ECE 385

Fall 2021

Experiment # 6

Simple Computer SLC-3.2 in SystemVerilog

Xu Ke / Zhu Xiaohan

LA4/Thursday & 18:00-20:50 Huang Tianhao

I. Introduction

In this experiment, we overall design a simple microprocessor using SystemVerilog. It consists of the LC-3 ISA, a 16-bit processor with 16-bit Program Counter, 16-bit instructions, and 16-bit registers.

More specifically, in week 1,we implement the FETCH instruction both in simulation and on the FPGA, which includes loading PC into MAR and loading MDR into IR. And in week 2, we implement the rest of the SLC3, including the DECODE instruction, which will check for the BEN bit and move to other parts of the state machine based on the opcode. Besides, we also implement the EXECUTE instruction: ADD/ADDi, AND/ANDi, NOT, BR, JMP, JSR, LDR, STR or PAUSE.

II. Written Description and Diagrams of SLC-3

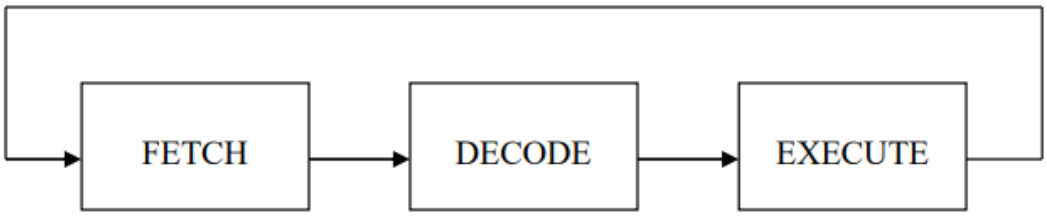
a. Summary of Operation

LC-3 is an assembly language created by the dynamic duo Yale N Patel and Sanjay J Patel. The SLC-3 that we create in this experiment differs from the original LC-3 for simplicity but also uses FETCH to get instructions, decodes them, and executes. The difference is that the SLC-3 lacks some instructions such as TRAP, STI, and there is no use of R signal. We deal with the lack of our ready signal R by adding more states to wait until the correct time based on the worst possible case. The reason for the lack of the R signal is due to the fact that the SRAM in use does not support it.

ADD Adds the contents of SR1 and SR2 and stores the result to DR. Sets the status register.
ADDi Add Immediate. Adds the contents of SR to the sign-extended value imm5 and stores the result to DR. Sets the status register.
AND ANDs the contents of SR1 with SR2 and stores the result to DR. Sets the status register.
ANDi And Immediate. ANDs the contents of SR with the sign-extended value imm5 and stores the result to DR. Sets the status register.
NOT Negates SR and stores the result to DR. Sets the status register.
BR Branch. If any of the condition codes match the condition stored in the status register, takes the branch; otherwise, continues execution. (An unconditional jump can be specified by setting NZP to 111.) Branch location is determined by adding the sign-extended PCoffset9 to the PC.
JMP Jump. Copies memory address from BaseR to PC.
JSR Jump to Subroutine. Stores current PC to R(7), adds sign-extended PCoffset11 to PC.
LDR Load using Register offset addressing. Loads DR with memory contents pointed to by (BaseR + SEXT(offset6)). Sets the status register.
STR Store using Register offset addressing. Stores the contents of SR at the memory location pointed to by (BaseR + SEXT(offset6)).
PAUSE Pauses execution until Continue is asserted by the user. Execution should only unpause if Continue is asserted during the current pause instruction; that is, when multiple

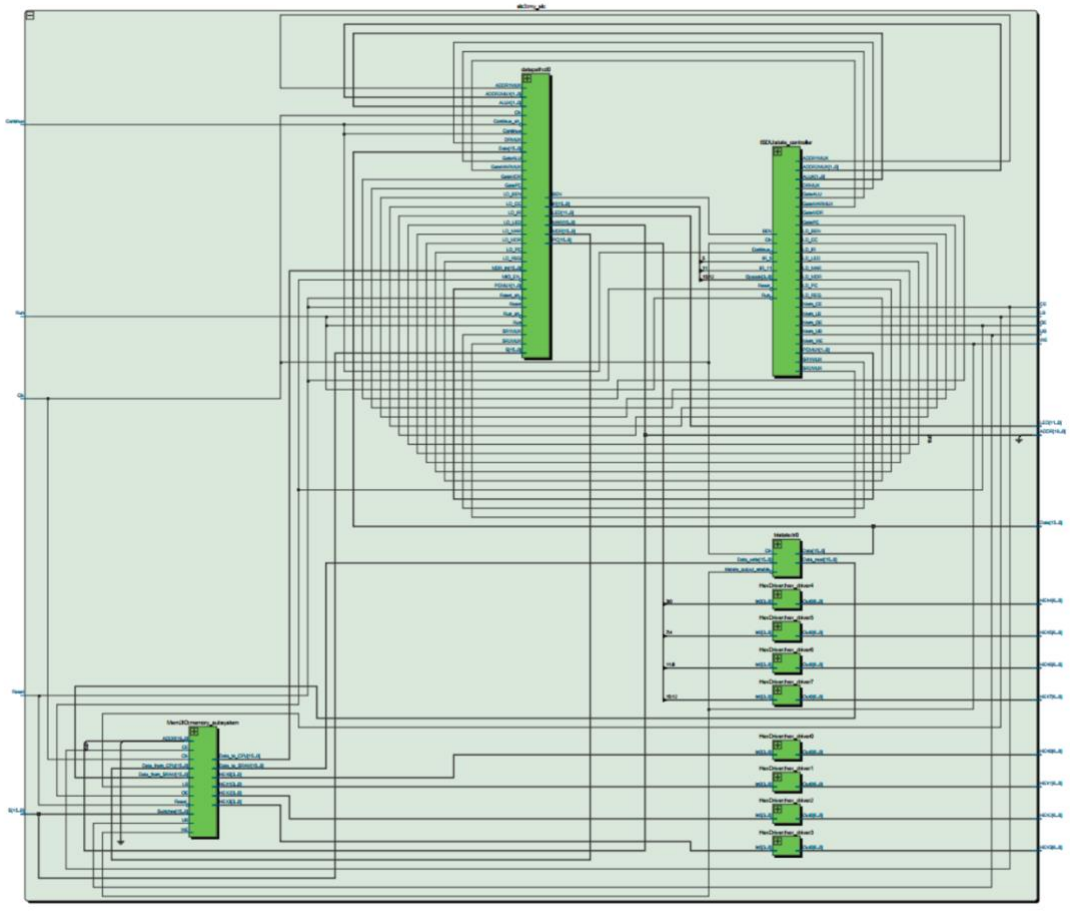
pause instructions are encountered, only one should be “cleared” per press of Continue. While paused, ledVect12 is displayed on the board LEDs. See I/O Specification section for usage notes.

