ECE 385 Experiment 5: Lab Report

Spring 2023
Eshaan Tibrewala (eshaant2) and Shivam Patel (shivamp6)
TA: Shitao Liu

Introduction

Experiment 5 was a two week lab where we designed a simple microprocessor using SystemVerilog. The microprocessor is a simplified version of the LC-3 ISA, which is a 16-bit processor with 16-bit program counter, instructions, and registers. Our SLC-3 (simplified LC-3) preprocessor has three main components. There is the central processing unit (CPU), memory for storing instructions and data, and the input/output (I/O) interface to communicate with other devices. There are three basic steps that the computer performs. First it fetches an instruction from memory, decodes it to determine the type of instruction, and then finally executes the instruction before returning to the fetch step. Our SLC-3 can perform 11 different instructions, which are selected based on their respective opcodes (`IR[15:12]`). The Instruction Sequencer/Decoder is where the CPU actually decodes the instruction, and contains a finite state machine which provides the other processor components with the proper control signals to carry out the instruction.

SLC-3 Processor

Summary of Operation

The SLC-3 processor performs its functions in three main steps: Fetch, Decode, and Execute. These steps are driven by a state machine (discussed further in the ISDU section) which transitions through different states to perform operations. Our simplified LC-3 processor is capable of performing 11 different operations. Table 1 shows these operations and their respective instruction bits. `IR[15:12]` hold the opcodes of each instruction, and tell the processor which operation to perform. Our processor works by first setting the PC. On the physical FPGA, this is done by setting the switches and toggling `Run`. One `Run` is toggled, the processor will see that memory is at `4'hffff`, and read the input from the switches. The PC will then jump to the corresponding location in memory, and begin to increment from that point. Our memory contains six different tests, so the user can select between six different starting points for the PC. Once PC is set, it will continue to increment using $PC \leftarrow PC + 1$ until the operation is either terminated by the `Reset` signal when both `Continue` and `Run` are toggled at the same time, or the processor enters the `PAUSE` instruction. Some tests such as MULTIPLY, SORT, and XOR use this to get additional input from the switches. If the processor reaches this state, it will remain until just `Continue` is toggled. This tells the processor that it can move on with its execution of the instruction. When the processor is completely done

executing the instruction, it will end in the `Halted` state, where it will wait for `Reset` to be toggled so that a new PC can be set and a new test can be fetched from memory.

Overall Functionality

The actual functionality of the processor occurs on the data path and ISDU. The ISDU essentially produces control signals, which are then used by the data path to execute the instructions. However before the processor can execute the instruction, it must fetch it from memory and then decode it.

Fetch

$$(1)\ MAR \leftarrow PC$$
$$(2)\ MDR \leftarrow M(MAR)$$
$$(3)\ IR \leftarrow MDR$$
$$(4)\ PC \leftarrow PC + 1$$

Fetching an instruction requires four steps. In the first step, the processor loads MAR with the value of PC. MAR is the *memory address register* and holds the address in memory. So since PC points to a space in memory, that value of PC is loaded into MAR so that it can hold the address of the instruction. In the second step, MDR, the *memory data register* is loaded with the data at MAR, as denoted by M(MAR). Since we know that MAR holds the address, the data at that address is the instruction we want and is loaded into the data register so that it can be loaded into the IR in the following step. In this third step, the *instruction register*, IR, is loaded with MDR, which holds the actual instruction do be executed. Once the actual instruction is loaded into IR via steps 1-3, PC is incremented by one so that the processor is ready to fetch the next instruction once the current instruction is decoded and executed. It is important to note the PC is incremented by one for all instructions, unless PC gets another value.

Decode

$$Instruction\ Sequencer/Decoder \leftarrow IR$$

After the instruction as been completely fetched by the processor, it is then decoded by the ISDU, which is the instruction sequencer/decoder unit. The ISDU reads the instruction, and generates the necessary control signals based on the opcode and also specify the necessary operation to the ALU so that the instruction can be correctly

executed. Each operation takes multiple cycles, so the ISDU needs to provide these signals at each cycle.

Execute

Once the instruction is decoded, the processor continues through the FSM and performs the operation based on the control signals generated. The execute step then writes the result to memory or the destination register based on the instruction, and then returns to the fetch state to start the cycle again.

Instructions and Operations of the SLC-3 Processor

Table 1: SLC-3 ISA

Instruction	Instruction(15 downto 0)																Operation
ADD	0001			DR			SR1			0	00			SR2			R(DR) ← R(SR1) + R(SR2)
ADDi	0001			DR			SR			1	imm5						R(DR) ← R(SR) + SEXT(imm5)
AND	0101			DR			SR1			0	00			SR2			R(DR) ← R(SR1) AND R(SR2)
ANDi	0101			DR			SR			1	imm5						R(DR) ← R(SR) AND SEXT(imm5)
NOT	1001			DR			SR			111111							R(DR) ← NOT R(SR)
BR	0000	n	z	p	PCoffset9												if ((nzp AND NZP) != 0) PC ← PC + SEXT(PCoffset9)
JMP	1100	000			BaseR			000000									PC ← R(BaseR)
JSR	0100	1	PCoffset11														R(7) ← PC; PC ← PC + SEXT(PCoffset11)
LDR	0110			DR			BaseR			offset6							R(DR) ← M[R(BaseR) + SEXT(offset6)]
STR	0111			SR			BaseR			offset6							M[R(BaseR) + SEXT(offset6)] ← R(SR)
PAUSE	1101	ledVect12															LEDs ← ledVect12; Wait on Continue

As shown in Table 1, our SLC-3 processor can perform 11 different operations. Each operation comes with its unique instruction that tells the processor which registers to use and how to reassign PC, if not using the standard PC incrementation.

ADD

The opcode of the ADD instruction is `0001`, and makes up the four most significant bits of the instruction. It occurs in State 1 of the ISDU, and involves three registers. The destination register gets the result of the add operation, and is declared in `IR[11:9]`. The other registers are the source registers, and store the data that will be added. SR1 is declared in `IR[8:6]`, and SR2 is declared in `IR[2:0]`. `IR[5]` tells the processor whether the add operation will occur between two registers, or one register and the immediate 5 bits of the instruction. For regular add operations between two registers `IR[5]` is set to 0. The actual occurs in the ALU, and the result is stored on the BUS via GateALU and driven back to the register file where it is stored in the corresponding destination register. ADD also sets CC, the condition code, at the end of its operation based on whether it is zero, negative, or positive. Set CC essentially tells the processor the sign of the result, which is necessary for the BRANCH instruction.

ADDi

ADDi performs immediate addition, and its instruction and operation is almost the same as ADD. The only difference is that `IR[5]` is high, indicating that the processor should add the data in the source register to the immediate 5 bits declared in the instruction bits.

AND/ANDi

The opcode for AND and ANDi instructions are `0101`. It occurs at State 5, and its instruction and operation are the same as the ADD and ADDi instructions, but instead of adding the data between source registers the ALU performs logical AND between the two inputs. Similar to the ADD and ADDi, the instruction of AND and ANDi also includes a `IR[5]` bit which selects immediate 5 bits when high or another source register when low. AND and ANDi also set CC once the operation is performed.

NOT

The opcode for the NOT instruction is `1001`, and occurs at State 9. Unlike ADD or AND, the instruction only declares two registers, the destination register in `IR[11:9]` and the source register in `IR[8:6]`. The remaining bits are all 1 because they are insignificant to the operation. NOT also occurs at the ALU which inverts the data from the source register. CC is set once the operation is performed.

BR

The opcode for the BR instruction is `0000`, and its execution starts in State 0. BR tests the condition codes in `IR[11:9]` with NZP. If any of the condition codes that are tested are set, the program reassigns PC so that it branches to the location specified by adding the sign extended value of `IR[8:0]` to the value of PC. These least significant 9 bits are referred to as PCOffset9 and is used to move the program counter.

JMP

The opcode for the JMP instruction is `1100` and its execution starts in State 12. Similar to the BR instruction, JMP reassigns the value of PC. However instead of testing the condition codes it changes the value of PC to the data stored in the base register which is declared in `IR[8:6]`.

JSR

The opcode for the JSR instruction is `0100` and its execution starts at State 4. Its purpose is similar to JMP, but it stores the value of PC in R7 (register 7) before changing the value of PC. The new value of PC becomes PC + PCOffset11. The 11 bits come from `IR[10:0]`.

LDR

The opcode for the LDR instruction is `0110` and its execution starts at State 6. Its purpose is to load the data into a specified destination register from a location in memory. This location is specified by adding the contents of a base register to an offset of 6 bits. So any data stored at a memory location within that range gets moved to the destination register declared in the instruction.

