ECE385
Spring 2023
Experiment #5

Simple Computer SLC-3.2 in SystemVerilog

V Verma & Matthew Guibord
3/20/2023
Hanfei Wang

Introduction

In this lab, we designed and implemented a microprocessor that uses a simplified version of the LC-3 Instruction Set Architecture (SLC-3) using SystemVerilog. Most processors, including ours, contain three main design components: a CPU (Central Processing Unit), memory, and an I/O (input/output) interface; we were provided with the interface between memory and the CPU (the memory read and write functionality), memory contents for testing, and the preliminary framework for the state machine the SLC-3 ISA functions off of.

Written Description and Diagrams of SLC-3

Summary of Operation

Describe in words how the SLC-3 performs its functions. You should describe the Fetch-Decode-Execute cycle as well as the various instructions the processor can perform.

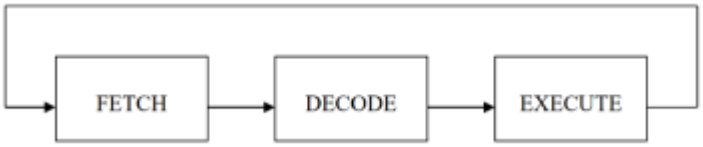

Figure 1: Visual representation of the Fetch, Decode, and Execute steps that are needed in the microprocessor

There are many parts to SLC-3 but its main function is a simple microprocessor which was implemented using System Verilog. It is a subset of the LC-3 ISA, a 16-bit processor with a 16-bit program counter, 16-bit instructions, and 16-bit registers. It can process 11 different instructions. The processor includes a state machine which includes a fetch, decode, and execute cycle. The first step of the cycle includes loading the program counter into the memory address register and incrementing the program counter by 1. This counter holds the memory address of the next instruction to be executed. The next state in the fetch step is to load with MDR with the data from memory. The SRAM can be accessed using the MEM2IO interface provided in this lab along with the fetch module containing the memory data register. This may have a delay as it takes time to access and read/write from memory so in our implementation we include three wait states after this state. The following and final state in the fetch step is to load the contents of the memory data register into the instruction register. This instruction register holds the instruction in which the programmer wants to execute. This instruction can be manipulated in memory by the programmer. The next step is the decode state which determines which is where the finite state machine branches off into a different sequence of following states depending on which instruction must be executed. This also utilizes logic, the NZP signals, and bits 11-9 in the instruction register to assign the branch enable signal. After this, there are many states in which the finite state machine can branch off to each pertaining to a different instruction. There are many of these instructions which are listed below with descriptions. The data path is another essential part of the SLC-3 implementation. This consists of the control unit which sends out control signals to the rest of the processor, a register unit consisting of 8 16-bit registers, an ALU, an adder, and registers including the instruction register, N, Z, P, BEN, and the program counter. Through the manipulation of the control signals from the control unit, the data path carries out the different functions of each state in the Fetch-Decode-Execute cycle.

- 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.

