ECE 385

Spring 2023
Experiment #5

Simple Computer SLC-3 in SystemVerilog

Max Ma & Dhruv Kulgod
NL / Friday 9:15 AM - 9:30 AM
Nick Lu

Purpose of Circuit

SLC-3 is a general purpose processor based on a Von Neumann architecture with 16-bit instruction and data registers. It contains a register file, arithmetic logic unit, and control unit to execute basic arithmetic, logic, store, load, and branching operations. Additionally, the SLC-3 processor stores data and instructions in an on-chip SRAM module for low latency memory operations. Finally, it incorporates an I/O controller to interface with memory mapped peripheral devices such as hex displays and switches.

Written Description of Circuit

Overview

The design of the SLC-3 can be categorized into the following: the Microarchitecture and the Instruction Set Architecture. The Microarchitecture is composed of the physical registers, memories, and control signals used to fetch, decode, and execute instructions on each clock cycle. The Instruction Set Architecture defines the instructions that the SLC-3 can perform as well as the step-by-step RTL operations needed to successfully execute these instructions.

Microarchitecture - The Puppet

As described early, SLC-3 is designed using a Von Neumann architecture, which means that instructions and data are coupled onto the same general purpose bus and share the same primary memory. In the SLC-3 Microarchitecture, there are four important registers: the Program Counter (PC), Instruction Register (IR), Memory Address Register (MAR), and Memory Data Register (MDR). These registers contain 16-bits, and they are critical to the operation of RTL instructions defined by the Instruction Set Architecture. The PC contains the memory location of the next instructions in primary memory. The IR contains the current instruction being executed by the Microarchitecture. The MAR and MDR hold the address and data of the current memory location, respectively. The Control Unit (ISDU.sv) sets control signals to store to and load from these registers during the execution of each instruction. Moreover, the Microarchitecture contains a simple Arithmetic Logic Unit (ALU) to perform ADD, AND, and NOT operations. It also utilizes a Register File containing eight 16-bit registers for low-latency, temporary storage required during the execution of instructions. This avoids having to constantly access primary memory, which can be slow by taking multiple clock cycles.

Instruction Set Architecture - The Puppet Master

The SLC-3 Instruction Set Architecture (ISA) uses 16-bit instructions to perform operations using the datapath described by the Microarchitecture. To perform the execution of any instruction, the SLC-3 must complete the all-important Fetch-Decode-Execute cycle. During the fetch state, the instruction at the memory location of the PC is loaded into the IR and the PC is incremented to the next instruction. In the decode state, the control unit uses the opcode (upper four bits in the IR) to determine the operation to perform. Finally, the execute state performs the set of RTL operations described by each operation. The ISA describes a hand full of arithmetic

[ADD, ADDi], logic [AND, ANDi, NOT], store [STR], load [LDR], branching [BR, JMP, JSR], and I/O [PAUSE] operations. To perform these operations, the control unit sets specific control signals during each clock cycle.

High Level Block Diagram

As we can see from the top-level RTL (Figure 1), the overarching structure is fairly simple. We have the slc3 processor itself, a RAM module to store instructions and data, and IO modules - 10 switches, two buttons for input and the hex display as output. We also have an “instantiate RAM” module to initialize the contents of RAM properly at the start.

The slc3 processor requires access to data from the RAM, a clock, as well as signals for “Run”, “Continue” and “Reset”. We will explore this module in depth later, but in essence, the slc3 processor will access the first instruction from RAM. This instruction contains details of a given operation to be executed (according to the ISA). The processor decodes this instruction into a set of signals that help it control the various hardware MUXs, register files etc. Once it completes the first instruction, it will access the next memory location and repeat the execution process.

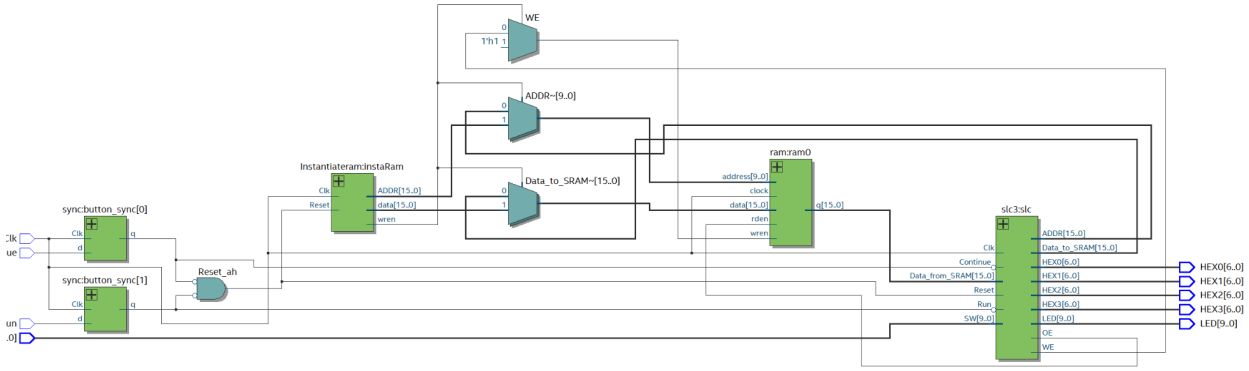

Figure 1: Top-level RTL

State Diagrams and Tables

The state diagram for the ISDU is very similar to the state diagram given to us except for one change. In our implementation, we do not have a memory "ready" signal (signified by 'R'). Instead, we allow the memory three clock cycles to have a memory value ready for us to read.