STR

The opcode for the STR instruction is `0111` and its execution starts at State 7. Its purpose and functionality is essentially the exact opposite of the LDR instruction. Whereas LDR loads data from memory, STR stores the contents from a register to the memory location declared in another register plus an offset of 6 bits.

PAUSE

The opcode of the PAUSE instruction is `1101` and its execution starts at State PAUSEIR1. Its purpose is to simply reflect the values of `IR[11:0]` on the LED display on our FPGA and wait for the `Continue` signal so that it can go the the next state and return to State 18 for the next instruction cycle.

Instruction Sequence Decoder Unit

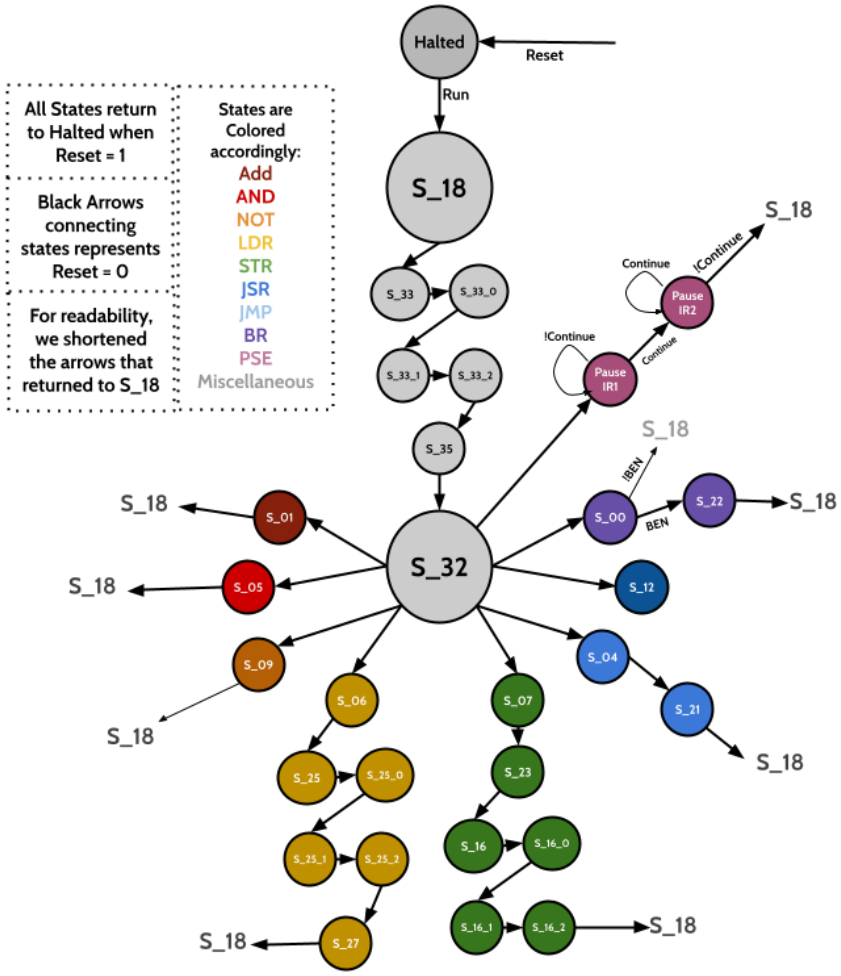

Figure 1: SLC-3 FSM State Diagram

Figure 1 shows the FSM for our SLC-3 processor. Each state is color coded to the corresponding instruction. This is a finite state machine that controls the SLC-3 operation by choosing which signals are enabled as well as which are disabled. As we proceed through the finite state machine, we turn on or off signals in each state that allow us to complete what we want to in each state to create the Fetch, Decode, and Execute parts of the SLC-3. We change various signals by following the SLC-3 datapath and setting our multiplexers and signals to their necessary values. The ISDU will begin at the state "halted" and if the user presses RUN then it will go to S_18 where it will load the PC value, Load MAR, and set our PCMux such that it increments our PC. It then moves onto the next few states where it completes FETCH so we can move onto S_32 which is the decode state. In this state we load bench enable since based on which instruction we have to complete from our IR, we want to go there allowing us to complete the execute stage. The ISDU will continue to execute and complete operations based on the fetched instructions. Since we use different parts of the Instruction Register to complete each operation and have to decide what Multiplexers to use or what tristate buffer to enable, each operation completes different things. With our ISDU, the SLC-3 can utilize different operations by storing and loading values from memory and executing instructions that are stored in the memory as well.

The ADD, AND, and NOT operation states are arithmetic so we need to compute values and store them in the destination register. To do this we have DRMux set to 0 and have our GateALU set to 1 and ALUK set accordingly. We also need to set our SR2Mux to IR_5 in the cases where we have Add or AND immediate and then finally set our condition code in the case of our next operation being BR.

For the LDR and STR operations we have to access memory because we are either reading from memory by loading or writing from memory to store and to do this we need to have multiple states. In the initial states, we set the SR1Mux, ADDR1Mux, and ADDR2Mux signals to 1,1, and 01 respectively. When we read or write to memory we included 3 more wait states so we have enough time to do so. For LDR we must read from memory and load it to a destination register so we have GATEMARMUX set to high as we read and GATEMDR and DRMux set accordingly when we want to store that value we read into a register. For STR we must write to memory so we do the opposite in a sense where we first have GATEMARMUX set to high and then load MAR so we can load MDR in the next state.

The JSR operation we want to go to a subroutine but we must first save our PC in R7 for when we must return. This is why in the first JSR state we have DRMux set to 1 which chooses R7. This is the only state where DRMux is set to 1. We also have GatePC set to high since we want to store our PC in R7 before proceeding to the next state. We then need to increment PC to our subroutine and that distance is defined by the sign extended

off11. To do this, we select the output of the address adder which is 11 in our PCMux and we also must load the PC, so we set it to high.

For the JMP and BR instructions, we have to jump to an intended address. The JMP function state loads our PC with the BaseR so to do this in our ISDU we set our load PC signal to high and set our SR1Mux, ADDR1Mux, and ADDR2Mux control signals to the necessary values. The BR function needs to check the condition codes so we have two states. In the first state, we check the BEN to see if we should branch or not so we do not change any signals in the state and just proceed to the next state if so. When we want to branch, we want to do the same as the JMP state where we set our SR1Mux, ADDR1Mux, and ADDR2Mux control signals to the necessary values.

Finally, the pause operations remain. We only load our LEDs in the control signals section but we set conditions in the next state section such that we only proceed based on if the continue button is pressed or not. If we press continue once we go from our first pause state to the second and stay in that state until we press continue again.

.SV Modules

Table 1: .SV Modules for Processor

MODULE	UNIT/PURPOSE
testbench.sv	testbench
registers.sv	registers
multiplexers.sv	multiplexers
datapath1.sv	SLC-3 data path
ISDU1.sv	control unit
test_memory.sv	memory for simulations
synchronizers.sv	asynchronous clock signals
slc3.sv	SLC-3 top level
slc3_testtop.sv	top level for simulations
sl3_sramtop.sv	top level for FPGA
SLC3_2.sv	describes opcodes
register_file.sv	register unit on data path

MODULE	UNIT/PURPOSE
memory_contents.sv	memory contents for testing
MEM2IO.sv	connect memory and device
instantiateram.sv	instantiate on-chip memory
HexDriver.sv	hex driver

testbench.sv

```
module testbench();
    logic [9:0] SW;
    logic Clk, Run, Continue;
    logic [9:0] LED;

    logic [6:0] HEX0, HEX1, HEX2, HEX3;
    logic [15:0] PC, IR, MAR, MDR;
```

The `testbench.sv` module creates all the local logic variables to instantiate the top level `slc3_testtop.sv` module for simulation testing. The purpose of the testbench is to debug the ISDU and data path and to make sure that our lab is passing all the given tests.

registers.sv

```
module sixteen_reg(input Clk, Reset, Load,
                   input [15:0] Data_In,
                   output logic [15:0] Data_Out);

module ten_reg(input Clk, Reset, Load,
                   input [9:0] Data_In,
                   output logic [9:0] Data_Out);

module nzp(input Clk, Reset, LD_CC,
           input [15:0] BUS,
           output logic [2:0] Data_Out);
```

```
12
13  module three_reg(input Clk, Reset, LD_CC,
14              input [2:0] Data_In,
15              output logic [2:0] Data_Out);
16
17 module ben(input Clk, Reset, LD_BEN,
18          input Data_In,
19          output logic Data_Out);
```

The `ten_reg` is used to store the `LED` and `three_reg` is used to store `CC` after executing the corresponding operations. `ben` is a simple 1-bit register, so essentially a flip-flop, which tells the ISDU if branch is enabled. The `module nzp` itself is not a register, but includes some logic and then instantiates the `three_reg` to store the values of `NZP`. The logic sets the bits based on whether the value on the `BUS` is positive, negative, or zero. Based on the sign, the corresponding bit of NZP is set to high and the other two are set to low.

