ELEC 374 – Phase 3 Report

Department of Electrical and Computer Engineering
Queen's University

Group 16

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

Date of Submission:
31 March 2023

"We do hereby verify that this written lab report is our own work and contains our own original ideas, concepts, and designs. No portion of this report has been copied in whole or in part from another source, with the possible exception of properly referenced material".

Table of Contents

Registers

General Register

hdl > registers > register.v

```
/* Representation of a register in Verilog HDL. */
/* Declared 3, 1-bit signals, 1, 32-bit D-input, and 1, 32-bit Q-output. */
/* Data type of each signal/input is a wire and output is a reg. */
module register(input clk, input clr, input enable, input [31:0] D, 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 = D;
    end
endmodule // Register end.
```

Memory Data Register

hdl > registers > md_register.v

```
/* 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.
```

Revised Register 0

hdl > registers > R0_revised.v

```
/* Representation of a register in Verilog HDL. */
/* Declared 3, 1-bit signals, 1, 32-bit D-input, and 1, 32-bit Q-output. */
/* Data type of each signal/input is a wire and output is a reg. */
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.
```

Program Counter

hdl > registers > program_counter.v

```
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
```

Arithmetic Logic Unit

hdl > alu > alu.v

```
module alu (
    // ALU Inputs
    input wire [31:0] A, B, Y,    //added Y
    input wire [4:0] opcode,

    // ALU Output
    output reg [63:0] result);

    parameter load = 5'b00000, loadImm = 5'b00001, store = 5'b00010, add = 5'b00011, sub = 5'b00100,
    AND = 5'b00101, OR = 5'b00110, right_shift = 5'b00111, arith_shift_right = 5'b01000, left_shift = 5'b01001,
    right_rotate = 5'b01010, left_rotate = 5'b01011, addImm = 5'b01100, AND_Imm = 5'b01101, OR_Imm = 5'b01110,
    mul = 5'b01111, div = 5'b10000, negate = 5'b10001, NOT = 5'b10010, branch = 5'b10011, jr = 5'b10100, jal = 5'b10101,
    in = 5'b10110, out = 5'b10111, mfhi = 5'b11000, mflo = 5'b11001;

    // And Operation
    wire [31:0] and_result;
    logical_and andInstance(.A(A), .B(B), .result(and_result));

    // Or Operation
    wire [31:0] or_result;
    logical_or orInstance(A, B, or_result);

    // Add Operation
    wire [31:0] sum_result;
    wire carryOut;
    adder adderInstance(A, B, sum_result, carryOut);

    // Subtractor Operation
    wire [31:0] difference_result;
    wire borrowOut;
    subtractor subtractorInstance(A, B, difference_result, borrowOut);

    // Multiplication Operation
    wire [63:0] product_result;
    multiplier multiplierInstance(A, B, product_result);

    // Division Operation
    wire [31:0] quotient_result, remainder_result;
    divider dividerInstance(A, B, quotient_result, remainder_result);

    // Left Shift Operation
    wire [31:0] leftShift_result;
    shift_left leftShiftInstance(A, B, leftShift_result);

    // Right Shift Operation
    wire [31:0] rightShift_result;
    shift_right rightShiftInstance(A, B, rightShift_result);

```

```
// Arithmetic Right Shift Operation
wire [31:0] arithShiftRight_result;
arithmetic_shift_right arithShiftRightInstance(A, B, arithShiftRight_result);

// Left Rotate Operation
wire [31:0] leftRotate_result;
rotate_left leftRotateInstance(A, B, leftRotate_result);

// Right Rotate Operation
wire [31:0] rightRotate_result;
rotate_right rightRotateInstance(A, B, rightRotate_result);

// Negate Operation
wire [31:0] negate_result;
negate negateInstance(B, negate_result);

// Not Operation
wire [31:0] not_result;
logical_not notInstance(B, not_result);

```

```
// Select operation
always @(*) begin
    case(opcode)
        // Add Operation
        add : begin
            result[31:0] <= sum_result[31:0];
            result[63:32] <= 32'd0;
        end

        // Subtract Operation
        sub : begin
            result[31:0] <= difference_result[31:0];
            result[63:32] <= 32'd0;
        end

        // And Operation
        AND, AND_Imm : begin
            result[31:0] <= and_result[31:0];
            result[63:32] <= 32'd0;
        end

        // Or Operation
        OR, OR_Imm : begin
            result[31:0] <= or_result[31:0];
            result[63:32] <= 32'd0;
        end

        // Right Shift Operation
        right_shift : begin
            result[31:0] <= rightShift_result[31:0];
            result[63:32] <= 32'd0;
        end

        // Arithmetic Right Shift Operation
        arith_shift_right : begin
            result[31:0] <= arithShiftRight_result[31:0];
            result[63:32] <= 32'd0;
        end

        // Left Shift Operation
        left_shift : begin
            result[31:0] <= leftShift_result[31:0];
            result[63:32] <= 32'd0;
        end

```

```
            // Right Rotate Operation
            right_rotate : begin
                result[31:0] <= rightRotate_result[31:0];
                result[63:32] <= 32'd0;
            end

            // Left Rotate Operation
            left_rotate : begin
                result[31:0] <= leftRotate_result[31:0];
                result[63:32] <= 32'd0;
            end

            // Multiply Operation
            mul : begin
                result[63:0] <= product_result[63:0];
            end

            // Divide Operation
            div : begin
                result[31:0] <= quotient_result[31:0];
                result[63:32] <= remainder_result[31:0];
            end

            // Negate Operation
            negate : begin
                result[31:0] <= negate_result[31:0];
                result[63:32] <= 32'd0;
            end

            // Not Operation
            NOT : begin
                result[31:0] <= not_result[31:0];
                result[63:32] <= 32'd0;
            end

            // Default Case
            default : begin
                result[63:0] <= 64'd0;
            end
        endcase
    end
endmodule
```

Bus System

Datapath

hdl > bus > datapath.v

```
/* Representation of the datapath in Verilog HDL. */
module datapath(
    // CPU signals
    input wire clk, clr,
    input wire [31:0] input_Data,
    output wire [31:0] outport_Data
    );

    /* Enable Signals */
    // General Register Enable Signals
    wire R15_enable;
    wire manual_R15_enable;
    wire [15:0] register_enable;

    // Program Counter Enable Signals
    wire PC_enable, PC_increment_enable;

    // Instruction Register Enable Signal
    wire IR_enable;

    // CON Flip Flop Enable Signal
    wire con_enable;

    // ALU 'Y' Register Enable Signal
    wire Y_enable;

    // Z_HI and Z_LO Enable Signal
    wire Z_enable;

    // HI and LO Register Enable Signals
    wire HI_enable, LO_enable;

    // Memory Register Enable Signals
    wire MAR_enable, MDR_enable, read, write;

    // Outport Register Enable Signal
    wire outport_enable;

    /* Select Signals */
    // General Register Select Signals
    wire [15:0] register_select;

    // 32-to-5 Encoder Output
    wire [4:0] bus_select;

    // PC Register Select Signal
    wire PC_select;

```

```
// HI/LO Register Select Signal
wire HI_select, LO_select;

// Z_HI/Z_LO Register Select Signal
wire Z_HI_select, Z_LO_select;

// MDR Select Signal
wire MDR_select;

// Inport Select Signal
wire inport_select;

// C_Sign_Extended Select Signal
wire c_select;

/* Data Signals */
wire [31:0] bus_Data; // Data currently in the bus

wire [63:0] aluResult;
wire [4:0] alu_instruction; // ALU Opcode

// General Register Contents
wire [31:0] R0_Data, R1_Data, R2_Data, R3_Data, R4_Data, R5_Data, R6_Data, R7_Data,
R8_Data, R9_Data, R10_Data, R11_Data, R12_Data, R13_Data, R14_Data, R15_Data;

// Instruction Register and Program Counter Contents
wire [31:0] PC_Data, IR_Data;

// ALU Input Register 'Y' Contents
wire [31:0] Y_Data;

// ALU Output Register Contents
wire [31:0] Z_HI_Data, Z_LO_Data;
wire [31:0] HI_Data, LO_Data;

// RAM Memory Address Register (MAR) Contents
wire [8:0] MAR_Data;

// RAM Memory Data Register (MDR) Contents
wire [31:0] MDR_Data, MDataIN;

// C Sign Extended Data
wire [31:0] C_sign_ext_Data;

// Port Register Contents
wire [31:0] inport_Data;