Block Diagram of slc3.sv Written Description of all .sv modules

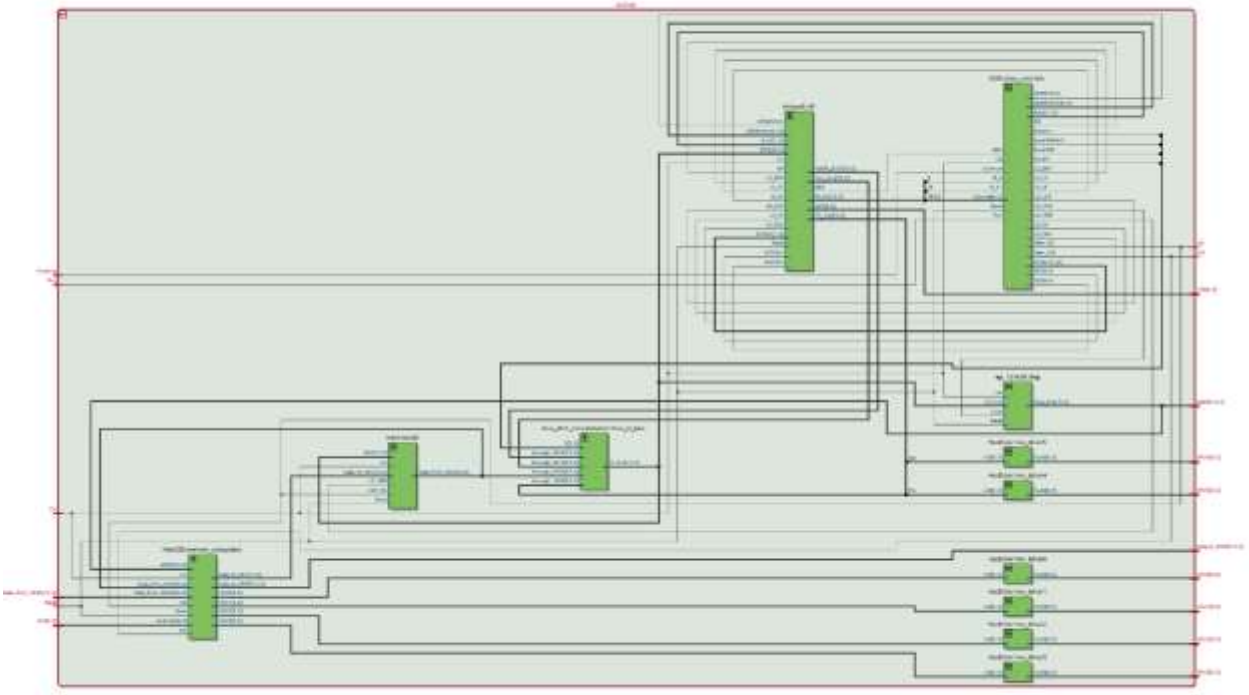

Figure 2:RTL Block Diagram of the SLC-3 generated by Quartus

Module: test_memory.sv
Inputs: Reset, Clk, data, address, rden, wren
Outputs: readout
Description: This creates memory with similar behavior to the SRAM IC on the DE2 board. It creates this memory in a way where there is no need to program the FPGA and initialize the ram in the FPGA but instead simulate the memory.
Purpose: This memory is used for simulation and the initial debugging of the lab.

Module: Synchronizers.sv
Inputs: Clk, d
Outputs: q
Description: This file represents a flipflop implementation to synchronize the push buttons and the switches with the clock.
Purpose: The purpose of this module is to act as a de-bouncer for the switches and push buttons

Module: slc3_testtop
Inputs: SW, Clk, Run, Continue
Outputs: LED, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5
Description: Creates a connection between the test memory, slc3, and the inputs (reset, continue, and run).
Purpose: Used for simulated inputs and simulated memory for ease of debugging before programming FPGA.

Module: slc3_sramtop
Inputs: SW, Run, Continue
Outputs: LED, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5
Description: Creates a connection between inputs on FPGA, instantiated ram on FPGA, and slc3
Purpose: Used for the "On-FPGA" version of the Lab 5 implementation meant to be programmed on the FPGA.

Module: slc3
Inputs: SW, Clk, Reset, Run, Continue, Data_from_SRAM
Outputs: LED, OE, WE, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, ADDR, Data_to_SRAM
Description: This module consists of Mem2IO, ISDU, fetch, mux_4to1_concatenation, reg_16, and Datapath to connect the bus, and all of the Datapath logic with fetching the memory and displaying the results to the hex drivers.
Purpose: Brings all of the upper-level modules including Datapath and ISDU to connect all of the control signals sent from the ISDU to the logic in Datapath as well as fetching the memory from Mem2IO.

Module: memory_parser
Inputs: N/A
Outputs: output logic[15:0] mem_array[0:size-1]
Description: Initializes the memory with LC3 commands which can be modified to create custom test programs.
Purpose: Used to create test programs for ease of debugging and lab demonstration.

Module: Mem2IO
Inputs: Clk, Reset, ADDR, OE, WE, Switches, Data_from_CPU, Data_from_SRAM
Outputs: Data_to_CPU, Data_to_SRAM, HEX0, HEX1, HEX2, HEX3
Description: Receives data from the Memory Address Register to fetch the data in SRAM and send it in a signal to be loaded into Memory Data Register
Purpose: Provides an interface between SRAM and the rest of the Datapath.