The overall purpose of the `registers.sv` module instantiates all the registers required for the data path by register size. We put all the different registers in one module so that our overall program was more organized and easier to debug once we started testing the data path.

multiplexers.sv

```
1  module driver_mux(input logic [3:0] BUS_MUX,
2              input logic [15:0] PC_Out, MDR_Out, ALU_Out,
   ADDR_Out,
3              output logic [15:0] BUS);
4
5   module two_one_16(input logic sel,
6            input logic [15:0] data0, data1,
7            output logic [15:0] muxOut);
8
9   module pc_mux(input logic [1:0] PC_select,
10           input logic [15:0] BUS, PC, ADDR,
11           output logic [15:0] mux_out);
12
```

```
13    module two_one_3(input logic sel,
14             input logic [2:0] data0, data1,
15             output logic [2:0] muxOut);
```

`multiplexers.sv` contains all the multiplexers that are instantiated in our data path. `two_one_16` instantiates all the 2-to-1 multiplexers in the data path that carry 16-bit data. These include the ADDR1MUX and the MDR mux, which uses `MIO_EN` as the select bit to drive data.

The `driver_mux` replaces the tri state buffer on the traditional LC-3 data path. Since our FPGA does not support tri state buffers, we used a 4-to-1 multiplexer instead. The select bits are determined by the concatenating the signals from `GatePC`, `GateMDR`, `GateALU`, and `GateMARMUX`. Since only one of these can be high at a time, the output of the multiplexer is the corresponding `_Out` data which then enters the BUS to carry out the operation described by the instruction.

The `pc_mux` is also its own special multiplexer, and its output is loaded into the PC register. The select signal has a size of 2-bits, and is determined by the `PC_select` signal. The possible output bits are `BUS`, which is the data on the BUS, `PC`, which is PC+1, and `ADDR`, which is the input from the ADDR.

The `two_one_3` module instantiates all 2-to-1 multiplexers which drive 3-bit data. These include DRMUX and SR1MUX, which are used to select different storage registers in the register file.

datapath1.sv

```
module datapath1(input logic Clk, Reset,
            input logic LD_MAR, LD_MDR, LD_IR, LD_BEN, LD_CC,
LD_REG, LD_PC, LD_LED,
            input logic GatePC, GateMDR, GateALU, GateMARMUX,
            input logic MIO_EN,
            input logic SR1MUX, SR2MUX, ADDR1MUX, MARMUX, DRMUX,
            output logic [9:0] LED,
            input logic [15:0] MDR_In,
            input logic [1:0] PCMUX, ALUK, ADDR2MUX,
            output logic BEN_Out,
            output logic [15:0] MDR_Out, MAR_Out, IR_Out, PC_Out
            );
```

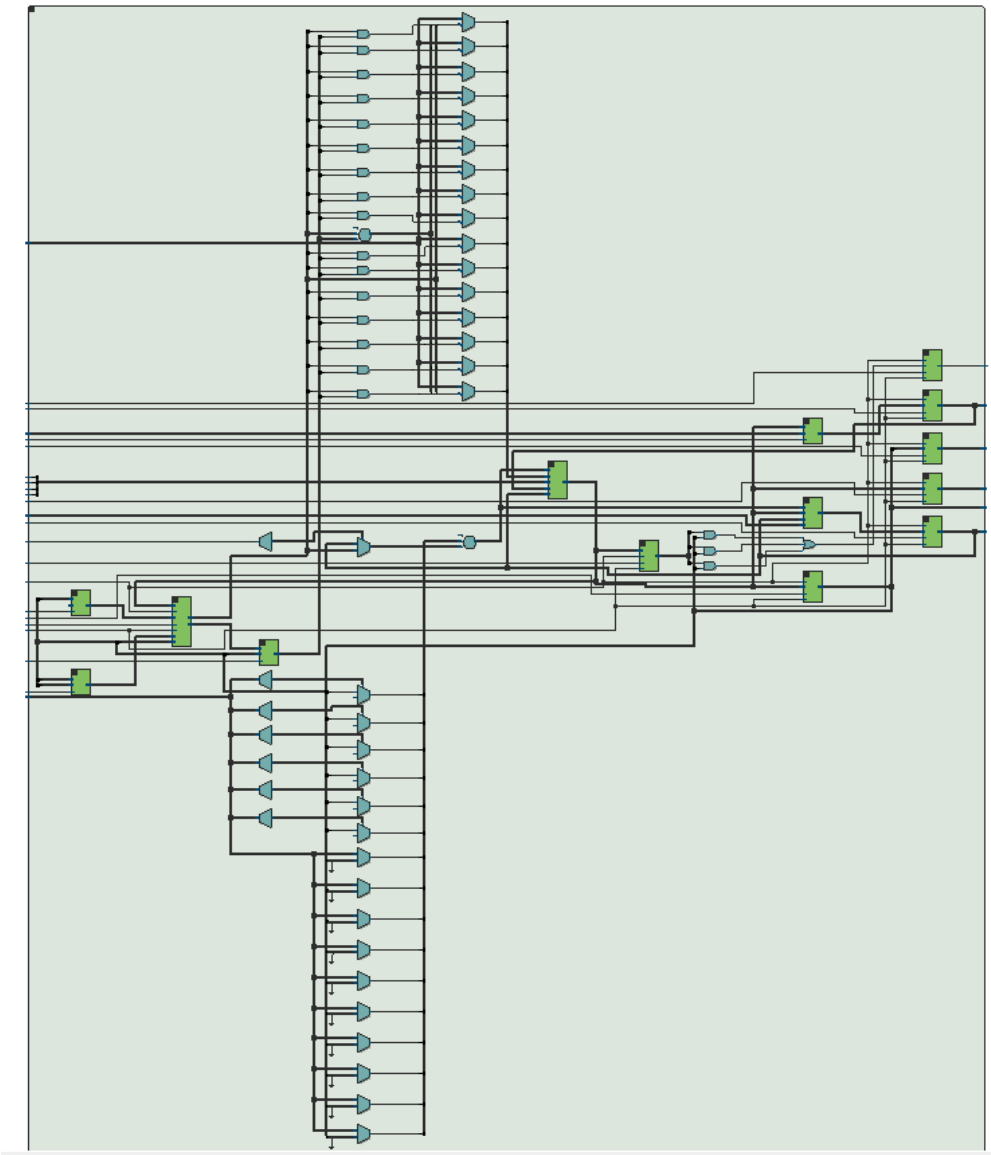

Figure 2: Data Path RTL Block Diagram

The `datapath1` module holds the entire SLC-3 data path. The `Clk` and `Reset` signals are mainly used by the registers to load and clear values. All the other inputs into the data path are our control signals which come from the ISDU. The MUX inputs are the select bits for the multiplexers and determine which data gets driven through the mux's. The outputs of our data path are the `LED`, `BEN_Out`, `MDR_Out`, `MAR_Out`, `IR_Out`, and `PC_Out`. These outputs correspond to the signals declared in the SLC3 module where the `datapath1` module is instantiated.

The purpose of the module is to instantiate all the multiplexers, registers, and other signals that are shown in the data path diagram. It performs sign extensions on the bits and loads memory. The inputs to the data path come from the ISDU, which is also instantiated in the `slc3` module. All the components of the data path are connected by the BUS which drives the 16-bit signals to different parts of the data path so that the instruction loaded from memory can be executed.