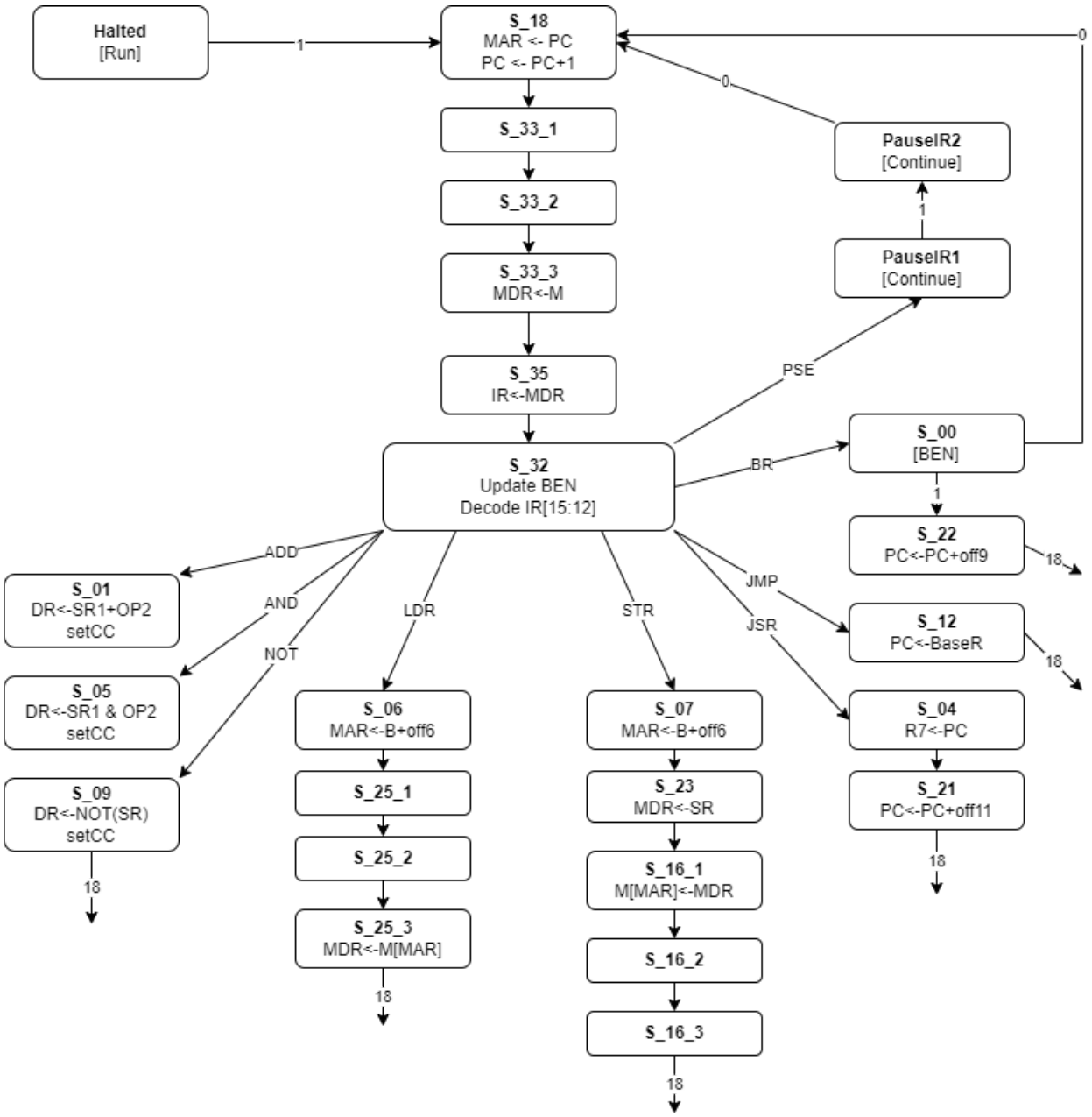

Figure 2: State Transition Diagram

For the most part, the next state logic has no conditions attached to it, except for a few cases. In S_32 (Decode), the next state is determined by the opcode present in the instruction register (IR[15:12]). In S_00 (conditional branch), if the condition is met (BEN = 1), the PC value jumps.

If the condition is not met, we simply execute the next instruction. The pause instruction routine has two states to account for the fact that we must physically press the button to continue, and the next instruction should not execute till we release the continue button.

For read operations, we read the memory output value three clock cycles after we update MAR. For write operations, we simply supply the correct address and value for 3 clock cycles while holding memory write-enable high.

The ISDU as a whole was written in SystemVerilog using the 2-always methodology. That is, have separate always blocks for assigning control signals to the datapath and for moving to the next state. This ensures a proper synchronous design.

Answers to Post-Lab

Design Resources and Statistic Table for Simple Computer SLC-3.2	
LUT	936
DSP	0
Memory	18,432
Flip-Flop	258
Frequency	24.01 MHz
Static Power	90 mW
Dynamic Power	9.71 mW
Total Power	110.55 mW

Figure 3: Design Resources and Statistic Table

What is MEM2IO used for, i.e. what is its main function?
The MEM2IO module is used to implement the memory mapped I/O functionality for SLC-3. It polls MAR checking for the memory address xFFFF. Reading from memory location xFFFF loads the switch values into MDR. Writing to memory location xFFFF sets the HEX display to the data in MDR.

What is the difference between BR and JMP instructions?
Although both instructions changed the value stored in the Program Counter, their scopes are different. The BR instruction will only modify the PC if the conditions codes match the NZP bits of the instruction. This will be primarily used in implementing control structures within the assembly language. Additionally, the BR instruction only adds a SEXT number to the PC, giving

it a small scope. On the other hand, the JMP instruction has no conditions and will change the PC value to the address in the Base Register upon completion. This gives the JMP instruction a large scope as it can change the PC to any location in memory.

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?

The R signal in Patt and Patel describes the Ready signal for reading from primary memory. If this signal is logic level high, the memory output is valid for the current memory address. In the design of the SLC-3, the SRAM memory module does not output a Ready signal. To compensate for this, we implement a minimum of three wait states for the data output from the SRAM memory module to be valid. Although this solution worked for the experiment, it could potentially lead to synchronization issues. For example, increasing the system clock could decrease the wait time, leading to invalid or inconsistent reads for the SRAM memory module.

