ELEC 374 – Phase 2 Report

Department of Electrical and Computer Engineering
Queen's University

Composed By:
Zeerak Asim (20237955)
Nicholas Seegobin (20246787)

Date of Submission:
31 March 2023

Datapath

```
/* Representation of the datapath in Verilog HDL. */
/* Connected outputs of encoder to select signals in multiplexer. */
module datapath(
    // CPU signals
    input wire clk,
    input wire clr,

    // Register write/enable signals
    input wire PC_enable, PC_increment_enable, IR_enable,
    input wire con_enable,
    input wire Y_enable, Z_enable,
    input wire MAR_enable, MDR_enable,
    input wire HI_enable, LO_enable,
    input wire manual_R15_enable,
    input wire outport_enable,

    // Memory Data Multiplexer Read/Select Signal
    input wire read, write,

    // Select and Encode Logic Inputs
    input wire Gra, Grb, Grc, r_enable, r_select, BAout,

    // Encoder Output Select Signals
    input wire PC_select,
    input wire HI_select, LO_select,
    input wire Z_HI_select, Z_LO_select,
    input wire MDR_select,
    input wire inport_select,
    input wire c_select,

    output wire [4:0] bus_select,
    output wire [15:0] register_select,

    // ALU Opcode
    input wire [4:0] alu_instruction,

    // Output Data Signals
    output wire [31:0] bus_Data, // Data currently in the bus
    // output wire [63:0] aluResult,

    // CON FF Module
    output wire con_output,

    output wire [31:0] R0_Data, R1_Data, R2_Data, R3_Data,
    output wire [31:0] R4_Data, R5_Data, R6_Data, R15_Data,
    output wire [31:0] debug_port_01, debug_port_02,
    output wire [31:0] outport_Data,

    output wire [31:0] PC_Data, IR_Data,
    output wire [31:0] Y_Data,
    output wire [31:0] HI_Data, LO_Data,
    output wire [31:0] Z_HI_Data, Z_LO_Data,
    output wire [8:0] MAR_Data,
    output wire [31:0] MDR_Data, MDataIN);
```

```
    // Enable Signals
    wire [15:0] register_enable;
    wire R15_enable;

    wire [63:0] aluResult;

    wire [31:0] R7_Data, R8_Data, R9_Data, R10_Data,
    R11_Data, R12_Data, R13_Data, R14_Data;

    wire [31:0] input_Data, inport_Data;
    wire [31:0] C_sign_ext_Data;

    assign input_Data = 32'd9;
    assign R15_enable = manual_R15_enable | register_enable[15];

    /* Bus Components */
    // 32-to-5 Encoder
    encoder encoder_instance(.register_select(register_select),
.HI_select(HI_select), .LO_select(LO_select),
    .Z_HI_select(Z_HI_select), .Z_LO_select(Z_LO_select), .PC_select(PC_select),
.MDR_select(MDR_select),
    .inport_select(inport_select), .c_select(c_select),
.selectSignal(bus_select));

    // 32-to-1 Multiplexer
    multiplexer multiplexer_instance(.selectSignal(bus_select),
.muxIN_r0(R0_Data),
    .muxIN_r1(R1_Data), .muxIN_r2(R2_Data), .muxIN_r3(R3_Data),
.muxIN_r4(R4_Data),
    .muxIN_r5(R5_Data), .muxIN_r6(R6_Data), .muxIN_r7(R7_Data),
.muxIN_r8(R8_Data),
    .muxIN_r9(R9_Data), .muxIN_r10(R10_Data), .muxIN_r11(R11_Data),
.muxIN_r12(R12_Data),
    .muxIN_r13(R13_Data), .muxIN_r14(R14_Data), .muxIN_r15(R15_Data),
.muxIN_HI(HI_Data),
    .muxIN_LO(LO_Data), .muxIN_Z_HI(Z_HI_Data), .muxIN_Z_LO(Z_LO_Data),
.muxIN_PC(PC_Data),
    .muxIN_MDR(MDR_Data), .muxIN_inport(inport_Data),
.muxIN_C_sign_ext(C_sign_ext_Data), .muxOut(bus_Data));

    // General purpose registers r0 -> r15
    R0_revised r0 (.clk(clk), .clr(clr), .enable(register_enable[0]),
.BAout(BAout),
                   .bus_Data(bus_Data), .R0_Data(R0_Data));
    register r1 (.clk(clk), .clr(clr), .enable(register_enable[1]),
.D(bus_Data), .Q(R1_Data));
```

```
    register r2 (.clk(clk), .clr(clr), .enable(register_enable[2]),
.D(bus_Data), .Q(R2_Data));
    register r3 (.clk(clk), .clr(clr), .enable(register_enable[3]),
.D(bus_Data), .Q(R3_Data));
    register r4 (.clk(clk), .clr(clr), .enable(register_enable[4]),
.D(bus_Data), .Q(R4_Data));
    register r5 (.clk(clk), .clr(clr), .enable(register_enable[5]),
.D(bus_Data), .Q(R5_Data));
    register r6 (.clk(clk), .clr(clr), .enable(register_enable[6]),
.D(bus_Data), .Q(R6_Data));
    register r7 (.clk(clk), .clr(clr), .enable(register_enable[7]),
.D(bus_Data), .Q(R7_Data));
    register r8 (.clk(clk), .clr(clr), .enable(register_enable[8]),
.D(bus_Data), .Q(R8_Data));
    register r9 (.clk(clk), .clr(clr), .enable(register_enable[9]),
.D(bus_Data), .Q(R9_Data));
    register r10 (.clk(clk), .clr(clr), .enable(register_enable[10]),
.D(bus_Data), .Q(R10_Data));
    register r11 (.clk(clk), .clr(clr), .enable(register_enable[11]),
.D(bus_Data), .Q(R11_Data));
    register r12 (.clk(clk), .clr(clr), .enable(register_enable[12]),
.D(bus_Data), .Q(R12_Data));
    register r13 (.clk(clk), .clr(clr), .enable(register_enable[13]),
.D(bus_Data), .Q(R13_Data));
    register r14 (.clk(clk), .clr(clr), .enable(register_enable[14]),
.D(bus_Data), .Q(R14_Data));
    register r15 (.clk(clk), .clr(clr), .enable(R15_enable), .D(bus_Data),
.Q(R15_Data));

    // ALU Output Registers
    register HI (.clk(clk), .clr(clr), .enable(HI_enable), .D(bus_Data),
.Q(HI_Data));
    register LO (.clk(clk), .clr(clr), .enable(LO_enable), .D(bus_Data),
.Q(LO_Data));
    register Z_HI (.clk(clk), .clr(clr), .enable(Z_enable),
.D(aluResult[63:32]), .Q(Z_HI_Data));
    register Z_LO (.clk(clk), .clr(clr), .enable(Z_enable), .D(aluResult[31:0]),
.Q(Z_LO_Data));
    register Y (.clk(clk), .clr(clr), .enable(Y_enable), .D(bus_Data),
.Q(Y_Data));

    // ALU Instance
    alu alu_instance(.A(Y_Data), .B(bus_Data), .opcode(alu_instruction),
.result(aluResult));
```

```
    // PC and IR Registers
    program_counter PC (.clk(clk), .clr(clr), .enable(PC_enable),
                        .incPC(PC_increment_enable), .PC_Input(bus_Data),
.PC_Output(PC_Data));
    register IR (.clk(clk), .clr(clr), .enable(IR_enable), .D(bus_Data),
.Q(IR_Data));

    // Memory Registers
    register MAR (.clk(clk), .clr(clr), .enable(MAR_enable), .D(bus_Data),
.Q(MAR_Data));
    md_register MDR (.clk(clk), .clr(clr), .enable(MDR_enable), .read(read),
                     .MDataIN(MDataIN), .bus_Data(bus_Data), .Q(MDR_Data));

    // Ram Instance
    ram ramInstance(.debug_port_01(debug_port_01),
.debug_port_02(debug_port_02),
                    .clk(clk), .read(read), .write(write), .data_in(MDR_Data),
                    .address_in(MAR_Data), .data_out(MDataIN));

    select_encode_logic selInstance(.instruction(IR_Data), .Gra(Gra), .Grb(Grb),
                                    .Grc(Grc), .r_enable(r_enable),
.r_select(r_select),
```

Memory Subsystem

Ram

```
module ram(
    input clk,          // RAM enable control signal (active high)
    input read,                 // Read control signal (active high)
    input write,                // Write control signal (active high)
    input [31:0] data_in,       // Input data
    input [8:0] address_in,     // Address input
    output wire [31:0] data_out, // Output data

    output [31:0] debug_port_01,
    output [31:0] debug_port_02
);

reg [31:0] mem [511:0]; // Memory array
reg [31:0] tempData;    // Temporary data storage

initial begin
    // $readmemh("load.mif", mem);
    // $readmemh("loadi.mif", mem);
    // $readmemh("store.mif", mem);
    // $readmemh("addi.mif", mem);
    // $readmemh("andi.mif", mem);
    // $readmemh("ori.mif", mem);
    // $readmemh("brzr.mif", mem);
    // $readmemh("brnz.mif", mem);
    // $readmemh("brpl.mif", mem);
    // $readmemh("brmi.mif", mem);
    // $readmemh("jr.mif", mem);
    // $readmemh("jal.mif", mem);
    // $readmemh("mfhi.mif", mem);
    // $readmemh("mflo.mif", mem);
    // $readmemh("inport.mif", mem);
    // $readmemh("outport.mif", mem);
end
```

```
always @(posedge clk) begin
    if (write) begin
        mem[address_in] <= data_in; // Write data to memory location
    end
    if (read) begin
        tempData <= mem[address_in]; // Read data from memory location
    end
end

assign data_out = tempData;

assign debug_port_01 = mem [144];
assign debug_port_02 = mem [247];

endmodule
```