ISDU1.sv

```
1  module ISDU (   input logic         Clk,
2                  Reset,
3                  Run,
4                  Continue,
5
6          input logic[3:0]    Opcode,
7          input logic         IR_5,
8          input logic         IR_11,
9          input logic         BEN,
10
11         output logic        LD_MAR,
12                 LD_MDR,
13                 LD_IR,
14                 LD_BEN,
15                 LD_CC,
16                 LD_REG,
17                 LD_PC,
18                 LD_LED, // for PAUSE instruction
19
20         output logic        GatePC,
21                 GateMDR,
```

```
22                  GateALU,
23                  GateMARMUX,
24
25          output logic [1:0]  PCMUX,
26          output logic        DRMUX,
27                  SR1MUX,
28                  SR2MUX,
29                  ADDR1MUX,
30          output logic [1:0]  ADDR2MUX,
31                  ALUK,
32
33          output logic        Mem_OE,
34                  Mem_WE
35          );
```

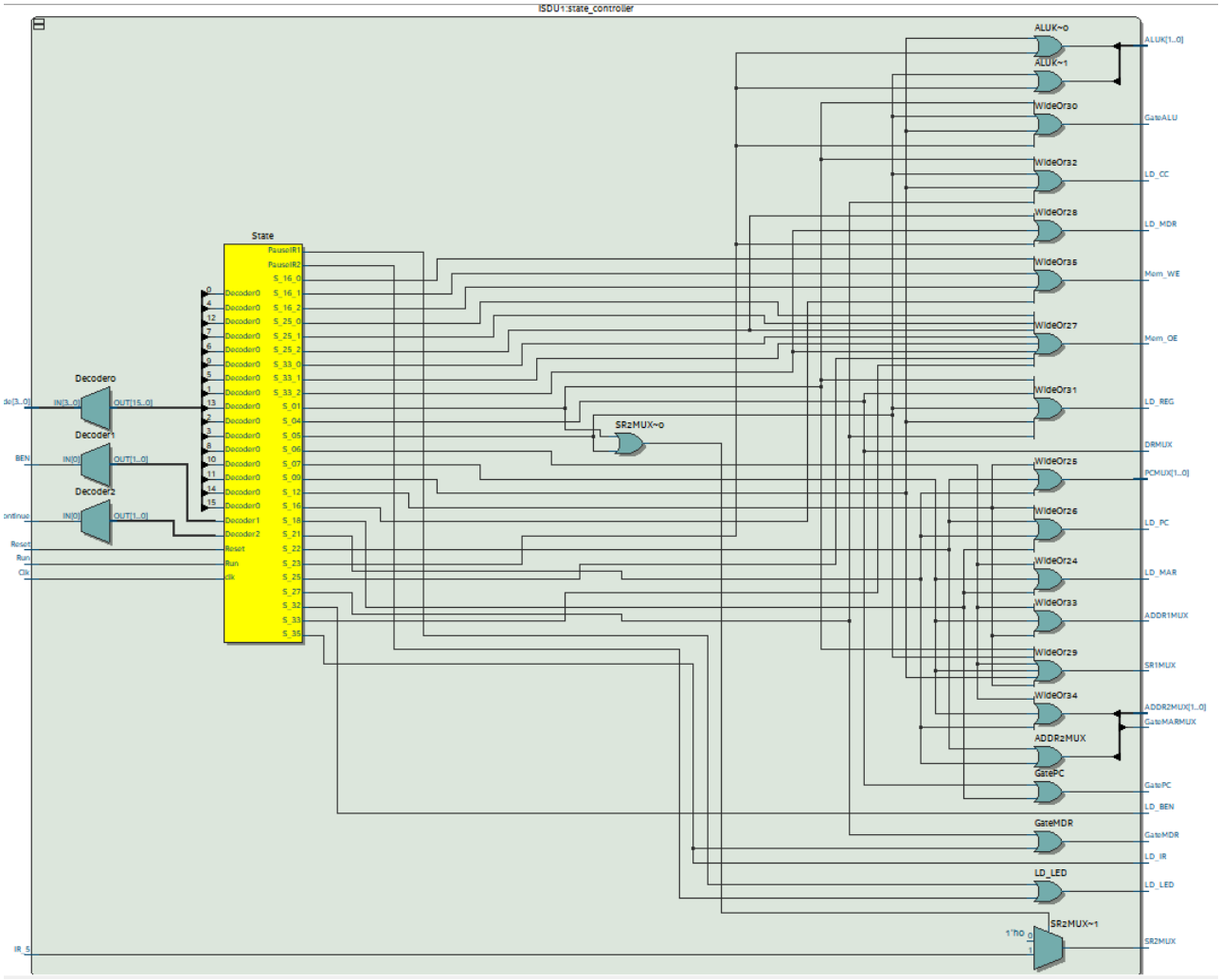

Figure 3: ISDU RTL Block Diagram

The `ISDU` module is the control unit of our processor. The inputs are `Opcode`, `IR_5`, `IR_11`, and `BEN`. The functionality of the control unit also depends on the clock, and the three driving signals, which are `Reset`, `Run`, and `Continue`. Those signals come from the buttons on the FPGA, and drive the operations. Reset completely resets the processor and clears the registers. Run starts the instruction cycle and initiates the fetch operation. `Continue` iterates through the instruction and steps through the states. The outputs of the `ISDU` module are all the control signals and multiplexer select bits which drive the signals and data in the data path. Each one corresponds to loading its respective register and driving the outputs of its respective multiplexers.

The `Opcode` input comes from `IR[15:12]`, and determines what kind of instruction to be executed. The `IR_5` bit selects whether the corresponding operation will be performed on the immediate 5 bits of the IR or a different register determined by `IR[2:0]`. `IR_11` is used by the JSR and JSRR instructions and tell the processor where to jump.

The overall purpose of the `ISDU` is to feed all the inputs to the data path. `Mem_OE` and `Mem_WE` are the memory signals and tell the data path whether to read or write from memory.

test_memory.sv

```
module test_memory ( input Reset,
                     input   Clk,
                     input [15:0]  data,
                     input [9:0]  address,
                     input   rden,
                     input   wren,
                     output logic [15:0]  readout);
```

The `test_memory` module behaves similar to SRAM on the FPGA. It is used for simulations only, so it is only instantiated by the `slc3_testtop` module. When `rden` is high, the module reads the memory at a given address and when `wren` is high the module writes to the memory stored at a specific address.

The overall purpose of the module is to simply simulate the memory unit for simulation and synthesis testing so that we can ensure that our program is functioning correctly before uploading and testing it on the FPGA.

synchronizers.sv

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

module sync_r0 (
  input Clk, Reset, d,
  output logic q
);

module sync_r1 (
  input Clk, Reset, d,
  output logic q
);
```

The purpose of the `Synchronizers` module is to bring the asynchronous signals into the FPGA. For switches and buttons it does not require a reset, but has a reset signal to 0 and 1 for D flip flops.

slc3.sv

```
module slc3(
  input logic [9:0] SW,
  input logic Clk, Reset, Run, Continue,
  output logic [9:0] LED,
  input logic [15:0] Data_from_SRAM,
  output logic OE, WE,
  output logic [6:0] HEX0, HEX1, HEX2, HEX3,
  output logic [15:0] ADDR,
  output logic [15:0] Data_to_SRAM
);
```

This is the top level module that for our processor. It instantiates `MEM2IO`, `ISDU1`, and `datapath1` using the inputs mapped from `slc3_testtop` or `slc3_sramtop` depending on where we are testing. It also outputs the ADDR, `Data_to_SRAM`, and OE and WE so that those signals can be used by the `MEM2IO` module to drive memory and IO. Depending on which test module is being instantiated, the inputs can either come from the local signals declared by the testbench (if testing via simulations) or the buttons and switches from the physical FPGA (if testing via FPGA). The HEX outputs are also mapped accordingly.

The overall purpose of this module is to actually instantiate the SLC-3 processor by the top level file.

slc3_testtop.sv

```
module slc3_testtop(
  input logic [9:0] SW,
  input logic Clk, Run, Continue,
  output logic [9:0] LED,
  output logic [6:0] HEX0, HEX1, HEX2, HEX3
);
```

This module is the top level module for testing done on simulation. The inputs come from the local variables declared in the testbench, and the outputs are also displayed on the local signals declared in the testbench. This module instantiates the basic ISDU and data path module for functionality, but utilizes different memory module since there isn't a physical SRAM for simulation testing. This simulated memory can be found in the `test_memory` module.

The overall purpose of this module is to be instantiated by the testbench so that all the inputs and outputs can be mapped to local variables and we can see how our FPGA is functioning in simulation via waveforms. This allows us to debug and ensure proper functionality before transferring to the FPGA for actual implementation of our processor.

slc3_sramtop.sv

```
module slc3_sramtop(
  input logic [9:0] SW,
  input logic Clk, Run, Continue,
  output logic [9:0] LED,
  output logic [6:0] HEX0, HEX1, HEX2, HEX3
);
```

This module is the top level module for testing done on the FPGA. It instantiates all the necessary memory and testing modules along with the ISDU and data path so that the processor can interact with the FPGA. Its functionality is similar to `slc3_testtop` but uses the physical RAM of the FPGA. The inputs come from the FPGA switches and buttons, and the outputs are reflected on the LED and Hex displays.

The overall purpose is to serve as a top level file to instantiate all the necessary modules so that the processor can function on the FPGA.

SLC3_2.sv

```
`ifndef _SLC3_2__SV
`define _SLC3_2__SV

package SLC3_2;
```

The `SLC3_2` module does not have any inputs or outputs, but declares and instantiates the opcodes and instructions that the processor can implement. It essentially creates a template or model for each corresponding opcode, and breaks down what goes where in the 16-bit instruction. The four most significant bits are always reserved for the opcode, but the lower 12 bits differ on what instruction is being initialized.

The overall purpose of the module is to create a format for the 16-bit instructions that drive the processor. This module is then utilized by the other modules so that the instructions can be stored and used from memory properly. The bits of the IR are broken down similar to the guidelines established in Patt and Patel's LC3.

register_file.sv