```

hdl > bus > datapath.v

```
    /* Select and Encode Signals */
    wire Gra, Grb, Grc, r_enable, r_select, ba_select;

    /* CON FF Output */
    wire con_output;

    /* 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]), .ba_select(ba_select), .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));

    assign R15_enable = manual_R15_enable | register_enable[15];
    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),
    .ba_select(ba_select), .C_sign_ext_Data(C_sign_ext_Data), .register_enable(register_enable), .register_select(register_select));

    con_ff conInstance(.bus_Data(bus_Data), .instruction(IR_Data), .con_enable(con_enable), .con_output(con_output));

    inport inportInstance(.clk(clk), .clr(clr), .input_Data(input_Data), .inport_Data(inport_Data));
    outport outportInstance(.clk(clk), .clr(clr), .enable(outport_enable), .bus_Data(bus_Data), .outport_Data(outport_Data));

    control_unit cuInstance(
        clk, reset, con_output, IR_Data, alu_instruction, read, write, Gra, Grb, Grc, r_enable, r_select, ba_select,
        PC_enable, PC_increment_enable, IR_enable, con_enable, Y_enable, Z_enable, HI_enable, LO_enable,
        MAR_enable, MDR_enable, outport_enable, manual_R15_enable,
        MDR_select, Z_HI_select, Z_LO_select, HI_select, LO_select, PC_select, inport_select, c_select);

endmodule // Datapath end.
```

Select Encode Logic

hdl > bus > select_encode_logic > select_encode_logic.v

```
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
```

Encoder

hdl > bus > encoder.v

```
/* Representation of an encoder in Verilog HDL. */
/* Declared 32, 1-bit inputs and 5 output select signals. */
/* Data type of each input is a wire. */
module encoder(input [15:0] register_select, input HI_select, input LO_select, input Z_HI_select, input Z_LO_select,
input PC_select, input MDR_select, input inport_select, input c_select, output reg [4:0] selectSignal);

    /* While loop to update the selectSignal output wire. */
    always @* begin
        if (register_select[0]) selectSignal <= 5'b00000;
        else if (register_select[1]) selectSignal <= 5'b00001;
        else if (register_select[2]) selectSignal <= 5'b00010;
        else if (register_select[3]) selectSignal <= 5'b00011;
        else if (register_select[4]) selectSignal <= 5'b00100;
        else if (register_select[5]) selectSignal <= 5'b00101;
        else if (register_select[6]) selectSignal <= 5'b00110;
        else if (register_select[7]) selectSignal <= 5'b00111;
        else if (register_select[8]) selectSignal <= 5'b01000;
        else if (register_select[9]) selectSignal <= 5'b01001;
        else if (register_select[10]) selectSignal <= 5'b01010;
        else if (register_select[11]) selectSignal <= 5'b01011;
        else if (register_select[12]) selectSignal <= 5'b01100;
        else if (register_select[13]) selectSignal <= 5'b01101;
        else if (register_select[14]) selectSignal <= 5'b01110;
        else if (register_select[15]) selectSignal <= 5'b01111;
        else if (HI_select) selectSignal <= 5'b10000;
        else if (LO_select) selectSignal <= 5'b10001;
        else if (Z_HI_select) selectSignal <= 5'b10010;
        else if (Z_LO_select) selectSignal <= 5'b10011;
        else if (PC_select) selectSignal <= 5'b10100;
        else if (MDR_select) selectSignal <= 5'b10101;
        else if (inport_select) selectSignal <= 5'b10110;
        else if (c_select) selectSignal <= 5'b10111;
        else selectSignal <= 5'b00000;  // optional, to avoid latch.
    end
endmodule // Encoder end.
```

Multiplexer

hdl > bus > multiplexer.v

```
/* Representation of a multiplexer in Verilog HDL. */
/* Declared 32, 32-bit inputs, 5 select signals, and  1, 32-bit output. */
/* Data type of each input/select signal is a wire and output is a reg. */

module multiplexer(input [4:0] selectSignal, input [31:0] muxIN_r0,
input [31:0] muxIN_r1, input [31:0] muxIN_r2, input [31:0] muxIN_r3,
input [31:0] muxIN_r4, input [31:0] muxIN_r5, input [31:0] muxIN_r6,
input [31:0] muxIN_r7, input [31:0] muxIN_r8, input [31:0] muxIN_r9,
input [31:0] muxIN_r10, input [31:0] muxIN_r11, input [31:0] muxIN_r12,
input [31:0] muxIN_r13, input [31:0] muxIN_r14, input [31:0] muxIN_r15,
input [31:0] muxIN_HI, input [31:0] muxIN_LO, input [31:0] muxIN_Z_HI,
input [31:0] muxIN_Z_LO, input [31:0] muxIN_PC, input [31:0] muxIN_MDR,
input [31:0] muxIN_inport, input [31:0] muxIN_C_sign_ext, output reg [31:0] muxOut);

    /* While loop to check for updates in the select signals, which updates the mux output. */
    always @(*) begin
            case (selectSignal)
                5'b00000: muxOut <= muxIN_r0;
                5'b00001: muxOut <= muxIN_r1;
                5'b00010: muxOut <= muxIN_r2;
                5'b00011: muxOut <= muxIN_r3;
                5'b00100: muxOut <= muxIN_r4;
                5'b00101: muxOut <= muxIN_r5;
                5'b00110: muxOut <= muxIN_r6;
                5'b00111: muxOut <= muxIN_r7;
                5'b01000: muxOut <= muxIN_r8;
                5'b01001: muxOut <= muxIN_r9;
                5'b01010: muxOut <= muxIN_r10;
                5'b01011: muxOut <= muxIN_r11;
                5'b01100: muxOut <= muxIN_r12;
                5'b01101: muxOut <= muxIN_r13;
                5'b01110: muxOut <= muxIN_r14;
                5'b01111: muxOut <= muxIN_r15;
                5'b10000: muxOut <= muxIN_HI;
                5'b10001: muxOut <= muxIN_LO;
                5'b10010: muxOut <= muxIN_Z_HI;
                5'b10011: muxOut <= muxIN_Z_LO;
                5'b10100: muxOut <= muxIN_PC;
                5'b10101: muxOut <= muxIN_MDR;
                5'b10110: muxOut <= muxIN_inport;
                5'b10111: muxOut <= muxIN_C_sign_ext;
                default: muxOut <= 32'd0;
            endcase
        end
endmodule // Multiplexer end.
```

Con Flip Flop

hdl > con_ff > con_ff.v

```
/* This module determines whether the correct condition has been met to cause branching to take place in a conditional branch instruction. */
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
```

Control Unit

hdl > control_unit > control_unit.v

```
`timescale 1ns/10ps
module control_unit(
    // Control Unit Inputs
    input wire clk, reset, con_output,
    input wire [31:0] IR_Data,

    // ALU Opcode
    output reg [4:0] alu_instruction,

    // RAM Read/Write Signals
    output reg read, write,

    // Select Encode Logic Signals
    output reg Gra, Grb, Grc, r_enable, r_select, ba_select,

    // Control Unit Enable Signal Outputs
    output reg PC_enable, PC_increment_enable, IR_enable, con_enable, Y_enable, Z_enable, HI_enable, LO_enable,
    MAR_enable, MDR_enable, outport_enable, manual_R15_enable,

    // Control Unit Select Signal Outputs
    output reg MDR_select, Z_HI_select, Z_LO_select, HI_select, LO_select, PC_select, inport_select, c_select);

parameter
// Initial State
reset_state = 8'b00000000,

// Instruction Fetch
fetch0 = 8'b00000001, fetch1 = 8'b00000010, fetch2= 8'b00000011,

// Add Instruction
add3 = 8'b00000100, add4= 8'b00000101, add5= 8'b00000110,

// Sub Instruction
sub3 = 8'b00000111, sub4 = 8'b00001000, sub5 = 8'b00001001,

// Multiply Instruction
mul3 = 8'b00001010, mul4 = 8'b00001011, mul5 = 8'b00001100, mul6 = 8'b00001101,

// Divide Instruction
div3 = 8'b00001110, div4 = 8'b00001111, div5 = 8'b00010000, div6 = 8'b00010001,

// Or Instruction
or3 = 8'b00010010, or4 = 8'b00010011, or5 = 8'b00010100, and3 = 8'b00010101,

// And Instruction
and4 = 8'b00010110, and5 = 8'b00010111,