Module: ISDU
Inputs: Clk, Reset, Run, Continue, IR_5, IR_11, BEN, Opcode
Outputs: LD_MAR, LD_MDR, LD_IR, LD_BEN, LD_CC, LD_REG, LD_PC, LD_LED, GatePC, GateMDR, GateALU, GateMARMUX, PCMUX, DR, SR1MUX, SR2MUX, ADDR1MUX, ADDR2MUX, ALUK, Mem_OE, Mem_WE
Description: Takes current state and IR_5, IR_11, BEN, and Opcode to choose the next states and control signals such as select bits, register load signals, and read/write enable signals. Includes a decode state and pause states for different lc3 instructions and ease of debugging.
Purpose: Is the finite state machine of the 16-bit processor unit and sends out control signals to the rest of the implementation.

Module: Instantiateram
Inputs: Reset, Clk
Outputs: ADDR, wren, data
Description: Allows for the instantiation of the ram while the FPGA remains on power, or else it would be required to unplug and reprogram the FPGA to instantiate the memory.
Purpose: Utilized in the hardware implementation of Lab 5 used for debugging, testing, and in the lab demonstration

Module: Hex_Driver.sv
Inputs: In0
Outputs: Out0
Description: This file takes in the input In0 and maps that value to a sequence of cases, each mapping to a different Out0 value. This Out0 value is representative of which LED's should receive high voltage to display the value In0.
Purpose: This module maps an In0 value to be displayed to which LED's should receive voltage to display this desired value, which is Out0

Module: datapath
Inputs: LD_PC, Reset, LD_IR, LD_REG, ADDR1MUX, SR2MUX, SR1MUX, LD_CC, LD_BEN, DR, LD_LED, BUS, PCMUX, ALUK, ADDR2MUX, Clk
Outputs: PC_Val, IR_Val, ALU_Out, ADDR_OUT, BEN, LED
Description: Utilizes many of the other supporting modules listed below to drive the bus of the processor unit as well as the BEN value and IR value.
Purpose: Defines the 4 different signals which drive the bus through the ISDU and IR as well as assigns the registers their values.

Module: fetch
Inputs: MIO_EN, LD_MDR, Clk, Reset, Data_to_CPU, BUS
Outputs: Data_from_CPU
Description: Includes the implementation and interface of Mem2IO into MDR and defines Data_from_CPU which drives GateMDR and feeds back into Mem2IO.
Purpose: I used to define MDR and load it into IR while also providing support for the interaction between the memory and the rest of the processor unit.

Module: mux_2to1
Inputs: S, through_1, through_0
Outputs: Q_Out
Description: Is a standard mux with two inputs and one output. Q_Out is defined based on the select (S) value. The bit length of Q_Out is parameterized.
Purpose: This mux is used in the implementation of ADDR1MUX, SR1MUX, DRMUX, MIO.EN, and SR2MUX.

Module: reg_16
Inputs: Clk, Reset, Load, D
Outputs: Data_Out
Description: This is a standard 16 bit register with a reset signal and loads D into Data_Out if the load signal is high.
Purpose: 16 Bit registers are used to store MDR, MAR, PC, IR, and the registers in RegFile

Module: reg_10
Inputs: Clk, Reset, Load, D
Outputs: Data_Out
Description: This is a standard 10 bit register with a reset signal and loads D into Data_Out if the load signal is high.
Purpose: A 10 bit register is used to hold the LED value.

Module: mux_4to1
Inputs: S, through_00, through_01, through_10, through_11
Outputs: Q_Out
Description: Is a standard mux with 4 inputs and one output. Q_Out is defined based on the select (S) value.
Purpose: This mux is used in the implementation of PCMUX and ADDR2MUX

Module: mux_4to1_concatenation
Inputs: S, through_0001, through_0010, through_0100, through_1000
Outputs: Q_Out
Description: Is similar to a standard 4_to_1 mux but only 1 bit of the 4 bit select (S) signal is at 1 meaning that the select signal is a product of the concatenation of 4 control signals.
Purpose: This mux is used in the implementation of the mux that drives the BUS using GatePC, GateMDR, GateALU, and GateMARMUX as the control signal concatenation.

Module: reg_file
Inputs: D, DRMUX, SR2, SR1_MUX, LD_REG, Clk, Reset
Outputs: SR2_OUT, SR1_OUT
Description: Contains 8 16-bit registers and uses 2 8-bit registers with control signals SR1_MUX and SR2 to choose the values of SR2_OUT and SR1_OUT which go to the rest of Datapath
Purpose: the 8 16-bit registers hold the values of R0-R7 in the processor unit.

