ECE 385

Fall 2014

Experiment #2

A Logic Processor

Dylan Bautista, Matthew Wei

Lab Section NL/ Jan 27 11:00 AM

Nick Lu

Experiment #2

A Logic Processor

I. Introduction/Purpose of Circuit:

The Logic Processor displays two sets of 4-Bit values held within dual shift registers, which can be initially loaded with values with Load A/Load B and switches. Upon activating “Execute”, every clock cycle the bit pairs shift out one set at a time and proceed to have a preselected computation conducted on the values, with the result being routed to one or neither register depending on the routing selection. Both the choice of bitwise operation as well as the routing designation of the inputs and outputs are chosen through a three- and two-bit selection respectively. After four shifts and with the operation complete, the registers are now loaded with whatever the preselects indicate and can be viewed or continued to be operated on.

II. Operation of the Logic Processor:

a. Once the Logic Processor is connected to a supply voltage and Clock signal, data can be loaded into a register by first entering the four-bit values via four switches. Then, depending on if you want to enter these values in Register A, B, or both, you will activate the corresponding “Load A” or “Load B” button. Loading values will always overwrite the current values in the registers.

b. Once the registers are full of the desired values, the desired computation is selected by finding the corresponding 3-bit value in the table and entering it in the 3 switches. Similarly, the desired routing operation is selected by finding the corresponding 2-bit value in the table and entering it in the 2 switches. Then, once “Execute” is activated four shifts should take place accordingly, regardless of the value of “Execute” immediately after.

III. Written description, block diagram and state machine diagram of logic processor.

a. Unit Descriptions:

The Register Unit houses two 4-bit shift registers with parallel load capability from the 4-bit switches D3-D0 and the Load A and Load B inputs within the control Unit. The carry out of these initial values are serially computed altogether pair-by-pair until all 4 values have been replaced by the carry ins, which come from the Routing Unit. LEDs are connected to all 8 held values so they can be observed during and after operations. This unit is connected to a clock signal which determines when the shift right occurs.

The Computation Unit takes 3 bits as input from switches F2-F0 which enter an internal mux to select which computation takes place. This unit is connected in sequence to the Register Unit in order to receive the serial inputs of bit pairs to conduct bitwise logic on. All the logic for the 8 different options is included in this Unit, and the selected result is directed toward the Routing Unit along with passing inputs A and B. Whatever option is chosen will be in effect for the entire duration of the 4 shift cycles.

The Routing Unit takes 2 bits as input from switches R1-F0 which enter an internal mux to select which values get fed into the Register Unit as the carry-ins. The Routing Unit accepts the values A, B, and the computation result F from the Computation Unit, and are used in the four possible combinations of the routing table. Whatever option is chosen will be in effect for the entire duration of the 4 shift cycles.

The Control Unit takes in three inputs from switches Load A, Load B, and Execute. When Load A or Load B are held high, the 4 bits from switches D3-D0 are parallel loaded into the corresponding register. When Execute is held high, it prompts the Register Unit to start shifting

and thus computing values. Once the 4 cycles are finished and the computation is complete, all shifting will halt. During the computation, the value of Execute is ignored until it is complete. Also, Execute must be turned low again, then high in order to initiate another computation.

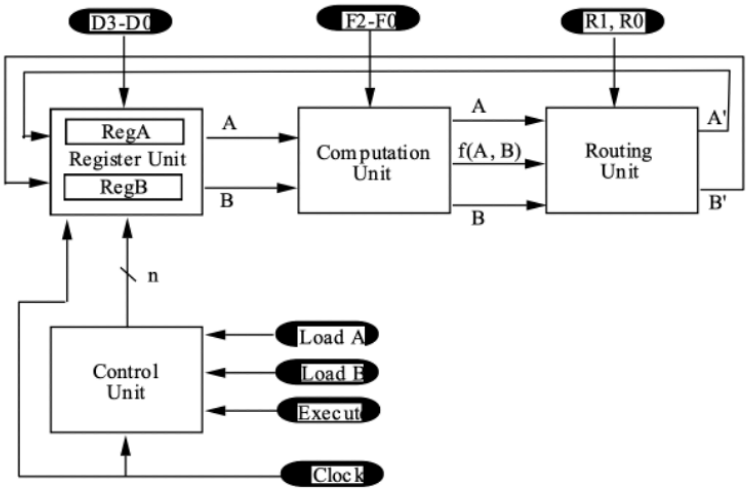
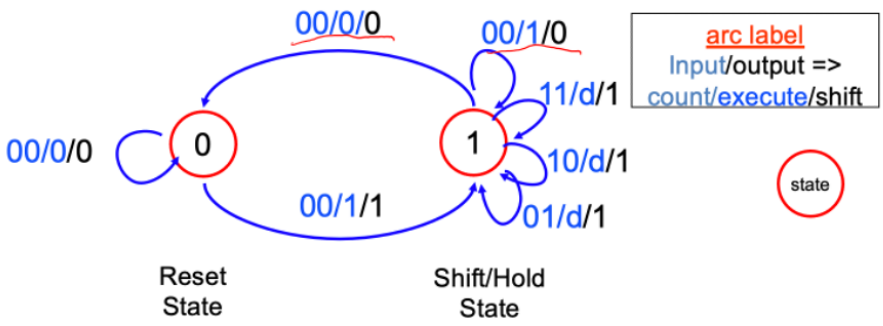


Figure 1: Block Diagram

c. State Machine Diagram:

Although a Moore Machine was created in order to understand the Control Unit procedures, a Mealy Machine was used in the final hardware design since less states were needed. Less states results in a simpler hardware implementation.