b. Performance of Function



In our design, instructions are FETCHed from memory and stored in the IR, from which we DECODE the opcode and thus go to the specified next state from the OPCODE. The OPCODE is responsible for choosing between the following operations: ADD/ADDi, AND/ANDi, NOT, BR, JMP, JSR, LDR, STR or PAUSE. The EXECUTE cycle is responsible for actually performing the operation, then looping back to initial state. In each state the FSM outputs control signals will get inputted into the various components such as muxes, registers, etc.

c. Block Diagram of slc3.sv



d. Written Description of all .sv modules

1) slc3.sv

```
//
module slc3(
    input logic [15:0] S,
    input logic Clk, Reset, Run, Continue,
    output logic [11:0] LED,
    output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, HEX6, HEX7,
    output logic CE, UB, LB, OE, WE,
    output logic [19:0] ADDR,
    inout wire [15:0] Data //tristate buffers need to be of type wire
);

// Declaration of push button active high signals
logic Reset_ah, Continue_ah, Run_ah;

//assign Reset_ah = ~Reset;
//assign Continue_ah = ~Continue;
//assign Run_ah = ~Run;

sync reset_sync (Clk, ~Reset, Reset_ah);
sync continue_sync (Clk, ~Continue, Continue_ah);
sync run_sync (Clk, ~Run, Run_ah);


// Internal connections
logic BEN;
logic LD_MAR, LD_MDR, LD_IR, LD_BEN, LD_CC, LD_REG, LD_PC, LD_LED;
logic GatePC, GateMDR, GateALU, GateMARMUX;
logic [1:0] PCMUX, ADDR2MUX, ALUK;
logic DRMUX, SR1MUX, SR2MUX, ADDR1MUX;
logic MIO_EN;

logic [15:0] MDR_In;
logic [15:0] MAR, MDR, IR, PC;
logic [15:0] Data_from_SRAM, Data_to_SRAM;

// Signals being displayed on hex display
logic [3:0][3:0] hex_4;
```

Input: [15:0] S, Clk, Reset, Run, Continue

Output: [11:0] LED, [6:0] HEX0, [6:0] HEX1, [6:0] HEX2, [6:0] HEX3, [6:0] HEX4, [6:0] HEX5, [6:0] HEX6, [6:0] HEX7, CE, UB, LB, OE, WE, [19:0] ADDR

Inouts: [15:0] Data

Description: This module is the main part of SLC-3 processor, in which contain all the sub modules that included in the processor, including datapath, Mem2IO, ISDU, etc. It will receive the input keys and switches to control the running of processor, and the output will be connected to the out-ship memory with data transmission.

Purpose: Connect the modules together and route the data and signals.

2) AlU.sv

```
module ALU(
    input logic [15:0] SR1OUT, SR2OUT,IMME,
    input logic sr2mux,
    input logic [1:0] ALUK,
    output logic [15:0] OUT
);
    logic [15:0] B, out_add,out_and,out_not;
    ripple_adder adder(.A(SR1OUT),.B(B),.Sum(out_add),.CO());
    assign B = sr2mux ? IMME : SR2OUT ;
    assign out_and = B & SR1OUT;
    assign out_not = ~SR1OUT;

    always_comb
    begin
        unique case (ALUK)
            3'b00:  OUT = out_add;
            3'b01:  OUT = out_and;
            3'b10:  OUT = out_not;
            3'b11:  OUT = SR1OUT;
        endcase
    end

endmodule
```

Inputs: [15:0] SR1OUT, SR2OUT, IMME [1:0] ALUK Sr2mux
Outputs: [15:0] OUT
Description: This module receives the outputs of register file and immediate number, ALUK control signal as input and the output is the result of the ALU calculation module.
Purpose: This module generates the arithmetic unit of SLC-3 processor.