```
module register_file(input Clk, Reset, LD_REG,
                     input [15:0] BUS,
                     input [2:0] DRMUX_out, SR1MUX_out, SR2,
                     output logic [15:0] SR1_Out, SR2_Out);
```

The `register_file` instantiates registers R0-R7. It also takes the inputs from the DRMUX, SR1MUX, and the SR2 signal. DRMUX determines the destination register for a given instruction, and SR1 and SR2 select the corresponding source registers for a given instruction. SR1 comes from a MUX because based on the SLC-3, SR1 can either be in `IR[8:6]` or `IR[11:9]`. However for any instruction involving a second source register the register is declared in `IR[2:0]`, so we directly mapped that to our SR2 input. The outputs from the `register_file` module are the actual data stored in the source registers for a given instruction. When called, the register file send the data through the ALU unit so that the correct data is pulled from the registers for the operation to be performed.

The overall purpose of this module is to store data from the bus to the corresponding destination registers, and output the data from the necessary source registers.

memory_contents.sv

```
import SLC3_2::*;
```

The `memory_contents` module stores all the instructions to different locations in memory. Each `mem_array[]` declaration holds a different instruction that the PC can point to. As PC moves through the memory contents, the instructions are fetched and then decoded by the ISDU. When testing on the FGPA, we set the switches to a specific location in memory, and when `Run` is toggled the PC moves through memory so that it can fetch the instructions necessary to perform the corresponding test.

The module does not really have inputs or outputs but utilizes the `SLC3_2` module which initializes all the opcodes and formats the instructions so that they can be stored in memory and eventually read by the processor.

MEM2IO.sv

```
module  Mem2IO (  input logic Clk, Reset,
          input logic [15:0]  ADDR,
          input logic OE, WE,
          input logic [9:0]  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 );
```

The `MEM2IO` module communicates between our external I/O device which is the FPGA, and our memory. It checks whether OE or WE are enabled so it knows whether to read or write to memory, and also checks the input `ADDR`. If `ADDR` is `4'hffff`, then MEM2IO tells the processor to either input data from the switches on our FPGA, or output data to the hex displays on our FPGA, depending on whether the operation is read or write. However is the `ADDR` is not `4'hffff`, then MEM2IO does not interact with our external FGPA and internally assigns inputs `Data_from_CPU` to `Data_to_SRAM` and the `Data_from_SRAM` to `Data_to_CPU` and displays `4'h0000` on the FPGA hex displays.

The overall purpose is to transfer memory contents to the I/O device if necessary for the instruction. When necessary, the ADDR will read `4'hffff`, otherwise the MEM2IO will handle memory transfer internally without interacting with the I/O device, which in our case is the FPGA.

instantiateram.sv

```
import SLC3_2::*;

module Instantiateram( input Reset,
                input Clk,
                output logic [15:0] ADDR,
                output logic wren,
                output logic [15:0] data);
```

The `instantiateram` module is similar to the `test_memory` module, but instantiates the actual RAM on the FPGA. It takes `Reset` and an internal `Clk` signal as its inputs. ADDR outputs the address in memory and data represents the instruction at that memory. It also outputs `wren` which is write enable, so when data is being written to memory.

The overall purpose of the module is to store the instructions for the test cases being run on the FPGA so that the PC and move through the instructions and carry out the operations.

HexDriver.sv