Module: decoder_3to8
Inputs: S, LD_REG
Outputs: Ld_signals
Description: This is a standard 3 to 8 decoder with DRMUX as the input and LD_REG as the single bit that toggles between 1 or 0 depending on the ISDU. This is optimal as if LD_REG is 1, then it acts as a standard decoder but if LD_REG is 0, then no matter what the input is, the output is x0.
Purpose: This is used as the register select in the reg_file.

Module: mux_8to1
Inputs: S, through_000, through_001, through_010, through_011, through_100, through_101, through_110, through_111
Outputs: Q_Out
Description: Is a standard mux with 8 inputs and one output. Q_Out is defined based on the select (S) value.
Purpose: This mux is used in the implementation of the SR1_Out select and SR2_Out select in reg_file.

Module: sext
Inputs: IR_Val
Outputs: sext_val
Description: This is a standard sign extend of the inputted 5, 6, 9, and 11-bit IR_Val to 16-bits.
Purpose: These sign-extend values are used as inputs in ADDR2MUX and SR2MUX.

Module: ALU
Inputs: S, A, B
Outputs: Q_Out
Description: The Arithmetic Logic Unit is a core part of datapath which takes in two inputs A/B, and outputs either A+B, A&B, ~A, or A depending on the select bit (S)
Purpose: This ALU unit is used in the Datapath module with inputs SR1_OUT, and SR2MUX_OUT with output as ALU_OUT and select of ALUK

Module: reg_16
Inputs: Clk, Reset, Load, D
Outputs: Data_Out
Description: This is a standard 1 bit register with a reset signal and loads D into Data_Out if the load signal is high.
Purpose: 1 Bit registers are used to store N, Z, and P

Description of the operation of the ISDU (Instruction Sequence Decoder Unit)

The ISDU (Instruction Sequence Decoder Unit) is the control unit for the SLC-3. It contains the Fetch, Decode, and Execute steps that are needed to have a functional microprocessor and implements them as a state machine (view Figure 3). The state machine for the SLC-3 has three states (not including wait states) dedicated to Fetch (S18, S33, S35), one Decode state (S32), and sixteen states (not including wait states) within the Execute step of which only a few are executed after the Decode state. The Execute step is dependent on the Fetch and Decode steps; the Decode state in specific determining the path of execution. Within the Decode state the Opcode of the instruction determines what the next state (within the Execute step) is, the state machine then goes through the required sequence of steps needed to complete the instruction. The ISDU uses the Datapath to generate control signals that are defined in every state. These signals then determine and implement the function of the state, which completes the desired instruction or moves us towards the completion of the desired instruction.