Ram MIF Files

File Name (.mif)	Memory Contents
addi	09800005 611FFFFD
andi	0980000C 69180019
brmi	0B07FFFB 9B180019
brnz	0B000005 9B080019
brpl	0B000005 9B100019
brzr	0B000000 9B000019
inport	B1800000
jal	0900000F A9000000
jr	0900000F A1000000
load	00800075 00080045 0x3 at address 0x75 in load.mif 0x5 at address 0xBA in load.mif
loadi	08800075 08080045
mfhi	09000005 C2000000
mflo	09000005 CB000000
ori	0980000C

	71180019
outport	09000045 B9000000
store	0A000067 12000090 12200090

Memory Data Register (MDR)

```verilog
/* Representation of a memory data register with 2 to 1 multiplexer in Verilog */
module md_register(input clk, input clr, input enable, input read, input [31:0] MDataIN, input [31:0] bus_Data, output reg [31:0] Q);

    /* While loop that iterates every positive clock edge. */
    always @(posedge clk)
    begin

        /* If clear signal is high, set Q output to 0. */
        if(clr)
            Q <= 0;

        /* If enable signal is high, set Q output to follow (or equal to) D input. */
        else if(enable)
            Q <= read ? MDataIN : bus_Data;
    end
endmodule // md_register end.
```

Select and Encode Logic

```
module select_encode_logic(
    // Instruction Register Data
    input [31:0] instruction,

    // Select and Encode Logic Control Signals
    input Gra, Grb, Grc, r_enable, r_select, ba_select,

    // Array of Enable and Select Output Signals
    output [15:0] register_enable, register_select,

    // Sign Extension of Constant C
    output [31:0] C_sign_ext_Data);

    // Instruction Register Content Variables
    reg [3:0] Ra, Rb, Rc;

    // 4 to 16 Decoder Input/Output Variable
    reg [3:0] decoder_input;
    reg [15:0] decoder_out;

    always @ (instruction, Gra, Grb, Grc) begin
        // Assigning Values to Instruction Register Content Variables
        Ra = instruction [26:23];        // Bit 23 -> 26 of the instruction register is register a
        Rb = instruction [22:19];        // Bit 19 -> 22 of the instruction register is register b
        Rc = instruction [19:15];        // Bit 15 -> 19 of the instruction register is register c

        if (Gra) decoder_input = Ra;
        else if (Grb) decoder_input = Rb;
        else if (Grc) decoder_input = Rc;
```

```
        case (decoder_input)
            4'd0: decoder_out = 16'd1;
            4'd1: decoder_out = 16'd2;
            4'd2: decoder_out = 16'd4;
            4'd3: decoder_out = 16'd8;
            4'd4: decoder_out = 16'd16;
            4'd5: decoder_out = 16'd32;
            4'd6: decoder_out = 16'd64;
            4'd7: decoder_out = 16'd128;
            4'd8: decoder_out = 16'd256;
            4'd9: decoder_out = 16'd512;
            4'd10: decoder_out = 16'd1024;
            4'd11: decoder_out = 16'd2048;
            4'd12: decoder_out = 16'd4096;
            4'd13: decoder_out = 16'd8192;
            4'd14: decoder_out = 16'd16384;
            4'd15: decoder_out = 16'd32768;
        endcase
    end

    // Assigning Values to the Outputs
    assign register_enable = {16{r_enable}} & decoder_out;
    assign register_select = ({16{ba_select}} | {16{r_select}}) & decoder_out;
    assign C_sign_ext_Data = {{13{instruction[18]}}, instruction[18:0]};

endmodule
```

Revision of R0 Register

```
module R0_revised(input clk, input clr, input enable, input ba_select, input [31:0] bus_Data, output wire [31:0] R0_Data);

    wire [31:0] registerOutput;

    register R0(clk, clr, enable, bus_Data, registerOutput);
    assign R0_Data = registerOutput & {32{!ba_select}};

endmodule // revised_register_R0 end.
```

CON FF Logic

```
module con_ff(
    // Bus Input
    input [31:0] bus_Data,

    // Instruction Register Input
    input [31:0] instruction,

    // Enable Signal for the CON Flip Flop
    input con_enable,

    // Q Output Signal from the CON Flip Flop
    output con_output);

    reg [3:0] decoder_out;
    reg FF_Output;
    assign con_output = FF_Output;

    always @ (instruction[20:19]) begin
        case (instruction[20:19])
            2'b00: begin
                decoder_out = 4'b0001;
            end

            2'b01: begin
                decoder_out = 4'b0010;
            end

            2'b10: begin
                decoder_out = 4'b0100;
            end

            2'b11: begin
                decoder_out = 4'b1000;
            end
        endcase
    end
```

```
always @ (bus_Data && con_enable) begin
    if (bus_Data == 0 && decoder_out[0]) begin
        FF_Output = 1;
    end else if (bus_Data != 0 && decoder_out[1]) begin
        FF_Output = 1;
    end else if (bus_Data >= 0 && decoder_out[2]) begin
        FF_Output = 1;
    end else if (bus_Data[31] && decoder_out[3]) begin
        FF_Output = 1;
    end else begin
        FF_Output = 0;
    end
end

endmodule
```

Input and Output Ports

```
module inport(
    input clk, clr,
    input wire [31:0] input_Data,
    output wire [31:0] inport_Data);

reg [31:0] tempData;
initial tempData = 32'h0;

always @(posedge clk)
begin
    if (clr) tempData <= {32{1'b0}};
    else tempData <= input_Data;
end

assign inport_Data = tempData[31:0];

endmodule
```

```
module outport(
    input clk, clr, enable,
    input wire [31:0] bus_Data,
    output wire [31:0] outport_Data);

reg [31:0] tempData;
initial tempData = 32'h0;

always @(posedge clk)
begin
    if (clr) tempData <= {32{1'b0}};
    else if (enable) tempData <= bus_Data;
end

assign outport_Data = tempData[31:0];

endmodule
```

PC Incrementor Unit

```
module program_counter(
    input wire clk, clr, enable, incPC,
    input wire [31:0] PC_Input,
    output wire [31:0] PC_Output
);

reg [31:0] Q;
reg incFlag; // flag to indicate that PC should be incremented on next clock edge

initial begin
    Q <= 0;
    incFlag = 1;
end

always @ (posedge clk) begin
     if (clr) begin
        Q <= 0;
    end
    else if (enable) begin
        Q <= PC_Input;
    end
    else begin
        if (incPC == 1 && incFlag == 1) begin
            Q <= Q + 1;
            incFlag <= 0;
        end
        else if (incPC == 0) begin
            incFlag <= 1;
        end
    end
end

assign PC_Output = Q;

endmodule
```

