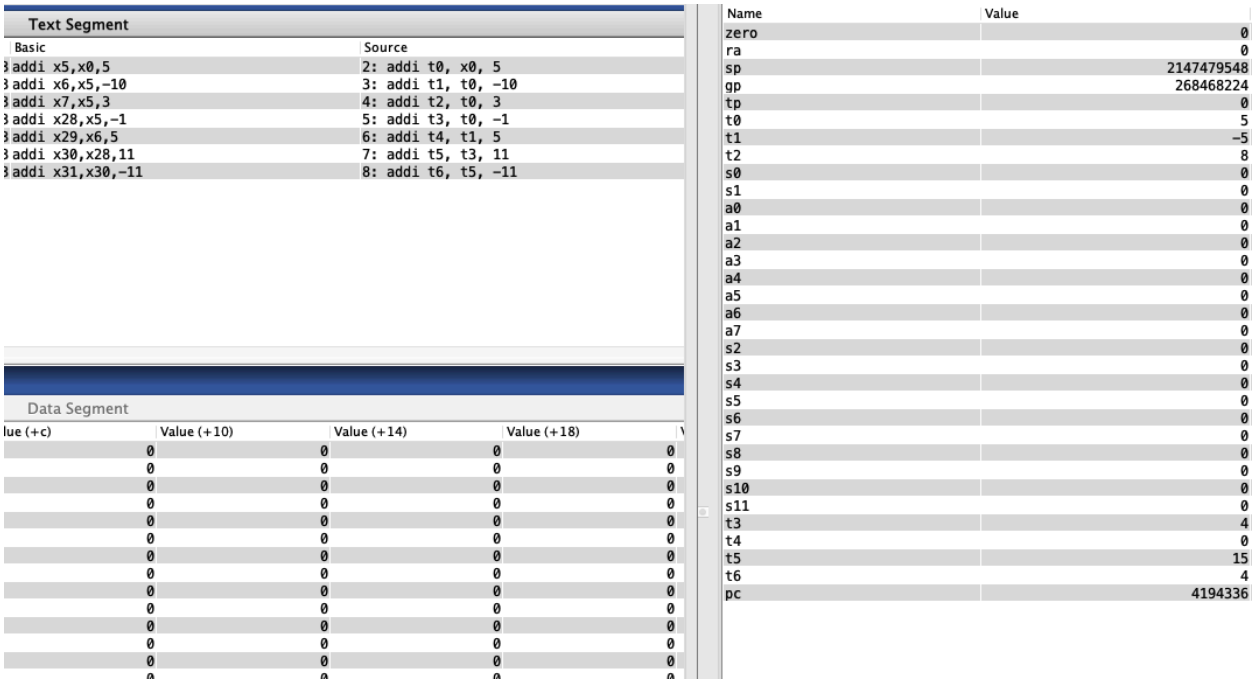
Data Segment

ue (+c)	Value (+10)	Value (+14)	Value (+18)
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Name	Value
zero	0
ra	0
sp	2147479548
gp	268468224
tp	0
t0	5
t1	-10
t2	3
s0	0
s1	0
a0	0
a1	0
a2	0
a3	0
a4	0
a5	0
a6	0
a7	0
s2	0
s3	0
s4	0
s5	0
s6	0
s7	0
s8	0
s9	0
s10	0
s11	0
t3	-1
t4	5
t5	11
t6	-6
pc	4194336

Addi_hazards.asm

```
.text
addi t0, x0, 5      #0+5=5
addi t1, t0, -10    #5+(-10)=-5
addi t2, t0, 3      #5+3=8
addi t3, t0, -1     #5+(-1)=-4
addi t4, t1, 5      #-5+5=0
addi t5, t3, 11     #4+11=15
addi t6, t5, -11   #15+(-11)=4
```



Text Segment

Basic	Source
addi x5,x0,5	2: addi t0, x0, 5
addi x6,x5,-10	3: addi t1, t0, -10
addi x7,x5,3	4: addi t2, t0, 3
addi x28,x5,-1	5: addi t3, t0, -1
addi x29,x6,5	6: addi t4, t1, 5
addi x30,x28,11	7: addi t5, t3, 11
addi x31,x30,-11	8: addi t6, t5, -11

Data Segment

lue (+c)	Value (+10)	Value (+14)	Value (+18)
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Name	Value
zero	0
ra	0
sp	2147479548
gp	268468224
tp	0
t0	5
t1	-5
t2	8
s0	0
s1	0
a0	0
a1	0
a2	0
a3	0
a4	0
a5	0
a6	0
a7	0
s2	0
s3	0
s4	0
s5	0
s6	0
s7	0
s8	0
s9	0
s10	0
s11	0
t3	4
t4	0
t5	15
t6	4
pc	4194336

Output files:

Design and Synthesize

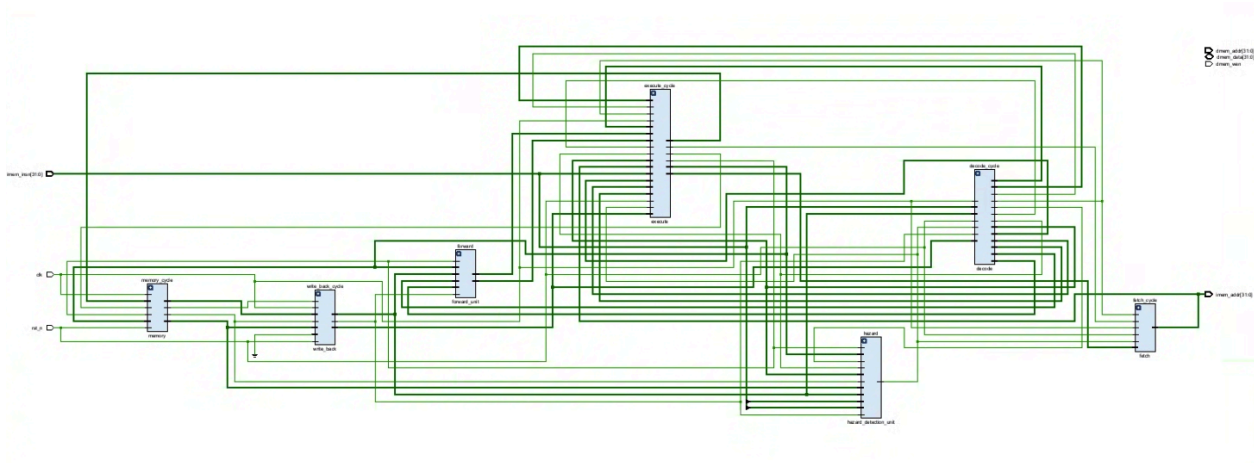


Figure 1: Design

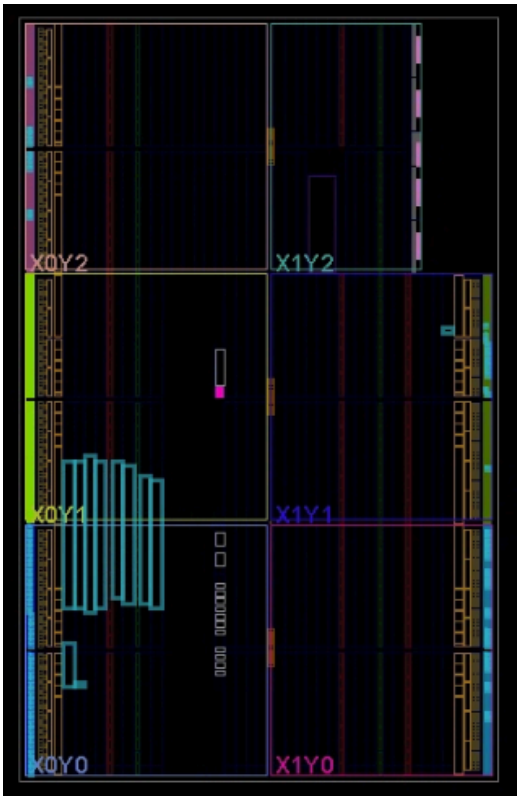


Figure 2: Synthesize design

Code implemented:

```
`timescale 1ns / 1ps

module cpu(
    input rst_n, clk,
    input [31:0] imem_insn,
    inout [31:0] dmem_data,
    output dmem_wen,
    output [31:0] imem_addr, dmem_addr
);
    // Control Signals
    wire ALUSrc, MemtoReg, MemRead, MemWrite, Branch, id_ex_RegWrite, Zero;
    wire ex_mem_RegWrite, mem_wb_RegWrite, stall, ex_mem_MemtoReg,
mem_wb_MemtoReg, wb_RegWrite;
```

```
    // Pipeline Registers & Signals
    wire [1:0] ALUOp, forwardA_sig, forwardB_sig;
    wire [3:0] ALUCtl;
    wire [4:0] rs1, rs2, id_ex_rd, ex_mem_rd, mem_wb_rd, reg_addr;
    wire [31:0] rd1, rd2, imm_val, ALU_result, target, dmem_data_cpu,
mem_wb_ALU_result, write_data;

    reg [15:0] clk_counter;

    // Fetch Stage
    fetch fetch_cycle (
        .rst_n(rst_n), .clk(clk), .Branch(Branch), .Zero(Zero),
.stall(stall),
        .target(target), .imem_addr(imem_addr)
    );

    // Decode Stage
    decode decode_cycle (
        .rst_n(rst_n), .clk(clk), .stall(stall), .wb_RegWrite(wb_RegWrite),
        .reg_addr(reg_addr), .imem_insn(imem_insn), .wd(write_data),
        .ALUSrc(ALUSrc), .MemtoReg(MemtoReg), .MemRead(MemRead),
        .MemWrite(MemWrite), .Branch(Branch),
.id_ex_RegWrite(id_ex_RegWrite),
        .ALUOp(ALUOp), .ALUCtl(ALUCtl),
        .rs1(rs1), .rs2(rs2), .id_ex_rd(id_ex_rd),
        .rd1(rd1), .rd2(rd2), .imm_val(imm_val)
    );

    // Execute Stage
    execute execute_cycle (
        .rst_n(rst_n), .clk(clk), .Branch(Branch), .ALUSrc(ALUSrc),
        .id_ex_RegWrite(id_ex_RegWrite), .id_ex_MemtoReg(MemtoReg),
.stall(stall),
        .ALUOp(ALUOp), .forwardA_ex(forwardA_sig),
.forwardB_ex(forwardB_sig),
```

```
        .ctl(ALUCtl), .id_ex_rd(id_ex_rd), .rd1(rd1), .rd2(rd2),
.imm_val(imm_val),
        .imem_addr(imem_addr), .imem_insn(imem_insn),
.wb_result(write_data),
        .Zero(Zero), .ex_mem_RegWrite(ex_mem_RegWrite),
.ex_mem_MemtoReg(ex_mem_MemtoReg),
        .ex_mem_rd(ex_mem_rd), .ALU_result(ALU_result), .target(target)
    );

    // Memory Stage
    memory memory_cycle (
        .rst_n(rst_n), .clk(clk), .ex_mem_RegWrite(ex_mem_RegWrite),
        .ex_mem_MemtoReg(ex_mem_MemtoReg), .ex_mem_rd(ex_mem_rd),
        .ex_mem_ALU_result(ALU_result),
        .mem_wb_RegWrite(mem_wb_RegWrite),
.mem_wb_MemtoReg(mem_wb_MemtoReg),
        .mem_wb_rd(mem_wb_rd), .mem_wb_ALU_result(mem_wb_ALU_result)
    );

    // Write Back Stage
    write_back write_back_cycle (
        .rst_n(rst_n), .clk(clk), .MemtoReg(mem_wb_MemtoReg),
        .mem_wb_RegWrite(mem_wb_RegWrite), .mem_wb_rd(mem_wb_rd),
        .read_data(32'd0), .mem_wb_ALU_result(mem_wb_ALU_result),
        .wb_RegWrite(wb_RegWrite), .reg_addr(reg_addr),
        .write_data(write_data)
    );

    // Hazard Detection Unit
    hazard_detection_unit hazard (
        .id_ex_MemRead(MemRead), .ex_mem_RegWrite(ex_mem_RegWrite),
        .id_ex_RegWrite(id_ex_RegWrite), .mem_wb_RegWrite(mem_wb_RegWrite),
        .wb_RegWrite(wb_RegWrite),
        .rs1(imem_insn[19:15]), .rs2(imem_insn[24:20]),
.id_ex_rd(id_ex_rd),
        .ex_mem_rd(ex_mem_rd), .mem_wb_rd(mem_wb_rd), .reg_addr(reg_addr),
```

```
        .stall(stall)
    );fha

    // Forwarding Unit
    forward_unit forward (
        .ex_mem_RegWrite(ex_mem_RegWrite), .wb_RegWrite(wb_RegWrite),
        .rs1(rs1), .rs2(rs2), .ex_mem_rd(ex_mem_rd), .reg_addr(reg_addr),
        .forwardA(forwardA_sig), .forwardB(forwardB_sig)
    );

    // File Handling
    integer file1, file2;
    initial begin
        file1 =
$fopen("C:\Users\Sora\LAB3CMPE140\LAB3CMPE140.srcs\sim_1\imports\Downloads
\ProgramCounter_Hazard.txt", "w");
        file2 =
$fopen("C:\Users\Sora\LAB3CMPE140\LAB3CMPE140.srcs\sim_1\imports\Downloads
\ProgramCounterUpdates_Hazard.txt", "w");

        //file1 =
$fopen("C:\Users\Sora\LAB3CMPE140\LAB3CMPE140.srcs\sim_1\imports\Downloads
\ProgramCounter_NoHazard.txt", "w");

//file2=$fopen("C:\Users\Sora\LAB3CMPE140\LAB3CMPE140.srcs\sim_1\imports\D
ownloads\ProgramCounterUpdates_NoHazard.txt", "w");
    end

    // Clock Counter for Debugging
    always @(posedge clk or negedge rst_n) begin
        if (~rst_n)
            clk_counter <= 16'd1;
        else begin
            clk_counter <= clk_counter + 1;
            $fwrite(file1, "program counter Hazard: %h\n", imem_addr);
```

```
            $fwrite(file2, "clk cycle %d | reg: %0d | write_data: %0h\n",
clk_counter, reg_addr, write_data);
            $display(
                "clk cycle %d | write_data: %0h | reg %0d | forwardA: %0h,
forwardB: %0h, stall: %0b | rs1: %0h | rs2: %0h | wb_RegWrite: %0h",
                clk_counter, write_data, reg_addr, forwardA_sig,
forwardB_sig, stall,
                imem_insn[19:15], imem_insn[24:20], wb_RegWrite
            );
        end
    end
endmodule


module fetch (
    input rst_n,
    input clk,
    input Branch,
    input Zero,
    input stall,
    input [31:0] target,
    output [31:0] imem_addr
);

    wire [31:0] PC, PCPlus4, PC_Next;