State Diagram of ISDU

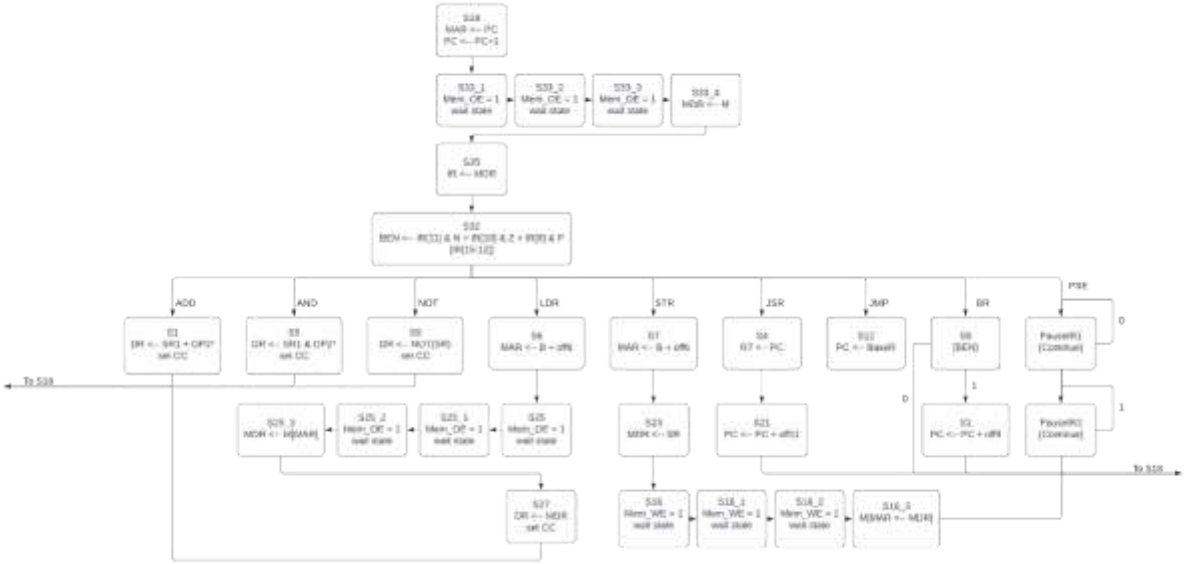

Figure 3: State Diagram of the ISDU for SLC-3

Annotated Simulations of SLC-3 Test Cases

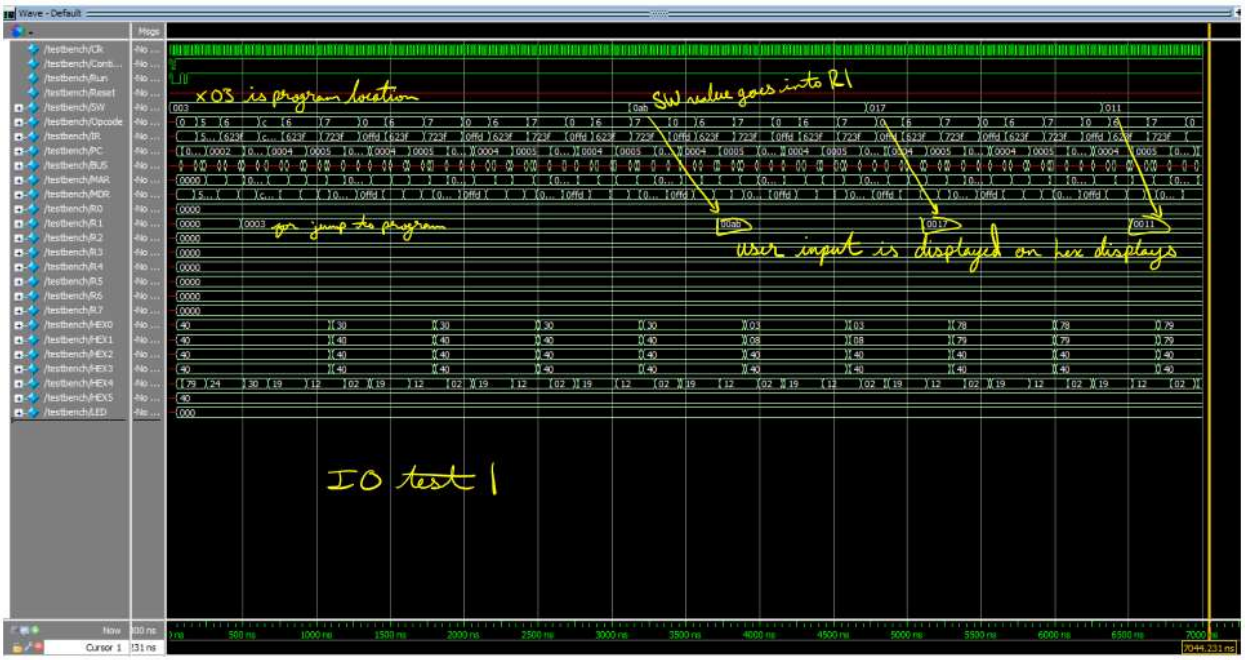

Figure 4: Annotated simulation for IO Test 1

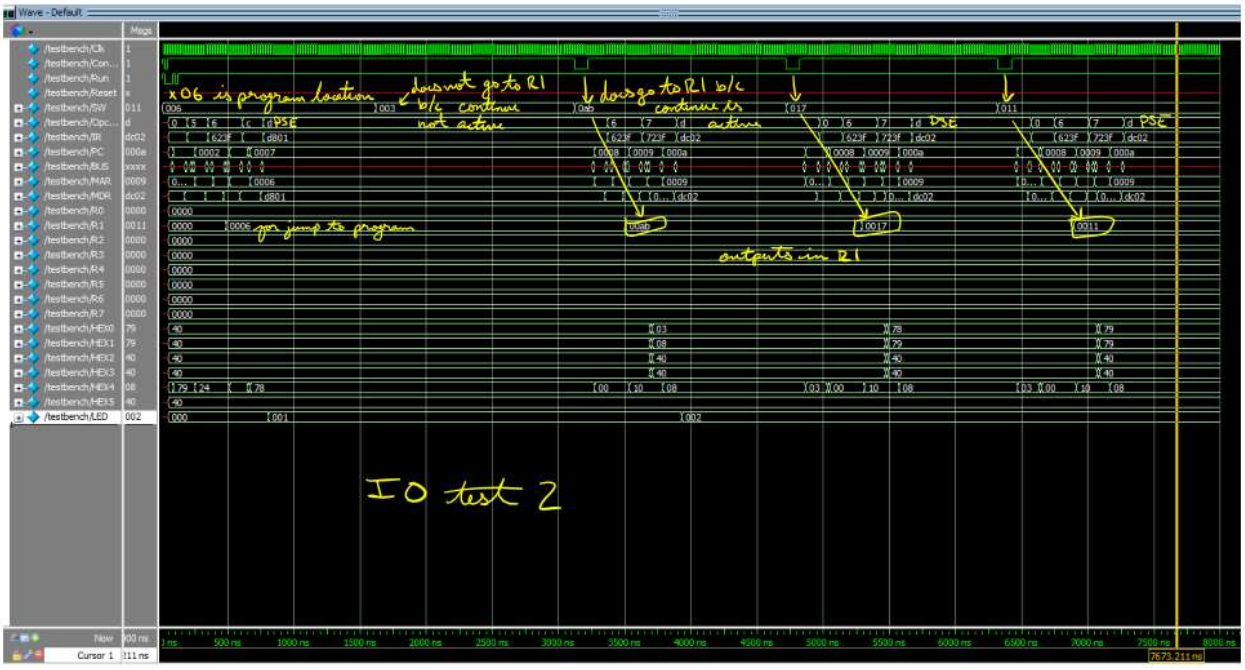