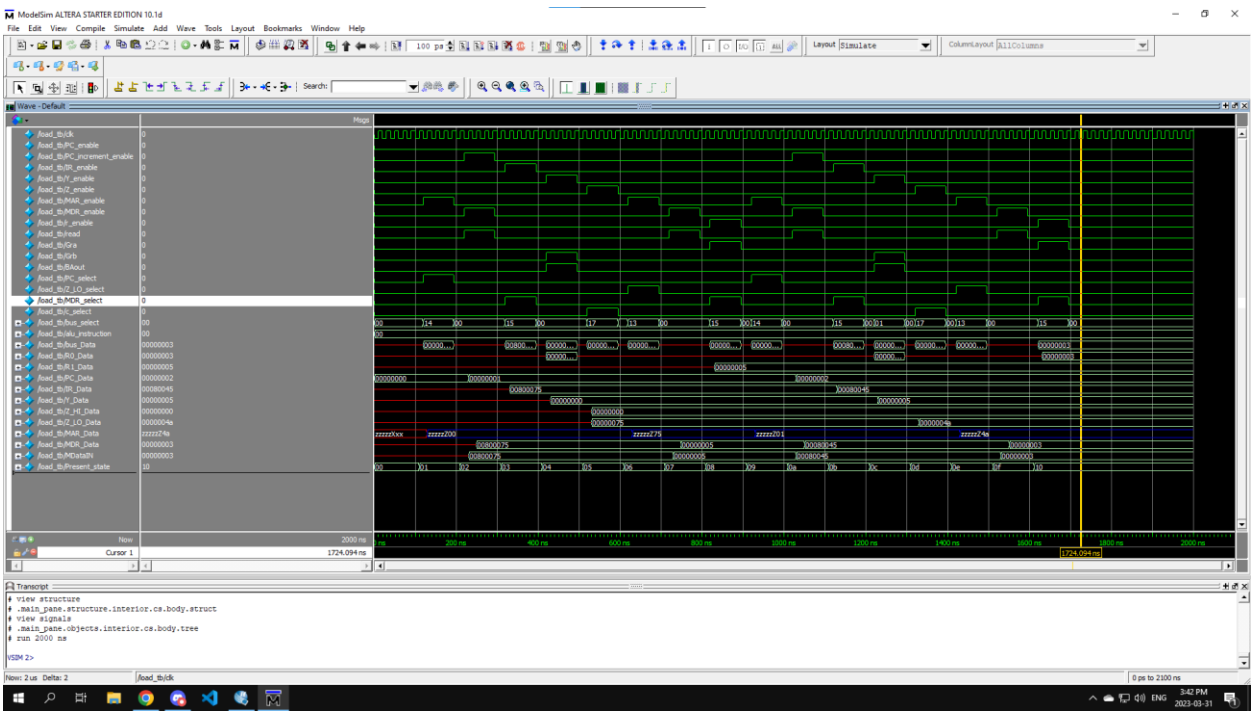
Memory Subsystem Testbench Files

Load Testbench

```
// Time Signals and Load Registers
parameter Default = 0, load_01_T0 = 1, load_01_T1 = 2, load_01_T2 = 3, load_01_T3 = 4, load_01_T4 = 5,
load_01_T5 = 6, load_01_T6 = 7, load_01_T7 = 8, load_02_T0 = 9, load_02_T1 = 10, load_02_T2 = 11, load_02_T3 = 12,
load_02_T4 = 13, load_02_T5 = 14, load_02_T6 = 15, load_02_T7 = 16;

reg [4:0] Present_state = Default;

initial begin clk = 0; Present_state = Default; end
always #10 clk = ~clk;

always @(posedge clk) // finite state machine; if clk rising-edge
    begin
        case (Present_state)
            Default: #100 Present_state = load_01_T0;
            load_01_T0 : #100 Present_state = load_01_T1;
            load_01_T1 : #100 Present_state = load_01_T2;
            load_01_T2 : #100 Present_state = load_01_T3;
            load_01_T3 : #100 Present_state = load_01_T4;
            load_01_T4 : #100 Present_state = load_01_T5;
            load_01_T5 : #100 Present_state = load_01_T6;
            load_01_T6 : #100 Present_state = load_01_T7;

            load_01_T7 : #100 Present_state = load_02_T0;
            load_02_T0 : #100 Present_state = load_02_T1;
            load_02_T1 : #100 Present_state = load_02_T2;
            load_02_T2 : #100 Present_state = load_02_T3;
            load_02_T3 : #100 Present_state = load_02_T4;
            load_02_T4 : #100 Present_state = load_02_T5;
            load_02_T5 : #100 Present_state = load_02_T6;
            load_02_T6 : #100 Present_state = load_02_T7;
        endcase
    end
```

```
always @(Present_state) // do the required job in each state
    begin
        case (Present_state) // assert the required signals in each clk cycle
            Default: begin
                // Enable Signals
                MDR_enable <= 0; MAR_enable <= 0;
                IR_enable <= 0;
                Y_enable <= 0; Z_enable <= 0;
                PC_enable <= 0; PC_increment_enable <= 0;
                r_enable <= 0;

                // Select Signals
                PC_select <= 0;
                MDR_select <= 0;
                Z_LO_select <= 0; read <= 0;
                c_select <= 0;

                // Select and Encode Signals
                Gra <= 0; Grb <= 0; ba_select <= 0;

                // Register Contents
                alu_instruction <= 0;
            end

            load_01_T0, load_02_T0: begin // see if you need to de-assert these signals
                #10 PC_select <= 1; MAR_enable <= 1;
                #75 PC_select <= 0; MAR_enable <= 0;
            end

            load_01_T1, load_02_T1: begin
                #10 PC_increment_enable <= 1; read <= 1; MDR_enable <= 1;
                #75 PC_increment_enable <= 0; read <= 0; MDR_enable <= 0;
            end
```

```
        load_01_T2, load_02_T2: begin
            #10 MDR_select <= 1; IR_enable <= 1;
            #75 MDR_select <= 0; IR_enable <= 0;
        end

        load_01_T3, load_02_T3: begin
            #10 Grb <= 1; ba_select <= 1; Y_enable <= 1;
            #75 Grb <= 0; ba_select <= 0; Y_enable <= 0;
        end

        load_01_T4, load_02_T4: begin
            #10 c_select <= 1; alu_instruction <= 5'b00000; Z_enable <= 1;
            #75 c_select <= 0; alu_instruction <= 0; Z_enable <= 0;
        end

        load_01_T5, load_02_T5: begin
            #10 Z_LO_select <= 1; MAR_enable <= 1;
            #75 Z_LO_select <= 0; MAR_enable <= 0;
        end

        load_01_T6, load_02_T6: begin
            #10 read <= 1; MDR_enable <= 1;
            #75 read <= 0; MDR_enable <= 0;
        end

        load_01_T7, load_02_T7: begin
            #10 MDR_select <= 1; Gra <= 1; r_enable <= 1;
            #75 MDR_select <= 0; Gra <= 0; r_enable <= 0;
        end
    endcase
end
```

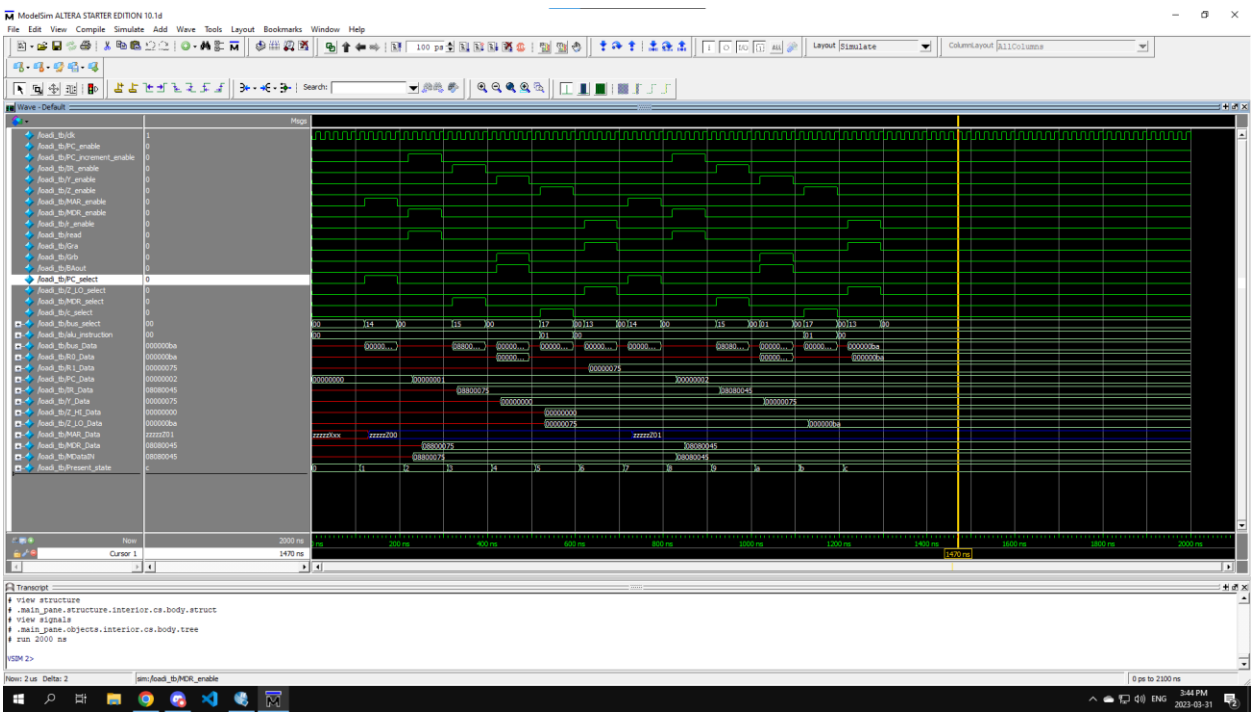


Load Immediate Testbench

```
// Time Signals and Load Registers
parameter Default = 0, loadi_01_T0 = 1, loadi_01_T1 = 2, loadi_01_T2 = 3, loadi_01_T3 = 4, loadi_01_T4 = 5,
loadi_01_T5 = 6, loadi_02_T0 = 7, loadi_02_T1 = 8, loadi_02_T2 = 9, loadi_02_T3 = 10, loadi_02_T4 = 11, loadi_02_T5 = 12;

reg [3:0] Present_state = Default;

initial begin clk = 0; Present_state = Default; end
always #10 clk = ~clk;

always @(posedge clk) // finite state machine; if clk rising-edge
    begin
        case (Present_state)
            Default: #100 Present_state = loadi_01_T0;
            loadi_01_T0 : #100 Present_state = loadi_01_T1;
            loadi_01_T1 : #100 Present_state = loadi_01_T2;
            loadi_01_T2 : #100 Present_state = loadi_01_T3;
            loadi_01_T3 : #100 Present_state = loadi_01_T4;
            loadi_01_T4 : #100 Present_state = loadi_01_T5;

            loadi_01_T5 : #100 Present_state = loadi_02_T0;
            loadi_02_T0 : #100 Present_state = loadi_02_T1;
            loadi_02_T1 : #100 Present_state = loadi_02_T2;
            loadi_02_T2 : #100 Present_state = loadi_02_T3;
            loadi_02_T3 : #100 Present_state = loadi_02_T4;
            loadi_02_T4 : #100 Present_state = loadi_02_T5;

        endcase
    end
```

```
always @(Present_state) // do the required job in each state
    begin
        case (Present_state) // assert the required signals in each clk cycle
            Default: begin
                // Enable Signals
                MDR_enable <= 0; MAR_enable <= 0;
                IR_enable <= 0;
                Y_enable <= 0; Z_enable <= 0;
                PC_enable <= 0; PC_increment_enable <= 0;
                r_enable <= 0;

                // Select Signals
                PC_select <= 0;
                MDR_select <= 0;
                Z_LO_select <= 0; read <= 0;
                c_select <= 0;

                // Select and Encode Signals
                Gra <= 0; Grb <= 0; ba_select <= 0;

                // Register Contents
                alu_instruction <= 0;
            end
```

```
        loadi_01_T0, loadi_02_T0: begin // see if you need to de-assert these signals
            #10 PC_select <= 1; MAR_enable <= 1;
            #75 PC_select <= 0; MAR_enable <= 0;
        end

        loadi_01_T1, loadi_02_T1: begin
            #10 PC_increment_enable <= 1; read <= 1; MDR_enable <= 1;
            #75 PC_increment_enable <= 0; read <= 0; MDR_enable <= 0;
        end

        loadi_01_T2, loadi_02_T2: begin
            #10 MDR_select <= 1; IR_enable <= 1;
            #75 MDR_select <= 0; IR_enable <= 0;
        end

        loadi_01_T3, loadi_02_T3: begin
            #10 Grb <= 1; ba_select <= 1; Y_enable <= 1;
            #75 Grb <= 0; ba_select <= 0; Y_enable <= 0;
        end

        loadi_01_T4, loadi_02_T4: begin
            #10 c_select <= 1; alu_instruction <= 5'b00001; Z_enable <= 1;
            #75 c_select <= 0; alu_instruction <= 0; Z_enable <= 0;
        end

        loadi_01_T5, loadi_02_T5: begin
            #10 Z_LO_select <= 1; Gra <= 1; r_enable <= 1;
            #75 Z_LO_select <= 0; Gra <= 0; r_enable <= 0;
        end
```