    // Multiplexer for selecting next PC value
    mux pc_mux (
        .a(PCPlus4),
        .b(target),
        .s(Branch && Zero),
        .c(PC_Next)
    );

    // Adder to calculate PC + 4
    adder pc_adder (
```

```
        .a(PC),
        .b(32'h00000004),
        .c(PCPlus4)
    );

    // Program Counter (PC) register
    pc pc (
        .clk(clk),
        .rst_n(rst_n),
        .stall(stall),
        .pc_next(PC_Next),
        .pc(PC)
    );

    // Output assignment
    assign imem_addr = PC;

endmodule

//======================
// Multiplexer (2:1)
//======================
module mux (
    input s,
    input [31:0] a,
    input [31:0] b,
    output [31:0] c
);
    assign c = ~s ? a : b;
endmodule

//======================
// 4-to-1 Multiplexer
//======================
module mux4_1 (
    input [1:0] s,
```

```
    input [31:0] a,
    input [31:0] b,
    input [31:0] c,
    output [31:0] d
);
    assign d = (s == 2'b01) ? b : (s == 2'b10) ? c : a;
endmodule

//======================
// 32-bit Adder
//======================
module adder (
    input [31:0] a,
    input [31:0] b,
    output [31:0] c
);
    assign c = a + b;
endmodule

//======================
// Program Counter (PC)
//======================
module pc (
    input clk,
    input rst_n,
    input stall,
    input [31:0] pc_next,
    output reg [31:0] pc
);
    always @(posedge clk or negedge rst_n) begin
        if (~rst_n)
            pc <= 32'h00000000;
        else if (~stall)
            pc <= pc_next;
    end
endmodule
```

```
//======================
// Instruction Decode
//======================
module decode (
   input rst_n,
   input clk,
   input stall,
   input wb_RegWrite,
   input [4:0] reg_addr,
   input [31:0] imem_insn,
   input [31:0] wd,
   output ALUSrc,
   output MemtoReg,
   output MemRead,
   output MemWrite,
   output Branch,
   output id_ex_RegWrite,
   output [1:0] ALUOp,
   output [3:0] ALUCtl,
   output [4:0] rs1,
   output [4:0] rs2,
   output [4:0] id_ex_rd,
   output [31:0] rd1,
   output [31:0] rd2,
   output [31:0] imm_val
);
   // Internal Wires
   wire ALUSrcD, MemtoRegD, RegWriteD, MemReadD, MemWriteD, BranchD;
   wire [1:0] ALUOpD;
   wire [3:0] ctl;
   wire [31:0] rd1_d, rd2_d, imm_valD;

   // Internal Registers
   reg ALUSrc_reg, MemtoReg_reg, MemRead_reg, MemWrite_reg, Branch_reg,
RegWrite_reg;
```

```
reg [1:0] ALUOp_reg;
reg [3:0] ctl_reg;
reg [4:0] rs1_reg, rs2_reg, id_ex_rd_reg;
reg [31:0] rd1_reg, rd2_reg, imm_val_reg;

// Register File Instance
register_file rf (
    .rst_n(rst_n),
    .clk(clk),
    .RegWrite(wb_RegWrite),
    .rr1(imem_insn[19:15]),
    .rr2(imem_insn[24:20]),
    .wr(reg_addr),
    .wd(wd),
    .rd1(rd1_d),
    .rd2(rd2_d)
);

// Control Unit Instance
control_unit cu (
    .opcode(imem_insn[6:0]),
    .ALUSrc(ALUSrcD),
    .MemtoReg(MemtoRegD),
    .RegWrite(RegWriteD),
    .MemRead(MemReadD),
    .MemWrite(MemWriteD),
    .Branch(BranchD),
    .ALUOp(ALUOpD)
);

// ALU Control Instance
alu_control alu_control (
    .ALUOp(ALUOpD),
    .Funct3(imem_insn[14:12]),
    .Funct7(imem_insn[31:25]),
    .op(ctl)
```

```
);

// Immediate Generator Instance
imm_gen imm_gen_module (
    .imem_insn(imem_insn),
    .imm_val(imm_valD)
);

// Pipeline Register: ID/EX
always @(posedge clk or negedge rst_n) begin
    if (~rst_n || stall) begin
        ALUSrc_reg    <= 1'b0;
        MemtoReg_reg  <= 1'b0;
        MemRead_reg   <= 1'b0;
        MemWrite_reg  <= 1'b0;
        Branch_reg    <= 1'b0;
        RegWrite_reg  <= 1'b0;
        ALUOp_reg     <= 2'b00;
        ctl_reg       <= 4'b0000;
        rs1_reg       <= 5'b00000;
        rs2_reg       <= 5'b00000;
        id_ex_rd_reg  <= 5'b00000;
        rd1_reg       <= 32'h00000000;
        rd2_reg       <= 32'h00000000;
        imm_val_reg   <= 32'h00000000;
    end
    else begin
        ALUSrc_reg    <= ALUSrcD;
        MemtoReg_reg  <= MemtoRegD;
        MemRead_reg   <= MemReadD;
        MemWrite_reg  <= MemWriteD;
        Branch_reg    <= BranchD;
        RegWrite_reg  <= RegWriteD;
        ALUOp_reg     <= ALUOpD;
        ctl_reg       <= ctl;
        rs1_reg       <= imem_insn[19:15];
```

```
            rs2_reg       <= imem_insn[24:20];
            id_ex_rd_reg  <= imem_insn[11:7];
            rd1_reg       <= rd1_d;
            rd2_reg       <= rd2_d;
            imm_val_reg   <= imm_valD;
        end
    end

    // Assigning Outputs
    assign ALUSrc        = ALUSrc_reg;
    assign MemtoReg      = MemtoReg_reg;
    assign MemRead       = MemRead_reg;
    assign MemWrite      = MemWrite_reg;
    assign Branch        = Branch_reg;
    assign id_ex_RegWrite = RegWrite_reg;
    assign ALUOp         = ALUOp_reg;
    assign ALUCtl        = ctl_reg;
    assign rs1           = rs1_reg;
    assign rs2           = rs2_reg;
    assign rd1           = rd1_reg;
    assign rd2           = rd2_reg;
    assign id_ex_rd      = id_ex_rd_reg;
    assign imm_val       = imm_val_reg;

endmodule

//================================
// 32-Register File Module
//================================
module register_file (
    input rst_n,
    input clk,