Figure 5: Annotated simulation for IO Test 2

Figure 6: Annotated simulation for Self Modifying Code Test

Figure 7: Annotated simulation for XOR Test

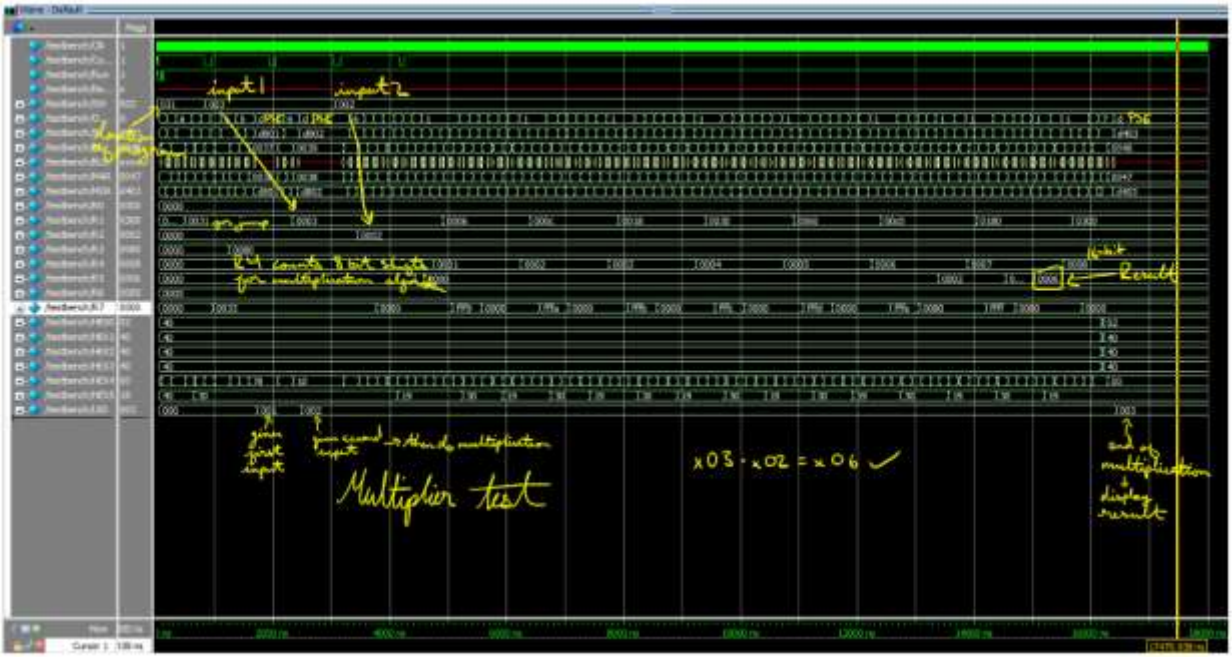

Figure 8: Annotated simulation for the Multiplier Test

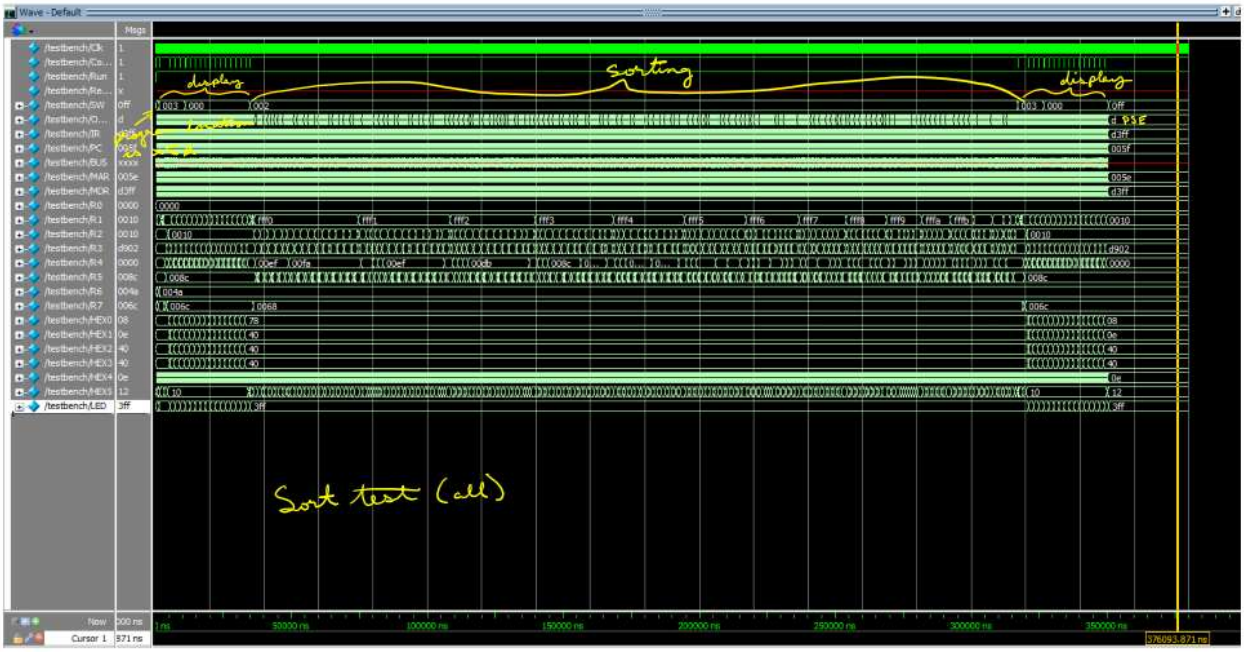

Figure 9: Annotated simulation for the full Sort Test which has three portions: pre-sort display, sorting, post-sort display
View Figures 6,7,8 for each specific portion of the full simulation

Figure 10: Annotated simulation for the pre-sort display portion of the Sort Test that displays the unsorted values