FPGA Labs: Block Diagram

The block diagram of slc3.sv is shown below in Figure 4. It incorporates separate Datapath, ISDU, Mem2IO, and Hex driver modules.

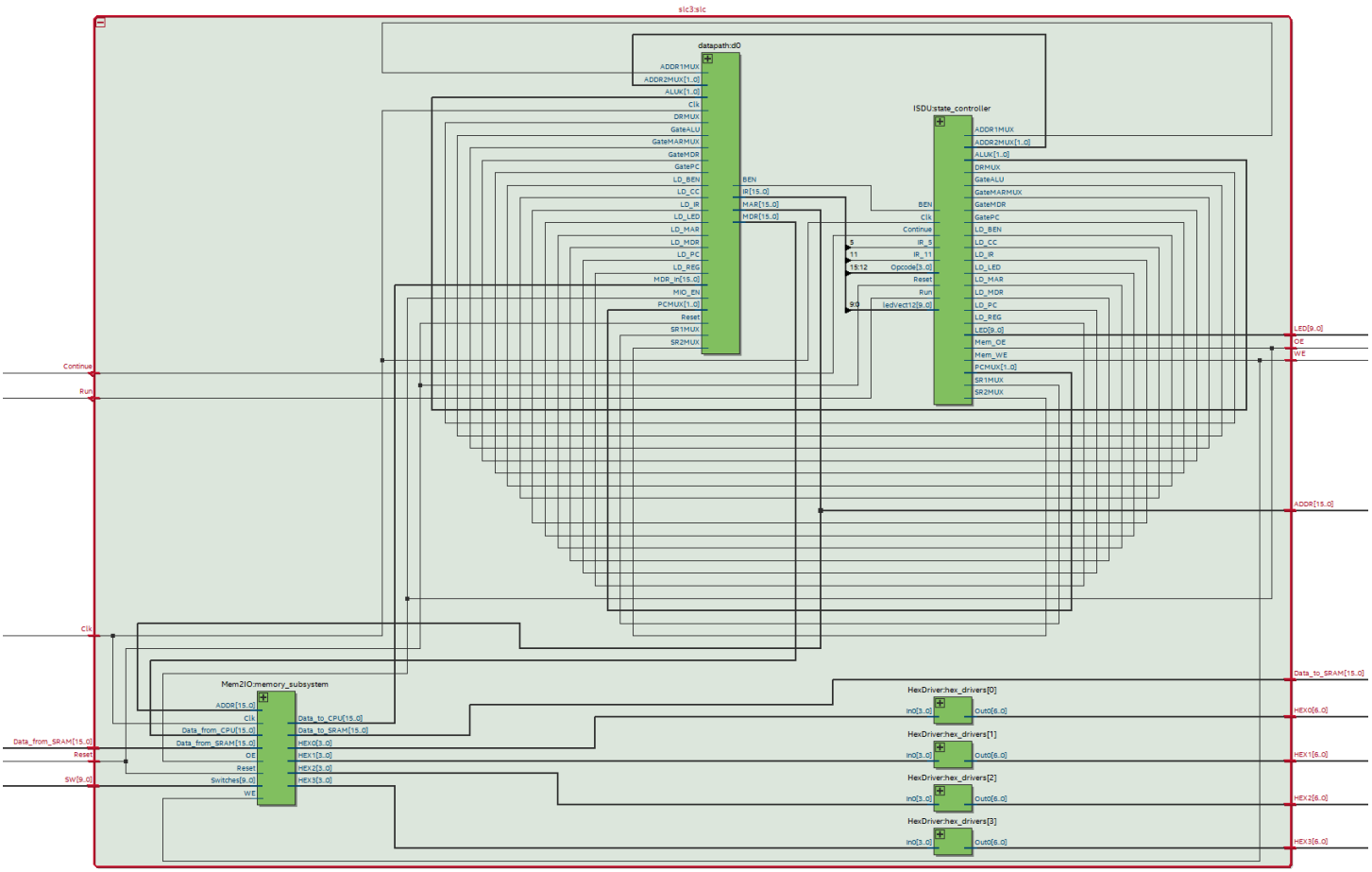

Figure 4: Block Diagram of slc3.sv

FPGA Labs: Design Module Information

Datapath:

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

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

Description: This module constructs the datapath of the SLC-3 processor. In essence, this module directs data to and from the various adders, registers and MUXes according to the control signals received from the ISDU. It includes logic for setting the CC register, calculating the branch condition, incrementing the PC counter, etc.