3) ADDRMUX.sv

```
module ADDRMUX(
    input logic [15:0] IR,
    input logic [1:0] ADDR2MUX,
    input logic ADDR1MUX,
    input logic [15:0] data_from_SR1OUT, data_from_PC,
    output logic [15:0] Data_Out, SEXT,
    output logic [3:0] Data_to_controller,
    output logic [2:0] Data_to_BEN
);

    logic [15:0] MUXOUT4to1, MUXOUT2to1, ZERO, offset6, offset9, offset11;

    ripple_adder adder(.A(MUXOUT4to1), .B(MUXOUT2to1), .Sum(Data_Out), .CO());
    mux2to1bit16 MUX1(.Din1(data_from_PC), .Din2(data_from_SR1OUT), .select(ADDR1MUX), .Dout(MUXOUT2to1));
    mux4to1bit16 MUX2(.Din1(ZERO), .Din2(offset6), .Din3(offset9), .Din4(offset11), .select(ADDR2MUX), .Dout(MUXOUT4to1));

    assign ZERO = 16'b0;

    always_comb
    begin
        if (IR[5] == 1'b0)
            offset6 = {10'b0, IR[5:0]};
        else
            offset6 = {10'b1111111111, IR[5:0]};
    end

    always_comb
    begin
        if (IR[8] == 1'b0)
            offset9 = {7'b0, IR[8:0]};
        else
            offset9 = {7'b1111111, IR[8:0]};
    end

    always_comb
    begin
        if (IR[10] == 1'b0)
            offset11 = {5'b0, IR[10:0]};
        else
            offset11 = {5'b11111, IR[10:0]};
    end

    always_comb
    begin
        if (IR[4] == 1'b0)
            SEXT = {11'b0, IR[4:0]};
        else
            SEXT = {11'b11111111111, IR[4:0]};
    end

    assign Data_to_controller = IR[15:12];
    assign Data_to_BEN = IR[11:9];

endmodule
```

Inputs: [15:0] IR, data_from_SR1OUT, data_from_PC [1:0] ADDR2MUX ADDR1MUX
Outputs:[15:0] Data_Out, SEXT [3:0] Data_to_controller [2:0] Data_to_BEN
Description: This module receives inputs from ALU, IR, PC, and control signals from state machine. And then it will decode the IR to generate the immediate value with signal extension and data to control unit and BEN module.
Purpose: This module decodes the IR instruction to get values that as the input to Ben module and controller.

4) Ben.sv

```
module ben_module(
    input logic Clk,reset,
    input logic[15:0] BUS,
    input logic LD_CC, LD_BEN,
    input logic [2:0] IR11_9,
    output logic BEN
);
    logic [2:0] NZP;
    logic n_out,z_out,p_out,ben_input;
    uni_reg #(.N(1)) N(.Clk(Clk),.Load(LD_CC),.reset(reset),.D(NZP[2]),.Data_Out(n_out));
    uni_reg #(.N(1)) Z(.Clk(Clk),.Load(LD_CC),.reset(reset),.D(NZP[1]),.Data_Out(z_out));
    uni_reg #(.N(1)) P(.Clk(Clk),.Load(LD_CC),.reset(reset),.D(NZP[0]),.Data_Out(p_out));
    assign ben_input = IR11_9[2]&n_out | IR11_9[1]&z_out | IR11_9[0]&p_out;
    uni_reg #(.N(1)) Ben(.Clk(Clk),.Load(LD_BEN),.reset(reset),.D(ben_input),.Data_Out(BEN));

    always_comb
    begin
      if(BUS == 16'b0)
         NZP =  3'b010;
      else if(BUS[15] == 1'b1)
         NZP = 3'b100;
      else
         NZP = 3'b001;
    end

endmodule
```

Inputs: [15:0] BUS Clk, reset, LD_CC, LD_BEN[2:0] IR11_9

Outputs: BEN

Description: Based on the value from BUS, this module will have the corresponding conditional code stored in the registers. And this module will generate the corresponding BEN value based on the conditional condes and the control signal and the BEN value will be used for judging whether jump the PC value or not.

Purpose: This module generates Conditional codes and BEN value for BR instruction.