```

hdl > control_unit > control_unit.v

```
// Shift Left Instruction
shl3 = 8'b00011000, shl4 = 8'b00011001, shl5 = 8'b00011010,

// Shift Right Instruction
shr3 = 8'b00011011, shr4 = 8'b00011100, shr5 = 8'b00011101,

// Rotate Left Instruction
rol3 = 8'b00011110, rol4 = 8'b00011111, rol5 = 8'b00100000,

// Rotate Right Instruction
ror3 = 8'b00100001, ror4 = 8'b00100010, ror5 = 8'b00100011,

// Negate Instruction
neg3 = 8'b00100100, neg4 = 8'b00100101, neg5 = 8'b00100110,

// Not Instruction
not3 = 8'b00100111, not4 = 8'b00101000, not5 = 8'b00101001,

// Load Instruction
ld3 = 8'b00101010, ld4 = 8'b00101011, ld5 = 8'b00101100, ld6 = 8'b00101101, ld7 = 8'b00101110,

// Load Immediate Instruction
ldi3 = 8'b00101111, ldi4 = 8'b00110000, ldi5 = 8'b00110001,

// Store Instruction
st3 = 8'b00110010, st4 = 8'b00110011, st5 = 8'b00110100, st6 = 8'b00110101,

// Add Immediate Instruction
addi3 = 8'b00110111, addi4 = 8'b00111000, addi5 = 8'b00111001,

// And Immediate Instruction
andi3 = 8'b00111010, andi4 = 8'b00111011, andi5 = 8'b00111100,

// Or Immediate Instruction
ori3 = 8'b00111101, ori4 = 8'b00111110, ori5 = 8'b00111111,

// Branch Instruction
br3 = 8'b01000000, br4 = 8'b01000001, br5 = 8'b01000010, br6 = 8'b01000011,

// Jump Instructions
jr3 = 8'b01000100,

jal3 = 8'b01000101, jal4 = 8'b01000110,

// Move from LO/HI Instruction
mfhi3 = 8'b01000111, mflo3 = 8'b01001000,
```

hdl > control_unit > control_unit.v

```
// In/Out Port Instruction
in3 = 8'b01001001, out3 = 8'b01001010,

// No Instruction
nop3 = 8'b01001011, halt3 = 8'b01001100,

// Shift Right Arithmetic Instruction
shra3 = 8'b01001101, shra4 = 8'b01001110, shra5 = 8'b01001111;

parameter ld = 5'b00000, ldi = 5'b00001, st = 5'b00010,
add = 5'b00011, sub = 5'b00100, AND = 5'b00101, OR = 5'b00110,
shr = 5'b00111, shra = 5'b01000, shl = 5'b01001, ror = 5'b01010, rol = 5'b01011,
mul = 5'b01111, div = 5'b10000, neg = 5'b10001, NOT = 5'b10010,
addi = 5'b01100, andi = 5'b01101, ori = 5'b01110,
br = 5'b10011, jr = 5'b10100, jal = 5'b10101,
in = 5'b10110, out = 5'b10111, mfhi = 5'b11000, mflo = 5'b11001,
nop = 5'b11010, halt = 5'b11011;

reg [7:0] present_state = reset_state;      // adjust the bit pattern based on the number of states

always @(posedge clk, posedge reset)  //con_ff finite state machine; if clk or reset rising-edge
begin
    if (reset == 1'b1) present_state = reset_state;
    else case (present_state)
        reset_state : present_state = fetch0;
        fetch0 : present_state = fetch1;
        fetch1 : present_state = fetch2;
        fetch2 : begin
        case (IR_Data[31:27]) // inst. decoding based on the opcode to set the next state
            5'b00011 : present_state = add3;
            5'b00100 : present_state = sub3;
            5'b01111 : present_state = mul3;
            5'b10000 : present_state = div3;
            5'b00111 : present_state = shr3;
                5'b01000 : present_state = shra3;
            5'b01001 : present_state = shl3;
            5'b01010 : present_state = ror3;
            5'b01011 : present_state = rol3;
            5'b00101 : present_state = and3;
            5'b00110 : present_state = or3;
            5'b10001 : present_state = neg3;
            5'b10010 : present_state = not3;
            5'b00000 : present_state = ld3;
            5'b00001 : present_state = ldi3;
            5'b00010 : present_state = st3;
            5'b01100 : present_state = addi3;
            5'b01100 : present_state = andi3;
            5'b01110 : present_state = ori3;
            5'b10011 : present_state = br3;
```

hdl > control_unit > control_unit.v

```
            5'b10100 : present_state = jr3;
            5'b10101 : present_state = jal3;
            5'b11000 : present_state = mfhi3;
            5'b11001 : present_state = mflo3;
            5'b10110 : present_state = in3;
            5'b10111 : present_state = out3;
            5'b11010 : present_state = nop3;
            5'b11011 : present_state = halt3;
        endcase
        end

        add3 : present_state = add4;
        add4 : present_state = add5;
        add5 : present_state = fetch0;

        addi3 : present_state = addi4;
        addi4 : present_state = addi5;
        addi5 : present_state = fetch0;

        sub3 : present_state = sub4;
        sub4 : present_state = sub5;
        sub5 : present_state = fetch0;

        mul3 : present_state = mul4;
        mul4 : present_state = mul5;
        mul5 : present_state = mul6;
        mul6 : present_state = fetch0;

        div3 : present_state = div4;
        div4 : present_state = div5;
        div5 : present_state = div6;
        div6 : present_state = fetch0;

        or3 : present_state = or4;
        or4 : present_state = or5;
        or5 : present_state = fetch0;

        and3 : present_state = and4;
        and4 : present_state = and5;
        and5 : present_state = fetch0;

        shl3 : present_state = shl4;
        shl4 : present_state = shl5;
        shl5 : present_state = fetch0;

        shr3 : present_state = shr4;
        shr4 : present_state = shr5;
        shr5 : present_state = fetch0;
```

hdl > control_unit > control_unit.v

```
            shra3 : present_state = shra4;
        shra4 : present_state = shra5;
        shra5 : present_state = fetch0;

        rol3 : present_state = rol4;
        rol4 : present_state = rol5;
        rol5 : present_state = fetch0;

        ror3 : present_state = ror4;
        ror4 : present_state = ror5;
        ror5 : present_state = fetch0;

        neg3 : present_state = neg4;
        neg4 : present_state = fetch0;

        not3 : present_state = not4;
        not4 : present_state = fetch0;

        ld3 : present_state = ld4;
        ld4 : present_state = ld5;
        ld5 : present_state = ld6;
        ld6 : present_state = ld7;
        ld7 : present_state = fetch0;

        ldi3 : present_state = ldi4;
        ldi4 : present_state = ldi5;
        ldi5 : present_state = fetch0;

        st3 : present_state = st4;
        st4 : present_state = st5;
        st5 : present_state = st6;
        st6 : present_state = fetch0;

        andi3 : present_state = andi4;
        andi4 : present_state = andi5;
        andi5 : present_state = fetch0;

        ori3 : present_state = ori4;
        ori4 : present_state = ori5;
        ori5 : present_state = fetch0;

        jal3 : present_state = jal4;
        jal4 : present_state = fetch0;

        jr3 : present_state = fetch0;

```

hdl > control_unit > control_unit.v

```
        br3 : present_state = br4;
        br4 : present_state = br5;
        br5 : present_state = br6;
        br6 : present_state = fetch0;

        out3 : present_state = fetch0;

        in3 : present_state = fetch0;

        mflo3 : present_state = fetch0;

        mfhi3 : present_state = fetch0;

        nop3 : present_state = fetch0;
    endcase
end

always @(present_state) // do the job for each state
begin
    case (present_state) // assert the required signals in each state
    reset_state: begin
    // RAM Read/Write Signals
    read <= 0; write <= 0;

    // Select Encode Logic Signals
    Gra <= 0; Grb <= 0; Grc <= 0; r_enable <= 0; r_select <= 0; ba_select <= 0;

    // Control Unit Enable Signal Outputs
    PC_enable <= 0; PC_increment_enable <= 0; IR_enable <= 0; con_enable <= 0; Y_enable <= 0;
    Z_enable <= 0; HI_enable <= 0; LO_enable <= 0; MAR_enable <= 0; MDR_enable <= 0; outport_enable <= 0;
    manual_R15_enable <= 0;

    // Control Unit Select Signal Outputs
    MDR_select <= 0; Z_HI_select <= 0; Z_LO_select <= 0; HI_select <= 0; LO_select <= 0; PC_select <= 0;
    inport_select <= 0; c_select <= 0;

    // ALU Instruction
    alu_instruction <= 0;

    end

    fetch0: begin
        #10 PC_select <= 1; MAR_enable <= 1;
        #75 PC_select <= 0; MAR_enable <= 0;
    end