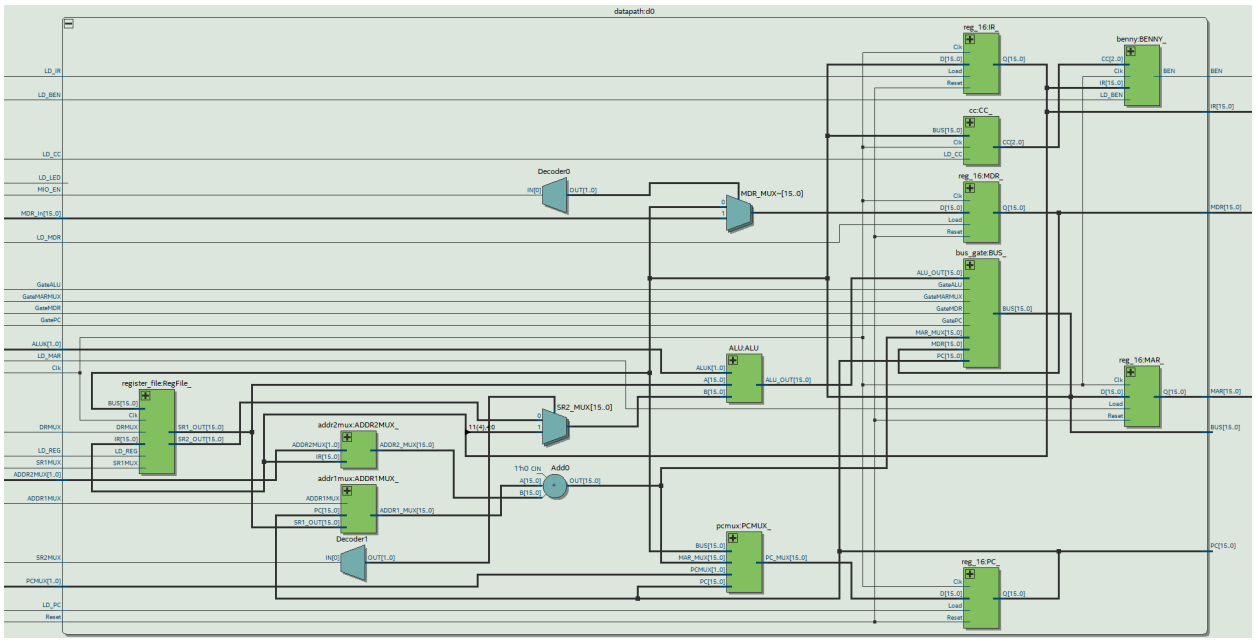

Figure 5: Block Diagram of dathpath.sv

Mem2IO:

Inputs: Clk, Reset, OE, WE,
[15:0] ADDR, Data_from_CPU, Data_from_SRAM
[9:0] Switches

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

Description: As the name suggests, this module handles the transfer of data to and from the memory and IO modules. Specifically, it helps us implement the memory-mapped IO implementation of our design. The two IO devices (HEX driver and switches) are

mapped to memory address xFFFF. Reading from this address returns the value at the switches, while writing to this address writes to the HEX display. Data to and from CPU interacts with MDR, while ADDR is taken from MAR.

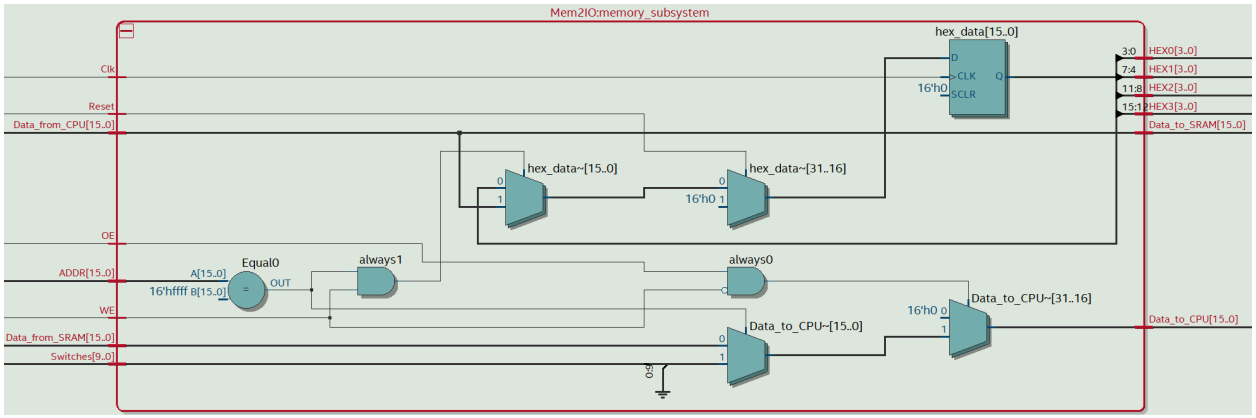

Figure 6: RTL of Mem2IO module

Slc3_sramtop:

Inputs: [9:0] SW,
Clk, Run, Continue
Outputs: [9:0] LED,
[6:0] HEX0, HEX1, HEX2, HEX3
Description: Top module meant for running on the FPGA. It only interacts with the IO of the device, and passes these values to the slc3 and RAM modules internally.

ISDU:

Inputs: Clk, Reset, Run, Continue, IR_5, IR_11, BEN,
[3:0] Opcode,
[9:0] ledVect12
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, Mem_OE, Mem_WE,
[1:0] PCMUX, ADDR2MUX, ALUK,
[9:0] LED
Description: This module contains the major aspect of the design - the state diagram and control signals. Using a 2-always technique, this module decides which state the control unit should move to next. Additionally, each state has a set of control signals it needs to specify, depending on the action the processor is supposed to take in that cycle. For example, in state 35, the processor is supposed to copy data from MDR to IR. It must pass the right control signals (here, set GateMDR = 1 and LD_IR = 1 while not passing any other data to the BUS) to execute this without conflict.