```
1  module HexDriver (input  logic [3:0]  In0,
2                    output logic [6:0]  Out0);
```

Assigns 4-bit inputs to corresponding hex values. The purpose of the `HexDriver` is to output the sum to the FPGA.

Block Diagram of SLC3.SV

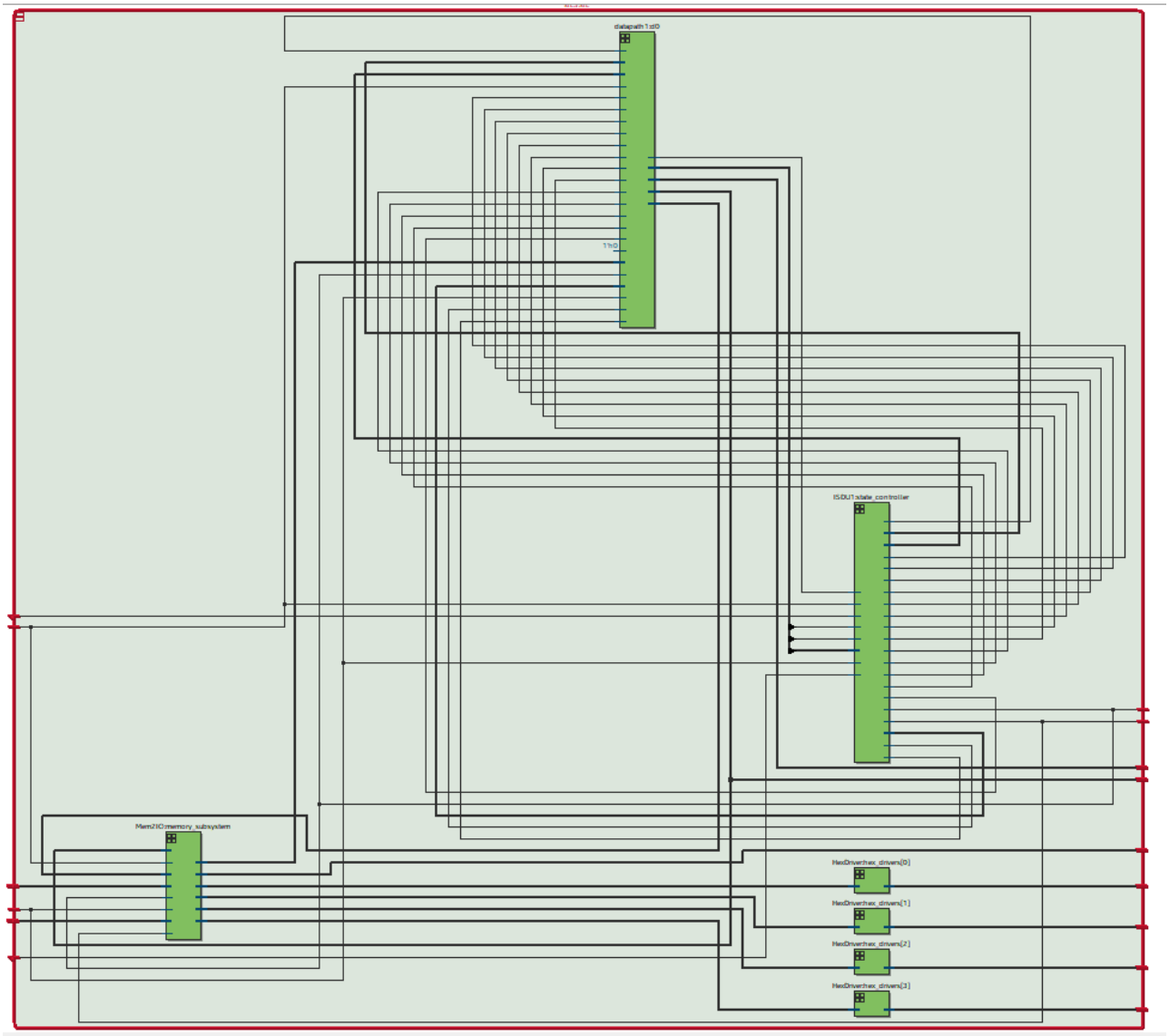

Figure 4: SLC3 Top Level RTL

Figure 4 shows the RTL block diagram of our slc3 module. It contains the ISDU1, datapath1, MEM2IO, and HexDriver modules. These sub level modules perform the different operations in the overall slc3 and output the results on the hex displays on the FPGA.

Simulations

Cursors are the yellow lines on the simulation figures

I/O Test 1

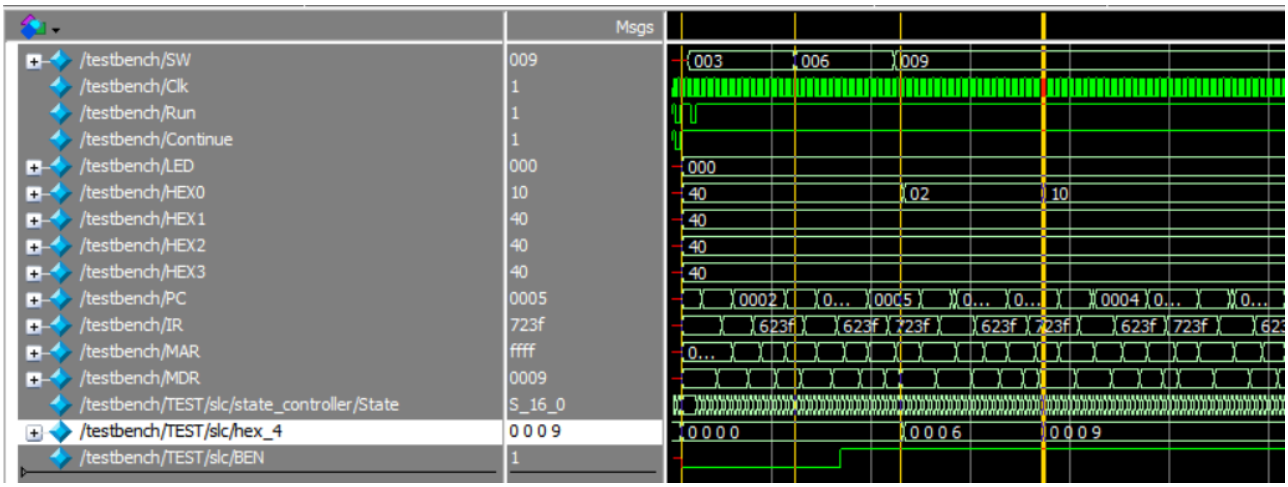

Figure 5: I/O Test 1 Simulation

Figure 5 shows the RTL simulation for I/O Test 1. The purpose of this test is to test instructions $ANDi$, LDR, STR, and BR. Essentially once `Run` is toggled, this test will display the value of the switches to the hex displays. Cursor 1 shows `003` begin loaded into the switches initially. This sets the value of `PC`, and tells our SLC-3 where to start the execution of test 1. Cursor 2 shows `x006` being loaded into the switches, and at Cursor 3, we see that `hex_4` also holds this value. In simulation, `hex_4` represents the value of the hex display on the FPGA, so we see that this test simulates successfully. Cursor 3 also shows that the switches now hold `x009`, and this new switch value is then shown in the hex display at Cursor 4, where `hex_4` also holds `x009`.

I/O Test 2

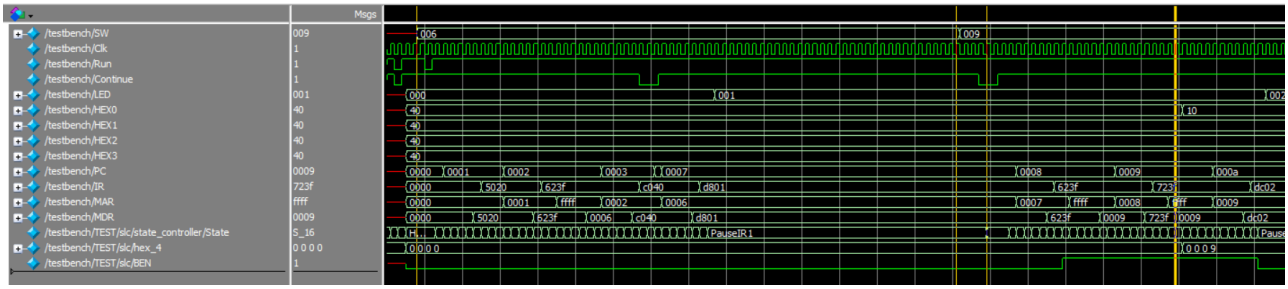

Figure 6: I/O Test 2 Simulation

I/O Test 2 is very similar to Test 1, except for that the value of the switches is not displayed on the FPGA until `Continue` is toggled. At Cursor 1, we see that the initial switch input executes the test by setting PC value. Cursor 2 shows the new value of the switches as `x009`, which is the value we want to display on `hex_4`. Cursor 3 shows

`Continue` being toggled, after which the value of `SW` is reflected in `hex_4` at Cursor 4. Figure 6 shows a successful cycle of Test 2 from setting up the test to reflecting the correct result.

Self-Modifying Code

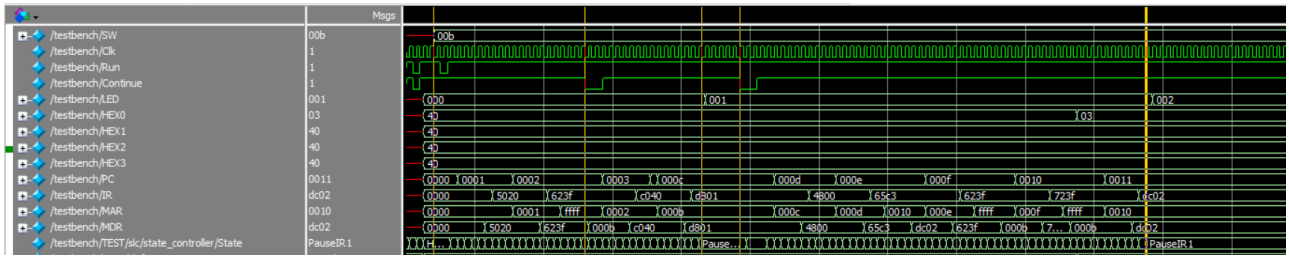

Figure 7: Self-Modifying Code Simulation

The Self-Modifying Code test essentially increments `LED` each time `Continue` is toggled. Cursor 1 shows the set up of the test at memory location `x00b`, after which `Run` is toggled. This begins the test. Cursor 2 shows the first toggle of `Continue`, after which Cursor 3 shows the data in `LED` changing from `000` to `001`. After then we toggle `Continue` again at Cursor 4, and `LED` is once again incremented at Cursor 5 from `001` to `002`.

XOR

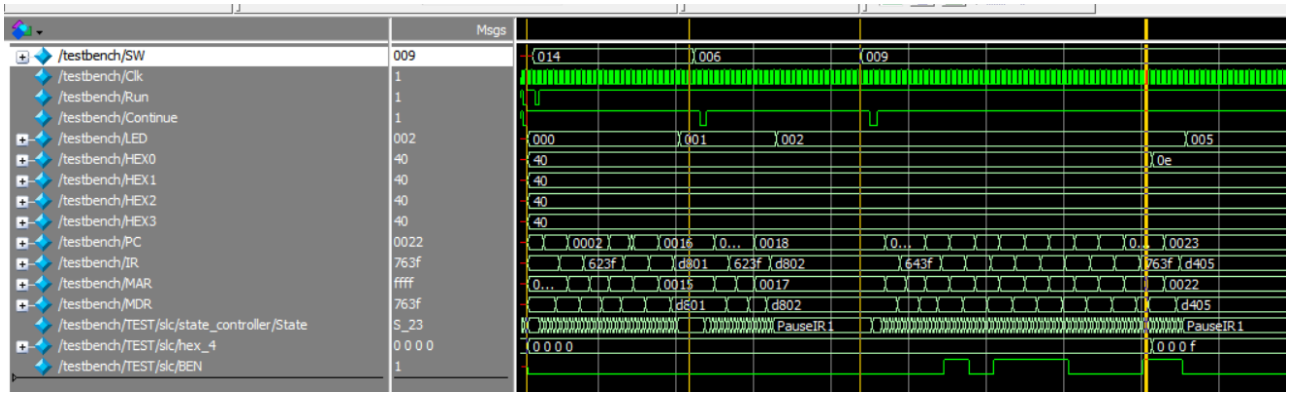

Figure 8: XOR Simulation

The XOR test takes two inputs A and B from the switches and returns $A \oplus B$. Cursor 1 shows this test being called by `PC`. Cursor 2 and Cursor 3 show the two values being computed and loaded into the processor. Cursor 2 shows the value `x006`, which is `0110`. Cursor 3 shows `x009`, which or `1001`. If we XOR these values, our expected result is `1111`

or `000f`, which is the value stored at `hex_4` at Cursor 4. Therefore Figure 8 shows a successful simulation of the XOR test.

Multiplier

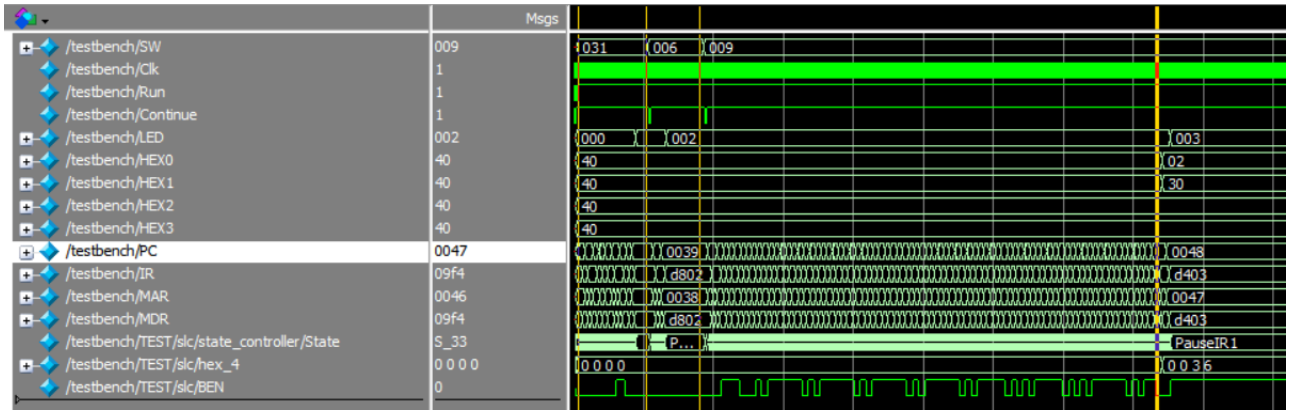

Figure 9: Multiplier Simulation

For our Multiplier test simulation, we used the values `x006` and `x009`, which when multiplied return a decimal product of 54, or `x0036` in hexadecimal. Figure 9 shows our simulation of the test. Cursor 1 shows initializing the test at memory location `x0031`. Cursor 2 shows loading the value `x006` from switches to the processor and Cursor 3 loads the value `x009`. Once `Continue` is toggled after `x009` is loaded in from switches, we see the results of the simulation at Cursor 4, which correctly shows the value `x0036` in `hex_4`, which is what we were expecting.

Sort

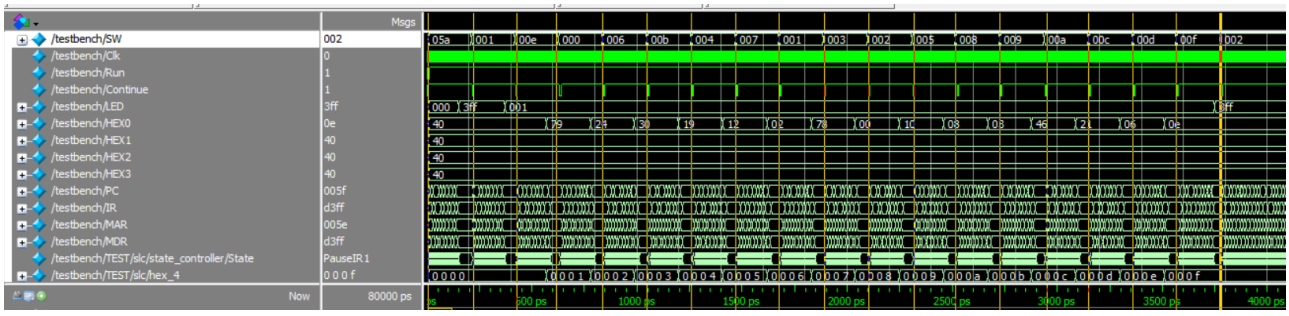

Figure 10: Sort Simulation Execution and Loading

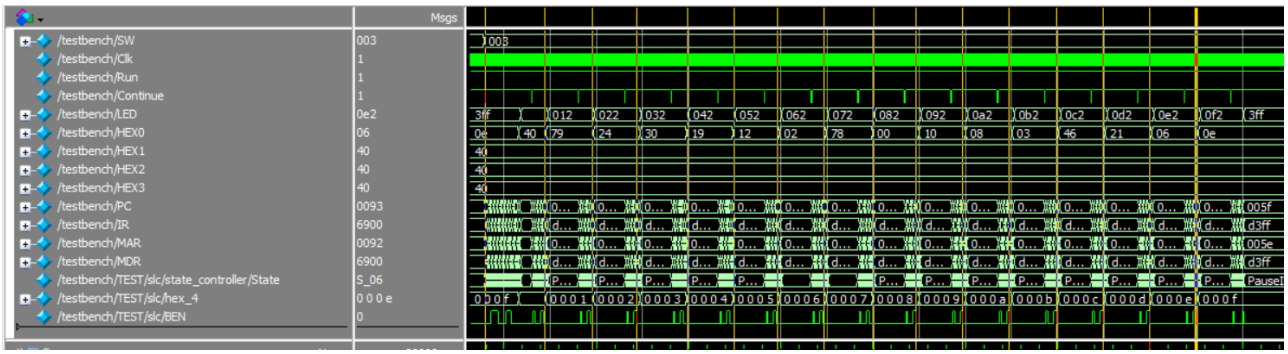

Figure 11: Sort Simulation Sort and Display

Figure 10 shows the set up and load part of the sort test. We call the sort test at Cursor 1 by setting the PC values. As per the sort test, the first thing we use the switches for is to select the menu. At `x01` we first load the 16 values to be sorted. For simplicity of the simulation, we used values `x0000 - xffff`. Cursors 3 - 19 show us loading the values to be sorted out of order. The `hex_4` values during this window increment to tell the user which index to be entered. Figure 11 shows the second half of the sort test, starting with the sort instruction. After the values to be sorted have been loaded, the test returns to the menu. Cursor 1 in Figure 11 shows the switches at `0x02`, which actually sorts the data. Once sorted, the switches show `0x03`, which then tells the test to display the sorted values. The remaining Cursors in Figure 11 show 16 toggles of the `Continue` signal, and `hex_4` shows the sorted values being displayed. The values are displayed in order, which shows that our simulation is successful for the sort test.

Extra Credit

XOR

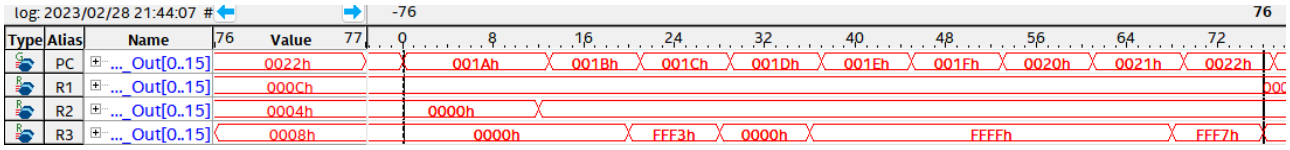

Figure 12: XOR Test Signal Tap

To calculate our MIPS value for our XOR function, we had to set our trigger to PC=0x1A. In our XOR we first set the switches to the first value and press continue and then change the switches to our second value and press continue again. We did an XOR between hex values C and 4 resulting in hex 8. We want to see how the PC changed during the actual computation and not when we set the values so that is why we set our trigger to a PC

value such that when we press the second continue and begin computing the XOR the SignalTap reflects that. The XOR instructions are quite straight-forward so there are only 8 instructions being done and 76 cycles. Therefore with a 50MHz frequency we have 50,000,000 cycles per second which we can use to find