Store Testbench

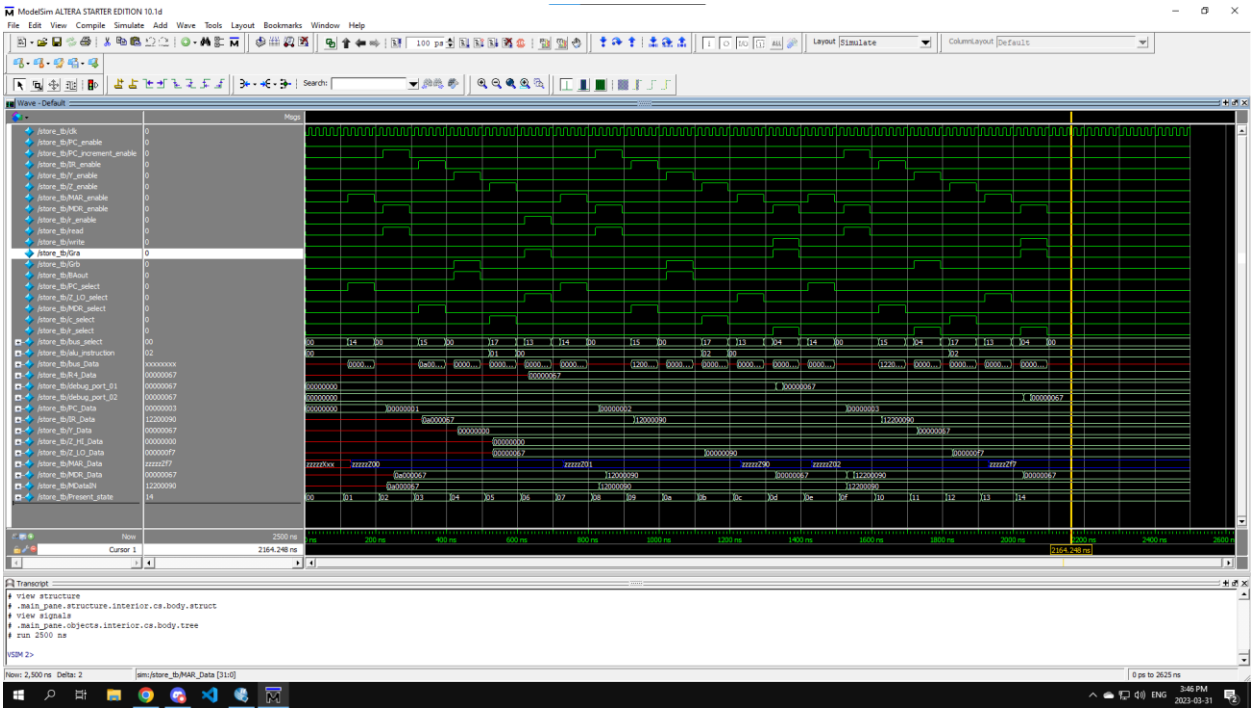
```
// Time Signals and Load Registers
parameter Default = 0, loadi_01_T0 = 1, loadi_01_T1 = 2, loadi_01_T2 = 3, loadi_01_T3 = 4, loadi_01_T4 = 5,
loadi_01_T5 = 6, store_01_T0 = 7, store_01_T1 = 8, store_01_T2 = 9, store_01_T3 = 10,
store_01_T4 = 11, store_01_T5 = 12, store_01_T6 = 13, store_02_T0 = 14, store_02_T1 = 15, store_02_T2 = 16, store_02_T3 = 17,
store_02_T4 = 18, store_02_T5 = 19, store_02_T6 = 20;

reg [4:0] Present_state = Default;

initial begin clk = 0; Present_state = Default; end
always #10 clk = ~clk;

always @(posedge clk) // finite state machine; if clk rising-edge
    begin
        case (Present_state)
            Default: #100 Present_state = loadi_01_T0;
            loadi_01_T0 : #100 Present_state = loadi_01_T1;
            loadi_01_T1 : #100 Present_state = loadi_01_T2;
            loadi_01_T2 : #100 Present_state = loadi_01_T3;
            loadi_01_T3 : #100 Present_state = loadi_01_T4;
            loadi_01_T4 : #100 Present_state = loadi_01_T5;

            loadi_01_T5 : #100 Present_state = store_01_T0;
            store_01_T0 : #100 Present_state = store_01_T1;
            store_01_T1 : #100 Present_state = store_01_T2;
            store_01_T2 : #100 Present_state = store_01_T3;
            store_01_T3 : #100 Present_state = store_01_T4;
            store_01_T4 : #100 Present_state = store_01_T5;
            store_01_T5 : #100 Present_state = store_01_T6;

            store_01_T6 : #100 Present_state = store_02_T0;
            store_02_T0 : #100 Present_state = store_02_T1;
            store_02_T1 : #100 Present_state = store_02_T2;
            store_02_T2 : #100 Present_state = store_02_T3;
            store_02_T3 : #100 Present_state = store_02_T4;
            store_02_T4 : #100 Present_state = store_02_T5;
            store_02_T5 : #100 Present_state = store_02_T6;
        endcase
    end
```

```
always @(Present_state) // do the required job in each state
    begin
        case (Present_state) // assert the required signals in each clk cycle
            Default: begin
                // Enable Signals
                MDR_enable <= 0; MAR_enable <= 0;
                IR_enable <= 0;
                Y_enable <= 0; Z_enable <= 0;
                PC_enable <= 0; PC_increment_enable <= 0;
                r_enable <= 0;

                // Select Signals
                PC_select <= 0;
                MDR_select <= 0;
                Z_LO_select <= 0; read <= 0;
                write <= 0; c_select <= 0;
                r_select <= 0;

                // Select and Encode Signals
                Gra <= 0; Grb <= 0; ba_select <= 0;

                // Register Contents
                alu_instruction <= 0;
            end

            loadi_01_T0: begin // see if you need to de-assert these signals
                #10 PC_select <= 1; MAR_enable <= 1;
                #75 PC_select <= 0; MAR_enable <= 0;
            end

            loadi_01_T1: begin
                #10 PC_increment_enable <= 1; read <= 1; MDR_enable <= 1;
                #75 PC_increment_enable <= 0; read <= 0; MDR_enable <= 0;
            end
```

```
        loadi_01_T2: begin
            #10 MDR_select <= 1; IR_enable <= 1;
            #75 MDR_select <= 0; IR_enable <= 0;
        end

        loadi_01_T3: begin
            #10 Grb <= 1; ba_select <= 1; Y_enable <= 1;
            #75 Grb <= 0; ba_select <= 0; Y_enable <= 0;
        end

        loadi_01_T4: begin
            #10 c_select <= 1; alu_instruction <= 5'b00001; Z_enable <= 1;
            #75 c_select <= 0; alu_instruction <= 0; Z_enable <= 0;
        end

        loadi_01_T5: begin
            #10 Z_LO_select <= 1; Gra <= 1; r_enable <= 1;
            #75 Z_LO_select <= 0; Gra <= 0; r_enable <= 0;
        end

        // Store Instruction 01
        store_01_T0: begin // see if you need to de-assert these signals
            #10 PC_select <= 1; MAR_enable <= 1;
            #75 PC_select <= 0; MAR_enable <= 0;
        end

        store_01_T1: begin
            #10 PC_increment_enable <= 1; read <= 1; MDR_enable <= 1;
            #75 PC_increment_enable <= 0; read <= 0; MDR_enable <= 0;
        end

        store_01_T2: begin
            #10 MDR_select <= 1; IR_enable <= 1;
            #75 MDR_select <= 0; IR_enable <= 0;
        end
```

```
        store_01_T3: begin
            #10 Grb <= 1; ba_select <= 1; Y_enable <= 1;
            #75 Grb <= 0; ba_select <= 0; Y_enable <= 0;
        end

        store_01_T4: begin
            #10 c_select <= 1; alu_instruction <= 5'b00010; Z_enable <= 1;
            #75 c_select <= 0; alu_instruction <= 0; Z_enable <= 0;
        end

        store_01_T5: begin
            #10 Z_LO_select <= 1; MAR_enable <= 1;
            #75 Z_LO_select <= 0; MAR_enable <= 0;
        end

        store_01_T6: begin
            #10 write <= 1; MDR_enable <= 1; Gra <= 1; r_select <= 1;
            #75 write <= 0; MDR_enable <= 0; Gra <= 0; r_select <= 0;
        end

        // Store Instruction 02
        store_02_T0: begin // see if you need to de-assert these signals
            #10 PC_select <= 1; MAR_enable <= 1;
            #75 PC_select <= 0; MAR_enable <= 0;
        end

        store_02_T1: begin
            #10 PC_increment_enable <= 1; read <= 1; MDR_enable <= 1;
            #75 PC_increment_enable <= 0; read <= 0; MDR_enable <= 0;
        end

        store_02_T2: begin
            #10 MDR_select <= 1; IR_enable <= 1;
            #75 MDR_select <= 0; IR_enable <= 0;
        end
```

```
        store_02_T3: begin
            #10 Grb <= 1; r_select <= 1; Y_enable <= 1;
            #75 Grb <= 0; r_select <= 0; Y_enable <= 0;
        end

        store_02_T4: begin
            #10 c_select <= 1; alu_instruction <= 5'b00010; Z_enable <= 1;
            #75 c_select <= 0; alu_instruction <= 5'b00010; Z_enable <= 0;
        end

        store_02_T5: begin
            #10 Z_LO_select <= 1; MAR_enable <= 1;
            #75 Z_LO_select <= 0; MAR_enable <= 0;
        end

        store_02_T6: begin
            #10 Gra <= 1; r_select <= 1; write <= 1; MDR_enable <= 1;
            #75 Gra <= 0; r_select <= 0; write <= 0; MDR_enable <= 0;
        end

    endcase
end
```