```

hdl > control_unit > control_unit.v

```
    fetch1: begin
        #10 PC_increment_enable <= 1; read <= 1; MDR_enable <= 1;
        #75 PC_increment_enable <= 0; read <= 0; MDR_enable <= 0;
    end

    fetch2: begin
        #10 MDR_select <= 1; IR_enable <= 1;
        #75 MDR_select <= 0; IR_enable <= 0;
    end

    // Addition Operation
    add3: begin
        #10 Grb <= 1; r_select <= 1; Y_enable <= 1;
        #75 Grb <= 0; r_select <= 0; Y_enable <= 0;
    end

    add4: begin
        #10 Grc <= 1; r_select <= 1; alu_instruction <= add; Z_enable <= 1;
        #75 Grc <= 0; r_select <= 0; alu_instruction <= 0; Z_enable <= 0;
    end

    add5: begin
        #10 Z_LO_select <= 1; Gra <= 1; r_enable <= 1;
        #75 Z_LO_select <= 0; Gra <= 0; r_enable <= 0;
    end

    // Subtraction Operation
    sub3: begin
        #10 Grb <= 1; r_select <= 1; Y_enable <= 1;
        #75 Grb <= 0; r_select <= 0; Y_enable <= 0;
    end

    sub4: begin
        #10 Grc <= 1; r_select <= 1; alu_instruction <= sub; Z_enable <= 1;
        #75 Grc <= 0; r_select <= 0; alu_instruction <= 0; Z_enable <= 0;
    end

    sub5: begin
        #10 Z_LO_select <= 1; Gra <= 1; r_enable <= 1;
        #75 Z_LO_select <= 0; Gra <= 0; r_enable <= 0;
    end

    // Or Operation
    or3: begin
        #10 Grb <= 1; r_select <= 1; Y_enable <= 1;
        #75 Grb <= 0; r_select <= 0; Y_enable <= 0;
    end
```

hdl > control_unit > control_unit.v

```
    or4: begin
        #10 Grc <= 1; r_select <= 1; alu_instruction <= OR; Z_enable <= 1;
        #75 Grc <= 0; r_select <= 0; alu_instruction <= 0; Z_enable <= 0;
    end

    or5: begin
        #10 Z_LO_select <= 1; Gra <= 1; r_enable <= 1;
        #75 Z_LO_select <= 0; Gra <= 0; r_enable <= 0;
    end

    // And Operation
    and3: begin
        #10 Grb <= 1; r_select <= 1; Y_enable <= 1;
        #75 Grb <= 0; r_select <= 0; Y_enable <= 0;
    end

    and4: begin
        #10 Grc <= 1; r_select <= 1; alu_instruction <= AND; Z_enable <= 1;
        #75 Grc <= 0; r_select <= 0; alu_instruction <= 0; Z_enable <= 0;
    end

    and5: begin
        #10 Z_LO_select <= 1; Gra <= 1; r_enable <= 1;
        #75 Z_LO_select <= 0; Gra <= 0; r_enable <= 0;
    end

    // SHR Operation
    shr3: begin
        #10 Grb <= 1; r_select <= 1; Y_enable <= 1;
        #75 Grb <= 0; r_select <= 0; Y_enable <= 0;
    end

    shr4: begin
        #10 Grc <= 1; r_select <= 1; alu_instruction <= shr; Z_enable <= 1;
        #75 Grc <= 0; r_select <= 0; alu_instruction <= 0; Z_enable <= 0;
    end

    shr5: begin
        #10 Z_LO_select <= 1; Gra <= 1; r_enable <= 1;
        #75 Z_LO_select <= 0; Gra <= 0; r_enable <= 0;
    end

    // SHL Operation
    shl3: begin
        #10 Grb <= 1; r_select <= 1; Y_enable <= 1;
        #75 Grb <= 0; r_select <= 0; Y_enable <= 0;
    end

```

```
    shl4: begin
        #10 Grc <= 1; r_select <= 1; alu_instruction <= shl; Z_enable <= 1;
        #75 Grc <= 0; r_select <= 0; alu_instruction <= 0; Z_enable <= 0;
    end

    shl5: begin
        #10 Z_LO_select <= 1; Gra <= 1; r_enable <= 1;
        #75 Z_LO_select <= 0; Gra <= 0; r_enable <= 0;
    end

    // shra Operation
    shra3: begin
        #10 Grb <= 1; r_select <= 1; Y_enable <= 1;
        #75 Grb <= 0; r_select <= 0; Y_enable <= 0;
    end

    shra4: begin
        #10 Grc <= 1; r_select <= 1; alu_instruction <= shra; Z_enable <= 1;
        #75 Grc <= 0; r_select <= 0; alu_instruction <= 0; Z_enable <= 0;
    end

    shra5: begin
        #10 Z_LO_select <= 1; Gra <= 1; r_enable <= 1;
        #75 Z_LO_select <= 0; Gra <= 0; r_enable <= 0;
    end

    // ror Operation
    ror3: begin
        #10 Grb <= 1; r_select <= 1; Y_enable <= 1;
        #75 Grb <= 0; r_select <= 0; Y_enable <= 0;
    end

    ror4: begin
        #10 Grc <= 1; r_select <= 1; alu_instruction <= ror; Z_enable <= 1;
        #75 Grc <= 0; r_select <= 0; alu_instruction <= 0; Z_enable <= 0;
    end

    ror5: begin
        #10 Z_LO_select <= 1; Gra <= 1; r_enable <= 1;
        #75 Z_LO_select <= 0; Gra <= 0; r_enable <= 0;
    end

    // rol Operation
    rol3: begin
        #10 Grb <= 1; r_select <= 1; Y_enable <= 1;
        #75 Grb <= 0; r_select <= 0; Y_enable <= 0;
    end

```

hdl > control_unit > control_unit.v

```
    rol4: begin
        #10 Grc <= 1; r_select <= 1; alu_instruction <= rol; Z_enable <= 1;
        #75 Grc <= 0; r_select <= 0; alu_instruction <= 0; Z_enable <= 0;
    end

    rol5: begin
        #10 Z_LO_select <= 1; Gra <= 1; r_enable <= 1;
        #75 Z_LO_select <= 0; Gra <= 0; r_enable <= 0;
    end

   // Multiplication Operation
    mul3: begin
        #10 Grb <= 1; r_select <= 1; Y_enable <= 1;
        #75 Grb <= 0; r_select <= 0; Y_enable <= 0;
    end

    mul4: begin
        #10 Grc <= 1; r_select <= 1; alu_instruction <= mul; Z_enable <= 1;
        #75 Grc <= 0; r_select <= 0; alu_instruction <= 0; Z_enable <= 0;
    end

    mul5: begin
        #10 Z_LO_select <= 1; LO_enable <= 1;
        #75 Z_LO_select <= 0; LO_enable <= 0;
    end

    mul6: begin
        #10 Z_HI_select <= 1; HI_enable <= 1;
        #75 Z_HI_select <= 0; HI_enable <= 0;
    end

    // Division Operation
    div3: begin
        #10 Grb <= 1; r_select <= 1; Y_enable <= 1;
        #75 Grb <= 0; r_select <= 0; Y_enable <= 0;
    end

    div4: begin
        #10 Grc <= 1; r_select <= 1; alu_instruction <= div; Z_enable <= 1;
        #75 Grc <= 0; r_select <= 0; alu_instruction <= 0; Z_enable <= 0;
    end

    div5: begin
        #10 Z_LO_select <= 1; LO_enable <= 1;
        #75 Z_LO_select <= 0; LO_enable <= 0;
    end