    input RegWrite,
    input [4:0] rr1, rr2, wr,
    input [31:0] wd,
    output [31:0] rd1, rd2
```

```
);

   reg [31:0] register [31:0]; // Register file with 32 registers
   integer i;

   always @(posedge clk or negedge rst_n) begin
       if (~rst_n) begin
           for (i = 0; i < 32; i = i + 1)
               register[i] <= 0;
       end
       else if (RegWrite) begin
           register[wr] <= wd;
       end
   end

   assign rd1 = register[rr1];
   assign rd2 = register[rr2];

endmodule

//==================================
// Control Unit
//==================================
module control_unit (
   input [6:0] opcode,
   output ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch,
   output [1:0] ALUOp
);

   // Instruction Type Opcodes
   wire [6:0] r_type = 7'b0110011;
   wire [6:0] lw = 7'b0000011;
   wire [6:0] sw = 7'b0100011;
   wire [6:0] beq = 7'b1100011;
   wire [6:0] i_type = 7'b0010011;
```

```
   // Control Signals
   assign ALUSrc   = (opcode == lw || opcode == sw || opcode == i_type);
   assign MemtoReg = (opcode == lw);
   assign RegWrite = (opcode == r_type || opcode == lw || opcode ==
i_type);
   assign MemRead  = (opcode == lw);
   assign MemWrite = (opcode == sw);
   assign Branch   = (opcode == beq);
   assign ALUOp[1] = (opcode == r_type || opcode == i_type);
   assign ALUOp[0] = (opcode == beq);

endmodule

//==================================
// ALU Control Unit
//==================================
module alu_control (
   input [1:0] ALUOp,
   input [2:0] Funct3,
   input [6:0] Funct7,
   output [3:0] op
);

   assign op[0] = ((ALUOp == 2'b10) && (Funct3 == 3'b110)) || // OR
                  ((ALUOp == 2'b10) && (Funct3 == 3'b100)) || // XOR
                  ((ALUOp == 2'b10) && (Funct3 == 3'b101) && (Funct7 ==
7'b0000000)) || // SRL
                  ((ALUOp == 2'b10) && (Funct3 == 3'b101) && (Funct7 ==
7'b0100000)) || // SRA
                  ((ALUOp == 2'b01) && (Funct3 == 3'b001)) || // BNE
                  ((ALUOp == 2'b01) && (Funct3 == 3'b101)) || // BGE
                  ((ALUOp == 2'b01) && (Funct3 == 3'b111));  // BGEU

   assign op[1] = (ALUOp == 2'b00) || // ADD
                  ((ALUOp == 2'b10) && (Funct3 == 3'b000)) || // ADD
                  ((ALUOp == 2'b10) && (Funct3 == 3'b100)) || // XOR
```

```
                  ((ALUOp == 2'b10) && (Funct7 == 7'b0100000) && (Funct3
== 3'b000)) || // SUB
                  ((ALUOp == 2'b10) && (Funct3 == 3'b011));  // SLTU

   assign op[2] = ((ALUOp == 2'b10) && (Funct3 == 3'b101)) || // SRL/SRA
                  ((ALUOp == 2'b10) && (Funct3 == 3'b001)) || // SLL
                  ((ALUOp == 2'b10) && (Funct3 == 3'b010));  // SLT

   assign op[3] = (ALUOp == 2'b01) || // Branch operations
                  (ALUOp == 2'b11);  // JAL/LUI

endmodule

//==================================
// Immediate Generator
//==================================
module imm_gen (
   input [31:0] imem_insn,
   output logic [31:0] imm_val
);

   wire [6:0] i_type = 7'b0010011;

   always_comb begin
       case(imem_insn[6:0])
           i_type: imm_val = {{20{imem_insn[31]}}, imem_insn[31:20]};
           default: imm_val = 32'h0;
       endcase
   end

endmodule

//==================================
// Execute Stage
//==================================
module execute (
```

```
    input rst_n, clk, Branch, ALUSrc, id_ex_RegWrite, id_ex_MemtoReg,
wb_RegWrite, stall,
    input [1:0] ALUOp, forwardA_ex, forwardB_ex,
    input [3:0] ctl,
    input [4:0] id_ex_rd,
    input [31:0] rd1, rd2, imm_val, imem_addr, imem_insn, wb_result,
mem_wb_MemtoReg,
    output Zero, ex_mem_RegWrite, ex_mem_MemtoReg,
    output [4:0] ex_mem_rd,
    output [31:0] ALU_result, target
);

    wire Zero_e;
    wire [31:0] operand, ALU_result_e, target_e;

    reg ex_mem_RegWrite_reg, Zero_reg, ex_mem_MemtoReg_reg;
    reg [4:0] ex_mem_rd_reg;
    reg [31:0] ALU_result_reg, target_reg;

    // ALU Operand Selection
    mux alu_mux (
        .s(ALUSrc),
        .a(rd2),
        .b(imm_val),
        .c(operand)
    );

    // ALU Execution
    alu alu (
        .ctl(ctl),
        .a(rd1),
        .b(operand),
        .Zero(Zero_e),
        .ALU_result(ALU_result_e)
    );
```

```
    // Target Address Calculation
    adder target_adder (
        .a(imem_addr),
        .b(imm_val << 1),
        .c(target_e)
    );

    always @(posedge clk or negedge rst_n) begin
        if (~rst_n) begin
            Zero_reg            <= 1'b0;
            ex_mem_RegWrite_reg <= 1'b0;
            ex_mem_MemtoReg_reg <= 1'b0;
            ex_mem_rd_reg       <= 5'b00000;
            ALU_result_reg      <= 32'd0;
            target_reg          <= 32'd0;
        end
        else begin
            Zero_reg            <= Zero_e;
            ex_mem_RegWrite_reg <= id_ex_RegWrite;
            ex_mem_MemtoReg_reg <= id_ex_MemtoReg;
            ex_mem_rd_reg       <= id_ex_rd;
            ALU_result_reg      <= ALU_result_e;
            target_reg          <= target_e;
        end
    end

    assign ALU_result     = ALU_result_reg;
    assign target         = target_reg;
    assign ex_mem_rd      = ex_mem_rd_reg;