Add Immediate Testbench

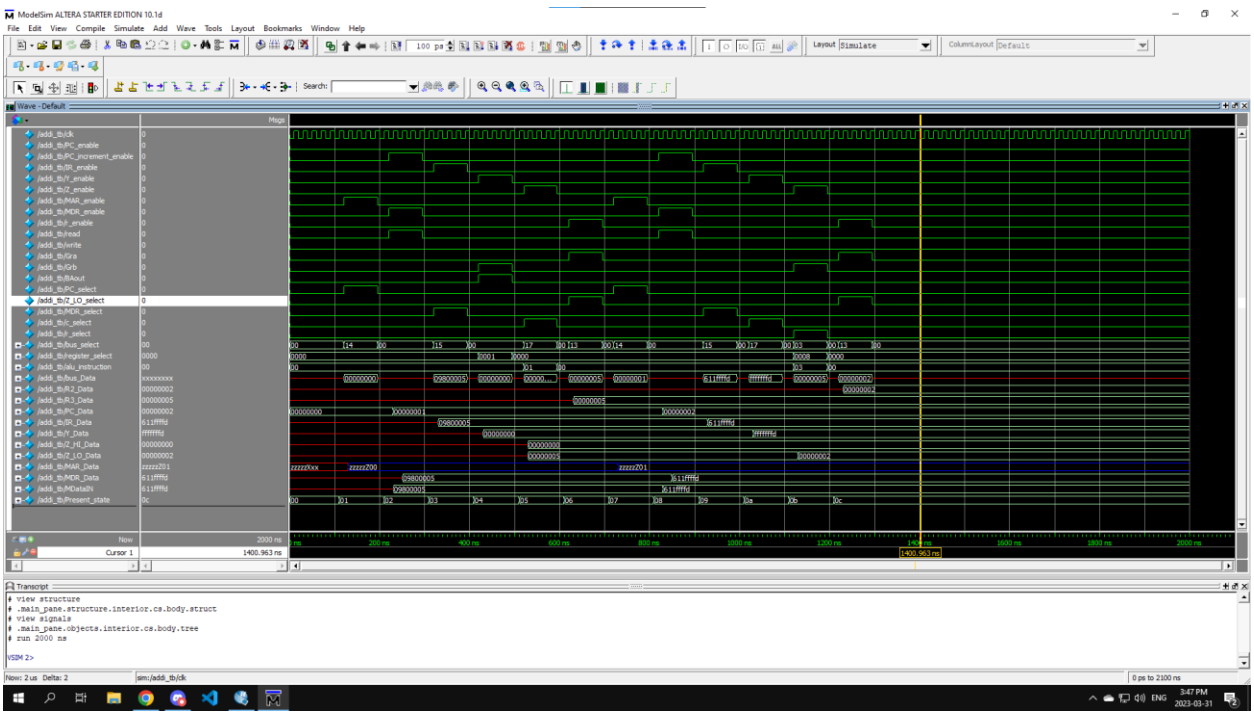
The add immediate (addi) instruction uses the same control sequence as the and immediate (andi) instruction and the or immediate (ori) instruction. The only difference is that each instruction uses the corresponding opcode to tell the ALU to execute one of the following operations: add, and, or.

```
// Time Signals and Load Registers
parameter Default = 0, loadi_01_T0 = 1, loadi_01_T1 = 2, loadi_01_T2 = 3, loadi_01_T3 = 4, loadi_01_T4 = 5,
loadi_01_T5 = 6, addi_01_T0 = 7, addi_01_T1 = 8, addi_01_T2 = 9, addi_01_T3 = 10, addi_01_T4 = 11, addi_01_T5 = 12;

reg [4:0] Present_state = Default;

initial begin clk = 0; Present_state = Default; end
always #10 clk = ~clk;

always @(posedge clk) // finite state machine; if clk rising-edge
    begin
        case (Present_state)
            Default: #100 Present_state = loadi_01_T0;
            loadi_01_T0 : #100 Present_state = loadi_01_T1;
            loadi_01_T1 : #100 Present_state = loadi_01_T2;
            loadi_01_T2 : #100 Present_state = loadi_01_T3;
            loadi_01_T3 : #100 Present_state = loadi_01_T4;
            loadi_01_T4 : #100 Present_state = loadi_01_T5;

            loadi_01_T5 : #100 Present_state = addi_01_T0;
            addi_01_T0 : #100 Present_state = addi_01_T1;
            addi_01_T1 : #100 Present_state = addi_01_T2;
            addi_01_T2 : #100 Present_state = addi_01_T3;
            addi_01_T3 : #100 Present_state = addi_01_T4;
            addi_01_T4 : #100 Present_state = addi_01_T5;

        endcase
    end
```

```
always @(Present_state) // do the required job in each state
    begin
        case (Present_state) // assert the required signals in each clk cycle
            Default: begin
                // Enable Signals
                MDR_enable <= 0; MAR_enable <= 0;
                IR_enable <= 0;
                Y_enable <= 0; Z_enable <= 0;
                PC_enable <= 0; PC_increment_enable <= 0;
                r_enable <= 0;

                // Select Signals
                PC_select <= 0;
                MDR_select <= 0;
                Z_LO_select <= 0; read <= 0;
                write <= 0; c_select <= 0;
                r_select <= 0;

                // Select and Encode Signals
                Gra <= 0; Grb <= 0; ba_select <= 0;

                // Register Contents
                alu_instruction <= 0;
            end
```

```
            loadi_01_T0: begin // see if you need to de-assert these signals
                #10 PC_select <= 1; MAR_enable <= 1;
                #75 PC_select <= 0; MAR_enable <= 0;
            end

            loadi_01_T1: begin
                #10 PC_increment_enable <= 1; read <= 1; MDR_enable <= 1;
                #75 PC_increment_enable <= 0; read <= 0; MDR_enable <= 0;
            end

            loadi_01_T2: begin
                #10 MDR_select <= 1; IR_enable <= 1;
                #75 MDR_select <= 0; IR_enable <= 0;
            end

            loadi_01_T3: begin
                #10 Grb <= 1; ba_select <= 1; Y_enable <= 1;
                #75 Grb <= 0; ba_select <= 0; Y_enable <= 0;
            end

            loadi_01_T4: begin
                #10 c_select <= 1; alu_instruction <= 5'b00001; Z_enable <= 1;
                #75 c_select <= 0; alu_instruction <= 0; Z_enable <= 0;
            end

            loadi_01_T5: begin
                #10 Z_LO_select <= 1; Gra <= 1; r_enable <= 1;
                #75 Z_LO_select <= 0; Gra <= 0; r_enable <= 0;
            end
```

```
        // Add Immediate Instruction
        addi_01_T0: begin // see if you need to de-assert these signals
            #10 PC_select <= 1; MAR_enable <= 1;
            #75 PC_select <= 0; MAR_enable <= 0;
        end

        addi_01_T1: begin
            #10 PC_increment_enable <= 1; read <= 1; MDR_enable <= 1;
            #75 PC_increment_enable <= 0; read <= 0; MDR_enable <= 0;
        end

        addi_01_T2: begin
            #10 MDR_select <= 1; IR_enable <= 1;
            #75 MDR_select <= 0; IR_enable <= 0;
        end

        addi_01_T3: begin
            #10 c_select <= 1; Y_enable <= 1;
            #75 c_select <= 0; Y_enable <= 0;
        end

        addi_01_T4: begin
            #10 Grb <= 1; r_select <= 1; alu_instruction <= 5'b00011; Z_enable <= 1;
            #75 Grb <= 0; r_select <= 0; alu_instruction <= 0; Z_enable <= 0;
        end

        addi_01_T5: begin
            #10 Z_LO_select <= 1; Gra <= 1; r_enable <= 1;
            #75 Z_LO_select <= 0; Gra <= 0; r_enable <= 0;
        end

    endcase
end
```



Branch Testbench

There are 4 different branch instructions, and they all use the same control sequence. The difference between the different branch instructions is the condition required to execute the branch. This report shows the waveforms for **BRNZ**.