$MIPS = (8/76) * 50,000,000 = 5263157.89$. This means that our CPU is capable of approximately 5.26 MIPS (millions of instructions per second) on the XOR test.

Multiplication

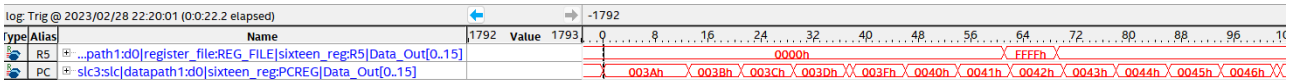

Figure 13: Multiplication Test Signal Tap Start

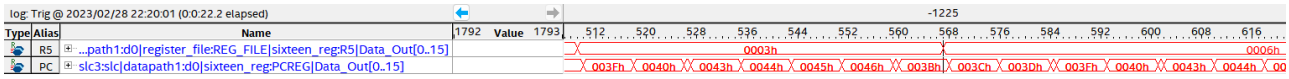

Figure 14: Multiplication Test Signal Tap End

To calculate our MIPS value for our Multiplication function, we had to set our trigger to PC=0x3A. To compute multiplication we first set the switches to the first value, pressed continue, and then changed the switches to our second value so we can begin computation on the second press of the continue button. We did an XOR between hex values 2 and 3 resulting in hex 6. We want to see how the PC changed during the actual computation and not when we set the values so that is why we set our trigger to a PC value such that when we press the second continue and begin computing the multiplication the SignalTap reflects that computation. Unlike the XOR operation there is a loop in our multiplication computation so we have to account when trying to find the number of instructions and cycles. In the image above we can compare the initial and final PC values and also multiply the amount of times we do the loop by the number of instructions in the loop to get an instruction count of 66 and cycle count of 568. This is much higher than the XOR computation. With a 50MHz frequency we have 50,000,000 cycles per second which we can use to find
$MIPS = 66/568 * 50,000,000 = 5809859.15$. This means that our CPU is capable of approximately 5.81 MIPS (millions of instructions per second) on the multiplication test.

Sort

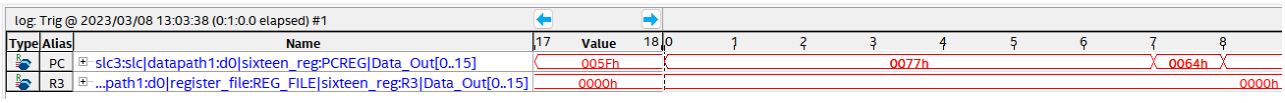

Figure 15: Sort Test Signal Tap Start

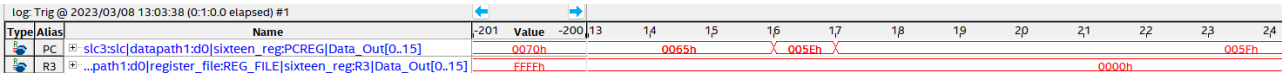

Figure 16: Sort Test Signal Tap End

To calculate our MIPS value for our SORT function, we had to set our trigger to PC=0x77. To sort all of our fifteen inputted values we have to input each of our values and press continue until the HEX LED displays F. We then set the switches to 0x02 and sort and then set the switches to 0x03 to display our sorted values. We inputted random values to confirm it was sorting and since we want to see how the PC changed during the actual sorting and not when we set the values so that is why we set our trigger to a PC value such that when we sort the SignalTap reflects it. Unlike the multiplication operation there is are two loops and not one so we have to account to both of them in finding the number of instructions and cycles. In the images above we can compare the initial and final PC values and also multiply the amount of times we do each the loop by the number of instructions in the loop to get an instruction count of 75 and cycle count of 738. This is much higher than the instruction and cycle counts of the XOR and multiplication tested. With a 50MHz frequency we have 50,000,000 cycles per second which we can use to find $MIPS = (75/738) * 50,000,000 = 5081300.81$. This means that our CPU is capable of approximately 5.08 MIPS (millions of instructions per second) on the sort test.

Post Lab Questions

Question 1

LUT	1128
DSP	0
Memory (BRAM	294912
Flip-Flop	1116
Frequency	66.41 MHz
Static Power	89.98 mW
Dynamic Power	3.46 mW
Total Power	102.25 mW

Figure 17: Design Statistics and Resources

From our design statistics table we can notice a few changes from the previous labs and see just how much more we are processing in this lab. Starting in the lookup tables, we are using an astounding 1128 LUTs whereas in lab 4 (8-bit multiplier), we only used 93 lookup tables. In this experiment we began to implement memory which we had not even looked into in previous labs and the ability to read and write to memory allowing us to refer to many different instructions and complete a variety of operations. Although the static power is similar to the previous lab, the dynamic power is much higher because we have to use more power when we switch as we have many different states to go to. Using over 1000 lookup tables and memory in this lab allows us to complete operations at the faster speeds than in previous labs which is quite surprising.

Question 2

What is MEM2IO used for, i.e. what is its main function?

In our lab, we need to connect our SLC-3 processor to the SRAM module since we need to connect our FPGA board memory to the FPGA chip and we can do so using the MEM2IO as a bridge. It allows us to read data from the switch ot the memory and display the output data onto the HEX display. We can also differentiate 0xFFFF as load and store addresses since at some point we want to load from switches and when we finish our operations we want to store out to the hex displays rather than storing in SRAM.

Question 3

What is the difference between BR and JMP instructions?

LC-3 has the condition code registers N (negative, Z (zero), and P (positive) that are set by any instruction that writes a value to a register (ADD, AND, NOT, LD, LDR, etc.). Exactly one is set at all times and based on the NZP values, we can use BR (Branch) and move to a target address. We can use it to create loops as it only branches if the condition code is true otherwise it continues execution. The branch location is determined by adding the sign-extended PCoffset9 to the PC. Jump, on the other hand, is an unconditional branch that moves to the any target address which is stored in the contents of a register. This means that we always jump to the target, whereas we only move to the target address in the BR operation if the condition code is true.

Question 4

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 implications does this have for synchronization?