    assign ex_mem_MemtoReg = ex_mem_MemtoReg_reg;
    assign Zero           = Zero_reg;
    assign ex_mem_RegWrite = ex_mem_RegWrite_reg;

endmodule

//=================================
```

```
// ALU Module
//==================================
module alu (
   input [3:0] ctl,
   input [31:0] a, b,
   output Zero,
   output reg [31:0] ALU_result
);

   assign Zero = (ALU_result == 0);

   always @(*) begin
       case (ctl)
           0:  ALU_result = a & b;
           1:  ALU_result = a | b;
           2:  ALU_result = $signed(a) + $signed(b);
           6:  ALU_result = $signed(a) - $signed(b);
           7:  ALU_result = (a < b) ? 32'd1 : 32'd0;
           12: ALU_result = ~(a | b);
           default: ALU_result = 32'd0;
       endcase
   end

endmodule

//==================================
// Memory Stage
//==================================
module memory (
   input rst_n, clk,
   input ex_mem_RegWrite, ex_mem_MemtoReg,
   input [4:0] ex_mem_rd,
   input [31:0] ex_mem_ALU_result,
   output mem_wb_RegWrite, mem_wb_MemtoReg,
   output [4:0] mem_wb_rd,
   output [31:0] mem_wb_ALU_result
```

```
);

   reg mem_wb_RegWrite_reg, mem_wb_MemtoReg_reg;
   reg [4:0] mem_wb_rd_reg;
   reg [31:0] mem_wb_ALU_result_reg;

   always @(posedge clk or negedge rst_n) begin
       if (~rst_n) begin
           mem_wb_RegWrite_reg   <= 1'b0;
           mem_wb_MemtoReg_reg   <= 1'b0;
           mem_wb_rd_reg         <= 5'b00000;
           mem_wb_ALU_result_reg <= 32'd0;
       end
       else begin
           mem_wb_RegWrite_reg   <= ex_mem_RegWrite;
           mem_wb_MemtoReg_reg   <= ex_mem_MemtoReg;
           mem_wb_rd_reg         <= ex_mem_rd;
           mem_wb_ALU_result_reg <= ex_mem_ALU_result;
       end
   end

   assign mem_wb_RegWrite   = mem_wb_RegWrite_reg;
   assign mem_wb_MemtoReg   = mem_wb_MemtoReg_reg;
   assign mem_wb_rd         = mem_wb_rd_reg;
   assign mem_wb_ALU_result = mem_wb_ALU_result_reg;

endmodule

//==================================
// Write Back Stage
//==================================
module write_back (
   input rst_n, clk,
   input MemtoReg, mem_wb_RegWrite,
   input [4:0] mem_wb_rd,
   input [31:0] read_data, mem_wb_ALU_result,
```

```
    output wb_RegWrite,
    output [4:0] reg_addr,
    output [31:0] write_data
);

    wire [31:0] write_data_w;

    reg wb_RegWrite_reg;
    reg [4:0] reg_addr_reg;
    reg [31:0] write_data_reg;

    // MUX to select between memory read data and ALU result
    mux write_back_mux (
        .s(MemtoReg),
        .a(mem_wb_ALU_result),
        .b(read_data),
        .c(write_data_w)
    );

    always @(posedge clk or negedge rst_n) begin
        if (~rst_n) begin
            wb_RegWrite_reg <= 1'b0;
            reg_addr_reg    <= 5'b00000;
            write_data_reg  <= 32'd0;
        end
        else begin
            wb_RegWrite_reg <= mem_wb_RegWrite;
            reg_addr_reg    <= mem_wb_rd;
            write_data_reg  <= write_data_w;
        end
    end

    assign wb_RegWrite = wb_RegWrite_reg;
    assign reg_addr    = reg_addr_reg;
    assign write_data  = write_data_reg;
```

```
endmodule

//==================================
// Multiplexers
//==================================
module mux (
   input s,
   input [31:0] a, b,
   output [31:0] c
);
   assign c = ~s ? a : b;
endmodule

module mux4_1 (
   input [1:0] s,
   input [31:0] a, b, c,
   output [31:0] d
);
   assign d = (s == 2'b01) ? b : (s == 2'b10) ? c : a;
endmodule

//==================================
// Hazard Detection Unit
//==================================
module hazard_detection_unit (
   input id_ex_MemRead, ex_mem_RegWrite, id_ex_RegWrite, mem_wb_RegWrite,
wb_RegWrite,
   input [4:0] rs1, rs2, id_ex_rd, ex_mem_rd, mem_wb_rd, reg_addr,
   output reg stall
);

   always @(*) begin
       if (id_ex_RegWrite && ((id_ex_rd == rs1) || (id_ex_rd == rs2)))
begin
           stall = 1;
       end
```

```
       else if (ex_mem_RegWrite && ((ex_mem_rd == rs1) || (ex_mem_rd ==
rs2))) begin
           stall = 1;
       end
       else if (mem_wb_RegWrite && ((mem_wb_rd == rs1) || (mem_wb_rd ==
rs2))) begin
           stall = 1;
       end
       else if (wb_RegWrite && ((reg_addr == rs1) || (reg_addr == rs2)))
begin
           stall = 1;
       end
       else if (id_ex_MemRead && ((id_ex_rd == rs1) || (id_ex_rd == rs2)))
begin
           stall = 1;
       end
       else begin
           stall = 0;
       end
   end

endmodule

//==================================
// Forwarding Unit
//==================================
module forward_unit (
   input ex_mem_RegWrite, wb_RegWrite,
   input [4:0] rs1, rs2, ex_mem_rd, reg_addr,
   output [1:0] forwardA, forwardB
);

   // Forwarding logic
   assign forwardA = ((ex_mem_RegWrite) && (ex_mem_rd != 0) && (ex_mem_rd
== rs1)) ? 2'b10 :
```

```
                    ((wb_RegWrite) && (reg_addr != 0) && (reg_addr ==
rs1)) ? 2'b01 : 2'b00;

   assign forwardB = ((ex_mem_RegWrite) && (ex_mem_rd != 0) && (ex_mem_rd
== rs2)) ? 2'b10 :
                    ((wb_RegWrite) && (reg_addr != 0) && (reg_addr ==
rs2)) ? 2'b01 : 2'b00;

endmodule
```