5) datapath.sv

```
module datapath(
    input logic Clk,
    input logic Reset_ah, Run_ah,
    input logic LD_MAR,LD_MDR,LD_IR,LD_BEN,LD_CC,LD_REG,LD_PC,
    input logic GatePC, GateMDR, GateALU,GateMARMUX,
    input logic[1:0] PCMUX,ADDR2MUX,ALUK,
    input logic DRMUX, SR1MUX, SR2MUX, ADDR1MUX,
    input logic MIO_EN,
    input logic[15:0] MDR_In,S,
    output logic[15:0] MAR,IR,MDR, PC,
    output logic BEN
);

logic[15:0] BUS;
logic[15:0] MDR_input,MDR_output, MAR_output, ALU_output, PC_output, MARMUX_output,IR_output,IR_input;
logic[15:0] sr1out,sr2out,IMME;


reg_parallel_16 MDR_unit(.Clk(Clk),.Load(LD_MDR),.reset(Reset_ah),.D(MDR_input),.Data_Out(MDR_output));
reg_parallel_16 MAR_unit(.Clk(Clk),.Load(LD_MAR),.reset(Reset_ah),.D(BUS),.Data_Out(MAR_output));
reg_parallel_16 IR_unit(.Clk(Clk),.Load(LD_IR),.reset(Reset_ah),.D(IR_input),.Data_Out(IR_output));

BUS_select bus_select(.MDR2BUS(MDR_output), .ALU2BUS(ALU_output), .PC2BUS(PC_output) ,.MARMUX2BUS(MARMUX_output),.GateMDR(GateMDR),.GateALU(GateALU),.GatePC(GatePC),.GateMARMUX(GateMARMUX),.BUS(BUS));
PC_module PC_unit(.Clk(Clk), .LD_PC(LD_PC),.PCMUX(PCMUX), .Data_from_BUS(BUS),.Data_from_addrmux_to_PC(MARMUX_output),.DataOut(PC_output),.reset(Reset_ah),.run(Run_ah),.S(S));
ADDRMUX addrmux(.IR(IR_output),.ADDR2MUX(ADDR2MUX),.ADDR1MUX(ADDR1MUX),.data_from_SR1OUT(sr1out), .data_from_PC(PC_output),.Data_Out(MARMUX_output), .SEXT(IMME),.Data_to_controller(),.Data_to_BEN());
ALU alu(.SR1OUT(sr1out), .SR2OUT(sr2out),.IMME(IMME),.sr2mux(SR2MUX),.ALUK(ALUK),.OUT(ALU_output));
reg_file regfile(.Clk(Clk),.reset(Reset_ah),.BUS(BUS),.IR11_9(IR_output[11:9]),.IR8_6(IR_output[8:6]),.SR2(IR_output[2:0]),.DR(DRMUX),.SR1(SR1MUX),.LD_REG(LD_REG),.SR1OUT(sr1out),.SR2OUT(sr2out));
ben_module ben_module(.Clk(Clk),.reset(Reset_ah),.BUS(BUS),.LD_CC(LD_CC),.LD_BEN(LD_BEN),.IR11_9(IR_output[11:9]),.BEN(BEN));

assign MDR = MDR_output;
assign MAR = MAR_output;
assign PC = PC_output;
assign IR_input = BUS;
assign IR = IR_output;
assign MDR_input = MIO_EN ? BUS : MDR_In;

endmodule
```

Inputs: Clk, Reset_ah, Run_ah, LD_MAR, LD_MDR, LD_IR, LD_BEN, LD_CC, LD_REG, LD_PC, GatePC, GateMDR, GateALU, GateMARMUX, DRMUX, SR1MUX, SR2MUX, ADDR1MUX, MIO_EN [1:0] PCMUX, ADDR2MUX, ALUK [15:0] MDR_In, S

Outputs: [15:0] MAR, IR, MDR, PC BEN

Description: This is the data path of the whole SLC-3, including the register file, PC-MUX module, instruction decoder, ALU and Conditional Code module. All these components will be connected and be set the control signals through datapath. This module receive all the control signals as input and the output will be the MAR, IR, PC, MDR and BEN.

Purpose: This module connects all the data path components together and set control signals to them.

6) lab6_toplevel.sv

```
module lab6_toplevel( input logic [15:0] S,
                      input logic Clk, Reset, Run, Continue,
                      output logic [11:0] LED,
                      output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, HEX6, HEX7,
                      output logic CE, UB, LB, OE, WE,
                      output logic [19:0] ADDR,
                      inout wire [15:0] Data);

slc3 my_slc(.*);
// Even though test memory is instantiated here, it will be synthesized into
// a blank module, and will not interfere with the actual SRAM.
// Test memory is to play the role of physical SRAM in simulation.
test_memory my_test_memory(.Reset(~Reset), .I_O(Data), .A(ADDR), .*);

endmodule
```

Inputs: [15:0] S Clk, Reset, Run, Continue

Inout: [15:0] Data

Outputs: [11:0] LED [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, HEX6, HEX7CE, UB, LB, OE, WE [19:0] ADDR

Descriptions: This is the top level of the SLC-3 processor, including the slc3 module connected with the test memory only for simulation

Purpose: This module works as the top level of lab6

7) Mem2IO.sv

```
module  Mem2IO (  input logic Clk, Reset,
                  input logic [19:0]  ADDR,
                  input logic CE, UB, LB, OE, WE,
                  input logic [15:0]  Switches,
                  input logic [15:0] Data_from_CPU, Data_from_SRAM,
                  output logic [15:0] Data_to_CPU, Data_to_SRAM,
                  output logic [3:0]  HEX0, HEX1, HEX2, HEX3 );

    logic [15:0] hex_data;

    // Load data from switches when address is xFFFF, and from SRAM otherwise.
    always_comb
     begin
         Data_to_CPU = 16'd0;
         if (WE && ~OE)
          if (ADDR[15:0] == 16'hFFFF)
             Data_to_CPU = Switches;
          else
             Data_to_CPU = Data_from_SRAM;
     end

     // Pass data from CPU to SRAM
    assign Data_to_SRAM = Data_from_CPU;

    // Write to LEDs when WE is active and address is xFFFF.
    always_ff @ (posedge Clk) begin
       if (Reset)
          hex_data <= 16'd0;
       else if ( ~WE & (ADDR[15:0] == 16'hFFFF) )
          hex_data <= Data_from_CPU;
     end

    assign HEX0 = hex_data[3:0];
    assign HEX1 = hex_data[7:4];
    assign HEX2 = hex_data[11:8];
    assign HEX3 = hex_data[15:12];

endmodule
```