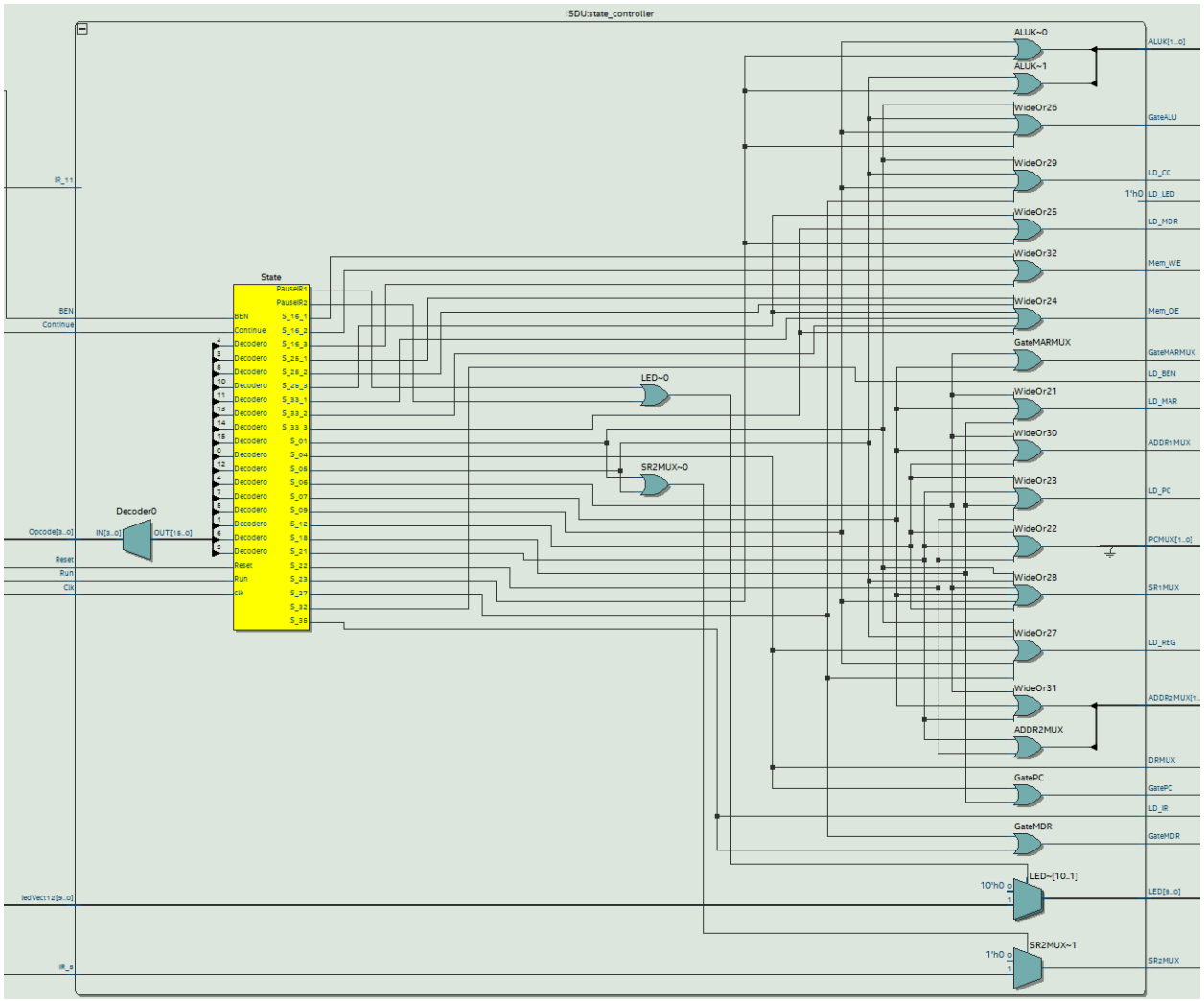

Figure 7: Block Diagram of ISDU.sv

Synchronizers:

Inputs: Clk, d

Outputs: q

Description: Synchronizes our physical buttons and switches i.e. ensures their values are only changed at each rising edge of the clock. This helps mitigate timing issues in the design.

ALU:

Inputs: [15:0] A, B,
[1:0] ALUK
Outputs: [15:0] ALU_OUT
Description: The Arithmetic and Logical unit of the CPU, capable of performing 4 basic operations on 16-bit values (AND, ADD, NOT and passthrough). These operations allow us to perform any logical operation (this set is logically complete) albeit makes it difficult to perform complication functions due to the lack of dedicated multiply, divide etc.

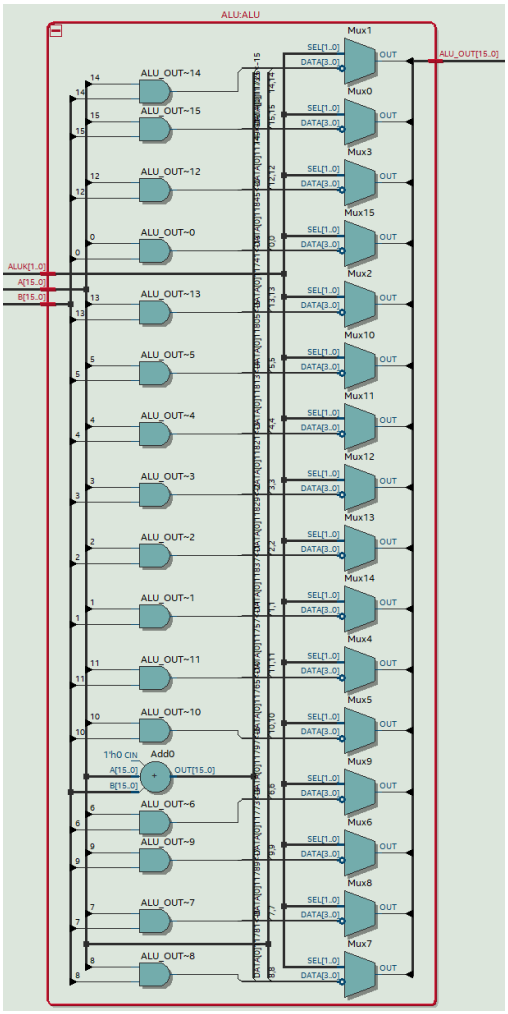

Figure 8: RTL of 16-bit ALU

HexDriver:

Inputs: [3:0] In0
Outputs: [6:0] Out0
Description: This module acts as a translator between binary to hexadecimal. A set of four binary bits are taken as input and linked to their corresponding pattern on the 7-Segment Hex Display using a case statement.

Test_memory:

Inputs: Reset, Clk, rden, wren,
[15:0] data,
[9:0] address
Outputs: [15:0] readout
Description: Non-synthesizable module meant to simulate the physical memory meant for testing. Intended to perform at least as well as actual RAM.