```

hdl > control_unit > control_unit.v

```
    div6: begin
        #10 Z_HI_select <= 1; HI_enable <= 1;
        #75 Z_HI_select <= 0; HI_enable <= 0;
    end

    // Not Operation
    not3: begin
        #10 Grb <= 1; r_select <= 1; alu_instruction <= NOT; Y_enable <= 1;
        #75 Grb <= 0; r_select <= 0; alu_instruction <= 0; Y_enable <= 0;
    end

    not4: begin
        #10 Z_LO_select <= 1; Gra <= 1; r_enable <= 1;
        #75 Z_LO_select <= 0; Gra <= 0; r_enable <= 0;
    end

    // Negate Operation
    neg3: begin
        #10 Grb <= 1; r_select <= 1; alu_instruction <= neg; Y_enable <= 1;
        #75 Grb <= 0; r_select <= 0; alu_instruction <= 0; Y_enable <= 0;
    end

    neg4: begin
        #10 Z_LO_select <= 1; Gra <= 1; r_enable <= 1;
        #75 Z_LO_select <= 0; Gra <= 0; r_enable <= 0;
    end

    // And Immediate Operation
    andi3: begin
        #10 c_select <= 1; Y_enable <= 1;
        #75 c_select <= 0; Y_enable <= 0;
    end

    andi4: begin
        #10 Grb <= 1; r_select <= 1; alu_instruction <= AND; Z_enable <= 1;
        #75 Grb <= 0; r_select <= 0; alu_instruction <= 0; Z_enable <= 0;
    end

    andi5: begin
        #10 Z_LO_select <= 1; Gra <= 1; r_enable <= 1;
        #75 Z_LO_select <= 0; Gra <= 0; r_enable <= 0;
    end

    // Add Immediate Operation
    addi3: begin
        #10 c_select <= 1; Y_enable <= 1;
        #75 c_select <= 0; Y_enable <= 0;
    end
```

hdl > control_unit > control_unit.v

```
    addi4: begin
        #10 Grb <= 1; r_select <= 1; alu_instruction <= add; Z_enable <= 1;
        #75 Grb <= 0; r_select <= 0; alu_instruction <= 0; Z_enable <= 0;
    end

    addi5: begin
        #10 Z_LO_select <= 1; Gra <= 1; r_enable <= 1;
        #75 Z_LO_select <= 0; Gra <= 0; r_enable <= 0;
    end

    // Or Immediate Operation
    ori3: begin
        #10 c_select <= 1; Y_enable <= 1;
        #75 c_select <= 0; Y_enable <= 0;
    end

    ori4: begin
        #10 Grb <= 1; r_select <= 1; alu_instruction <= OR; Z_enable <= 1;
        #75 Grb <= 0; r_select <= 0; alu_instruction <= 0; Z_enable <= 0;
    end

    ori5: begin
        #10 Z_LO_select <= 1; Gra <= 1; r_enable <= 1;
        #75 Z_LO_select <= 0; Gra <= 0; r_enable <= 0;
    end

    // Load Operation
    ld3: begin
        #10 Grb <= 1; ba_select <= 1; Y_enable <= 1;
        #75 Grb <= 0; ba_select <= 0; Y_enable <= 0;
    end

    ld4: begin
        #10 c_select <= 1; alu_instruction <= add; Z_enable <= 1;
        #75 c_select <= 0; alu_instruction <= 0; Z_enable <= 0;
    end

    ld5: begin
        #10 Z_LO_select <= 1; MAR_enable <= 1;
        #75 Z_LO_select <= 0; MAR_enable <= 0;
    end

    ld6: begin
        #10 read <= 1; MDR_enable <= 1;
        #75 read <= 0; MDR_enable <= 0;
    end

```

hdl > control_unit > control_unit.v

```
    ld7: begin
        #10 MDR_select <= 1; Gra <= 1; r_enable <= 1;
        #75 MDR_select <= 0; Gra <= 0; r_enable <= 0;
    end

    // Load Immediate Operation
    ldi3: begin
        #10 Grb <= 1; ba_select <= 1; Y_enable <= 1;
        #75 Grb <= 0; ba_select <= 0; Y_enable <= 0;
    end

    ldi4: begin
        #10 c_select <= 1; alu_instruction <= add; Z_enable <= 1;
        #75 c_select <= 0; alu_instruction <= 0; Z_enable <= 0;
    end

    ldi5: begin
        #10 Z_LO_select <= 1; Gra <= 1; r_enable <= 1;
        #75 Z_LO_select <= 0; Gra <= 0; r_enable <= 0;
    end

    //Store Operation
    st3: begin
        #10 Grb <= 1; ba_select <= 1; Y_enable <= 1;
        #75 Grb <= 0; ba_select <= 0; Y_enable <= 0;
    end

    st4: begin
        #10 c_select <= 1; alu_instruction <= add; Z_enable <= 1;
        #75 c_select <= 0; alu_instruction <= 0; Z_enable <= 0;
    end

    st5: begin
        #10 Z_LO_select <= 1; MAR_enable <= 1;
        #75 Z_LO_select <= 0; MAR_enable <= 0;
    end

    st6: begin
        #10 write <= 1; MDR_enable <= 1; Gra <= 1; r_select <= 1;
        #75 write <= 0; MDR_enable <= 0; Gra <= 0; r_select <= 0;
    end

    // Jump Register Operation
    jr3: begin
        #10 Gra <= 1; r_select <= 1; PC_enable <= 1;
        #75 Gra <= 0; r_select <= 0; PC_enable <= 0;
    end

```

```
    //Jump and Link Operation
    jal3: begin
        #10 manual_R15_enable <= 1; PC_select <= 1;
        #75 manual_R15_enable <= 0; PC_select <= 0;
    end

    jal4: begin
        #10 Gra <= 1; r_select <= 1; PC_enable <= 1;
        #75 Gra <= 0; r_select <= 0; PC_enable <= 0;
    end

    // Move from Hi Register Operation
    mfhi3: begin
        #10 Gra <= 1; r_enable <= 1; HI_select <= 1;
        #75 Gra <= 0; r_enable <= 0; HI_select <= 0;
    end

    // Move from Lo Register Operation
    mflo3: begin
        #10 Gra <= 1; r_enable <= 1; LO_select <= 1;
        #75 Gra <= 0; r_enable <= 0; LO_select <= 0;
    end

    // Inputting Operation
    in3: begin
        #10 Gra <= 1; r_enable <= 1; inport_select <= 1;
        #75 Gra <= 0; r_enable <= 0; inport_select <= 0;
    end

    // Outputting Operation
    out3: begin
        #10 Gra <= 1; r_select <= 1; outport_enable <= 1;
        #75 Gra <= 0; r_select <= 0; outport_enable <= 0;
    end

    // Branch Operation
    br3: begin
        #10 Gra <= 1; r_select <= 1; con_enable <= 1;
        #75 Gra <= 0; r_select <= 0; con_enable <= 0;
    end

    br4: begin
        #10 PC_select <= 1; Y_enable <= 1;
        #75 PC_select <= 0; Y_enable <= 0;
    end

```

```
        br5: begin
            #10 c_select <= 1; alu_instruction <= add; Z_enable <= 1;
            #75 c_select <= 0; alu_instruction <= 0; Z_enable <= 0;
        end

        br6: begin
            #10 Z_LO_select <= 1; PC_enable <= con_output;
            #75 Z_LO_select <= 0; PC_enable <= 0;
        end

        // No Operation
        nop3: begin
            MDR_select <= 0; IR_enable <= 0;
        end

        // Halt Operation (Stops instruction execution)
        halt3: begin
            MDR_select <= 0; IR_enable <= 0;
        end

        //Any other input would be default to do nothing
        default: begin
        end
        endcase
    end
    endmodule
```

Memory Subsystem

Ram

hdl > memory_subsystem > ram.v

```
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("ram.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 File