Input: Clk Reset [19:0] ADDR CE UB LB OE WE [15:0] Switches [15:0] Data_from_CPU [15:0] Data_from_SRAM

Output: [15:0] Data_to_CPU, [15:0] Data_to_SRAM, [3:0] HEX0, HEX1, HEX2, HEX3

Description: This module uses the data from MAR and input the address which is used for READ and WRITE memory, aided by the signals WE and OE in the ISDU.

Purpose: This module serves as the link between the datapath module and memory_contents

module and enables the modification of memory outside of registers.

8)ISDU.sv

```
module ISDU (   input logic         Clk,
                            Reset,
                            Run,
                            Continue,

            input logic[3:0]    Opcode,
            input logic         IR_5,
            input logic         IR_11,
            input logic         BEN,

            output logic        LD_MAR,
                            LD_MDR,
                            LD_IR,
                            LD_BEN,
                            LD_CC,
                            LD_REG,
                            LD_PC,
                            LD_LED, // for PAUSE instruction

            output logic        GatePC,
                            GateMDR,
                            GateALU,
                            GateMARMUX,

            output logic [1:0]  PCMUX,
            output logic        DRMUX,
                            SR1MUX,
                            SR2MUX,
                            ADDR1MUX,
            output logic [1:0]  ADDR2MUX,
                            ALUK,

            output logic        Mem_CE,
                            Mem_UB,
                            Mem_LB,
                            Mem_OE,
                            Mem_WE
            );
    enum logic [4:0] {  Halted,
                    PauseIR1,
                    PauseIR2,
                    S_18,
                    S_33_1,
                    S_33_2,
                    S_35,
                    S_32,
                    S_01,
                    S_05,
                    S_09,
                    S_06,
                    S_25_1,
                    S_25_2,
                    S_27,
                    S_07,
                    S_23,
                    S_16_1,
                    S_16_2,
                    S_04,
                    S_21,
                    S_12,
                    S_0,
                    S_22}   State, Next_state;   // Internal state logic

    always_ff @ (posedge Clk)
    begin
       if (Reset)
          State <= Halted;
       else
          State <= Next_state;
    end

    always_comb
    begin
       // Default next state is staying at current state
       Next_state = State;
```

Inputs: [3:0] Opcode Clk, Reset, Run, Continue, IR_5, IR_11, BEN

Outputs: LD_MAR, LD_MDR, LD_IR, LD_BEN, LD_CC, LD_REG, LD_PC, LD_LED, GatePC, GateMDR, GateALU, GateMARMUX, DRMUX, SR1MUX, SR2MUX, ADDR1MUX, ALUK, Mem_CE, Mem_UB, Mem_LB, Mem_OE, Mem_WE [1:0] PCMUX, ADDR2MUX

Description: This module defines our state machine (next state and gate variables for current state),

load signals and mux select signals, etc.
Purpose: This module regulates the states that are specified by the given lc3 state diagram and assign proper values to the gate and mux select signals to make the system function properly, and the data can be stored or loaded in the desired way.

9) tristate.sv

```
module tristate #(N = 16) (
    input logic Clk,
    input logic tristate_output_enable,
    input logic [N-1:0] Data_write, // Data from Mem2IO
    output logic [N-1:0] Data_read, // Data to Mem2IO
    inout wire [N-1:0] Data // inout bus to SRAM
);

// Registers are needed between synchronized circuit and asynchronized SRAM
logic [N-1:0] Data_write_buffer, Data_read_buffer;

always_ff @(posedge Clk)
begin
    // Always read data from the bus
    Data_read_buffer <= Data;
    // Always updated with the data from Mem2IO which will be written to the bus
    Data_write_buffer <= Data_write;
end

// Drive (write to) Data bus only when tristate_output_enable is active.
assign Data = tristate_output_enable ? Data_write_buffer : {N{1'bz}};

assign Data_read = Data_read_buffer;

endmodule
```

Inputs: Clk, tristate_output_enable [15:0] Data_write,
Inout: [15:0] Data
Outputs: [15:0] Data_read
Description: This module is the tristate buffer between Memory IO module and the SRAM. With the control signal from control unit, data can flow along bus. Also it is connected to the SRAM with a bi-direction data.
Purpose: This module works as the tristate buffer that connect the SRAM and the memory io module.

10) synchronizers.sv

```
module sync (
   input  logic Clk, d,
   output logic q
);

always_ff @ (posedge Clk)
begin
   q <= d;
end

endmodule
```

Inputs: Clk, d
Output: q
Description: This module serves as the synchronizer for the input from keys or switches. It is a register to synchronize the input signal and help avoid the potential problems caused by the synchronous signals of inputs
Purpose: This module will synchronize the asynchronous signals to the modules in FPGA.