Trace files:

```
WARNING: Too many words specified in data file addi_nohazard.dat
clk cycle     1 | write_data: 0 | reg 0 | forwardA: 0, forwardB: 0, stall: 0 | rs1: 0 | rs2: 5 | wb_RegWrite: 0
clk cycle     2 | write_data: 0 | reg 0 | forwardA: 0, forwardB: 0, stall: 0 | rs1: 0 | rs2: 16 | wb_RegWrite: 0
clk cycle     3 | write_data: 0 | reg 0 | forwardA: 0, forwardB: 0, stall: 0 | rs1: 0 | rs2: 3 | wb_RegWrite: 0
clk cycle     4 | write_data: 0 | reg 0 | forwardA: 0, forwardB: 0, stall: 0 | rs1: 0 | rs2: 1f | wb_RegWrite: 0
clk cycle     5 | write_data: 5 | reg 5 | forwardA: 0, forwardB: 0, stall: 1 | rs1: 0 | rs2: 5 | wb_RegWrite: 1
clk cycle     6 | write_data: fffffff6 | reg 6 | forwardA: 0, forwardB: 0, stall: 0 | rs1: 0 | rs2: 5 | wb_RegWrite: 1
clk cycle     7 | write_data: 3 | reg 7 | forwardA: 0, forwardB: 0, stall: 0 | rs1: 0 | rs2: b | wb_RegWrite: 1
clk cycle     8 | write_data: ffffffff | reg 28 | forwardA: 0, forwardB: 0, stall: 0 | rs1: 5 | rs2: 15 | wb_RegWrite: 1
clk cycle     9 | write_data: 0 | reg 0 | forwardA: 0, forwardB: 0, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: 0
clk cycle    10 | write_data: 5 | reg 29 | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: 1
clk cycle    11 | write_data: b | reg 30 | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: 1
clk cycle    12 | write_data: fffffffa | reg 31 | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: 1
clk cycle    13 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    14 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    15 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    16 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    17 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    18 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    19 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    20 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    21 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    22 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    23 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    24 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    25 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    26 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    27 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    28 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    29 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    30 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    31 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    32 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    33 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    34 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    35 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    36 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    37 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    38 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    39 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    40 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    41 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    42 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    43 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    44 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    45 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    46 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    47 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    48 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
relaunch_sim: Time (s): cpu = 00:00:02 ; elapsed = 00:00:10 . Memory (MB): peak = 1957.391 ; gain = 0.000
```