```
1   08800002
2   08080000
3   01000068
4   0917FFFC
5   00900001
6   09800069
7   99980004
8   09980002
9   039FFFFD
10  D0000000
11  9B900002
12  09000005
13  09880002
14  19918000
15  63B80002
16  8BB80000
17  93B80000
18  6BB8000F
19  50880000
20  7388001C
21  43B80000
22  39180000
23  11000052
24  59100000
25  31180000
26  28908000
27  11880060
28  21918000
29  48900000
30  0A000006
31  0A800032
32  7AA00000
33  C3800000
34  CB000000
35  82A00000
36  0C27FFFF
37  0CAFFFED
38  0D300000
39  0DB80000
40  AD000000
41  D8000000
42  ffffffff
43  ffffffff
44  ffffffff
45  ffffffff
46  ffffffff
47  ffffffff
48  ffffffff
49  ffffffff
50  ffffffff
51  ffffffff
52  ffffffff
53  ffffffff
54  ffffffff
55  ffffffff
56  ffffffff
57  ffffffff
58  ffffffff
59  ffffffff
60  ffffffff
61  ffffffff
62  ffffffff
63  ffffffff
64  ffffffff
65  ffffffff
66  ffffffff
67  ffffffff
68  ffffffff
69  ffffffff
70  ffffffff
71  ffffffff
72  ffffffff
73  ffffffff
74  ffffffff
75  ffffffff
76  ffffffff
77  ffffffff
78  ffffffff
79  ffffffff
80  ffffffff
81  ffffffff
82  ffffffff
83  00000026
84  ffffffff
85  ffffffff
86  ffffffff
87  ffffffff
88  ffffffff
89  ffffffff
90  ffffffff
91  ffffffff
92  ffffffff
93  ffffffff
94  ffffffff
95  ffffffff
96  ffffffff
97  ffffffff
98  ffffffff
99  ffffffff
100 ffffffff
101 ffffffff
102 ffffffff
103 ffffffff
104 ffffffff
105 00000055
106 ffffffff
107 ffffffff
108 ffffffff
109 ffffffff
110 ffffffff
111 ffffffff
112 ffffffff
113 ffffffff
114 ffffffff
115 ffffffff
116 ffffffff
117 ffffffff
118 ffffffff
119 ffffffff
120 ffffffff
121 ffffffff
122 ffffffff
123 ffffffff
124 ffffffff
125 ffffffff
126 ffffffff
127 ffffffff
128 ffffffff
129 ffffffff
130 ffffffff
131 ffffffff
132 ffffffff
133 ffffffff
134 ffffffff
135 ffffffff
136 ffffffff
137 ffffffff
138 ffffffff
139 ffffffff
140 ffffffff
141 ffffffff
142 ffffffff
143 ffffffff
144 ffffffff
145 ffffffff
146 ffffffff
147 ffffffff
148 ffffffff
149 ffffffff
150 ffffffff
151 ffffffff
152 ffffffff
153 ffffffff
154 ffffffff
155 ffffffff
156 ffffffff
157 ffffffff
158 ffffffff
159 ffffffff
160 ffffffff
161 ffffffff
162 ffffffff
163 ffffffff
164 ffffffff
165 ffffffff
166 ffffffff
167 ffffffff
168 ffffffff
169 ffffffff
170 ffffffff
171 ffffffff
172 ffffffff
173 ffffffff
174 ffffffff
175 ffffffff
176 ffffffff
177 ffffffff
178 ffffffff
179 ffffffff
180 ffffffff
181 ffffffff
182 ffffffff
183 ffffffff
184 ffffffff
185 ffffffff
186 ffffffff
187 ffffffff
188 ffffffff
189 ffffffff
190 ffffffff
191 ffffffff
192 ffffffff
193 ffffffff
194 ffffffff
195 ffffffff
196 ffffffff
197 ffffffff
198 ffffffff
199 ffffffff
200 ffffffff
201 ffffffff
202 ffffffff
203 ffffffff
204 ffffffff
205 ffffffff
206 ffffffff
207 ffffffff
208 ffffffff
209 ffffffff
210 ffffffff
211 ffffffff
212 ffffffff
213 ffffffff
214 ffffffff
215 ffffffff
216 ffffffff
217 ffffffff
```

```
218 ffffffff
219 ffffffff
220 ffffffff
221 ffffffff
222 ffffffff
223 ffffffff
224 ffffffff
225 ffffffff
226 ffffffff
227 ffffffff
228 ffffffff
229 ffffffff
230 ffffffff
231 ffffffff
232 ffffffff
233 ffffffff
234 ffffffff
235 ffffffff
236 ffffffff
237 ffffffff
238 ffffffff
239 ffffffff
240 ffffffff
241 ffffffff
242 ffffffff
243 ffffffff
244 ffffffff
245 ffffffff
246 ffffffff
247 ffffffff
248 ffffffff
249 ffffffff
250 ffffffff
251 ffffffff
252 ffffffff
253 ffffffff
254 ffffffff
255 ffffffff
256 ffffffff
257 ffffffff
258 ffffffff
259 ffffffff
260 ffffffff
261 ffffffff
262 ffffffff
263 ffffffff
264 ffffffff
265 ffffffff
266 ffffffff
267 ffffffff
268 ffffffff
269 ffffffff
270 ffffffff
271 ffffffff
272 ffffffff
273 ffffffff
274 ffffffff
275 ffffffff
276 ffffffff
277 ffffffff
278 ffffffff
279 ffffffff
280 ffffffff
281 ffffffff
282 ffffffff
283 ffffffff
284 ffffffff
285 ffffffff
286 ffffffff
287 ffffffff
288 ffffffff
289 ffffffff
290 ffffffff
291 ffffffff
292 ffffffff
293 ffffffff
294 ffffffff
295 ffffffff
296 ffffffff
297 ffffffff
298 ffffffff
299 ffffffff
300 ffffffff
301 1EC50000
302 264D8000
303 26EE0000
304 A7800000
305 ffffffff
306 ffffffff
307 ffffffff
308 ffffffff
309 ffffffff
310 ffffffff
311 ffffffff
312 ffffffff
313 ffffffff
314 ffffffff
315 ffffffff
316 ffffffff
317 ffffffff
318 ffffffff
319 ffffffff
320 ffffffff
321 ffffffff
322 ffffffff
323 ffffffff
324 ffffffff
325 ffffffff
326 ffffffff
327 ffffffff
328 ffffffff
329 ffffffff
330 ffffffff
331 ffffffff
332 ffffffff
333 ffffffff
334 ffffffff
335 ffffffff
336 ffffffff
337 ffffffff
338 ffffffff
339 ffffffff
340 ffffffff
341 ffffffff
342 ffffffff
343 ffffffff
344 ffffffff
345 ffffffff
346 ffffffff
347 ffffffff
348 ffffffff
349 ffffffff
350 ffffffff
351 ffffffff
352 ffffffff
353 ffffffff
354 ffffffff
355 ffffffff
356 ffffffff
357 ffffffff
358 ffffffff
359 ffffffff
360 ffffffff
361 ffffffff
362 ffffffff
363 ffffffff
364 ffffffff
365 ffffffff
366 ffffffff
367 ffffffff
368 ffffffff
369 ffffffff
370 ffffffff
371 ffffffff
372 ffffffff
373 ffffffff
374 ffffffff
375 ffffffff
376 ffffffff
377 ffffffff
378 ffffffff
379 ffffffff
380 ffffffff
381 ffffffff
382 ffffffff
383 ffffffff
384 ffffffff
385 ffffffff
386 ffffffff
387 ffffffff
388 ffffffff
389 ffffffff
390 ffffffff
391 ffffffff
392 ffffffff
393 ffffffff
394 ffffffff
395 ffffffff
396 ffffffff
397 ffffffff
398 ffffffff
399 ffffffff
400 ffffffff
401 ffffffff
402 ffffffff
403 ffffffff
404 ffffffff
405 ffffffff
406 ffffffff
407 ffffffff
408 ffffffff
409 ffffffff
410 ffffffff
411 ffffffff
412 ffffffff
413 ffffffff
414 ffffffff
415 ffffffff
416 ffffffff
417 ffffffff
418 ffffffff
419 ffffffff
420 ffffffff
421 ffffffff
422 ffffffff
423 ffffffff
424 ffffffff
425 ffffffff
426 ffffffff
427 ffffffff
428 ffffffff
429 ffffffff
430 ffffffff
431 ffffffff
432 ffffffff
433 ffffffff
434 ffffffff
435 ffffffff
436 ffffffff
437 ffffffff
438 ffffffff
439 ffffffff
440 ffffffff
441 ffffffff
442 ffffffff
443 ffffffff
444 ffffffff
445 ffffffff
446 ffffffff
447 ffffffff
448 ffffffff
449 ffffffff
```

```
450  ffffffff
451  ffffffff
452  ffffffff
453  ffffffff
454  ffffffff
455  ffffffff
456  ffffffff
457  ffffffff
458  ffffffff
459  ffffffff
460  ffffffff
461  ffffffff
462  ffffffff
463  ffffffff
464  ffffffff
465  ffffffff
466  ffffffff
467  ffffffff
468  ffffffff
469  ffffffff
470  ffffffff
471  ffffffff
472  ffffffff
473  ffffffff
474  ffffffff
475  ffffffff
476  ffffffff
477  ffffffff
478  ffffffff
479  ffffffff
480  ffffffff
481  ffffffff
482  ffffffff
483  ffffffff
484  ffffffff
485  ffffffff
486  ffffffff
487  ffffffff
488  ffffffff
489  ffffffff
490  ffffffff
491  ffffffff
492  ffffffff
493  ffffffff
494  ffffffff
495  ffffffff
496  ffffffff
497  ffffffff
498  ffffffff
499  ffffffff
500  ffffffff
501  ffffffff
502  ffffffff
503  ffffffff
504  ffffffff
505  ffffffff
506  ffffffff
507  ffffffff
508  ffffffff
509  ffffffff
510  ffffffff
511  ffffffff
512  ffffffff
```

Ports

Inport

hdl > ports > inport.v