11) reg_file.sv

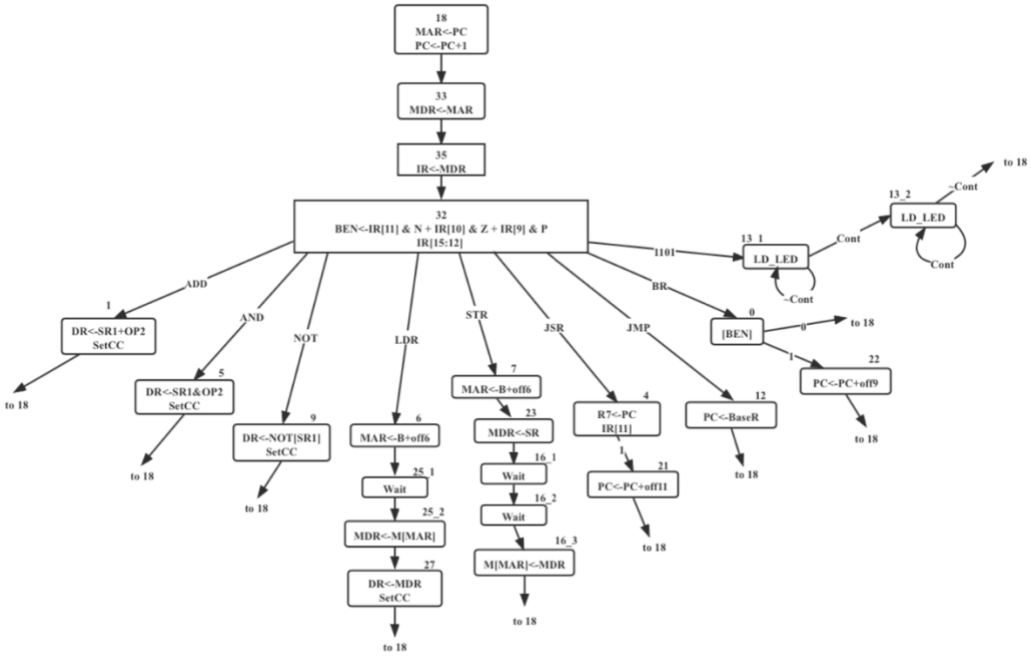
```
module reg_file(
    input logic Clk,
    input logic reset,
    input logic [15:0] BUS,
    input logic [2:0] IR11_9,IR8_6,SR2,
    input logic DR,SR1,LD_REG,
    output logic [15:0] SR1OUT, SR2OUT
);
    logic [2:0] drmux_out, sr1mux_out;
    logic [7:0] reg_ld;
    logic [15:0]outr0, outr1,outr2,outr3,outr4,outr5,outr6,outr7;

    assign drmux_out = DR ? 3'b111 : IR11_9;
    assign sr1mux_out = SR1 ? IR11_9 : IR8_6;

    reg_parallel_16 reg0(.Clk(Clk),.reset(reset),.Load(LD_REG&reg_ld[0]),.D(BUS),.Data_Out(outr0));
    reg_parallel_16 reg1(.Clk(Clk),.reset(reset),.Load(LD_REG&reg_ld[1]),.D(BUS),.Data_Out(outr1));
    reg_parallel_16 reg2(.Clk(Clk),.reset(reset),.Load(LD_REG&reg_ld[2]),.D(BUS),.Data_Out(outr2));
    reg_parallel_16 reg3(.Clk(Clk),.reset(reset),.Load(LD_REG&reg_ld[3]),.D(BUS),.Data_Out(outr3));
    reg_parallel_16 reg4(.Clk(Clk),.reset(reset),.Load(LD_REG&reg_ld[4]),.D(BUS),.Data_Out(outr4));
    reg_parallel_16 reg5(.Clk(Clk),.reset(reset),.Load(LD_REG&reg_ld[5]),.D(BUS),.Data_Out(outr5));
    reg_parallel_16 reg6(.Clk(Clk),.reset(reset),.Load(LD_REG&reg_ld[6]),.D(BUS),.Data_Out(outr6));
    reg_parallel_16 reg7(.Clk(Clk),.reset(reset),.Load(LD_REG&reg_ld[7]),.D(BUS),.Data_Out(outr7));

    always_comb
    begin
        unique case (drmux_out)
            3'b000 :    reg_ld = 8'b00000001;
            3'b001 :    reg_ld = 8'b00000010;
            3'b010 :    reg_ld = 8'b00000100;
            3'b011 :    reg_ld = 8'b00001000;
            3'b100 :    reg_ld = 8'b00010000;
            3'b101 :    reg_ld = 8'b00100000;
            3'b110 :    reg_ld = 8'b01000000;
            3'b111 :    reg_ld = 8'b10000000;
            default :   reg_ld = 8'b00000000;
        endcase

        unique case(SR2)
            3'b000 :    SR2OUT = outr0;
            3'b001 :    SR2OUT = outr1;
            3'b010 :    SR2OUT = outr2;
            3'b011 :    SR2OUT = outr3;
            3'b100 :    SR2OUT = outr4;
            3'b101 :    SR2OUT = outr5;
            3'b110 :    SR2OUT = outr6;
            3'b111 :    SR2OUT = outr7;
        endcase

        unique case(sr1mux_out)
            3'b000 :    SR1OUT = outr0;
            3'b001 :    SR1OUT = outr1;
            3'b010 :    SR1OUT = outr2;
            3'b011 :    SR1OUT = outr3;
            3'b100 :    SR1OUT = outr4;
            3'b101 :    SR1OUT = outr5;
            3'b110 :    SR1OUT = outr6;
            3'b111 :    SR1OUT = outr7;
        endcase
    end

endmodule
```