Figure 3: no_hardzards trace file

```
clk cycle     1 | write_data: 0 | reg 0 | forwardA: 0, forwardB: 0, stall: 0 | rs1: 0 | rs2: 5 | wb_RegWrite: 0
clk cycle     2 | write_data: 0 | reg 0 | forwardA: 0, forwardB: 0, stall: 1 | rs1: 5 | rs2: 16 | wb_RegWrite: 0
clk cycle     3 | write_data: 0 | reg 0 | forwardA: 0, forwardB: 0, stall: 1 | rs1: 5 | rs2: 16 | wb_RegWrite: 0
clk cycle     4 | write_data: 0 | reg 0 | forwardA: 0, forwardB: 0, stall: 1 | rs1: 5 | rs2: 16 | wb_RegWrite: 0
clk cycle     5 | write_data: 5 | reg 5 | forwardA: 0, forwardB: 0, stall: 1 | rs1: 5 | rs2: 16 | wb_RegWrite: 1
clk cycle     6 | write_data: 0 | reg 0 | forwardA: 0, forwardB: 0, stall: 0 | rs1: 5 | rs2: 16 | wb_RegWrite: 0
clk cycle     7 | write_data: 0 | reg 0 | forwardA: 0, forwardB: 0, stall: 0 | rs1: 5 | rs2: 3 | wb_RegWrite: 0
clk cycle     8 | write_data: 0 | reg 0 | forwardA: 0, forwardB: 0, stall: 0 | rs1: 5 | rs2: 1f | wb_RegWrite: 0
clk cycle     9 | write_data: 0 | reg 0 | forwardA: 0, forwardB: 0, stall: 1 | rs1: 6 | rs2: 5 | wb_RegWrite: 0
clk cycle    10 | write_data: fffffffb | reg 6 | forwardA: 0, forwardB: 0, stall: 1 | rs1: 6 | rs2: 5 | wb_RegWrite:
clk cycle    11 | write_data: 8 | reg 7 | forwardA: 0, forwardB: 0, stall: 0 | rs1: 6 | rs2: 5 | wb_RegWrite: 1
clk cycle    12 | write_data: 4 | reg 28 | forwardA: 0, forwardB: 0, stall: 1 | rs1: 1c | rs2: b | wb_RegWrite: 1
clk cycle    13 | write_data: 0 | reg 0 | forwardA: 0, forwardB: 0, stall: 0 | rs1: 1c | rs2: b | wb_RegWrite: 0
clk cycle    14 | write_data: 0 | reg 0 | forwardA: 0, forwardB: 0, stall: 1 | rs1: 1e | rs2: 15 | wb_RegWrite: 0
clk cycle    15 | write_data: 0 | reg 29 | forwardA: 0, forwardB: 0, stall: 1 | rs1: 1e | rs2: 15 | wb_RegWrite: 1
clk cycle    16 | write_data: 0 | reg 0 | forwardA: 0, forwardB: 0, stall: 1 | rs1: 1e | rs2: 15 | wb_RegWrite: 0
clk cycle    17 | write_data: f | reg 30 | forwardA: 0, forwardB: 0, stall: 1 | rs1: 1e | rs2: 15 | wb_RegWrite: 1
clk cycle    18 | write_data: 0 | reg 0 | forwardA: 0, forwardB: 0, stall: 0 | rs1: 1e | rs2: 15 | wb_RegWrite: 0
clk cycle    19 | write_data: 0 | reg 0 | forwardA: 0, forwardB: 0, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: 0
clk cycle    20 | write_data: 0 | reg 0 | forwardA: X, forwardB: X, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: 0
clk cycle    21 | write_data: 0 | reg 0 | forwardA: X, forwardB: X, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: 0
clk cycle    22 | write_data: 4 | reg 31 | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: 1
clk cycle    23 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    24 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    25 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    26 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    27 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    28 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    29 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    30 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    31 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    32 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    33 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    34 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    35 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    36 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    37 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    38 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    39 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    40 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    41 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    42 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    43 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    44 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    45 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    46 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    47 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
clk cycle    48 | write_data: 0 | reg x | forwardA: x, forwardB: x, stall: 0 | rs1: xx | rs2: xx | wb_RegWrite: x
```