```
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
```

Outport

hdl > ports > outport.v

```
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
```

Control Unit Testbench

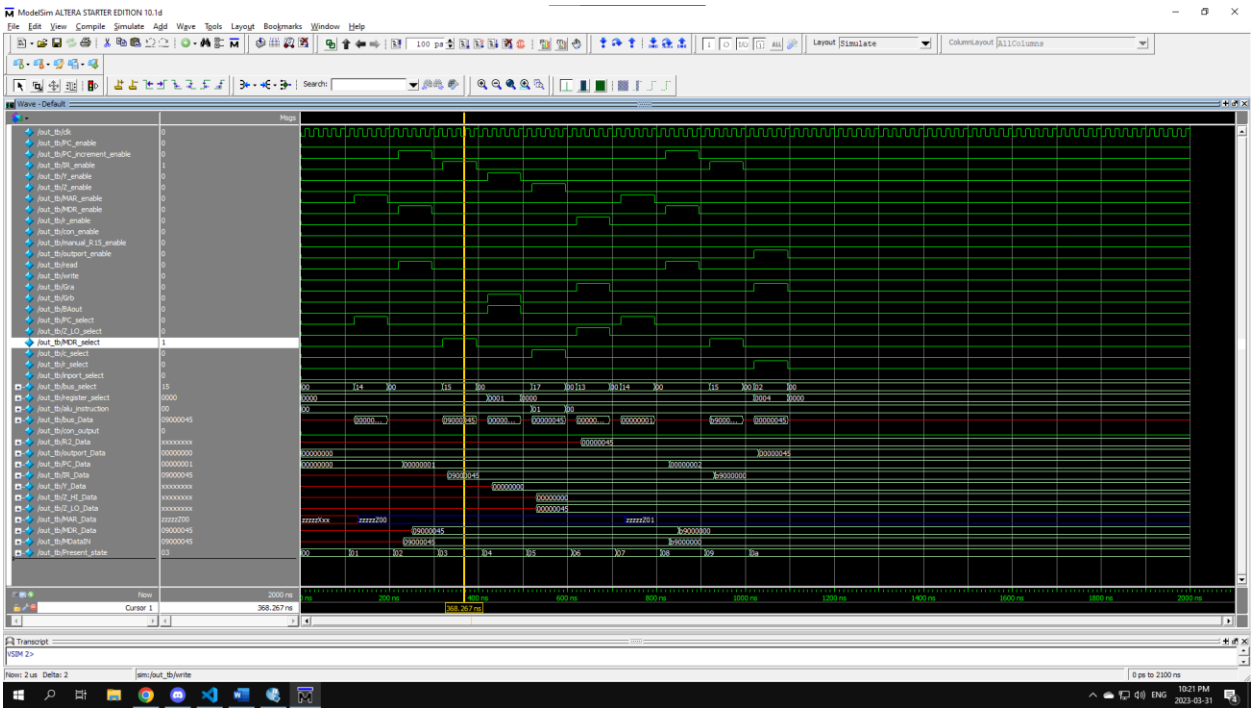
Our group finished the code for the control unit but since there were no specified instructions, we were not sure how to implement a testbench for this phase. In this section, we used our phase 2 simulations to show that most features required still work.

What features actually work:

- IR holds the instruction from the .MIF file
- PC is a dedicated module and it will only increment once every time the increment PC signal is enabled
- PC works with all branch instructions
- All registers can hold data from the bus and the select and encode logic works
- R15_enable is also modified to enable from the select encode logic and another manual enable signal
- Instruction fetch (shown in control unit) and decode work

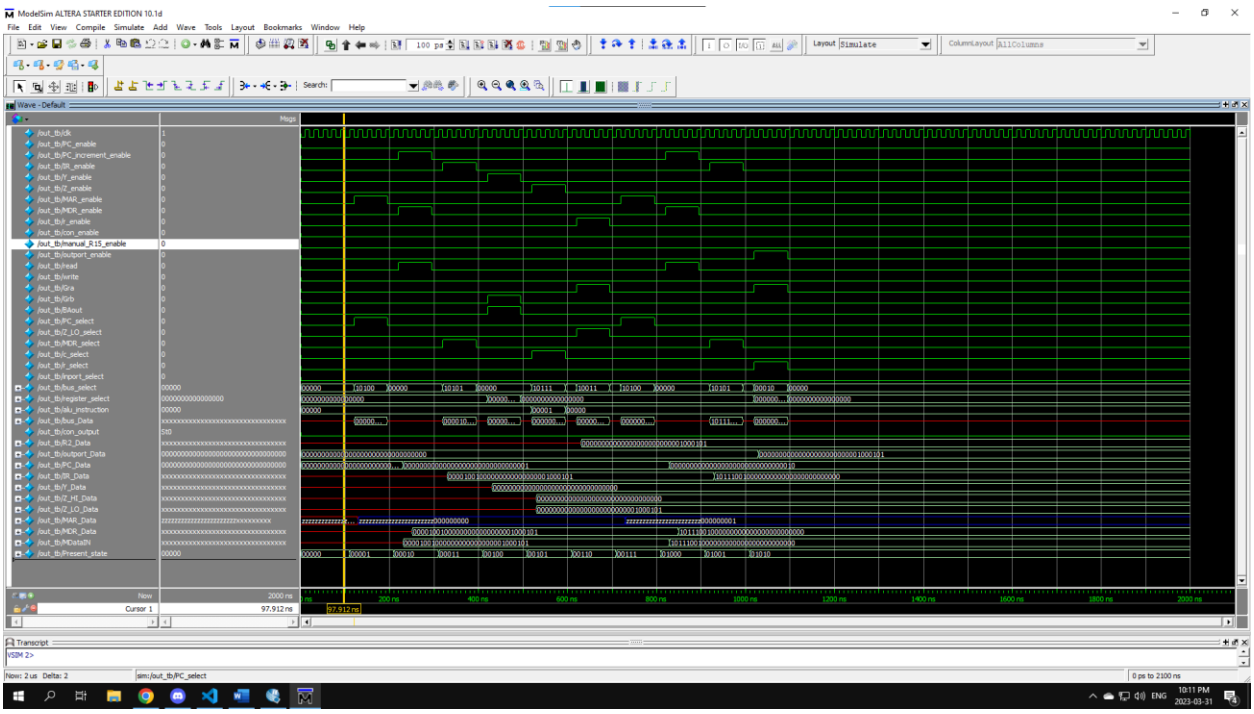
IR Storing Data from Ram

Example of IR Holding the Instruction from Memory:

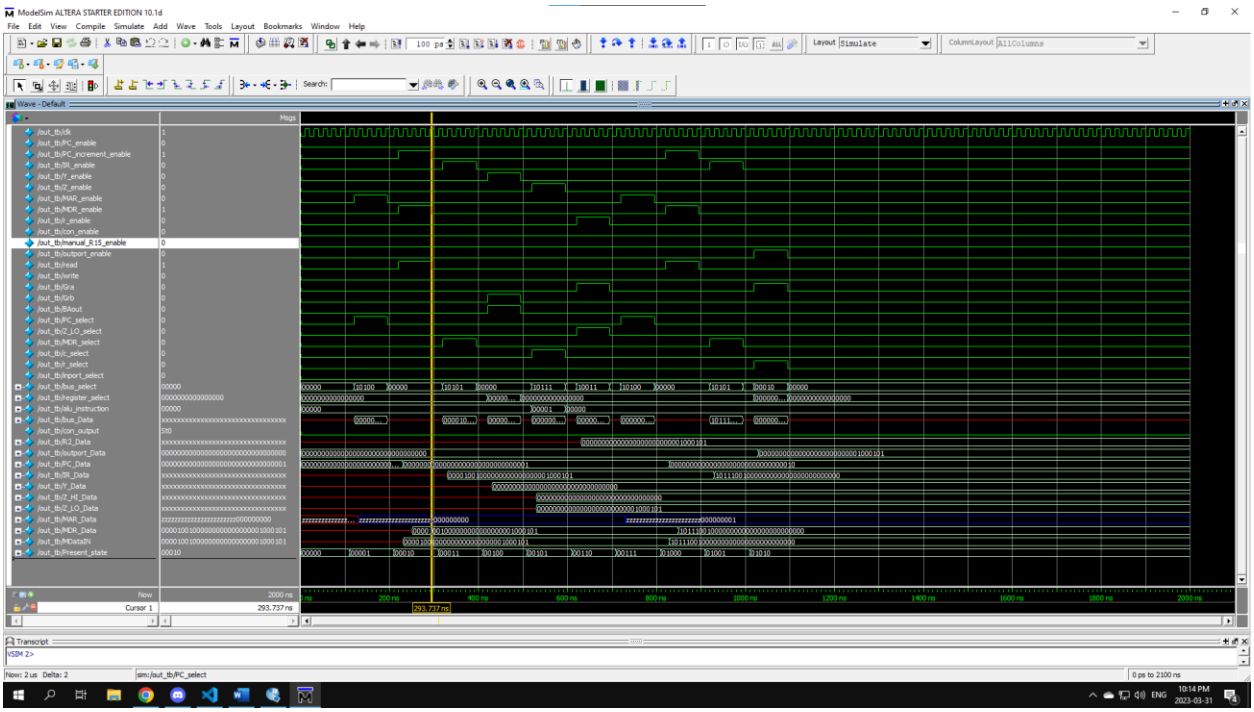


Dedicated PC Incrementor

Example Testbench at the Start:

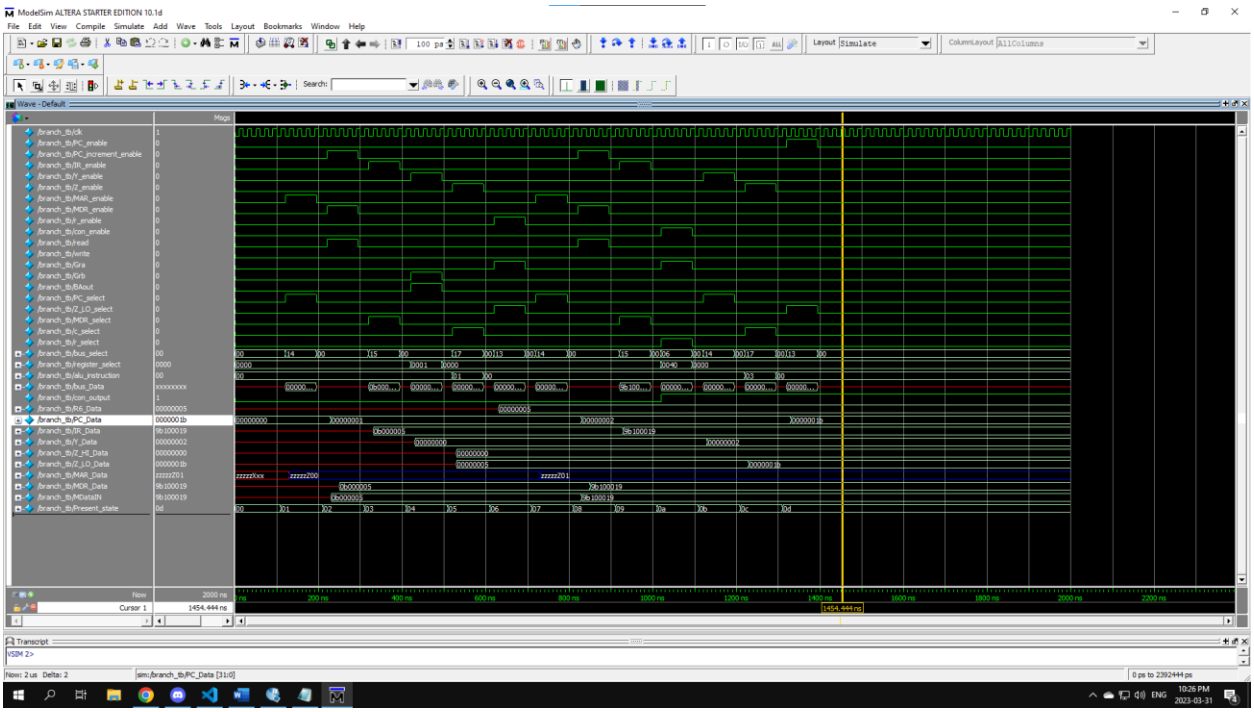