Register_file:

Inputs: Clk, SR1MUX, DRMUX, LD_REG,
[15:0] BUS, IR
Outputs: [15:0] SR1_OUT, SR2_OUT
Description: Module meant to implement the functionality of the register file of the lc3 datapath. It is basically an unpacked [7:0] array of 16-bit registers. Depending on the registers chosen, the module will pass the values of the right register to the datapath for processing instructions.

Instantiateram:

Inputs: Reset, Clk
Outputs: wren,
[15:0] ADDR, data
Description: Module meant to instantiate the processor's RAM module on startup. This puts all the instructions to be run into RAM so the processor knows what to do.

Memory_contents:

Inputs: N.A.
Outputs: N.A.
Description: Contains the actual memory contents to be written to memory. If we were to write our own program to run on the FPGA, we would have to write it here.

SLC3_2:

Inputs: N.A.
Outputs: N.A.
Description: In essence, this is a library to help us write programs in lc3 binary. It allows us full control over all possible instructions without requiring us to write in actual 1s and 0s.

FPGA Labs: Annotated Simulation Waveform

The annotated simulation waveforms are shown below (Figures 9-15) for all 6 test programs.

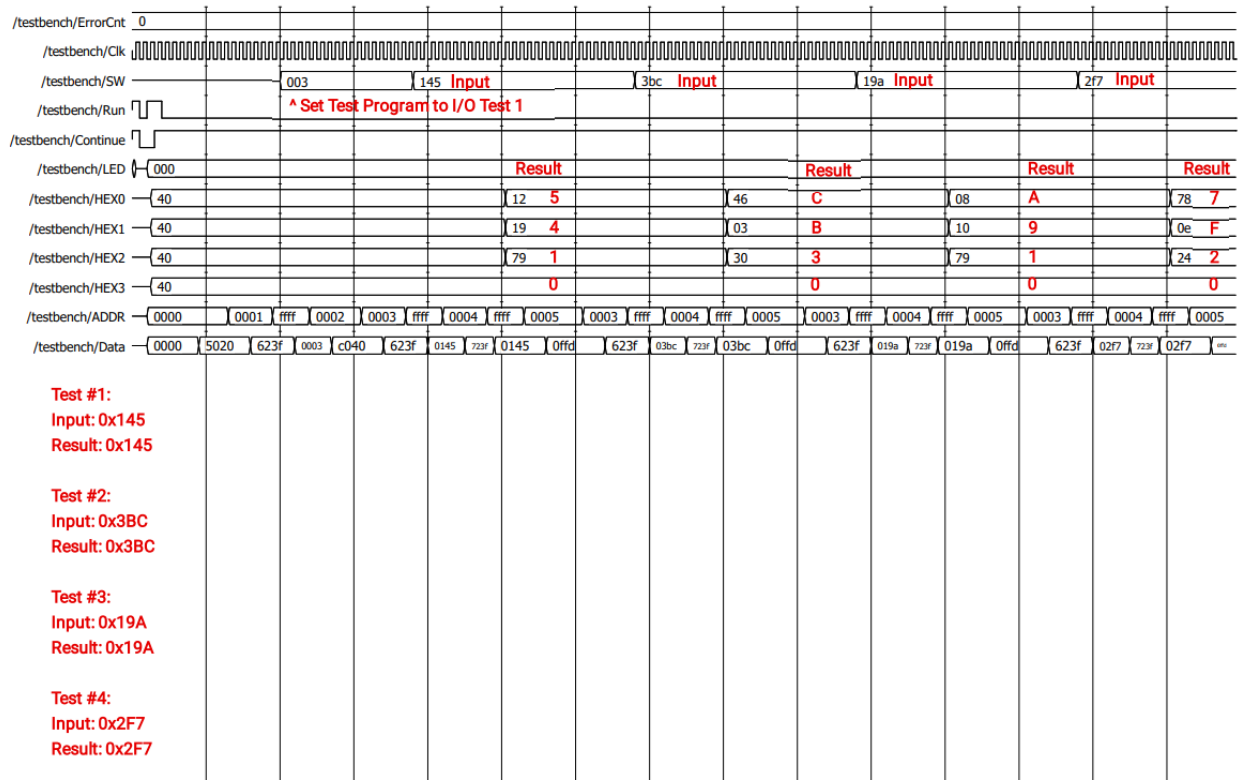

Figure 9: Annotated Simulation Waveform for I/O Test 1

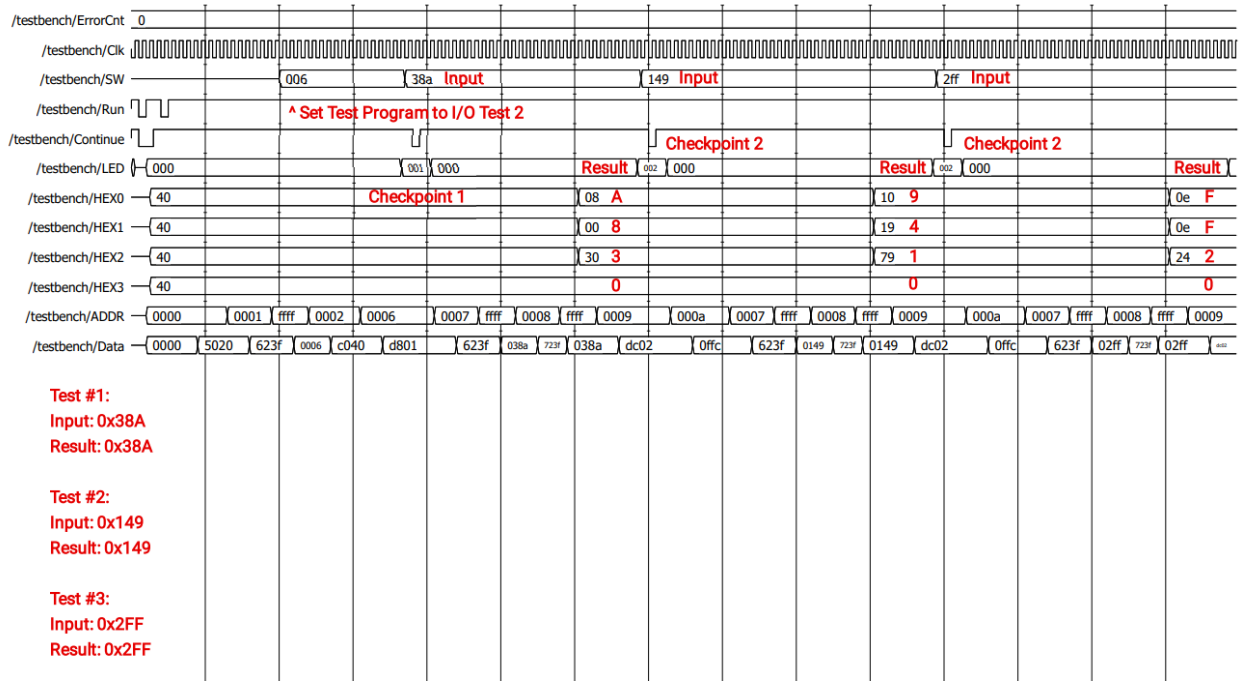

Figure 10: Annotated Simulation Waveform for I/O Test 2

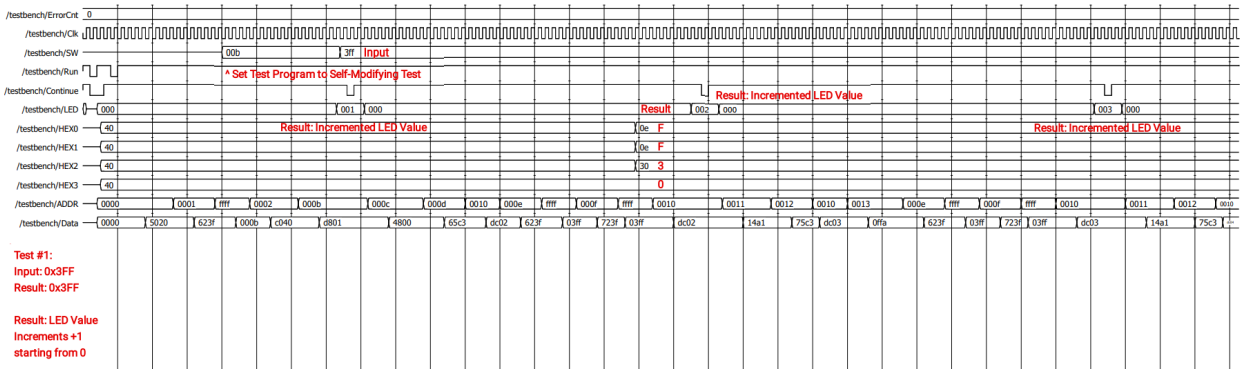

Figure 11: Annotated Simulation Waveform for Self-Modifying Test

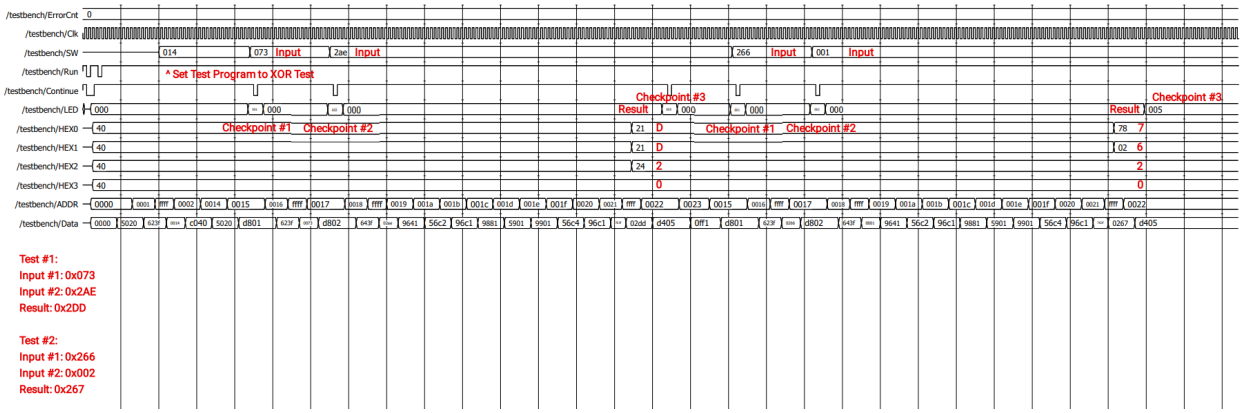

Figure 12: Annotated Simulation Waveform for XOR Test

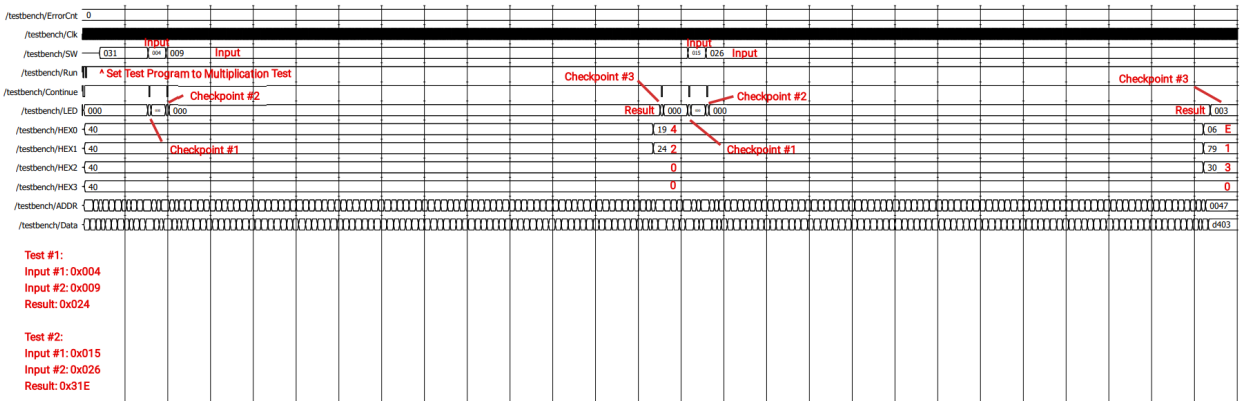

Figure 13: Annotated Simulation Waveform for Multiplier Test