Branch Conditions

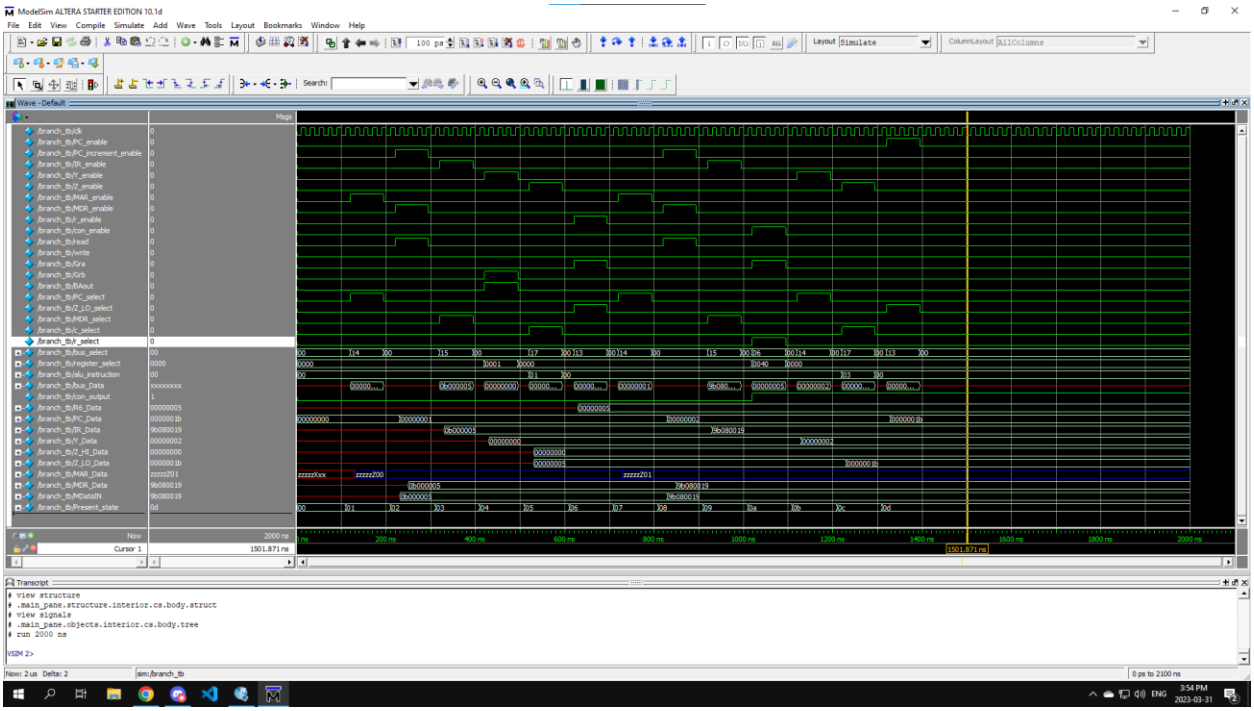
- brzr: Branch if Ra = 0
- brnz: Branch if Ra ≠ 0
- brpl: Branch if Ra = Positive Number
- brim: Branch if Ra = Negative Number

```
// Time Signals and Load Registers
parameter Default = 0, loadi_T0 = 1, loadi_T1 = 2, loadi_T2 = 3, loadi_T3 = 4,
loadi_T4 = 5, loadi_T5 = 6, branch_T0 = 7, branch_T1 = 8, branch_T2 = 9, branch_T3 = 10,
branch_T4 = 11, branch_T5 = 12, branch_T6 = 13;

reg [4:0] Present_state = Default;

initial begin clk = 0; Present_state = Default; end
always #10 clk = ~clk;

always @(posedge clk) // finite state machine; if clk rising-edge
    begin
        case (Present_state)
            Default: #100 Present_state = loadi_T0;
            loadi_T0 : #100 Present_state = loadi_T1;
            loadi_T1 : #100 Present_state = loadi_T2;
            loadi_T2 : #100 Present_state = loadi_T3;
            loadi_T3 : #100 Present_state = loadi_T4;
            loadi_T4 : #100 Present_state = loadi_T5;

            loadi_T5 : #100 Present_state = branch_T0;
            branch_T0 : #100 Present_state = branch_T1;
            branch_T1 : #100 Present_state = branch_T2;
            branch_T2 : #100 Present_state = branch_T3;
            branch_T3 : #100 Present_state = branch_T4;
            branch_T4 : #100 Present_state = branch_T5;
            branch_T5 : #100 Present_state = branch_T6;
        endcase
    end
```

```
always @(Present_state) // do the required job in each state
    begin
        case (Present_state) // assert the required signals in each clk cycle
            Default: begin
                // Enable Signals
                MDR_enable <= 0; MAR_enable <= 0;
                IR_enable <= 0;
                Y_enable <= 0; Z_enable <= 0;
                PC_enable <= 0; PC_increment_enable <= 0;
                r_enable <= 0; con_enable <= 0;

                // Select Signals
                PC_select <= 0;
                MDR_select <= 0;
                Z_LO_select <= 0; read <= 0;
                write <= 0; c_select <= 0;
                r_select <= 0;

                // Select and Encode Signals
                Gra <= 0; Grb <= 0; ba_select <= 0;

                // Register Contents
                alu_instruction <= 0;
            end
```

```
            loadi_T0: begin // see if you need to de-assert these signals
                #10 PC_select <= 1; MAR_enable <= 1;
                #75 PC_select <= 0; MAR_enable <= 0;
            end

            loadi_T1: begin
                #10 PC_increment_enable <= 1; read <= 1; MDR_enable <= 1;
                #75 PC_increment_enable <= 0; read <= 0; MDR_enable <= 0;
            end

            loadi_T2: begin
                #10 MDR_select <= 1; IR_enable <= 1;
                #75 MDR_select <= 0; IR_enable <= 0;
            end

            loadi_T3: begin
                #10 Grb <= 1; ba_select <= 1; Y_enable <= 1;
                #75 Grb <= 0; ba_select <= 0; Y_enable <= 0;
            end

            loadi_T4: begin
                #10 c_select <= 1; alu_instruction <= 5'b00001; Z_enable <= 1;
                #75 c_select <= 0; alu_instruction <= 0; Z_enable <= 0;
            end

            loadi_T5: begin
                #10 Z_LO_select <= 1; Gra <= 1; r_enable <= 1;
                #75 Z_LO_select <= 0; Gra <= 0; r_enable <= 0;
            end
```

```
// Branch Instruction
branch_T0: begin // see if you need to de-assert these signals
    #10 PC_select <= 1; MAR_enable <= 1;
    #75 PC_select <= 0; MAR_enable <= 0;
end

branch_T1: begin
    #10 PC_increment_enable <= 1; read <= 1; MDR_enable <= 1;
    #75 PC_increment_enable <= 0; read <= 0; MDR_enable <= 0;
end

branch_T2: begin
    #10 MDR_select <= 1; IR_enable <= 1;
    #75 MDR_select <= 0; IR_enable <= 0;
end

branch_T3: begin
    #10 Gra <= 1; r_select <= 1; con_enable <= 1;
    #75 Gra <= 0; r_select <= 0; con_enable <= 0;
end

branch_T4: begin
    #10 PC_select <= 1; Y_enable <= 1;
    #75 PC_select <= 0; Y_enable <= 0;
end

branch_T5: begin
    #10 c_select <= 1; alu_instruction <= 5'b00011; Z_enable <= 1;
    #75 c_select <= 0; alu_instruction <= 0; Z_enable <= 0;
end

branch_T6: begin
    #10 Z_LO_select <= 1; PC_enable <= con_output;
    #75 Z_LO_select <= 0; PC_enable <= 0;
end
```



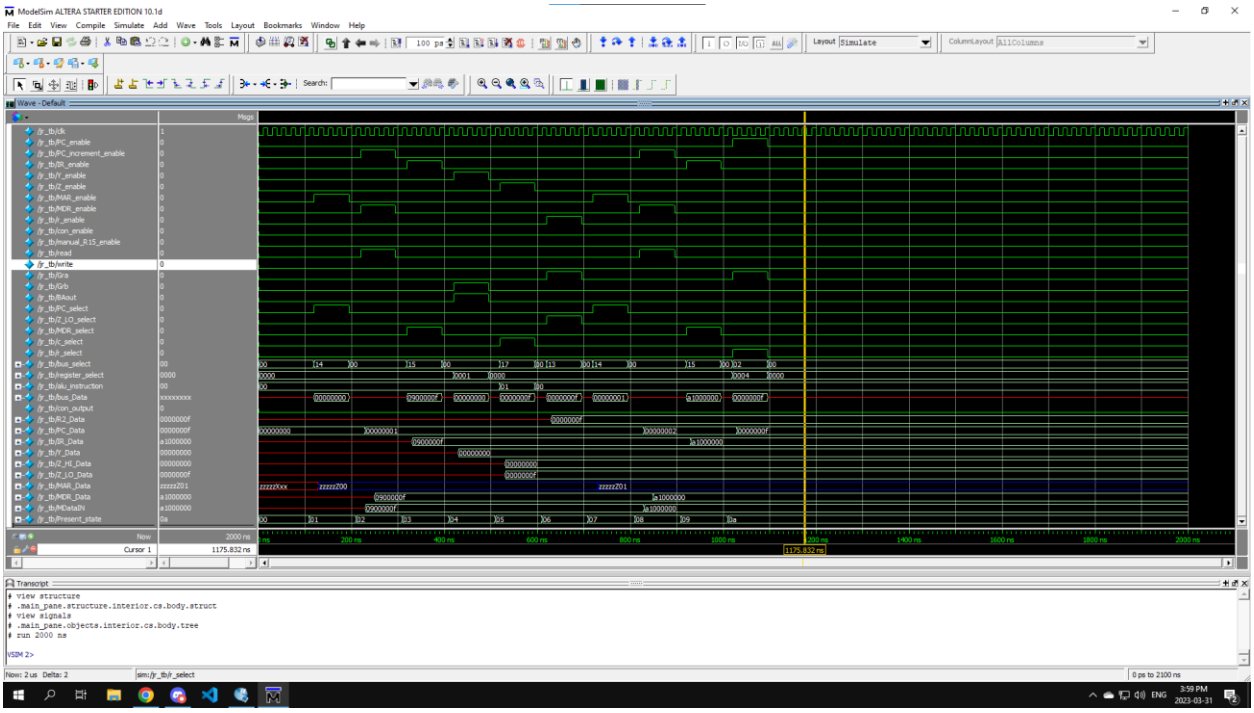
Jr Testbench

```
// Time Signals and Load Registers
parameter Default = 0, loadi_T0 = 1, loadi_T1 = 2, loadi_T2 = 3, loadi_T3 = 4,
loadi_T4 = 5, loadi_T5 = 6, jr_T0 = 7, jr_T1 = 8, jr_T2 = 9, jr_T3 = 10;

reg [4:0] Present_state = Default;

initial begin clk = 0; Present_state = Default; end
always #10 clk = ~clk;

always @(posedge clk) // finite state machine; if clk rising-edge
    begin
        case (Present_state)
            Default: #100 Present_state = loadi_T0;
            loadi_T0 : #100 Present_state = loadi_T1;
            loadi_T1 : #100 Present_state = loadi_T2;
            loadi_T2 : #100 Present_state = loadi_T3;
            loadi_T3 : #100 Present_state = loadi_T4;
            loadi_T4 : #100 Present_state = loadi_T5;

            loadi_T5 : #100 Present_state = jr_T0;
            jr_T0 : #100 Present_state = jr_T1;
            jr_T1 : #100 Present_state = jr_T2;
            jr_T2 : #100 Present_state = jr_T3;

        endcase
    end
```

```
  always @(Present_state) // do the required job in each state
      begin
          case (Present_state) // assert the required signals in each clk cycle
              Default: begin
                  // Enable Signals
                  MDR_enable <= 0; MAR_enable <= 0;
                  IR_enable <= 0;
                  Y_enable <= 0; Z_enable <= 0;
                  PC_enable <= 0; PC_increment_enable <= 0;
                  r_enable <= 0; con_enable <= 0; manual_R15_enable <= 0;

                  // Select Signals
                  PC_select <= 0;
                  MDR_select <= 0;
                  Z_LO_select <= 0; read <= 0;
                  write <= 0; c_select <= 0;
                  r_select <= 0;

                  // Select and Encode Signals
                  Gra <= 0; Grb <= 0; ba_select <= 0;

                  // Register Contents
                  alu_instruction <= 0;
              end
```

```
        loadi_T0: begin // see if you need to de-assert these signals
            #10 PC_select <= 1; MAR_enable <= 1;
            #75 PC_select <= 0; MAR_enable <= 0;
        end

        loadi_T1: begin
            #10 PC_increment_enable <= 1; read <= 1; MDR_enable <= 1;
            #75 PC_increment_enable <= 0; read <= 0; MDR_enable <= 0;
        end

        loadi_T2: begin
            #10 MDR_select <= 1; IR_enable <= 1;
            #75 MDR_select <= 0; IR_enable <= 0;
        end

        loadi_T3: begin
            #10 Grb <= 1; ba_select <= 1; Y_enable <= 1;
            #75 Grb <= 0; ba_select <= 0; Y_enable <= 0;
        end

        loadi_T4: begin
            #10 c_select <= 1; alu_instruction <= 5'b00001; Z_enable <= 1;
            #75 c_select <= 0; alu_instruction <= 0; Z_enable <= 0;
        end

        loadi_T5: begin
            #10 Z_LO_select <= 1; Gra <= 1; r_enable <= 1;
            #75 Z_LO_select <= 0; Gra <= 0; r_enable <= 0;
        end
```

```
        // Branch Instruction
        jr_T0: begin
            #10 PC_select <= 1; MAR_enable <= 1;
            #75 PC_select <= 0; MAR_enable <= 0;
        end

        jr_T1: begin
            #10 PC_increment_enable <= 1; read <= 1; MDR_enable <= 1;
            #75 PC_increment_enable <= 0; read <= 0; MDR_enable <= 0;
        end

        jr_T2: begin
            #10 MDR_select <= 1; IR_enable <= 1;
            #75 MDR_select <= 0; IR_enable <= 0;
        end

        jr_T3: begin
            #10 Gra <= 1; r_select <= 1; PC_enable <= 1;
            #75 Gra <= 0; r_select <= 0; PC_enable <= 0;
        end

    endcase
end
```