Example Testbench Showing that PC Incremented only 1 time:

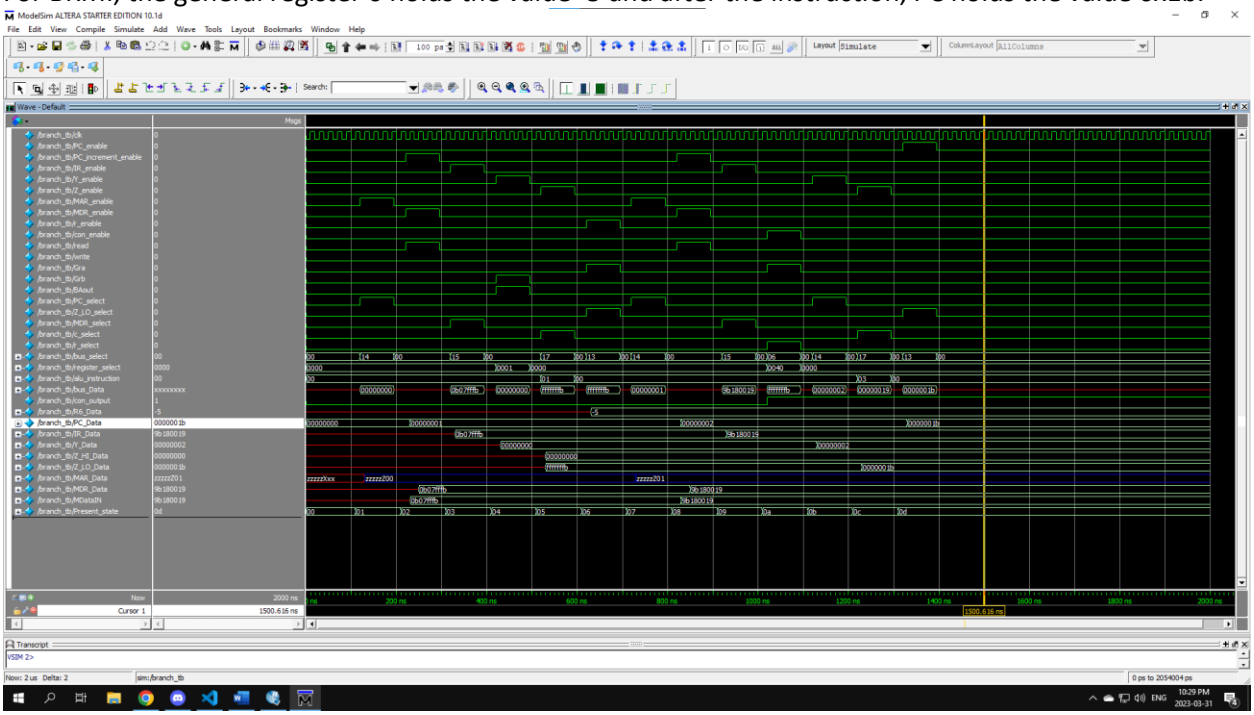


PC Changing for BRMI and BRPL

For BRPL, the general register 6 holds the value 5 and after the instruction, PC holds the value 0x1b.



For BRMI, the general register 6 holds the value -5 and after the instruction, PC holds the value 0x1b.



Instruction Fetch and Decode

```
fetch0: begin
    #10 PC_select <= 1; MAR_enable <= 1;
    #75 PC_select <= 0; MAR_enable <= 0;
end

fetch1: begin
    #10 PC_increment_enable <= 1; read <= 1; MDR_enable <= 1;
    #75 PC_increment_enable <= 0; read <= 0; MDR_enable <= 0;
end

fetch2: begin
    #10 MDR_select <= 1; IR_enable <= 1;
    #75 MDR_select <= 0; IR_enable <= 0;
end
```

```
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]};
```

The decoder allows the select and encode logic to determine which register needs to be enabled or selected and then the r_enable, r_select, and ba_select signals allow the select and encode logic to output the proper enable and select signals. The C_sign_extended_Data variable holds the first 19 bits which is usually the immediate value. For some instructions bits 15-19 can be Rc.