ECE 385 Lab 5 Report

Team Members:

Siddarth Natarajan (sn28)
Vishnu Shastry (vishnu5)

Introduction:

SLC-3 stands for Simplified Little Computer - 3. It is a simplified version of the LC3 processor that was made by Sanjay Patel and Yale Patt. SLC-3 has an intuitive 16 bit instruction design, where the user can communicate with the processor using a sequence of instructions put forward by the Instruction Set Architecture.

SLC-3 has an easy to understand, 11 instruction architecture with traditional operations such as ADD, ADDi, AND, ANDi, LDR, STR, and BR. These commands are easy to understand and allow us to interact with a computer at the hardware level using 16-bit commands. The first 4 bits of the command are known as the opcode. The opcode of the processor specifies which instruction is currently being executed. Following these 4 bits, depending upon the command, we are able to specify the location of the destination register, source register, as well as any offset that may be needed for a particular command to be executed.

The processor has three primary phases: FETCH, DECODE, and EXECUTE. In the FETCH phase, instructions are retrieved from memory, while the DECODE phase interprets the instruction, and the EXECUTE phase carries out the desired operation.

In essence, the processor can be split up into 3 main files: a cpu file that allows us to instantiate all of the multiplexers that are required in the design as well as any external logic that may be required in order to perform arithmetic and logical operations on register values, a control file that defines the finite state machine that governs the transition and next state logic for the required variables in the design by giving us a mapping for the inputs to the different selection lines / selection signals that are used within the program, and a register file that is used in order to handle all the values for the registers in the design.

In accordance with this, the SLC-3 has a streamlined datapath that allows us to clearly visualize signals and how they may be used in order to handle a particular computation. Additionally, it uses HEX displays for real-time visualization of the instruction register.

While the SLC-3 processor only contains 11 commands (instructions), the LC-3 processor contains 19 different instructions with all 16 opcodes being used in the computation of particular values. In particular, if we look on a broader level, the LC-3 processor has a TRAP command, something that is missing in SLC-3. Other commands that are missing include LEA, ST, STI , LD , LDI, JSRR, and RET. It also supports additional addressing modes to provide flexibility. The LC-3 datapath includes more components, such as multiple registers, more complex control logic, and extended capabilities for instruction processing for more complex instructions and programs.

Written Description and Diagrams of SLC-3

Summary of Operation

In order to comprehensively address the operation of the SLC-3, as mentioned previously, it is beneficial to split up the operation of the processor into 3 different phases: FETCH, DECODE, and EXECUTE.

FETCH Phase Implementation

During the first week, the FETCH phase was implemented to retrieve instructions from the memory. The key operations included:

- MAR loading: The PC value was loaded into MAR to specify the address from which to fetch the instruction.
- MDR loading: The instruction was fetched from the memory into the MDR.
- IR loading: The content of MDR was then transferred to the IR for decoding.
- PC increment: After fetching the instruction, the PC was incremented to point to the next instruction.

DECODE Phase Implementation:

1. **Instruction Sequencer/Decoder**: The IR provides the instruction to the Instruction Sequencer/Decoder, which interprets the opcode and prepares the control signals for the next phase. The decoding process includes determining the type of instruction and identifying the source and destination registers.
2. **Control Signals Generation**: Based on the decoded instruction, the control unit generates specific signals to direct the ALU and registers. For instance, if the

instruction is an ADD operation, control signals will direct the ALU to perform addition and indicate which registers to use.

EXECUTE Phase Implementation:

1. **ALU Operation**: The ALU receives control signals from the control unit and performs the specified operation (e.g., addition, subtraction). The inputs to the ALU come from either the register file or immediate values extracted from the instruction.
2. **Writing Back Results**: After the ALU completes the operation, the results are written back to the destination register or memory. The status register (NZP) is updated based on the result, indicating whether it is negative, zero, or positive.

Memory Interface Components

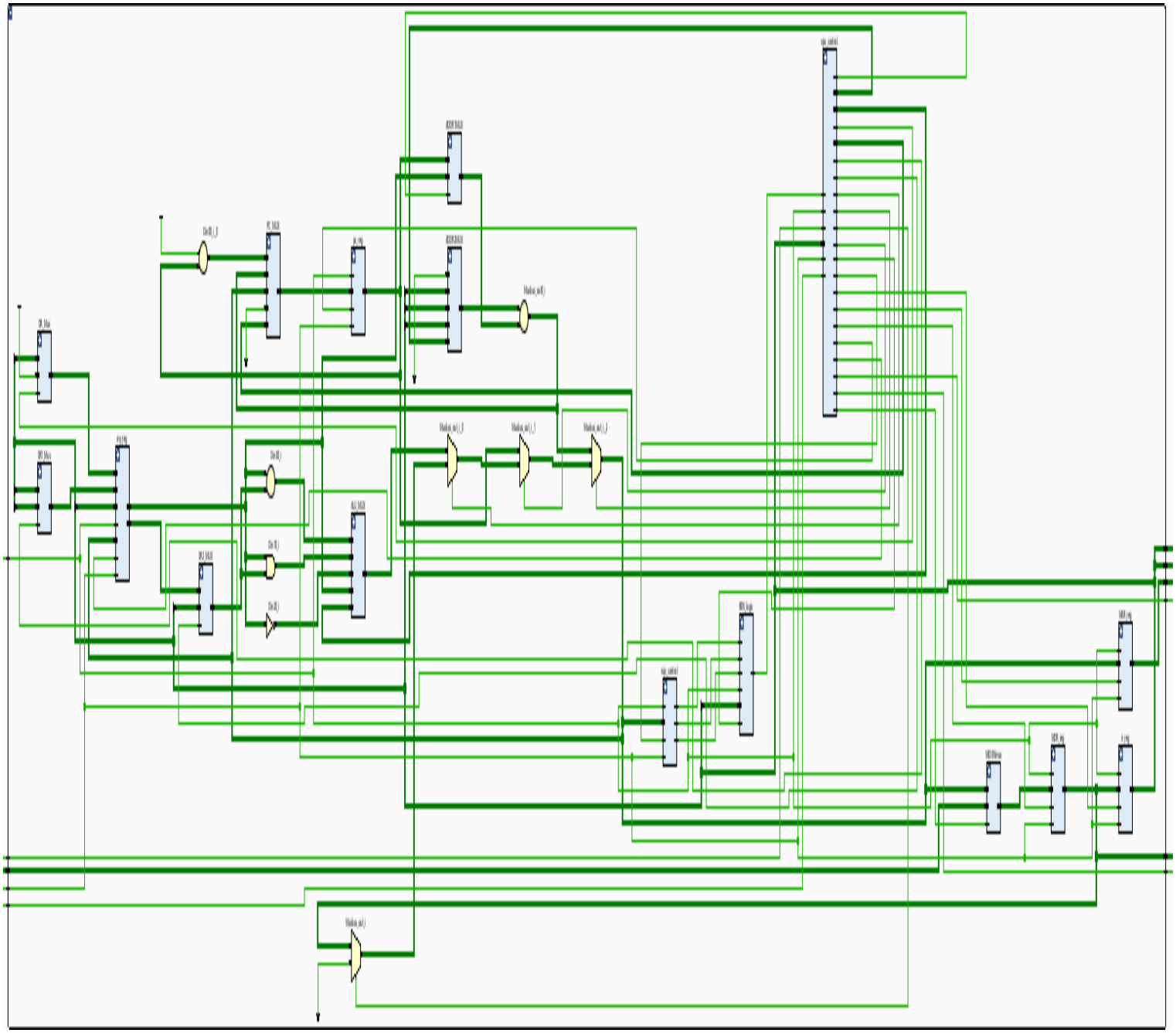
1. **Memory Address Register (MAR)**
    - The MAR holds the address of the memory location to be accessed. This register is updated during the fetch and load operations, pointing to the specific location in BRAM from which data will be read or to which data will be written.
    - The MAR can interface directly with the BRAM, allowing the control unit to specify which address to access during memory operations.
2. **Memory Data Register (MDR)**
    - The MDR serves as a temporary storage location for data being read from or written to BRAM. When data is fetched from memory, it is loaded into the MDR before being sent to the ALU or a destination register.
    - Similarly, when writing data back to memory, the MDR holds the data that will be written to the address specified by the MAR.
3. **Control Signals**
    - Control signals are crucial for managing the read and write operations to BRAM. These signals are generated by the control unit based on the current instruction being executed.
    - **Read Enable Signal**: Activates the read operation, allowing data to be fetched from BRAM into the MDR.
    - **Write Enable Signal**: Activates the write operation, allowing data from the MDR to be stored in the specified memory location.

Memory Interface Overview

In the context of an FPGA-based microprocessor, the memory interface is critical for facilitating communication between the CPU and the on-chip memory. For the SLC-3.2

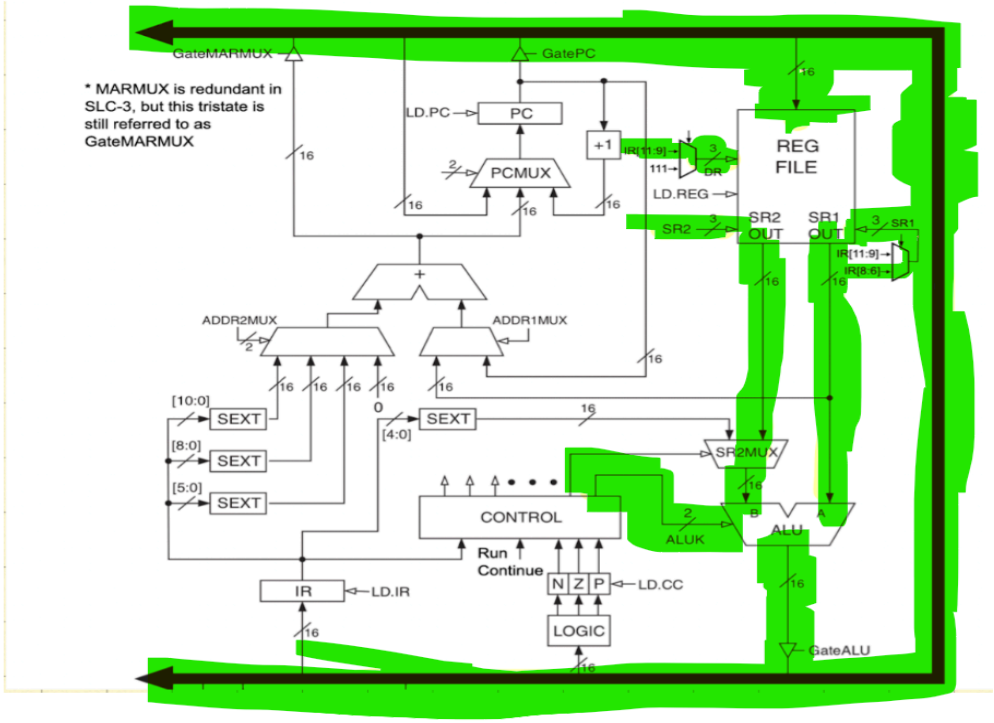
processor implemented in this lab, Block RAM (BRAM) on the FPGA serves as the primary memory resource.