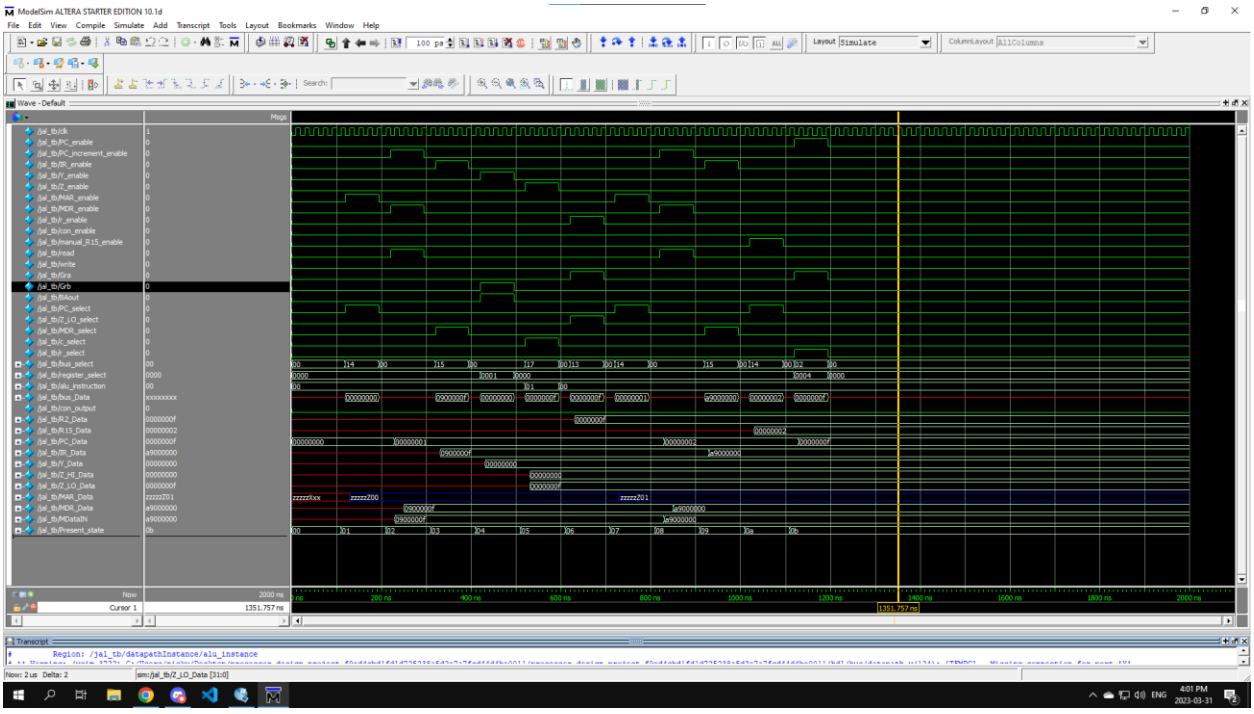
Jal Testbench

```
// Time Signals and Load Registers
parameter Default = 0, loadi_T0 = 1, loadi_T1 = 2, loadi_T2 = 3, loadi_T3 = 4,
loadi_T4 = 5, loadi_T5 = 6, jal_T0 = 7, jal_T1 = 8, jal_T2 = 9, jal_T3 = 10, jal_T4 = 11;

reg [4:0] Present_state = Default;

initial begin clk = 0; Present_state = Default; end
always #10 clk = ~clk;

always @(posedge clk) // finite state machine; if clk rising-edge
    begin
        case (Present_state)
            Default: #100 Present_state = loadi_T0;
            loadi_T0 : #100 Present_state = loadi_T1;
            loadi_T1 : #100 Present_state = loadi_T2;
            loadi_T2 : #100 Present_state = loadi_T3;
            loadi_T3 : #100 Present_state = loadi_T4;
            loadi_T4 : #100 Present_state = loadi_T5;

            loadi_T5 : #100 Present_state = jal_T0;
            jal_T0 : #100 Present_state = jal_T1;
            jal_T1 : #100 Present_state = jal_T2;
            jal_T2 : #100 Present_state = jal_T3;
            jal_T3 : #100 Present_state = jal_T4;
        endcase
    end
```

```
always @(Present_state) // do the required job in each state
    begin
        case (Present_state) // assert the required signals in each clk cycle
            Default: begin
                // Enable Signals
                MDR_enable <= 0; MAR_enable <= 0;
                IR_enable <= 0;
                Y_enable <= 0; Z_enable <= 0;
                PC_enable <= 0; PC_increment_enable <= 0;
                r_enable <= 0; con_enable <= 0; manual_R15_enable <= 0;

                // Select Signals
                PC_select <= 0;
                MDR_select <= 0;
                Z_LO_select <= 0; read <= 0;
                write <= 0; c_select <= 0;
                r_select <= 0;

                // Select and Encode Signals
                Gra <= 0; Grb <= 0; ba_select <= 0;

                // Register Contents
                alu_instruction <= 0;
            end
```

```
            loadi_T0: begin // see if you need to de-assert these signals
                #10 PC_select <= 1; MAR_enable <= 1;
                #75 PC_select <= 0; MAR_enable <= 0;
            end

            loadi_T1: begin
                #10 PC_increment_enable <= 1; read <= 1; MDR_enable <= 1;
                #75 PC_increment_enable <= 0; read <= 0; MDR_enable <= 0;
            end

            loadi_T2: begin
                #10 MDR_select <= 1; IR_enable <= 1;
                #75 MDR_select <= 0; IR_enable <= 0;
            end

            loadi_T3: begin
                #10 Grb <= 1; ba_select <= 1; Y_enable <= 1;
                #75 Grb <= 0; ba_select <= 0; Y_enable <= 0;
            end

            loadi_T4: begin
                #10 c_select <= 1; alu_instruction <= 5'b00001; Z_enable <= 1;
                #75 c_select <= 0; alu_instruction <= 0; Z_enable <= 0;
            end

            loadi_T5: begin
                #10 Z_LO_select <= 1; Gra <= 1; r_enable <= 1;
                #75 Z_LO_select <= 0; Gra <= 0; r_enable <= 0;
            end
```

```
        // Branch Instruction
        jal_T0: begin
            #10 PC_select <= 1; MAR_enable <= 1;
            #75 PC_select <= 0; MAR_enable <= 0;
        end

        jal_T1: begin
            #10 PC_increment_enable <= 1; read <= 1; MDR_enable <= 1;
            #75 PC_increment_enable <= 0; read <= 0; MDR_enable <= 0;
        end

        jal_T2: begin
            #10 MDR_select <= 1; IR_enable <= 1;
            #75 MDR_select <= 0; IR_enable <= 0;
        end

        jal_T3: begin
            #10 manual_R15_enable <= 1; PC_select <= 1;
            #75 manual_R15_enable <= 0; PC_select <= 0;
        end

        jal_T4: begin
            #10 Gra <= 1; r_select <= 1; PC_enable <= 1;
            #75 Gra <= 0; r_select <= 0; PC_enable <= 0;
        end

    endcase
end
```