Inputs: input logic LD_REG, Reset, Clk, input logic [15:0] in, input logic [2:0] SR1, SR2, DR,
Outputs: output logic [15:0] SR1_OUT, SR2_OUT)
Description:
The register file is a unit with eight 16-bit registers that with input logic DR, which will choose which register to load and input logic LD_REG for when to load. SR1, SR2 select which registers to output.
Purpose: For operations in the SLC-3 we need to be able to store “answers” into 8 data registers

e. Description of the Operation of the ISDU

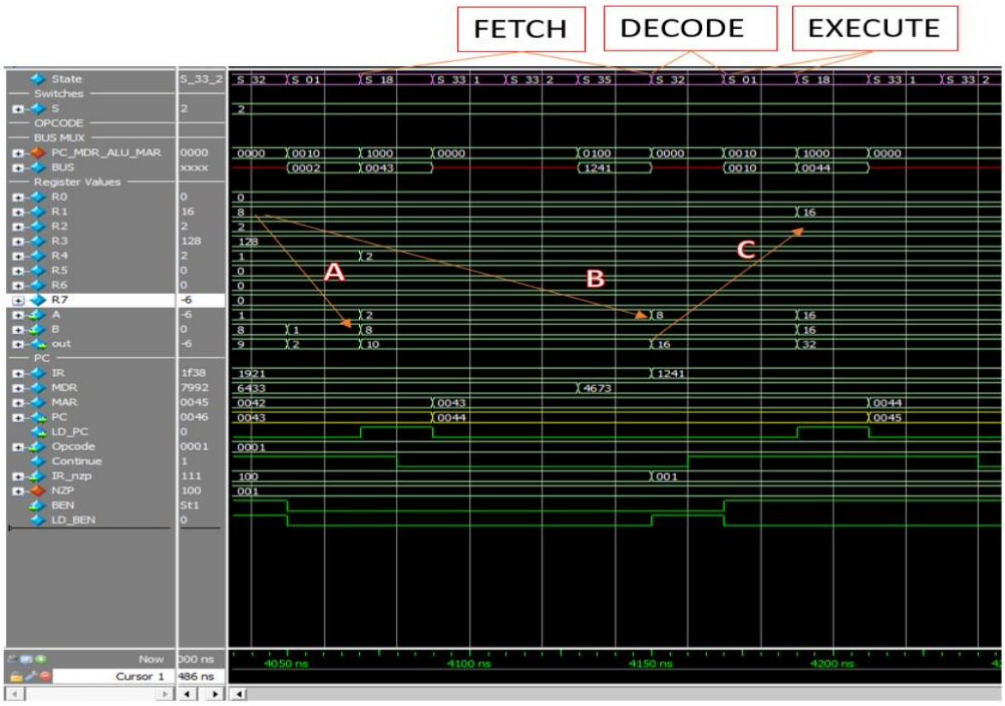
The ISDU module includes signal assignments for all the valid states, controlling many components in the slc3 system with the enable bits that are predefined in each state. The module ISDU. In addition, in this module we define the state transitions for states using various toggle bits to ensure the functionality of the components and the operations of states.

f. State Diagram of ISDU



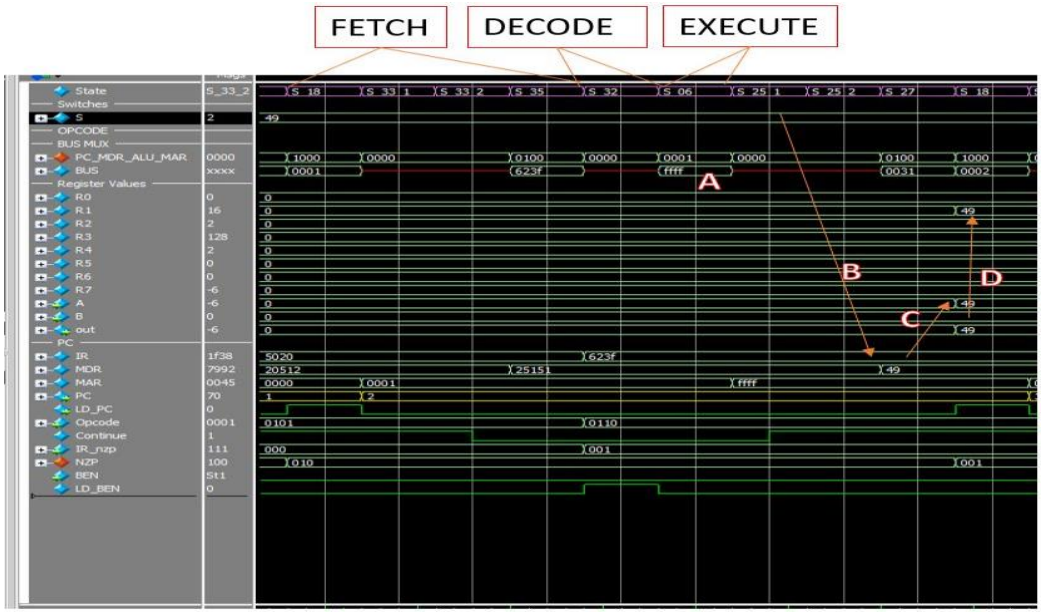
III. Simulation

a. opADD(R1, R1, R1)