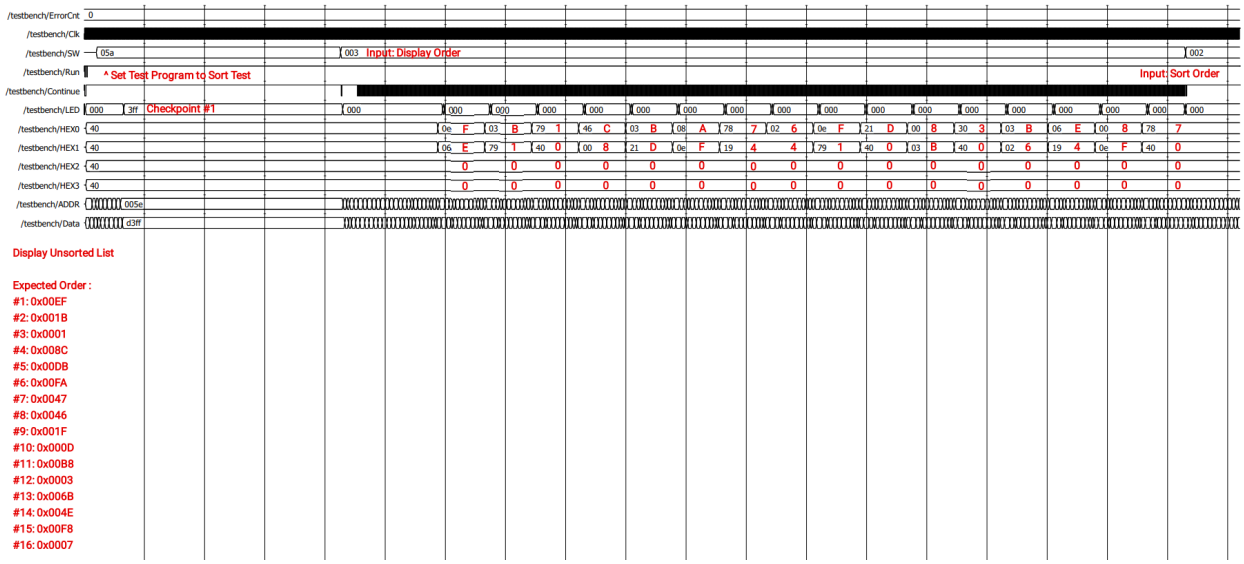

Figure 14: Annotated Simulation Waveform for Sort Test: Unordered Display

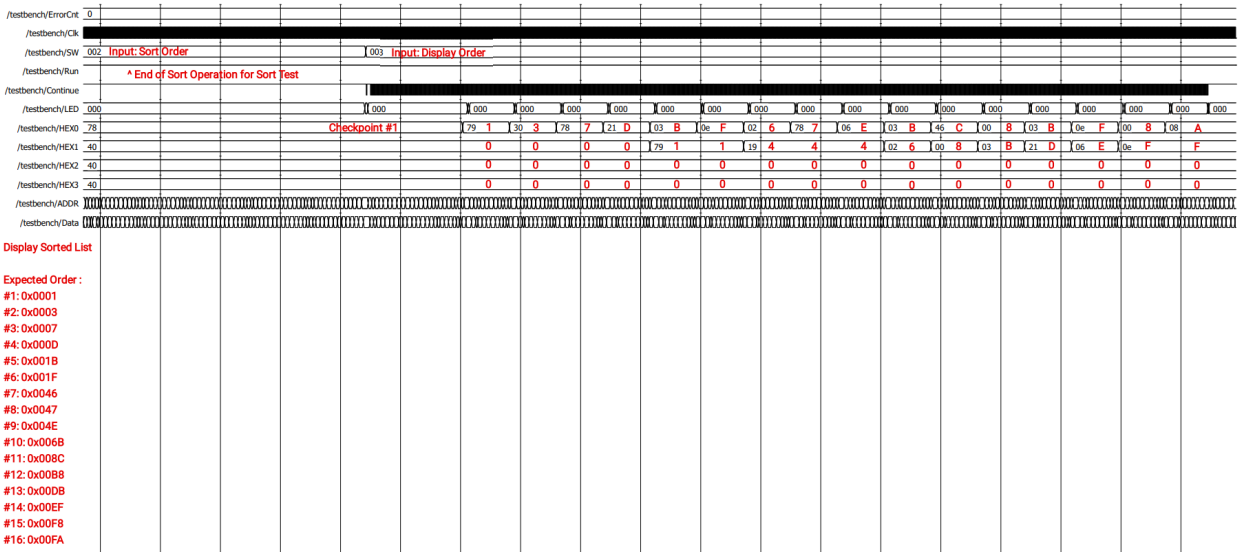

Figure 15: Annotated Simulation Waveform for Sort Test: Ordered Display

Conclusions

Although the completion of this experiment was smooth, we encountered two problems. One, we did not allocate three clock cycles to read from the primary memory, which caused inconsistent fetch behavior. Adding two additional wait states to our control unit when reading data from memory solved this problem. Second, we were not able to pass some of the tests in Lab 5.2 because of incorrect branching behavior. After fixing the transitions from our branch instruction, we were able to solve this problem. Overall, careful planning and diligent execution helped us minimize debugging time, leading to a successful, well-planned experiment.