Move from HI and Move from LO Testbench

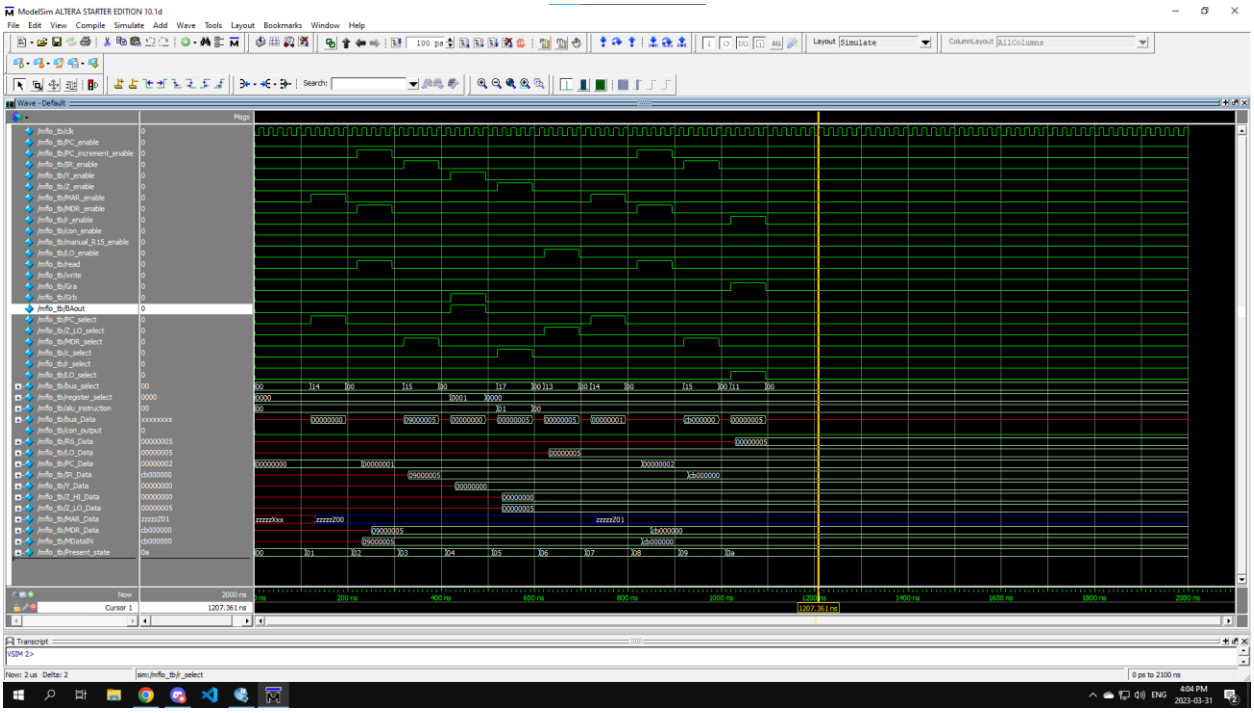
MFHI and MFLO use the same control sequence except MFHI moves a value from the HI register to a general register and MFLO moves a value from the LO register to a general register.

```
// Time Signals and Load Registers
parameter Default = 0, loadi_T0 = 1, loadi_T1 = 2, loadi_T2 = 3, loadi_T3 = 4,
loadi_T4 = 5, loadi_T5 = 6, mflo_T0 = 7, mflo_T1 = 8, mflo_T2 = 9, mflo_T3 = 10;

reg [4:0] Present_state = Default;

initial begin clk = 0; Present_state = Default; end
always #10 clk = ~clk;

always @(posedge clk) // finite state machine; if clk rising-edge
    begin
        case (Present_state)
            Default: #100 Present_state = loadi_T0;
            loadi_T0 : #100 Present_state = loadi_T1;
            loadi_T1 : #100 Present_state = loadi_T2;
            loadi_T2 : #100 Present_state = loadi_T3;
            loadi_T3 : #100 Present_state = loadi_T4;
            loadi_T4 : #100 Present_state = loadi_T5;

            loadi_T5 : #100 Present_state = mflo_T0;
            mflo_T0 : #100 Present_state = mflo_T1;
            mflo_T1 : #100 Present_state = mflo_T2;
            mflo_T2 : #100 Present_state = mflo_T3;

        endcase
    end
```

```
always @(Present_state) // do the required job in each state
    begin
        case (Present_state) // assert the required signals in each clk cycle
            Default: begin
                // Enable Signals
                MDR_enable <= 0; MAR_enable <= 0;
                IR_enable <= 0; LO_enable <= 0;
                Y_enable <= 0; Z_enable <= 0;
                PC_enable <= 0; PC_increment_enable <= 0;
                r_enable <= 0; con_enable <= 0; manual_R15_enable <= 0;
                LO_enable <= 0;

                // Select Signals
                PC_select <= 0;
                MDR_select <= 0;
                Z_LO_select <= 0; read <= 0;
                write <= 0; c_select <= 0;
                r_select <= 0; LO_select <= 0;

                // Select and Encode Signals
                Gra <= 0; Grb <= 0; ba_select <= 0;

                // Register Contents
                alu_instruction <= 0;
            end
```

```
            loadi_T0: begin // see if you need to de-assert these signals
                #10 PC_select <= 1; MAR_enable <= 1;
                #75 PC_select <= 0; MAR_enable <= 0;
            end

            loadi_T1: begin
                #10 PC_increment_enable <= 1; read <= 1; MDR_enable <= 1;
                #75 PC_increment_enable <= 0; read <= 0; MDR_enable <= 0;
            end

            loadi_T2: begin
                #10 MDR_select <= 1; IR_enable <= 1;
                #75 MDR_select <= 0; IR_enable <= 0;
            end

            loadi_T3: begin
                #10 Grb <= 1; ba_select <= 1; Y_enable <= 1;
                #75 Grb <= 0; ba_select <= 0; Y_enable <= 0;
            end

            loadi_T4: begin
                #10 c_select <= 1; alu_instruction <= 5'b00001; Z_enable <= 1;
                #75 c_select <= 0; alu_instruction <= 0; Z_enable <= 0;
            end

            loadi_T5: begin
                #10 Z_LO_select <= 1; LO_enable <= 1;
                #75 Z_LO_select <= 0; LO_enable <= 0;
            end
```

```
            // mflo Instruction
            mflo_T0: begin
                #10 PC_select <= 1; MAR_enable <= 1;
                #75 PC_select <= 0; MAR_enable <= 0;
            end

            mflo_T1: begin
                #10 PC_increment_enable <= 1; read <= 1; MDR_enable <= 1;
                #75 PC_increment_enable <= 0; read <= 0; MDR_enable <= 0;
            end

            mflo_T2: begin
                #10 MDR_select <= 1; IR_enable <= 1;
                #75 MDR_select <= 0; IR_enable <= 0;
            end

            mflo_T3: begin
                #10 Gra <= 1; r_enable <= 1; LO_select <= 1;
                #75 Gra <= 0; r_enable <= 0; LO_select <= 0;
            end

        endcase
    end
```



Inport and Outport Testbench

The inport and outport instructions use the same control sequence. The difference is that the inport instruction takes a value as input from outside the processor and stores it in a general register and the outport instruction takes an internal value and sends it out as an output.

```
// Time Signals and Load Registers
parameter Default = 0, loadi_01_T0 = 1, loadi_01_T1 = 2, loadi_01_T2 = 3, loadi_01_T3 = 4, loadi_01_T4 = 5,
loadi_01_T5 = 6, out_T0 = 7, out_T1 = 8, out_T2 = 9, out_T3 = 10;

reg [4:0] Present_state = Default;

initial begin clk = 0; Present_state = Default; end
always #10 clk = ~clk;

always @(posedge clk) // finite state machine; if clk rising-edge
    begin
        case (Present_state)
            Default: #100 Present_state = loadi_01_T0;
            loadi_01_T0 : #100 Present_state = loadi_01_T1;
            loadi_01_T1 : #100 Present_state = loadi_01_T2;
            loadi_01_T2 : #100 Present_state = loadi_01_T3;
            loadi_01_T3 : #100 Present_state = loadi_01_T4;
            loadi_01_T4 : #100 Present_state = loadi_01_T5;

            loadi_01_T5: #100 Present_state = out_T0;
            out_T0 : #100 Present_state = out_T1;
            out_T1 : #100 Present_state = out_T2;
            out_T2 : #100 Present_state = out_T3;
        endcase
    end
```

```
always @(Present_state) // do the required job in each state
    begin
        case (Present_state) // assert the required signals in each clk cycle
            Default: begin
                // Enable Signals
                MDR_enable <= 0; MAR_enable <= 0;
                IR_enable <= 0;
                Y_enable <= 0; Z_enable <= 0;
                PC_enable <= 0; PC_increment_enable <= 0;
                r_enable <= 0; con_enable <= 0; manual_R15_enable <= 0;
                outport_enable <= 0;

                // Select Signals
                PC_select <= 0;
                MDR_select <= 0;
                Z_LO_select <= 0; read <= 0;
                write <= 0; c_select <= 0;
                r_select <= 0; inport_select <= 0;

                // Select and Encode Signals
                Gra <= 0; Grb <= 0; ba_select <= 0;

                // Register Contents
                alu_instruction <= 0;
            end
```

```
            loadi_01_T0: begin // see if you need to de-assert these signals
                #10 PC_select <= 1; MAR_enable <= 1;
                #75 PC_select <= 0; MAR_enable <= 0;
            end

            loadi_01_T1: begin
                #10 PC_increment_enable <= 1; read <= 1; MDR_enable <= 1;
                #75 PC_increment_enable <= 0; read <= 0; MDR_enable <= 0;
            end

            loadi_01_T2: begin
                #10 MDR_select <= 1; IR_enable <= 1;
                #75 MDR_select <= 0; IR_enable <= 0;
            end

            loadi_01_T3: begin
                #10 Grb <= 1; ba_select <= 1; Y_enable <= 1;
                #75 Grb <= 0; ba_select <= 0; Y_enable <= 0;
            end

            loadi_01_T4: begin
                #10 c_select <= 1; alu_instruction <= 5'b00001; Z_enable <= 1;
                #75 c_select <= 0; alu_instruction <= 0; Z_enable <= 0;
            end

            loadi_01_T5: begin
                #10 Z_LO_select <= 1; Gra <= 1; r_enable <= 1;
                #75 Z_LO_select <= 0; Gra <= 0; r_enable <= 0;
            end
```

```
        // out Instruction
        out_T0: begin
            #10 PC_select <= 1; MAR_enable <= 1;
            #75 PC_select <= 0; MAR_enable <= 0;
        end

        out_T1: begin
            #10 PC_increment_enable <= 1; read <= 1; MDR_enable <= 1;
            #75 PC_increment_enable <= 0; read <= 0; MDR_enable <= 0;
        end

        out_T2: begin
            #10 MDR_select <= 1; IR_enable <= 1;
            #75 MDR_select <= 0; IR_enable <= 0;
        end

        out_T3: begin
            #10 Gra <= 1; r_select <= 1; outport_enable <= 1;
            #75 Gra <= 0; r_select <= 0; outport_enable <= 0;
        end

    endcase
end
```