In Patt and Patel, the R signal is used in three states as the state has a self loop on !R. The R signal means that data from memory has been read and we need this for states that need to read memory otherwise we will move onto the next state with the wrong things being done, leading to the overall task being incorrect. States 16, 25, and 33 need to read from memory so they self loop until memory has been read and then they can move onto the next state. Our FPGA does not have the ready signal so to compensate for this we added several wait states in each of those three states. For example, instead of state 16 we would use state 16, 16_0, 16_1, and 16_2 to represent wait states and allow enough time for memory to be read and the operation to then be completed. Our change has no implications for synchronization since it will always take a determined amount of clock cycles to complete the operation and would not require synchronization.

Conclusion

Design Functionality

In week one, we ran into minimal errors. One we ran into, however, was that we ran into a problem with our PC not incrementing. It was because we had variables PCMux and PC_Mux which had two different intended uses but we had PCMux where we should have used PC_Mux. In week two, we ran into various errors which ruined the functionality of our design. In our ISDU file, we had states that were missing control signals that were integral in their functionality and some of our control signals were set high where they needed to have been set low. We found this out when we began testing in ModelSim and seeing at which state our project would not load or compute properly. In our arithmetic states, we did not have DRMux signals so they failed to store the computed value in a register. We did not have Bench Enable conditions set so our Branch operation would not work and we fixed this by adding them. Additionally, when we tested on our FPGA, we were unable to change the displays to show any new values so we looked into our hex_4 values. There, we found out that our test worked on ModelSim but when ADDR[15] was equal to 0xFFFF, the value would not get loaded to our hex_4 to be displayed on the HEX LEDs. We realized that our LD_Reg signal was not functioning properly so we fixed that in order to output our hex_4 to the HEX displays. Another mistake we made was that we did not follow the provided data path control signals in the multiplexers so sometimes the wrong value of PCMux would get chosen. For example, the provided control signal for PCMUX when the select bit was 00 was to select “PC+1” but our PCMUX selected the opposite: “ADDER”. After fixing the orientation of the PCMUX our code was one step closer to being fully functional. Finally, a bug that we had was that we would get stuck in our pause states because we had incorrect conditions set. When we noticed this in our simulation, we changed the condition so we move on when continue is pressed, and fixing this bug allowed us to move on to finding more critical bugs and finally finish our code so it could function properly.

Lab Manual Feedback

Out of the documents provided, the lab manual explained what we needed to do and where to start. The testing document, however, was slightly confusing since not all of the tests would explicitly provide a procedure on what buttons to initially press and what we should see as a correct output on the LEDs or HEX displays. Although we were able to understand what to do at some time we could have understood better how to test if there was a procedure on what to press and what to look for when testing for functionality. Additionally, we had some trouble perfectly tracing an operation through the SLC-3 datapath since it had been some time since we did, and the method we had used provided us with control signals that were missing a few elements. Several other groups during office hours ran into similar problems where their ISDU control signals for several states

may have been incorrect and required debugging. We could have benefitted from a tutorial that explained the best method of tracing one of the signals through the SLC-3 data path.

Summary

In this lab, we designed a microprocessor using Verilog. It was to be a subset of the LC-3 ISA with a 16-bit Program counter, instructions, and registers. We were to accomplish the eleven operations (ADD, ADDi, AND, ANDi, NOT, BR, JMP, JSR, LDR, STR, and PSE) with provided opcodes in the instruction register IR. There are three basic states of SLC-3: Fetch, Decode and Execute. We will first fetch the instruction by loading our PC into the MAR and incrementing the PC. We then load data from CPU from SRAM to the MDR and read from the address in the MAR so we may store it in the IR. Then we will decode which operation it is by comparing all of the bits in the IR to find the appropriate state in the ISDU that we want to begin with so we may proceed to execute the operation. Finally, we execute by going through the states of the ISDU and making necessary changes to the registers or memory to complete the operation. After this is all done, we increment the program counter and fetch again. We had the lab split up into two weeks where in week one we wanted to display the fetch state of SLC-3 and in week two we wanted to build upon that and complete decode and execute as well so we may demonstrate our completed simple microprocessor. For the arithmetic logic operations (ADD, ADDi, AND, ANDi, and NOT) our SLC-3 will compute with two registers or one register and an offset then store that value accordingly by the chosen destination register. The BR instruction will occur based on bench enable (BEN) and move based on the condition codes that may have been set after completing prior instructions. The LDR and STR instructions will have memory read or write to the SRAM memory and in our JMP instruction our program counter will move to the target memory address. JSP will do the same as JMP but it will also store PC in our R7 register. Finally, in PSE we are in the pause states so our LEDs will light up based on which checkpoint we are in and change when we press continue, moving on to the next state. Altogether, we were able to successfully implement the aforementioned operations and use them in succession to complete longer and more complicated computations and tests.