Block Diagram of cpu.sv

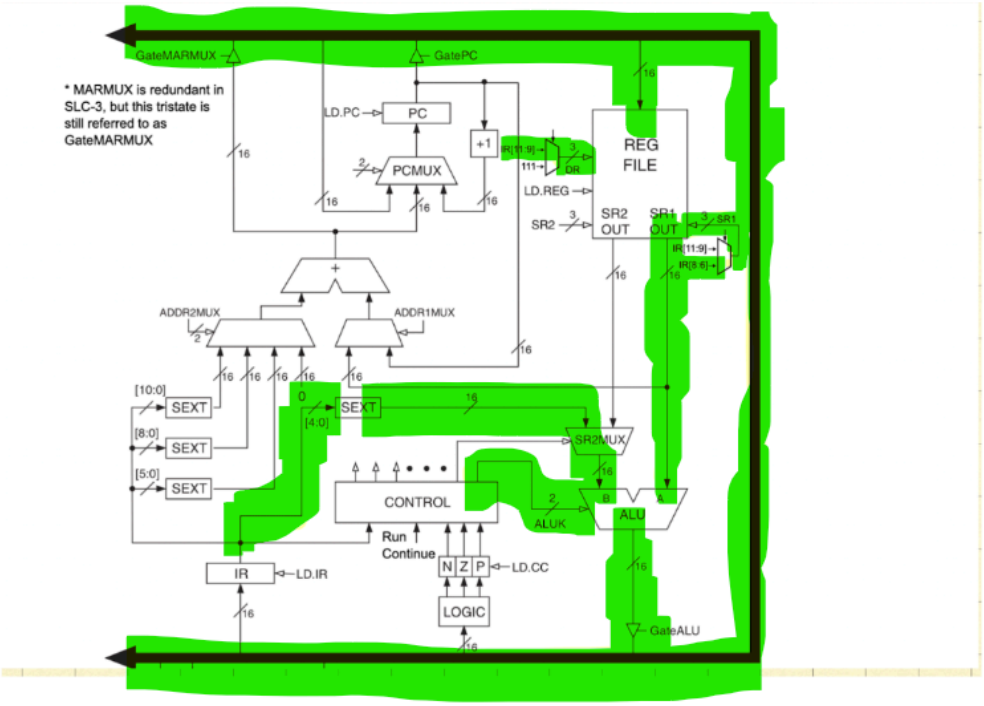


Data paths traced for each instructions (traced in green):

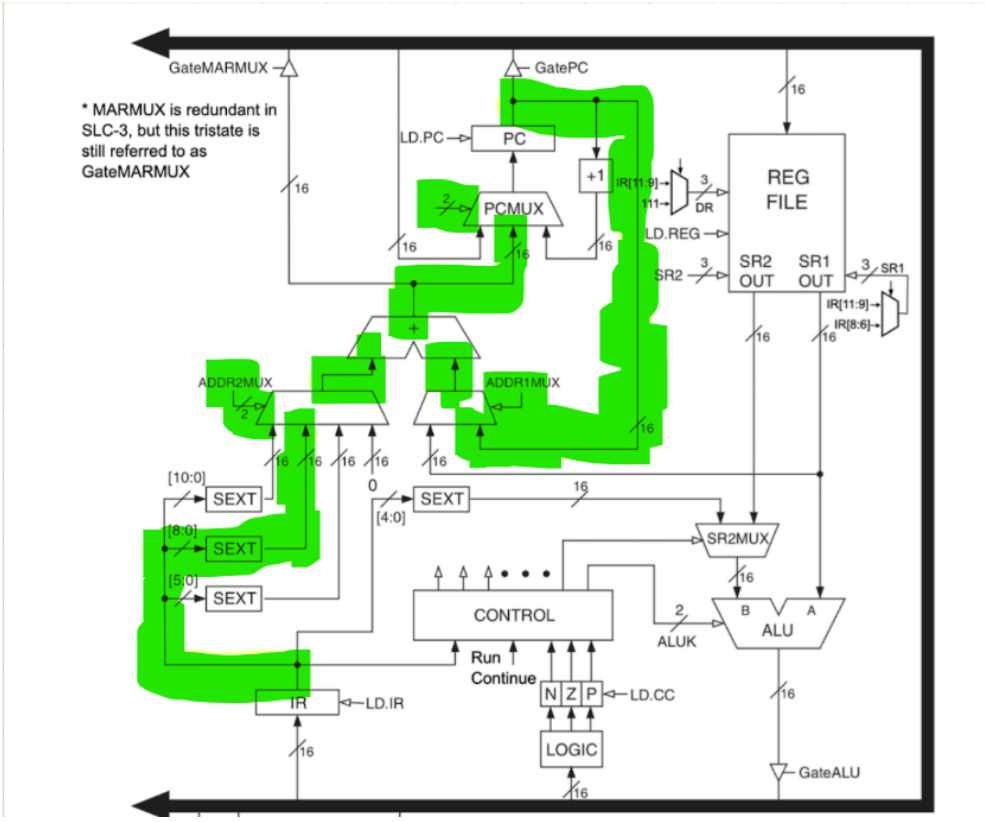
Add and And instruction:



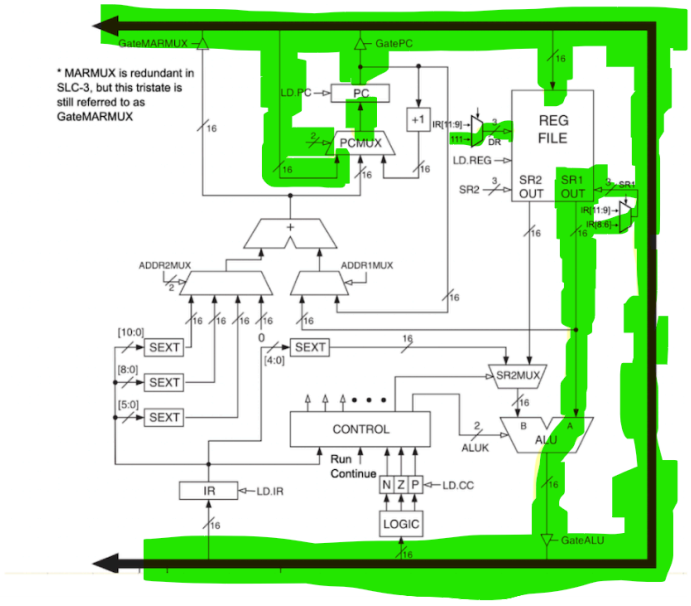
Addi and Andi instruction:



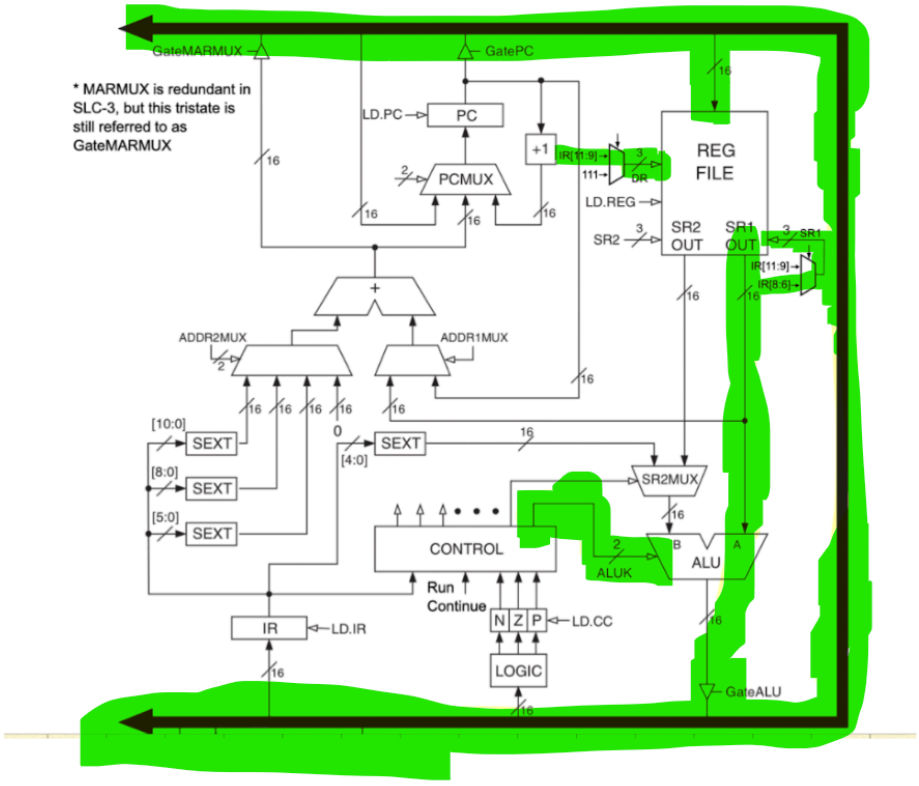
Branch instruction



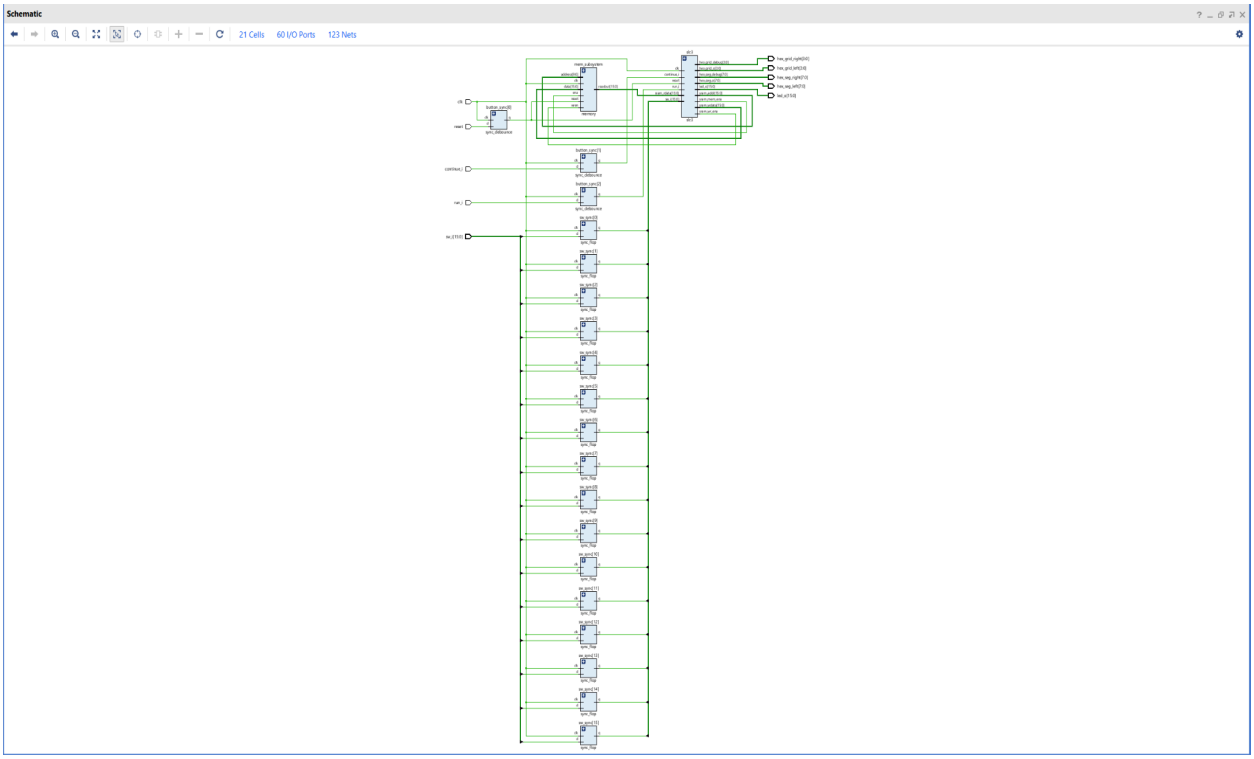
JSR instruction



Not instruction:



Block Diagram of processor top level



control.sv Description:

The control.sv module is responsible for controlling the overall flow of the SLC-3 (Simple LC-3) microprocessor core. It generates the necessary control signals that guide the execution of instructions by coordinating the loading, gating, and selection operations across different components in the CPU. The control logic operates based on the instruction register (ir), branch enable (ben), and external signals like run_i and continue_i.

- Inputs:
    - clk and reset: Clock and reset signals.
    - ir: Instruction Register containing the current instruction.
    - ben: A flag used to determine whether a branch is to be taken.
    - continue_i, run_i: External signals to control CPU state transitions.
- Outputs:
    - Signals to load various CPU components (ld_mar, ld_mdr, ld_ir, ld_pc, ld_led, ld_reg, ld_ben, ld_cc).
    - Control of multiplexers (SR1_MUX_select, SR2_MUX_select, ADDR1_MUX_select, ADDR2_MUX_select, DR_MUX_select, PCMUX_select, ALUK).
    - Gating signals (gate_pc, gate_mdr, gate_marmux, gate_alu).
    - Memory control (mem_mem_ena, mem_wr_ena).
- State Machine: The internal state machine drives the flow of instructions, with states like:
    - halted: Initial state waiting for run_i.
    - s_18: Begins execution, preparing to load the MAR with the PC.
    - s_33_1, s_33_2, s_33_3: States handling memory enable signals, accounting for wait states in synchronous BRAM.
    - Execution states for different instructions like ADD, AND, NOT, LDR, STR, JSR, JMP, etc.
- The state transitions depend on the opcode in the ir and flags like ben, with typical instruction execution cycles passing through fetch (s_18), decode, and execute phases.

cpu.sv Description:

The cpu.sv module represents the core of the SLC-3 microprocessor, integrating various components and datapath elements to execute instructions. It interacts with memory, processes instructions, and drives outputs like LED displays and hex debug displays.

- Inputs:
    - clk and reset: Clock and reset signals.
    - run_i and continue_i: External signals controlling CPU state transitions.
    - mem_rdata: Data read from memory.
- Outputs:

    - hex_display_debug, led_o: Debugging outputs and control for LEDs.
    - mem_wdata, mem_addr: Write data and memory address for external memory.
    - mem_mem_ena, mem_wr_ena: Signals to enable memory and perform write operations.
- Internal Signals:
    - Control Signals: Signals generated by the control module such as ld_mar, ld_mdr, ld_ir, ld_pc, and others, are used to load values into CPU registers and control execution flow.
    - Multiplexers: Various multiplexers such as SR1_MUX_select, SR2_MUX_select, PCMUX_select control the data flow in the ALU and register file.
    - Registers: Internal registers like mar (Memory Address Register), mdr (Memory Data Register), ir(Instruction Register), pc (Program Counter), and ben are used for instruction execution.
    - ALU Operations: Arithmetic and logic operations are handled by the ALU, controlled by signals like ALUK, gate_alu, and inputs from SR1 and SR2.
- Datapath: The module implements the SLC-3 datapath, which includes connections between the register file, memory interface, and the ALU. It handles instruction fetching, decoding, execution, and writing back results to the register file or memory.

The CPU state is controlled by the control module and moves through different stages of instruction execution, including fetching the instruction from memory, decoding it, performing the necessary computations (using the ALU), and writing results back to memory or registers.

Register unit description

The following System Verilog module, named register_unit is a simple register file unit for a processor containing a set of eight 16-bit registers. It features two source registers, named SR1_reg and SR2_reg, from which data can be read, and one destination register, called DR_reg into which data is written. The input clk ensures that register updates are synchronized. Reset sets all the registers to zero. The ld_reg signal enables data to be written to the destination register at each clock cycle.
The module contains two always blocks: the first, always_comb, is combinational logic continuously reading values from the register file and assigning the values to the output ports-SR1_OUT_reg and SR2_OUT_reg. The always_ff does synchronous updating of the register file on a positive edge of a clock. In this design, flexible data manipulation is possible, for example, data processing and storage; this is why it can be used in a simplified CPU datapath or in a microcontroller design.

Code for each unit:

CPU.sv

```
//-------------------------------------------------------------------------------
// Company:              UIUC ECE Dept.
// Engineer:             Stephen Kempf
//
// Create Date:
// Design Name:    ECE 385 Given Code - SLC-3 core
// Module Name:    SLC3
//
// Comments:
//    Revised 03-22-2007
//    Spring 2007 Distribution
//    Revised 07-26-2013
//    Spring 2015 Distribution
//    Revised 09-22-2015
//    Revised 06-09-2020
//      Revised 03-02-2021
//    Xilinx vivado
//    Revised 07-25-2023
//    Revised 12-29-2023
//    Revised 09-25-2024
//-------------------------------------------------------------------------------

module cpu (
    input   logic        clk,
    input   logic        reset,

    input   logic        run_i,
    input   logic        continue_i,
    output  logic [15:0] hex_display_debug,
    output  logic [15:0] led_o,

    input   logic [15:0] mem_rdata,
    output  logic [15:0] mem_wdata,
    output  logic [15:0] mem_addr,
    output  logic        mem_mem_ena,
    output  logic        mem_wr_ena
);


// Internal connections, follow the datapath block diagram and add the additional needed
signals
logic ld_mar;
logic ld_mdr;
logic ld_ir;
logic ld_pc;
logic ld_led;
logic ld_reg;
```

```
logic ld_cc;
logic ld_ben;

logic gate_pc;
logic gate_mdr;
logic gate_alu;
logic gate_marmux;

logic mio_en;

logic [1:0] pcmux;
logic [1:0] ALUK;

logic  SR1_MUX_select;
logic  SR2_MUX_select;
logic  DR_MUX_select;
logic [1:0] ADDR2_MUX_select;
logic ADDR1_MUX_select;
logic [1:0] PCMUX_select;

logic [15:0] mar;
logic [15:0] mdr;
logic [15:0] ir;
logic [15:0] pc;
logic ben;


logic [15:0] Mioen_out;
logic [15:0] Muxbus_out;
logic [15:0] PCmux_out;
logic [15:0] ALU_MUX_OUT;
logic [2:0] SR1_MUX_out;
logic [15:0] SR2_MUX_out;
logic [2:0] DR_MUX_out;
logic [15:0] ADDR2_MUX_OUT;
logic [15:0] A_ALU;
logic [15:0] B_ALU;
logic [15:0] ADDR1_MUX_OUT;

logic [1:0] mux_bux_select;


logic [15:0]SR2_Reg_In;

assign mem_addr = mar;
assign mem_wdata = mdr;


// REGISTER UNIT DECLARAtion below

logic [15:0] reg_in_bus;
assign reg_in_bus = Muxbus_out;
```

```
logic [2:0] DR_reg_unit;
assign DR_reg_unit =  DR_MUX_out;

logic [2:0] SR1_reg_unit;
assign SR1_reg_unit =  SR1_MUX_out;

logic [2:0] SR2_reg_unit;
assign SR2_reg_unit = ir[2:0] ;

logic [15:0] SR1_OUT_reg_unit;
assign A_ALU = SR1_OUT_reg_unit;

logic [15:0] SR2_OUT_reg_unit;
assign SR2_Reg_In = SR2_OUT_reg_unit;

// ADDR1 DECLARATION

logic [15:0] ADDR1_from_SR1;
assign ADDR1_from_SR1 = SR1_OUT_reg_unit;

// NZP Logic

logic [15:0] BUS_TO_NZP;
assign BUS_TO_NZP = Muxbus_out;

logic N_out, Z_out, P_out;
logic N_in,Z_in, P_in;
assign N_in = N_out;
assign Z_in = Z_out;
assign P_in = P_out;

// State machine, you need to fill in the code here as well
// .* auto-infers module input/output connections which have the same name
// This can help visually condense modules with large instantiations,
// but can also lead to confusing code if used too commonly
control cpu_control (
    .*
);


assign led_o = ir;
assign hex_display_debug = ir;

load_reg #(.DATA_WIDTH(16)) ir_reg (
    .clk    (clk),
    .reset  (reset),

    .load   (ld_ir),
    .data_i (mdr),

    .data_q (ir)
```

```
);

load_reg #(.DATA_WIDTH(16)) pc_reg (
    .clk(clk),
    .reset(reset),

    .load(ld_pc),
    .data_i(PCmux_out),

    .data_q(pc)
);

logic [15:0] mar_in;
assign mar_in = Muxbus_out;

load_reg #(.DATA_WIDTH(16)) MAR_reg (
    .clk    (clk),
    .reset  (reset),

    .load   (ld_mar),
    .data_i (mar_in),

    .data_q (mar)
);

load_reg #(.DATA_WIDTH(16)) MDR_reg (
    .clk    (clk),
    .reset  (reset),

    .load   (ld_mdr),
    .data_i (Mioen_out),

    .data_q (mdr)
);

logic [15:0] mio_en_bus;
assign mio_en_bus = Muxbus_out;

mux2_1 MIOENmux(
     .sel  (mio_en),
     .Din0 (mio_en_bus),
     .Din1 (mem_rdata),
     .Dout (Mioen_out)
);

logic [15:0] PC_MUX_bus;
assign PC_MUX_bus = Muxbus_out;

mux4_1 PC_MUX(

    .sel (PCMUX_select),
    .Din0 (pc+1),
```

```
    .Din1 (ADDR2_MUX_OUT + ADDR1_MUX_OUT),
    .Din2 (PC_MUX_bus),
    .Din3 (16'b0),
    .Dout (PCmux_out)

);

always_comb
begin
if(gate_marmux)
    begin
        Muxbus_out = ADDR2_MUX_OUT + ADDR1_MUX_OUT;
    end

else if(gate_pc)
    begin
        Muxbus_out = pc;
    end
else if(gate_alu)
    begin
        Muxbus_out = ALU_MUX_OUT;
    end

else if(gate_mdr)
    begin
        Muxbus_out = mdr;
    end

else
    begin
        Muxbus_out = 16'b0;
    end

end

assign B_ALU = SR2_MUX_out;

mux4_1 ALU_MUX (

    .sel (ALUK),
    .Din0 (A_ALU + B_ALU),
    .Din1 (A_ALU & B_ALU),
    .Din2 (~A_ALU),
    .Din3 (A_ALU),
    .Dout (ALU_MUX_OUT)
);

SR1_DR_MUX SR1_Mux(

    .sel (SR1_MUX_select),
    .Din0 (ir[11:9]),
    .Din1 (ir[8:6]),
```

```
    .Dout (SR1_MUX_out)
);

SR1_DR_MUX DR_Mux(

    .sel (DR_MUX_select),
    .Din0 (ir[11:9]),
    .Din1 (3'b111),
    .Dout (DR_MUX_out)
);

//mux2_1 mar_mux (

//    .sel  (1'b1),
//    .Din0 ({8'b0 , ir[7:0]}),
//    .Din1 (mem_rdata),
//    .Dout (Mioen_out)
//);

mux4_1 ADDR2MUX(

    .sel  (ADDR2_MUX_select),
    .Din0 (16'b0),
    .Din1 ({{10{ir[5]}}, ir[5:0]}),
    .Din2 ({{7{ir[8]}}, ir[8:0]}),
    .Din3 ({{6{ir[10]}}, ir[10:0]}),
    .Dout (ADDR2_MUX_OUT)
);


mux2_1 ADDR1MUX (
    .sel (ADDR1_MUX_select),
    .Din0 (pc),
    .Din1 (ADDR1_from_SR1), // ADD THE OUTPUT SR1 FROM THE REG FILE HERE.
    .Dout (ADDR1_MUX_OUT)
);

mux2_1 SR2_MUX (
    .sel (SR2_MUX_select),
    .Din0 (SR2_Reg_In),
    .Din1 ({{12{ir[4]}}, ir[4:0]}),
    .Dout (SR2_MUX_out)
);



register_unit my_reg(

    .clk(clk),
    .reset (reset),
    .data_in (reg_in_bus),
    .DR_reg (DR_reg_unit),
```

```
    .SR1_reg(SR1_reg_unit),
    .SR2_reg(SR2_reg_unit),
    .ld_reg(ld_reg),
    .SR1_OUT_reg(SR1_OUT_reg_unit),
    .SR2_OUT_reg(SR2_OUT_reg_unit)
);

NZP nzp_control (

   .clk (clk),
   .reset (reset),
   .data_in (BUS_TO_NZP),
   .ld_cc (ld_cc),
   .N (N_out),
   .Z (Z_out) ,
   .P (P_out)

);

BEN_to_control BEN_logic (


    .clk (clk),
    .ld_ben (ld_ben),
    .N (N_in),
    .Z (Z_in),
    .P (P_in),
    .ir_ben (ir[11:9]),
    .ben (ben)
);

endmodule
```

Control Unit

```
//------------------------------------------------------------------------------
// Company:          UIUC ECE Dept.
// Engineer:         Stephen Kempf
//
// Create Date:    17:44:03 10/08/06
// Design Name:    ECE 385 Given Code - Incomplete ISDU for SLC-3
// Module Name:    Control - Behavioral
//
// Comments:
//    Revised 03-22-2007
//    Spring 2007 Distribution
//    Revised 07-26-2013
//    Spring 2015 Distribution
//    Revised 02-13-2017
//    Spring 2017 Distribution
//    Revised 07-25-2023
//    Xilinx Vivado
//        Revised 12-29-2023
```

```
//        Spring 2024 Distribution
//        Revised 6-22-2024
//        Summer 2024 Distribution
//        Revised 9-27-2024
//        Fall 2024 Distribution
//------------------------------------------------------------------------------

    module control (
        input logic                     clk,
        input logic                     reset,

        input logic  [15:0]     ir,
        input logic                     ben,

        input logic             continue_i,
        input logic             run_i,

        output logic            ld_mar,
        output logic            ld_mdr,
        output logic            ld_ir,
        output logic            ld_pc,
        output logic        ld_led,
        output logic        ld_reg,
        output logic        ld_ben,
        output logic        ld_cc,
        output logic        SR1_MUX_select,
        output logic        SR2_MUX_select,
        output logic        ADDR1_MUX_select,
        output logic        [1:0] ADDR2_MUX_select,
        output logic        DR_MUX_select,
        output logic        [1:0] ALUK,
        output logic        mio_en,
        output logic       [1:0] PCMUX_select,

        output logic            gate_pc,
        output logic            gate_mdr,
        output logic        gate_marmux,
        output logic        gate_alu,

        //You should add additional control signals according to the SLC-3 datapath design

        output logic            mem_mem_ena, // Mem Operation Enable
        output logic            mem_wr_ena  // Mem Write Enable
);

        enum logic [5:0] {
                halted,
                pause_ir1,
                pause_ir2,
                s_18,
                s_33_1,
                s_33_2,
                s_33_3,
                s_35,
                s_32,
                s_1_add,
                s_5_and,
                s_9_not,
                s_6_ldr,
```

```
        s_25_1,
        s_25_2,
        s_25_3,
        s_27,
        s_7_str,
        s_23,
        s_16_1,
        s_16_2,
        s_16_3,
        s_0_br,
        s_22,
        s_12_jmp,
        s_4_jsr,
        s_21
} state, state_nxt;   // Internal state logic


always_ff @ (posedge clk)
begin
        if (reset)
                state <= halted;
        else
                state <= state_nxt;
end

always_comb
begin

        // Default controls signal values so we don't have to set each signal
        // in each state case below (If we don't set all signals in each state,
        // we can create an inferred latch)
        ld_mar = 1'b0;
        ld_mdr = 1'b0;
        ld_ir = 1'b0;
        ld_pc = 1'b0;
        ld_led = 1'b0;
        ld_reg = 1'b0;
        ld_ben = 1'b0;
        ld_cc = 1'b0;

        gate_pc = 1'b0;
        gate_mdr = 1'b0;
        gate_marmux = 1'b0;
        gate_alu = 1'b0;

        SR1_MUX_select = 1'b0;
        SR2_MUX_select = 1'b0;
        ADDR1_MUX_select = 1'b0;
        ADDR2_MUX_select = 2'b0;
        ALUK = 2'b0;
        PCMUX_select = 2'b0;
        mio_en = 1'b1;
        DR_MUX_select = 1'b0;

        mem_mem_ena = 1'b0;
        mem_wr_ena = 1'b0;
```

```
                // Assign relevant control signals based on current state
                case (state)
                        halted: ;
                        s_18 :
                                begin
                                        gate_pc = 1'b1;
                                        ld_mar = 1'b1;
                                        PCMUX_select = 2'b00;
                                        ld_pc = 1'b1;
                                end
                        s_33_1, s_33_2, s_33_3 : //you may have to think about this as well to adapt to
ram with wait-states
                                begin
                                        mem_mem_ena = 1'b1;
                                        ld_mdr = 1'b1;
                                end
                        s_35 :
                                begin
                                        gate_mdr = 1'b1;
                                        ld_ir = 1'b1;
                                end
                        pause_ir1: ld_led = 1'b1;
                        pause_ir2: ld_led = 1'b1;
                        // you need to finish the rest of state output logic.....

                        s_32 :
                          begin
                             ld_ben = 1'b1;
                          end

                        s_7_str :
                          begin
                             SR1_MUX_select = 1'b1;
                             ADDR1_MUX_select = 1'b1;
                             ADDR2_MUX_select = 2'b01;
                             gate_marmux = 1'b1;
                             ld_mar = 1'b1;
                          end

                        s_23:
                         begin
                             SR1_MUX_select = 1'b0;
                             gate_alu = 1'b1;
                         mio_en = 1'b0;
                             ALUK = 2'b11;
                             ld_mdr = 1'b1;
                          end

                        s_16_1 , s_16_2,  s_16_3:
                         begin
                             mem_wr_ena = 1'b1;
                             mem_mem_ena = 1'b1;
                         end

                    s_1_add :
                       begin
                            ld_reg = 1'b1;
                            ld_cc = 1'b1;
                            gate_alu = 1'b1;
```

```
            DR_MUX_select = 1'b0;
            SR1_MUX_select = 1'b1;
            ALUK = 2'b00;
            SR2_MUX_select = ir[5];
        end

     s_5_and :
        begin
            ld_reg = 1'b1;
            ld_cc = 1'b1;
            gate_alu = 1'b1;
            DR_MUX_select = 1'b0;
            SR1_MUX_select = 1'b1;
            ALUK = 2'b01;
            SR2_MUX_select = ir[5];
        end

     s_9_not :
        begin
            ld_reg = 1'b1;
            ld_cc = 1'b1;
            gate_alu = 1'b1;
            DR_MUX_select = 1'b0;
            SR1_MUX_select = 1'b1;
            ALUK = 2'b10;
        end

     s_0_br:
        begin
        end

    s_22:
        begin
        ADDR1_MUX_select = 1'b0;
        ADDR2_MUX_select = 2'b10;
        PCMUX_select = 2'b01;
        ld_pc = 1'b1;
        end

    s_6_ldr:
        begin
           SR1_MUX_select = 1'b1;
                ADDR1_MUX_select = 1'b1;
                ADDR2_MUX_select = 2'b01;
                gate_marmux = 1'b1;
                ld_mar = 1'b1;
            end

    s_25_1 , s_25_2:
        begin
            mem_mem_ena = 1'b1;
        end

   s_25_3:
        begin
            mem_mem_ena = 1'b1;
            ld_mdr = 1'b1;
        end
```

```
                s_27:
                    begin

                    ld_cc = 1'b1;
                    gate_mdr = 1'b1;
                    ld_reg = 1'b1;

                    end

                 s_12_jmp:
                    begin
                    SR1_MUX_select = 1'b1;
                    gate_alu = 1'b1;
                    ALUK = 2'b11;
                    PCMUX_select = 2'b10;;
                    ld_pc = 1'b1;
                    end

                 s_4_jsr:
                    begin
                    ld_reg = 1'b1;
                    gate_pc = 1'b1;
                    DR_MUX_select = 1'b1;
                    end

                 s_21:
                    begin
                    ADDR1_MUX_select = 1'b0;
                    ADDR2_MUX_select = 2'b11;
                    PCMUX_select = 2'b01;
                    ld_pc = 1'b1;
                    end


                      default : ;
            endcase
    end


    always_comb
    begin
            // default next state is staying at current state
            state_nxt = state;

            unique case (state)
                    halted :
                            if (run_i)
                                    state_nxt = s_18;
                    s_18 :
                            state_nxt = s_33_1; //notice that we usually have 'r' here, but you
will need to add extra states instead
                    s_33_1 :                //e.g. s_33_2, etc. how many? as a hint, note that the
bram is synchronous, in addition,
                            state_nxt = s_33_2;   //it has an additional output register.
                    s_33_2 :
                            state_nxt = s_33_3;
                    s_33_3 :
                            state_nxt = s_35;
                    s_35 :
```

```
        state_nxt = s_32;
// pause_ir1 and pause_ir2 are only for week 1 such that TAs can see
// the values in ir.
pause_ir1 :
        if (continue_i)
                state_nxt = pause_ir2;
pause_ir2 :
        if (~continue_i)
                state_nxt = s_18;
// you need to finish the rest of state transition logic.....

s_32:
    if (ir[15:12] == 4'b0001)
    begin
     state_nxt = s_1_add;
    end

    else if (ir[15:12] == 4'b0101)
    begin
     state_nxt = s_5_and;
    end

    else if (ir[15:12] == 4'b1001)
    begin
     state_nxt = s_9_not;
    end

    else if (ir[15:12] == 4'b0110)
    begin
     state_nxt = s_6_ldr;
    end

    else if (ir[15:12] == 4'b0111)
    begin
     state_nxt = s_7_str;
    end

    else if (ir[15:12] == 4'b0100)
    begin
     state_nxt = s_4_jsr;
    end

    else if (ir[15:12] == 4'b1100)
    begin
     state_nxt = s_12_jmp;
    end

    else if (ir[15:12] == 4'b0000)
    begin
     state_nxt = s_0_br;
    end

    else if (ir[15:12] == 4'b1101)
    begin
     state_nxt = pause_ir1;
    end

    else
    begin
```