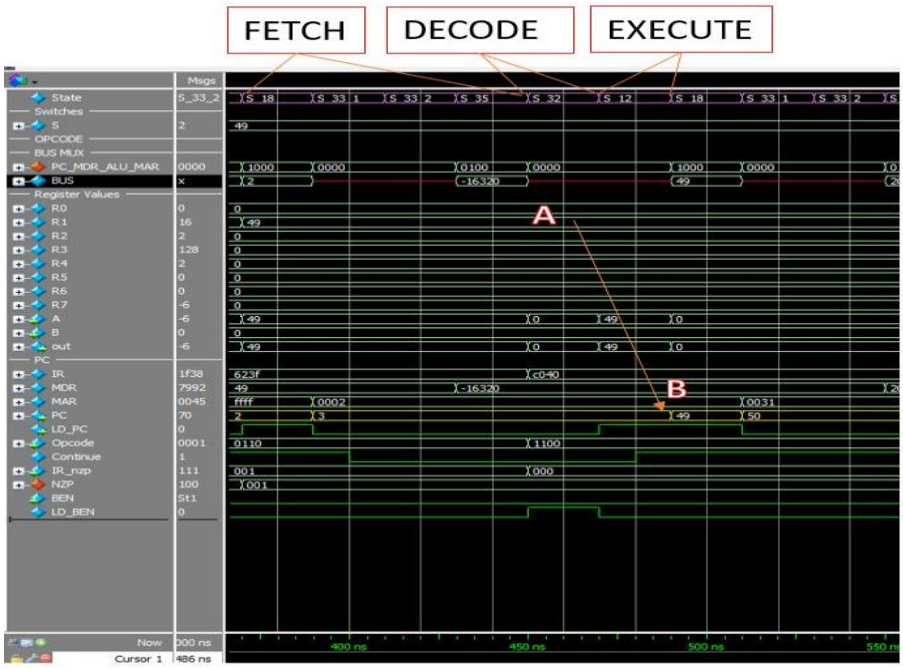
At A, we can easily see that 8 from R1 goes into port B of the ALU (from SR2MUX output). Similarly, at **B we can see that 8 fro**m R1 goes into port A of ALU (from SR1 OUT). At C, we see that 16 is stored back into R1.

b. opLDR(R1, R0, inSW)



At A, we can easily see that xffff in the bus indicates that we will be reading from switches. At B, the value from the switches is loaded into MDR. At C this goes into the ALU (ADD), so 49 + 0 (R0 value) = 49, and at D the output goes into R1

c. opJMP(R1)



At A, we can easily see the value at R1 is 49. When the LD_PC signal goes high right before we see B, PC gets updated the next clock cycle to 49.

IV. Post-Lab Questions

Q1.

LUT	603
DSP	0
Memory (BRAM)	0
Flip-flop	283
Frequency	85.61
Static Power	98.64mW
Dynamic Power	0.00mW
Total Power	172.29mW

In week 1, we didn't change the code in testbench, thus our code cannot be compiled. Also, there were too many ports that need to be wired to datapath, we took a lot of time to make it work. It's useful to look at the diagram of LC-3 to do our work.

In week 2, we forgot to use BEN bit in a BR execution, so it always branching. We find this error according to the output result and traced back, that would be easier to find the error.

Q2.1

MEM2IO is used to interface with the SDRAM and the switches, only if the address is xFFFF, it will read from the switches, otherwise, it will read from SDRAM. And we can do reading and writing from SDRAM. MEM2IO interfaces with tristate buffers so data is either from SRAM to CPU or CPU to SRAM, that is, SRAM is bidirectional.

Q2.2

BR is the condition based on NZP bit from the instruction register. JMP is unconditional since it copies memory address from BaseR to PC. The differences is that, JMP can branch the instruction to any indicated memory position while BR can only branch the instruction located at surrounding positions for the current PC.

Q2.3

The R signal in Patt and Patel is a ready signal, which indicate whether LC3 has finished reading and writing from memory. LC3 will wait for R signal to execute the next instruction. However, we don't need R signal in SRAM, so we add the extra states in ISDU, there will be enough time in these states so that the read and write will definitely finish. In LC3, thw R signal is generated form FSM so it's asynchronized, in real situation, R imply that there is an unintended behaviour.

V. Conclusion

Luckily, our design functions well on both analog and FPGA boards. We pass the most difficult XOR tests and SORT tests which is used to determine if all of our operations worked.

We want to thank the lab manual, which is very helpful in explaining the base pipeline about how to design the SLC3 using the SystemVerilog. However, according to our memory, we think that the lab manual doesn't talk much about the memory organization. We use the provided memory code to get familiar with memory organization. We hope it can be specified with more information in the future to help students avoid misunderstanding about this part.