Figure 3: addi hardzards trace file

Graph Behaviors:

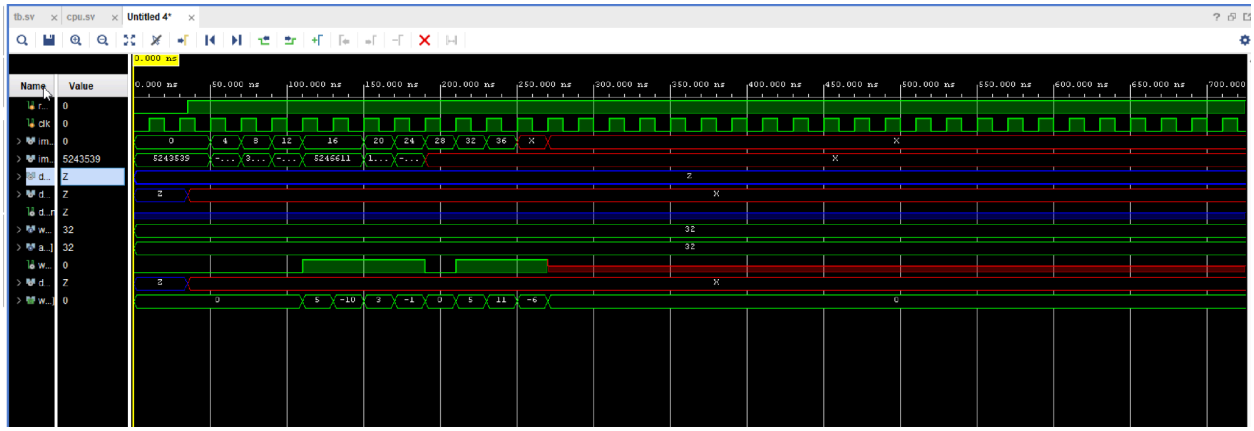


Figure 5: Addi no hardzards

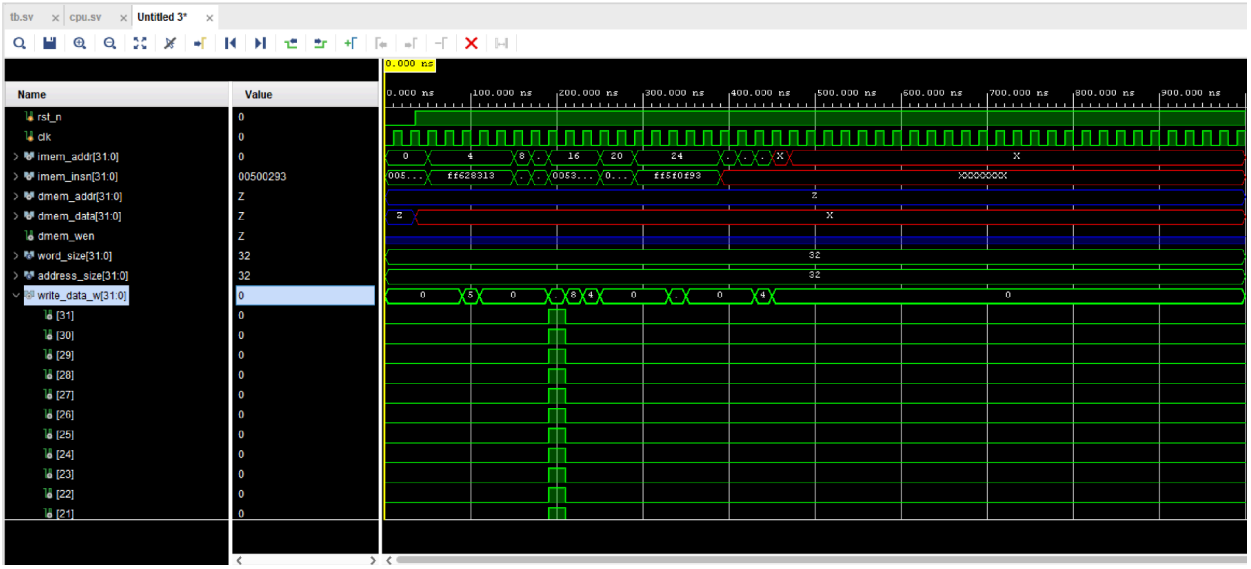


Figure 6: Addi with hardzards

What did we learned from this lab design?

In this lab we were tasked to create a pipelined RISC-V processor. The processor was implemented with a five-stage pipeline; instruction fetch, instruction decode, execute, memory access, and write back to analyze how the data is transferred at each stage. This setup taught us how important correct execution order is. The lab focuses on handling data hazards which are needed for pipeline stalls to make sure the execution is correct. The challenging section of the lab was getting it together and making sure everything was running smoothly. Throughout this lab we got a better understanding of pipelining, CPU design, and understanding the development of the processor.