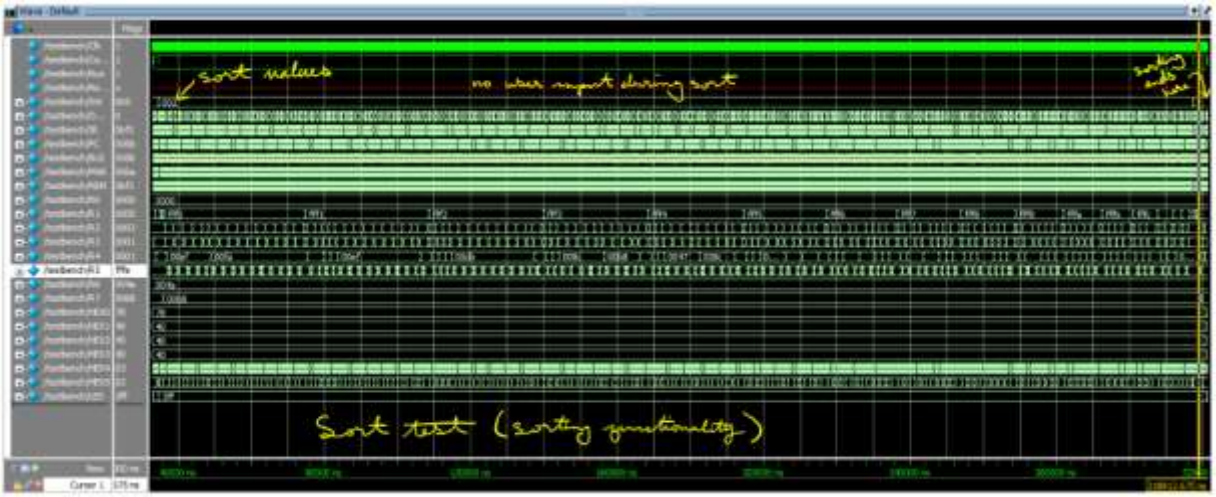

Figure 11: Annotated simulation for the portion of the Sort Test that is visualizing the sorting functionality using the inputted values

Figure 12: Annotated simulation for the post-sort display portion of the Sort Test that displays the sorted values

Post-Lab Questions

LUT	896
DSP	0
Memory (BRAM)	18,432 / 1,677,312 (1 %)
Flip-Flop	271
Frequency	66.57 MHz
Static Power	90.00 mW
Dynamic Power	11.96 mW
Total Power	101.96 mW

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

The main purpose of the MEM2IO is to facilitate the process of interacting with the 1.638Mbit of on-chip memory on the MAX10 FPGA. Whenever the address is xFFFF in MAR, the data is loaded from the switches but otherwise is loaded from SRAM. When the write enable signal is active and the address is xFFFF, MEM2IO writes the switch values to the LEDs. In all, MEM2IO assigns data to be passed to the CPU from the SRAM and data to be passed to the SRAM from the CPU.

What is the difference between BR and JMP instructions?

Though BR and JMP are very similar in many aspects, they are not entirely the same. JMP can change the current instruction to anywhere in the memory while BR is relative and can only jump a limited number of instructions away. JMP is generally used for subroutines with separate chunks of memory separated for those specific instructions to be executed while BR is typically used for conditional statements and loops.

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 is designed to compensate for the delay of fetching data from the memory. If R is 0, then the finite state machine stays at the current state. This means that the data is not ready, and the control signals stay consistent for fetching for the whole duration of the fetch. Since we don't have ready signal for fetching memory from data, we account for this by having an arbitrary amount of wait states so the control signals stay consistent for the whole duration of the fetch. In our case, 3 to 4 wait states should suffice. In terms of synchronization, the time it takes to fetch the data from memory is inconsistent and there isn't enough time allotted to fetch this data, there is uncertainty that the memory being fetched is from the current fetch, previous fetch, or inaccurate data.

Conclusion

The full design for the microprocessor is able to use the full functionality of SLC-3 on the DE10-Lite FPGA Board provided to us in this class. Every state works correctly and all programs (both provided and theoretically possible) written in SLC-3 are able to run properly.

During the design and implementation of the microprocessor there were various errors that hindered progress significantly however the item that most halted testing and debugging was the creation of the testbench for this lab. The implementation of the testbench for this lab was unintuitively explained within the provided resources and lab manual. It can be argued that there was little to no guidance on the implementation of the testbench within the provided resources making it near impossible for us to create a working testbench without accessing outside resources. Though with the help of Course Assistants and peers we were able to create a functional testbench having a guide on how to create a testbench for labs as complex as SLC-3 would be preferred as if this was better explained the time taken to complete this lab and frustration wrought upon my soul would have decreased exponentially. I would call the creation of the testbench unnecessarily difficult and poorly explained.

Other than the creation of the testbench for debugging, there were various issues within our ISDU due to small oversights within a few states which were difficult to find even with ModelSim. Staring at ModelSim for hours did help us better understand what values are important for debugging large circuits such as SLC-3 in Quartus, however it was mind numbing and hurt our brains.

This lab was fully completed and the final implementation contains no errors that can be found through the given test programs or through programs that we have written ourselves. We were able to become proficient in debugging in SystemVerilog and gained a deep understanding of ModelSim and its functionalities. This lab was also good for helping us move away from using modules to create functionality in SystemVerilog and move towards more implicit implementations. Week 1 of this lab was simple and well explained, however, week 2 (with fair warning) was much more difficult and time consuming.