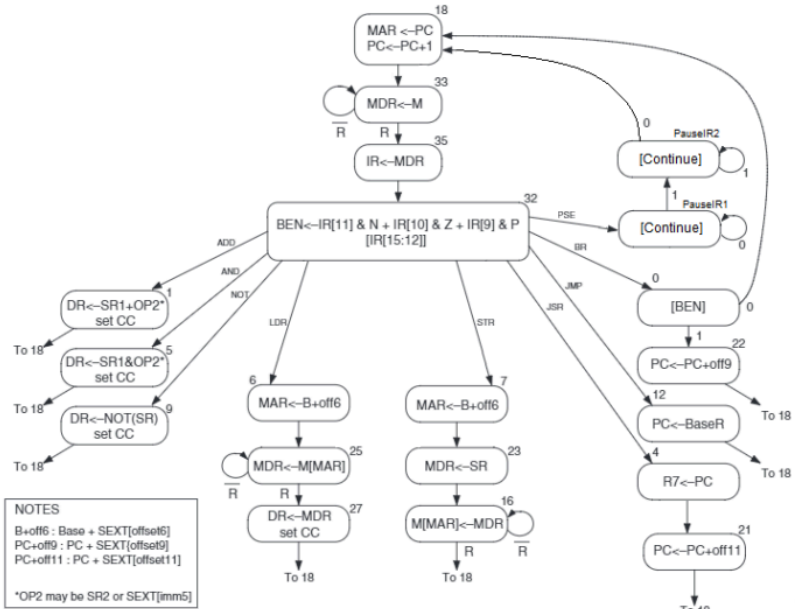
```
                    state_nxt = s_32;
                end

            s_1_add:
                state_nxt = s_18;
            s_5_and:
                state_nxt = s_18;
            s_9_not:
                state_nxt = s_18;
        s_6_ldr:
            state_nxt = s_25_1;
        s_25_1:
            state_nxt = s_25_2;
        s_25_2:
            state_nxt = s_25_3;
        s_25_3:
            state_nxt = s_27;
        s_27:
            state_nxt = s_18;
        s_7_str:
            state_nxt = s_23;
        s_23:
            state_nxt = s_16_1;
       s_16_1:
            state_nxt = s_16_2;
       s_16_2:
            state_nxt = s_16_3;
        s_16_3:
            state_nxt = s_18;
        s_0_br:
            if (ben)
            begin
              state_nxt = s_22;
            end
            else
            begin
              state_nxt = s_18;
            end
        s_22:
            state_nxt = s_18;
        s_12_jmp:
            state_nxt = s_18;
        s_4_jsr:
            state_nxt = s_21;
        s_21:
            state_nxt = s_18;
            default :;
    endcase
end

endmodule
```

Register Unit

```
module register_unit (
    input logic reset,
```

```
    input logic clk,
    input logic [15:0] data_in,
    input logic [2:0] DR_reg,
    input logic [2:0] SR1_reg,
    input logic [2:0] SR2_reg,
    input logic ld_reg,

    output logic [15:0] SR1_OUT_reg,
    output logic [15:0] SR2_OUT_reg
);



    logic [15:0] reg_file[8];

    always_comb
    begin

    SR1_OUT_reg = reg_file[SR1_reg];
    SR2_OUT_reg = reg_file[SR2_reg];

    end


    always_ff @(posedge clk) begin

    if (reset) begin
            // Reset all registers to 0
            for (int i = 0; i < 8; i++) begin
                reg_file[i] <= 16'b0;
            end
        end else if (ld_reg)
        begin
            // Write data to the destination register
            reg_file[DR_reg] <= data_in;
        end
    end

endmodule
```

State Diagram of Control Unit:

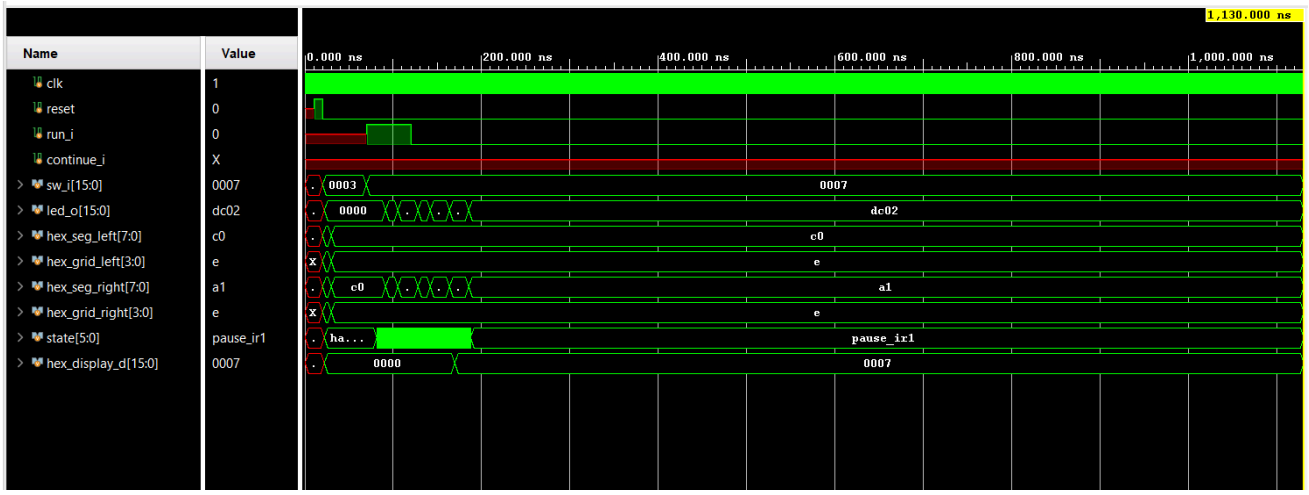


Simulations:

I/O Test 1:

In I/O test 1, we test functions AND, LDR, STR and BR. In this, we first set the switches on the board to be x0003, and then, it goes into I/O test 1 mode. After this, the value on the hex display is the same value as the switches. In order to see the instructions being executed, we can take a look at the types.sv file. For this instruction, these are the commands:

```
16'd0:   memContents =    instr_clr(r0)                ;       // Clear
16'd1:   memContents =    instr_ldr(r1, r0, inSW)      ;       // Load s
16'd2:   memContents =    instr_jmp(r1)                ;       // Jump t

                                                     // Basic I/O test 1
16'd3:   memContents =    instr_ldr(r1, r0, inSW)      ;       // Load s
16'd4:   memContents =    instr_str(r1, r0, outHEX)    ;       // Output
16'd5:   memContents =    instr_br(nzp, -3)            ;       // Repeat
```



As we can see here, the “hex_display_d” variable shows the value on the switches, x0007.

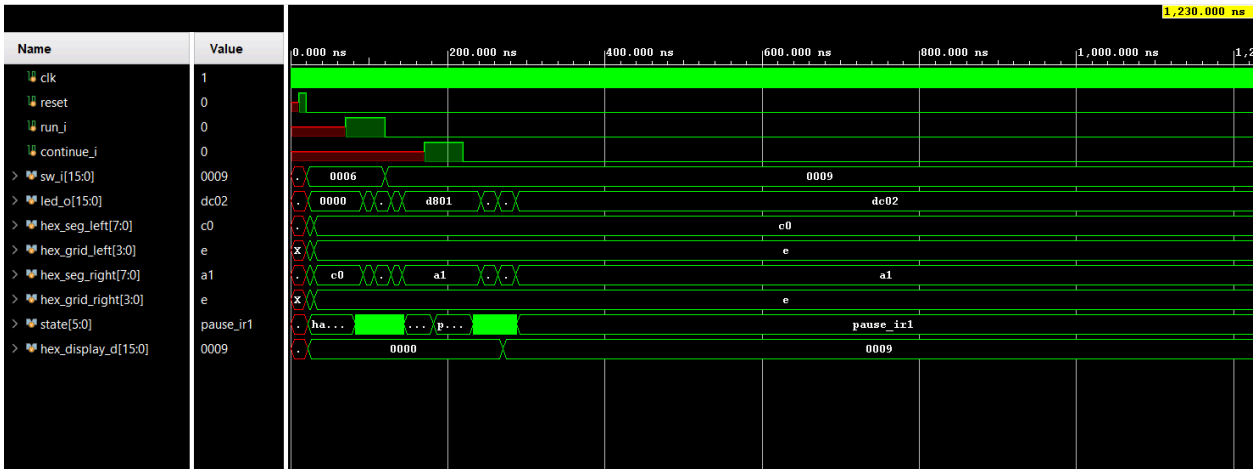
I/O Test 2:

In I/O test 2, we are testing all of the commands of I/O test 1, but we include the PSE state in testing. In order to do this, we should first set the switches to x0006, and then

upon each continue press, we see the value of the switches on the actual hex display. The code that is being used in order to test this is given in types.sv.

Here is the code being used:

```
16'd6:   memContents =    instr_pse(12'h801)          ;       //
Checkpoint 1 - prepare to input
            16'd7:   memContents =    instr_ldr(r1, r0, inSW)      ;
// Load switches
            16'd8:   memContents =    instr_str(r1, r0, outHEX)    ;
// Output
            16'd9:   memContents =    instr_pse(12'hC02)           ;
// Checkpoint 2 - read output, prepare to input
            16'd10:  memContents =    instr_br(nzp, -4)            ;
// Repeat
```



As we see here, the hex display takes up the value that is being inserted into the switches, which is x0009, upon the pressing of continue. Thus, our AND, LDR, STR, BR and PSE states do work as expected.

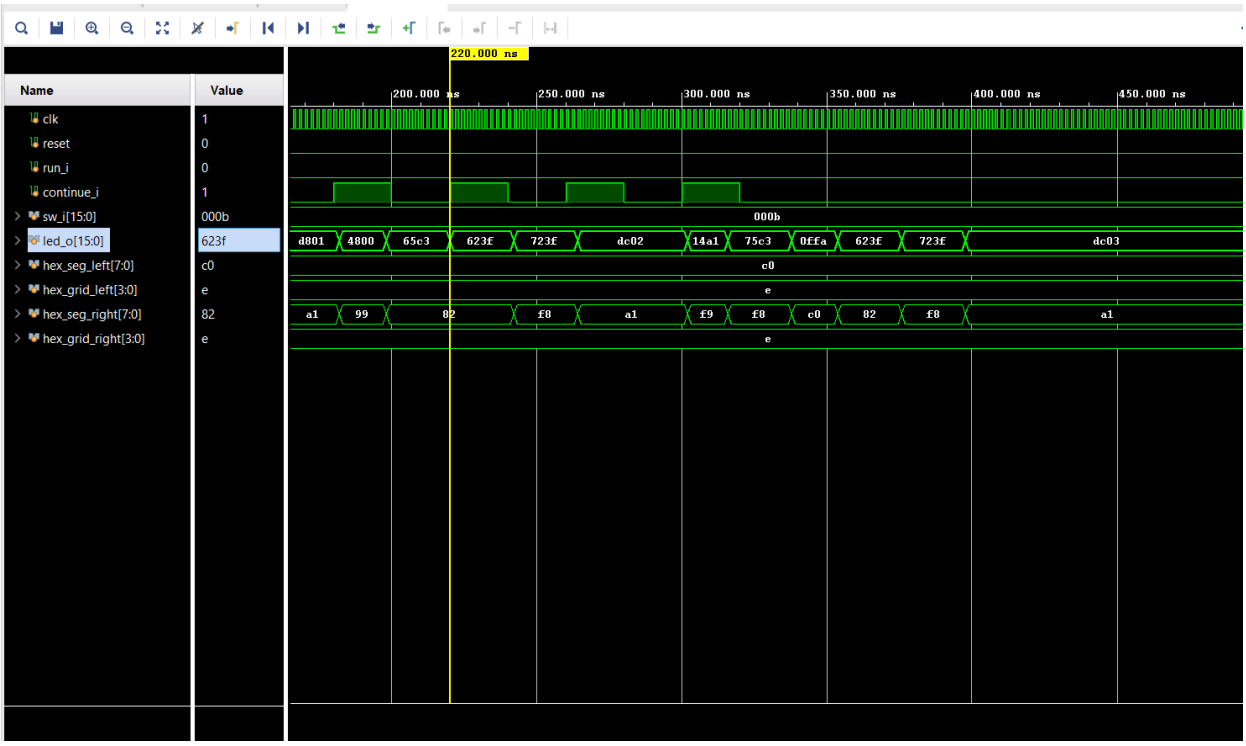
Self-Modifying Code Test:

In this test, we essentially check that the result is that with each iteration of the loop, the pause instruction will display a value on the LEDs one greater than the value in the last iteration. These are not displayed on the hex displays like IO test 1 and 2, but rather on the leds in the FPGA and the standard codes ran for testing in simulation are given in types.sv as shown below:

```
16'd11:  memContents =    instr_pse(12'h801)          ;       // Checkpoint 1 - prepare to input
           16'd12:  memContents =    instr_jsr(0)                ;       // Get PC address
           16'd13:  memContents =    instr_ldr(r2,r7,3)          ;       // Load pause instruction as data
           16'd14:  memContents =    instr_ldr(r1, r0, inSW)     ;       // Load switches
           16'd15:  memContents =    instr_str(r1, r0, outHEX)   ;       // Output
           16'd16:  memContents =    instr_pse(12'hC02)          ;       // Checkpoint 2 - read output, prepare to input
           16'd17:  memContents =    instr_inc(r2)               ;       // Increment checkpoint number
           16'd18:  memContents =    instr_str(r2,r7,3)          ;       // Store new checkpoint instruction (self-modifying code)
           16'd19:  memContents =    instr_br(nzp, -6)           ;       // Repeat

           16'd20:  memContents =    instr_clr(r0)               ;       // XOR test
           16'd21:  memContents =    instr_pse(12'h801)          ;       // Checkpoint 1 - prepare to input (upper)
           16'd22:  memContents =    instr_ldr(r1, r0, inSW)     ;       // Load switches
           16'd23:  memContents =    instr_pse(12'h802)          ;       // Checkpoint 2 - prepare to input (lower)
           16'd24:  memContents =    instr_ldr(r2, r0, inSW)     ;       // Load switches again
           16'd25:  memContents =    instr_not(r3, r1)           ;       // r3: A'
           16'd26:  memContents =    instr_and(r3, r3, r2)       ;       // r3: A'B
           16'd27:  memContents =    instr_not(r3, r3)           ;       // r3: (A'B)'
           16'd28:  memContents =    instr_not(r4, r2)           ;       // r4: B'
           16'd29:  memContents =    instr_and(r4, r4, r1)       ;       // r4: B'A
           16'd30:  memContents =    instr_not(r4, r4)           ;       // r4: (B'A)'
           16'd31:  memContents =    instr_and(r3, r3, r4)       ;       // r3: (A'B)'(B'A)'
           16'd32:  memContents =    instr_not(r3, r3)           ;       // r3: ((A'B)'(B'A)')' XOR using only and-not
           16'd33:  memContents =    instr_str(r3, r0, outHEX)   ;       // Output
           16'd34:  memContents =    instr_pse(12'h405)          ;       // Checkpoint 5 - read output
           16'd35:  memContents =    instr_br(nzp, -15)          ;       // Repeat
           16'd36:  memContents =    NO_OP                       ;       // Place Holder
           16'd37:  memContents =    NO_OP                       ;       // Place Holder
           16'd38:  memContents =    NO_OP                       ;       // Place Holder
           16'd39:  memContents =    NO_OP                       ;       // Place Holder
           16'd40:  memContents =    NO_OP                       ;       // Place Holder
           16'd41:  memContents =    NO_OP                       ;       // Place Holder

           16'd42:  memContents =    instr_clr(r0)               ;       // Run once test (also for JMP)
           16'd43:  memContents =    instr_clr(r1)               ;       // clear r1
           16'd44:  memContents =    instr_jsr(0)                ;       // get jumpback address
           16'd45:  memContents =    instr_str(r1, r0, outHEX)   ;       // output r1; LOOP DEST
           16'd46:  memContents =    instr_pse(12'h401)          ;       // Checkpoint 1 - read output
           16'd47:  memContents =    instr_inc(r1)               ;       // increment r1
           16'd48:  memContents =    instr_ret()                 ;       // repeat

           16'd49:  memContents =    instr_clr(r0)               ;       // Multiplier Program
           16'd50:  memContents =    instr_jsr(0)                ;       // r7 <- PC (for loading bit test mask)
           16'd51:  memContents =    instr_ldr(r3, r7, 22)       ;       // load mask;
           16'd52:  memContents =    instr_clr(r4)               ;       // clear r4 (iteration tracker),  ; START
           16'd53:  memContents =    instr_clr(r5)               ;       // r5 (running total)
           16'd54:  memContents =    instr_pse(12'h801)          ;       // Checkpoint 1 - prepare to input
           16'd55:  memContents =    instr_ldr(r1, r0, inSW)     ;       // Input operand 1
           16'd56:  memContents =    instr_pse(12'h802)          ;       // Checkpoint 2 - prepare to input
           16'd57:  memContents =    instr_ldr(r2, r0, inSW)     ;       // Input operand 2
           16'd58:  memContents =    instr_add(r5, r5, r5)       ;       // shift running total; LOOP DEST
           16'd59:  memContents =    instr_and(r7, r3, r1)       ;       // apply mask
           16'd60:  memContents =    instr_br(z, 1)              ;       // test bit and jump over...
           16'd61:  memContents =    instr_add(r5, r5, r2)       ;       // ... the addition
           16'd62:  memContents =    instr_addi(r4, r4, 0)       ;       // test iteration == 0 (first iteration)
           16'd63:  memContents =    instr_br(p,2)               ;       // if not first iteration, jump over negation
           16'd64:  memContents =    instr_not(r5, r5)           ;       // 2's compliment negate r5
           16'd65:  memContents =    instr_inc(r5)               ;       //   (part of above)
           16'd66:  memContents =    instr_inc(r4)               ;       // increment iteration
           16'd67:  memContents =    instr_add(r1, r1, r1)       ;       // shift operand 1 for mask comparisons
           16'd68:  memContents =    instr_addi(r7, r4, -8)      ;       // test for last iteration
           16'd69:  memContents =    instr_br(n, -12)            ;       // branch back to LOOP DEST if iteration < 7
           16'd70:  memContents =    instr_str(r5, r0, outHEX)   ;       // Output result
           16'd71:  memContents =    instr_pse(12'h403)          ;       // Checkpoint 3 - read output
           16'd72:  memContents =    instr_br(nzp, -21)          ;       // loop back to start
           16'd73:  memContents =    16'h0080                    ;       // bit test mask

           16'd74:  memContents =    16'h00ef                    ;       // Data for Bubble Sort
           16'd75:  memContents =    16'h001b                    ;
           16'd76:  memContents =    16'h0001                    ;
           16'd77:  memContents =    16'h008c                    ;
           16'd78:  memContents =    16'h00db                    ;
           16'd79:  memContents =    16'h00fa                    ;
           16'd80:  memContents =    16'h0047                    ;
```

```
16'd81:  memContents =    16'h0046                  ;
16'd82:  memContents =    16'h001f                  ;
16'd83:  memContents =    16'h000d                  ;
16'd84:  memContents =    16'h00b8                  ;
16'd85:  memContents =    16'h0003                  ;
16'd86:  memContents =    16'h006b                  ;
16'd87:  memContents =    16'h004e                  ;
16'd88:  memContents =    16'h00f8                  ;
16'd89:  memContents =    16'h0007                  ;
16'd90:  memContents =    instr_clr(r0)             ;       // Bubblesort Program start
16'd91:  memContents =    instr_jsr(0)              ;
16'd92:  memContents =    instr_addi(r6, r7, -16)   ;       // Store data location in r6
16'd93:  memContents =    instr_addi(r6, r6, -2)    ;       //   (data location is 18 above the address from JSR)
16'd94:  memContents =    instr_pse(12'h3FF)        ;       // Checkpoint -1 - select function; LOOP DEST
16'd95:  memContents =    instr_ldr(r1, r0, inSW)   ;
16'd96:  memContents =    instr_br(z, -3)           ;       // If 0, retry
16'd97:  memContents =    instr_dec(r1)             ;
16'd98:  memContents =    instr_br(np, 2)           ;       // if selection wasn't 1, jump over
16'd99:  memContents =    instr_jsr(9)              ;       //   ...call to entry function
16'd100: memContents =    instr_br(nzp, -7)         ;
16'd101: memContents =    instr_dec(r1)             ;
16'd102: memContents =    instr_br(np, 2)           ;       // if selection wasn't 2, jump over
16'd103: memContents =    instr_jsr(15)             ;       //   ...call to sort function
16'd104: memContents =    instr_br(nzp, -11)        ;
16'd105: memContents =    instr_dec(r1)             ;
16'd106: memContents =    instr_br(np, -13)         ;       // if selection wasn't 3, retry
16'd107: memContents =    instr_jsr(29)             ;       //   call to display function
16'd108: memContents =    instr_br(nzp, -15)        ;       // repeat menu
16'd109: memContents =    instr_clr(r1)             ;       // ENTRY FUNCTION
16'd110: memContents =    instr_str(r1, r0, outHEX) ;       // r5 is temporary index into data; r1 is counter; LOOP DEST
16'd111: memContents =    instr_pse(12'hC01)        ;       // Checkpoint 1 - read data (index) and write new value
16'd112: memContents =    instr_ldr(r2, r0, inSW)   ;
16'd113: memContents =    instr_add(r5, r6, r1)     ;       // generate pointer to data
16'd114: memContents =    instr_str(r2, r5, 0)      ;       // store data
16'd115: memContents =    instr_inc(r1)             ;       // increment counter
16'd116: memContents =    instr_addi(r3, r1, -16)   ;       // test for counter == 16
16'd117: memContents =    instr_br(n, -8)           ;       // less than 16, repeat
16'd118: memContents =    instr_ret()               ;       // ENTRY FUNCTION RETURN
16'd119: memContents =    instr_addi(r1, r0, -16)   ;       // i = -16; SORT FUNCTION
16'd120: memContents =    instr_addi(r2, r0, 1)     ;       // j = 1; OUTER LOOP DEST
16'd121: memContents =    instr_add(r3, r6, r2)     ;       // generate pointer to data; INNER LOOP DEST
16'd122: memContents =    instr_ldr(r4, r3, -1)     ;       // r4 = data[j-1]
16'd123: memContents =    instr_ldr(r5, r3, 0)      ;       // r5 = data[j]
16'd124: memContents =    instr_not(r5, r5)         ;
16'd125: memContents =    instr_addi(r5, r5, 1)     ;       // r5 = -data[j]
16'd126: memContents =    instr_add(r5, r4, r5)     ;       // r5 = data[j-1]-data[j]
16'd127: memContents =    instr_br(nz, 3)           ;       // if data[j-1] > data[j]
16'd128: memContents =    instr_ldr(r5, r3, 0)      ;       // { r5 = data[j]
16'd129: memContents =    instr_str(r5, r3, -1)     ;       //   data[j-1] = data[j]
16'd130: memContents =    instr_str(r4, r3, 0)      ;       //   data[j] = r4 } // old data[j-1]
16'd131: memContents =    instr_inc(r2)             ;
16'd132: memContents =    instr_add(r3, r1, r2)     ;       // Compare i and j
16'd133: memContents =    instr_br(n, -13)          ;       // INNER LOOP BACK
16'd134: memContents =    instr_inc(r1)             ;
16'd135: memContents =    instr_br(n, -16)          ;       // OUTER LOOP BACK
16'd136: memContents =    instr_ret()               ;       // SORT FUNCTION RETURN
16'd137: memContents =    instr_clr(r1)             ;       // DISPLAY FUNCTION
16'd138: memContents =    instr_add(r4, r7, r0)     ;       // JSR shuffle to get PC value in r5
16'd139: memContents =    instr_jsr(0)              ;
16'd140: memContents =    instr_add(r5, r7, r0)     ;
16'd141: memContents =    instr_add(r7, r4, r0)     ;       // shuffle done
16'd142: memContents =    instr_ldr(r3, r5, 15)     ;       // r3 = op_PSE(12'b802)
16'd143: memContents =    instr_addi(r2, r0, 8)     ;
16'd144: memContents =    instr_add(r2, r2, r2)     ;       // r2 = 16
16'd145: memContents =    instr_add(r4, r6, r1)     ;       // generate pointer to data; LOOP DEST
16'd146: memContents =    instr_ldr(r4, r4, 0)      ;       // load data
16'd147: memContents =    instr_str(r4, r0, outHEX) ;       // display data
16'd148: memContents =    instr_pse(12'h802)        ;       // Checkpoint 2 - read data (self-modified instruction)
16'd149: memContents =    instr_add(r3, r3, r2)     ;       // modify register with code
16'd150: memContents =    instr_str(r3, r5, 8)      ;       // store modified code
16'd151: memContents =    instr_inc(r1)             ;       // increment counter
16'd152: memContents =    instr_addi(r4, r1, -16)   ;       // test for counter == 16
16'd153: memContents =    instr_br(n, -9)           ;       // less than 16, repeat
16'd154: memContents =    instr_ret()               ;       // DISPLAY FUNCTION RETURN
16'd155: memContents =    instr_pse(12'h802)                 ;
```

As we can see, the last 2 digits of the led keep increasing from 01 to 02 to 03 as we press continue. This will keep continuing as we keep pressing continue but we have only pressed it 3 times since the led were preset to 01.

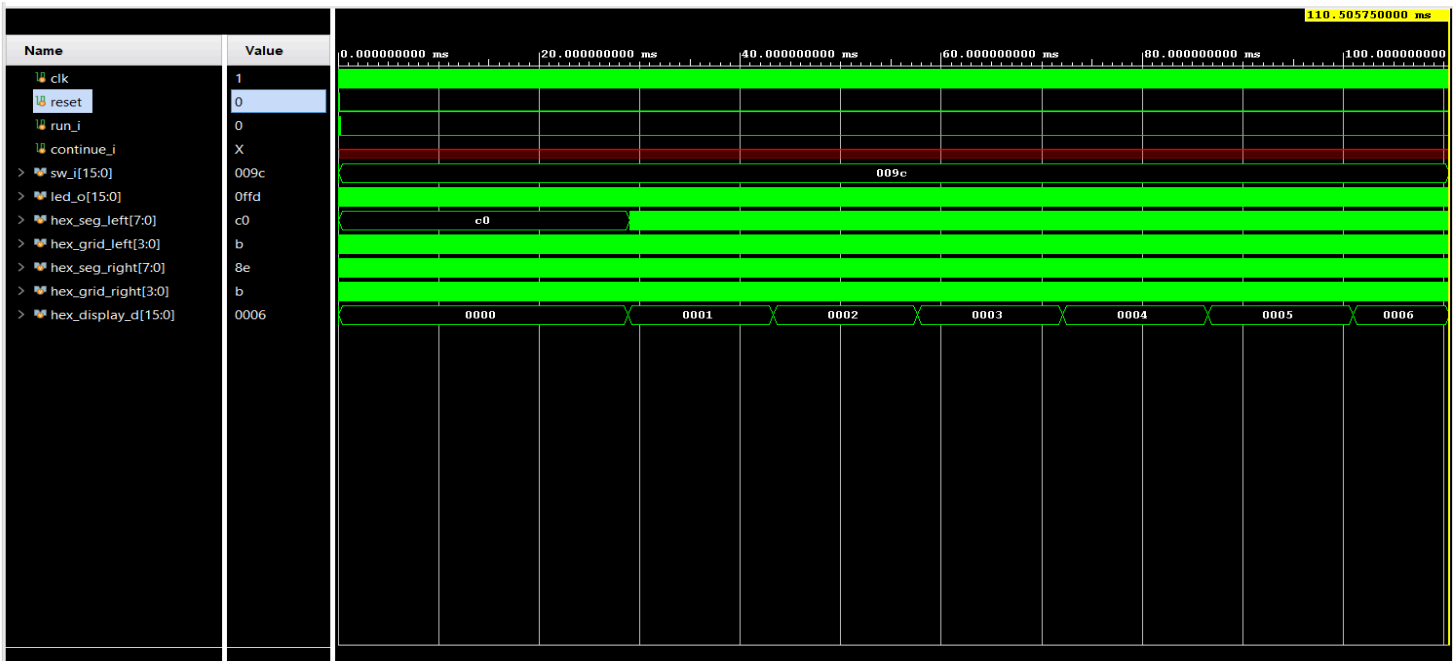


Auto Counting Test

The auto counting test essentially keeps counting on the hex displays as soon as we set the switches to x009C and hit run. Once we click on run, we should see all of the hex displays keep increasing in value until the maximum 4 bit hex value be attained, and then it resets back to 0 and keeps counting. The code being run in the backend in types.sv for this is shown below:

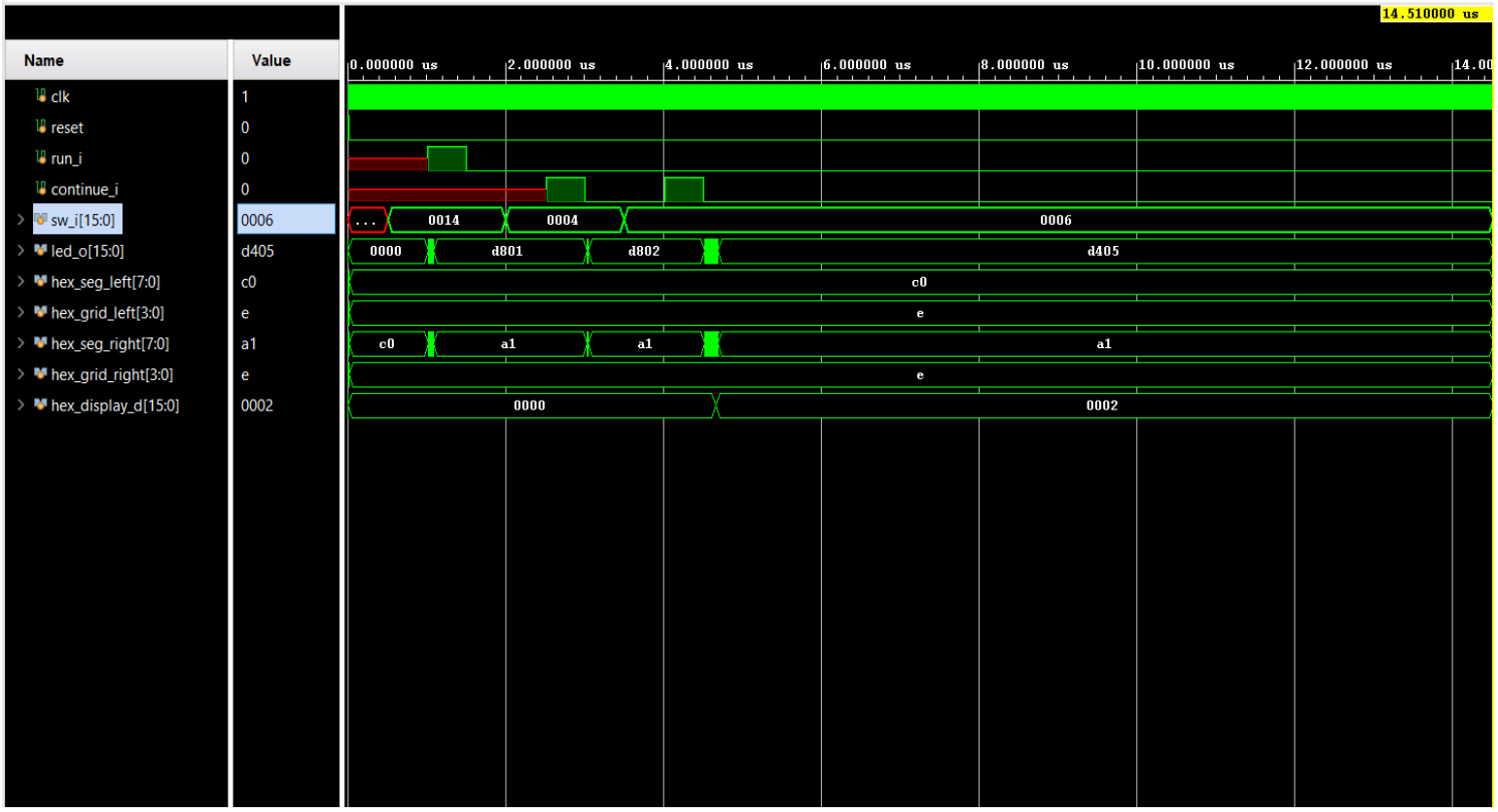
```
16'd156: memContents =    instr_clr(r0)              ;       // r0 = 0
16'd157: memContents =    instr_clr(r1)              ;       // r1 = 0  (r1 will be used as loop counter 1)
16'd158: memContents =    instr_clr(r2)              ;       // r2 = 0  (r2 will be used as loop counter 2)
16'd159: memContents =    instr_clr(r3)              ;       // r3 = 0  (r3 will be displayed to hex displays)
16'd160: memContents =    instr_jsr(0)               ;       // r7 <- PC = 161 (161 because PC <- PC+1 after fetch)
// INIT:      (PC = 161)
16'd161: memContents =    instr_ldr(r1, r7, 12)      ;       // r1 <- xFFFF
16'd162: memContents =    instr_ldr(r2, r7, 13)      ;       // r2 <- d5
// 1ST LOOP:  (PC = 163)
16'd163: memContents =    instr_dec(r1)              ;       // Decrement first loop counter
16'd164: memContents =    instr_br(z, 1)             ;       // (Go to 2ND LOOP) - r1 =  0, go to second loop
16'd165: memContents =    instr_br(nzp, -3)          ;       // (Go to 1ST LOOP) - r1 != 0, repeat first loop
```

```
// 2ND LOOP:  (PC = 166)
16'd166: memContents =    instr_dec(r2)               ;       // Decrement second loop counter
16'd167: memContents =    instr_br(z, 2)              ;       // (Go to DISPLAY) -  r2 = 0, show new number on hex displays
16'd168: memContents =    instr_ldr(r1, r7, 12)       ;       // r1 <- xFFFF (reset loop 1 counter)
16'd169: memContents =    instr_br(nzp, -7)           ;            // (Go to 1ST LOOP) - r2 != 0, repeat first loop
// DISPLAY:   (PC = 170)
16'd170: memContents =    instr_str(r3, r0, outHEX)   ;       // Display counter to hex display
16'd171: memContents =    instr_inc(r3)               ;
16'd172: memContents =    instr_br(nzp, -12)          ;       // (Go to INIT) - Repeat double for loop counting
// CONSTANTS: (PC = 173)
16'd173: memContents =    16'hFFFF                    ;      // Constant for loading into r1 for counting/delay purposes xFFF
16'd174: memContents =    16'd5                       ;      // Constant for loading into r2 for counting/delay purposes d750
```



XOR Test:

With the 2 presses of continue, we XOR x0006 and x0004 and the hex_display outputted the correct result of x0002.



Multiplication Test:

With 2 presses of continue we multiplied x0004 with x0003 and the correct result of x000E is displayed on hex_display.

Sort Test:

After pressing run and sorting, with each press of continue we can see the values stored being displayed on hex_display in increasing order.

Post lab questions:

Answer at least the following questions in the lab report
•What is CPU_TO_IO used for, i.e., what is its main function?

What is the difference between BR and JMP instructions?

What is the purpose of the R signal in Patt and Patel? How do we compensate for the lack
of the signal in our design? What impact does this have on performance?

Answer:

1. Table:

LUT	424
DSP	0
Memory(BRAM)	0.5
Flip-flop	339
Latches*	0
Frequency	0.101 Mhz
Static Power	0.074 W
Dynamic Power	0.015 w
Total Power	0.089 W

The CPU_TO_IO serves as the critical interface linking the FPGAs memory system with external components such as switches and displays. Specifically, the address `0xFFFF` is designated for reading input from the switches, allowing the CPU to gather real-time user inputs. Additionally, when the CPU writes to the instruction located at `0xFFFF`, it sends the appropriate values to the hexadecimal display for output, effectively rendering the results visible to users. For memory operations involving the Load (LDR) and Store (STR) instructions—where the addresses differ from `0xFFFF`—the FPGA utilizes `CPU_TO_IO` to facilitate reading from and writing to memory, ensuring data is stored and retrieved correctly.

Difference Between BR and JMP Instructions: The JMP (Jump) instruction alters the Program Counter (PC) to point to a subroutine address. Upon executing the subroutine, the RET (Return) instruction is invoked, which reverts the PC to its original value, enabling the CPU to continue executing subsequent instructions from that point. In contrast, the BR (Branch) instruction modifies the PC conditionally based on the status flags (negative, zero, positive) set by previous operations. Unlike JMP, the PC may only change with BR if the specified condition is satisfied. Moreover, the change in the PC for BR can be of a smaller magnitude compared to that of JMP, and it does not inherently return to the initial point of execution, which can affect the flow of program control in a more flexible manner.

Purpose of the R Signal and Compensation Strategy: The R signal is crucial in signaling the completion of memory access, indicating that data is ready for processing. Without this signal in our design, we compensate by implementing three wait states for memory access. This delay ensures that the memory system is adequately prepared before the CPU proceeds to the next state in the execution sequence. However, this workaround can negatively impact performance, as it introduces additional latency into the system. The CPU may spend extra cycles waiting for data readiness, which could lead to slower overall processing speeds, especially in memory-intensive applications where rapid data access is essential.

Conclusion:

A simple microprocessor based on the LC-3 ISA showed, in a simplified form, some of the most relevant stages of instruction processing, namely FETCH, DECODE, and EXECUTE. This design was able to overcome the challenges presented during design, and the results from simulation show that the operations were correct. This project has improved our knowledge with respect to principles of microprocessor design and the necessity of handling control signals in synchronous systems.