Mealy Machine State Diagram

IV. Design steps taken and detailed circuit schematic diagram:

a. Written procedure of the design steps taken:

Exec. Switch ('E')	Q	C1	C0	Reg. Shift ('S')	Q^+	$C1^+$	$C0^+$
0	0	0	0	0	0	0	0
0	0	0	1	d	d	d	D
0	0	1	0	d	d	d	D
0	0	1	1	d	d	d	D
0	1	0	0	0	0	0	0
0	1	0	1	1	1	1	0
0	1	1	0	1	1	1	1
0	1	1	1	1	1	0	0
1	0	0	0	1	1	0	1
1	0	0	1	d	d	d	D
1	0	1	0	d	d	d	D
1	0	1	1	d	d	d	D
1	1	0	0	0	1	0	0
1	1	0	1	1	1	1	0
1	1	1	0	1	1	1	1
1	1	1	1	1	1	0	0

Control unit state transition table using the Mealy state machine

	C1C0 (00)	C1C0 (01)	C1C0 (11)	C1C0 (10)
EQ (00)	0	X	X	X
EQ (01)	0	1	1	1
EQ (11)	0	1	1	1
EQ (10)	1	X	X	X

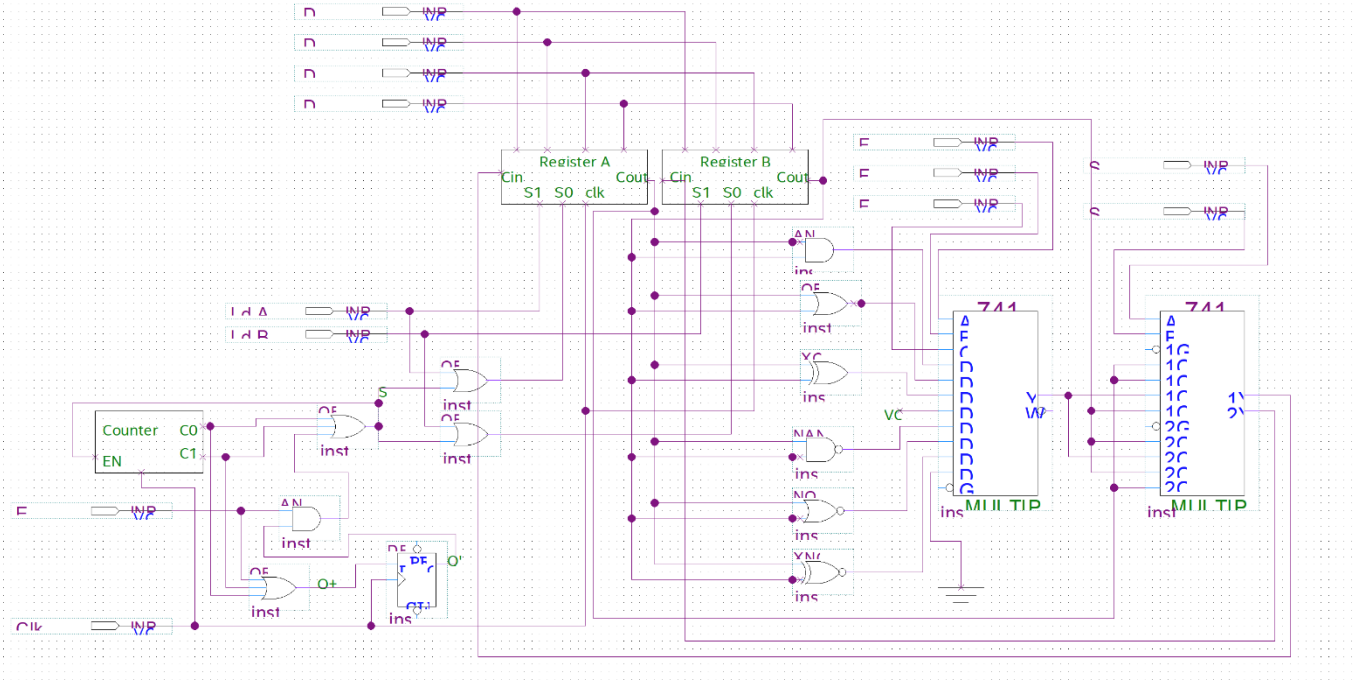
Reg Shift S = C1 + C0 +EQ’

Register Shift K-map

	C1C0 (00)	C1C0 (01)	C1C0 (11)	C1C0 (10)
EQ (00)	0	X	X	X
EQ (01)	0	1	1	1
EQ (11)	1	1	1	1
EQ (10)	1	X	X	X

Next State Q+ = E+ C1+C0

As stated before, we decided to use a Mealy Machine since it only had two states which could be represented via a single bit. Therefore, next state logic was relatively simple to implement by just creating a truth table with the three inputs, Execute, C1, C0, as well as the current state Q. Then, next states were created for both the output Register Shift S and the next state Q. The other two next-state values of C1+ and C0+ do not need k-maps since they are produced within the counter ICs. Regarding hardware, multiple implementations were tested, but we ultimately settled on a design that utilized a flip flop to hold the state Q and a IC counter to produce a continuous stream of C1 and C0.



Gate-level Circuit Schematic

V. Breadboard View/Layout Sheet:

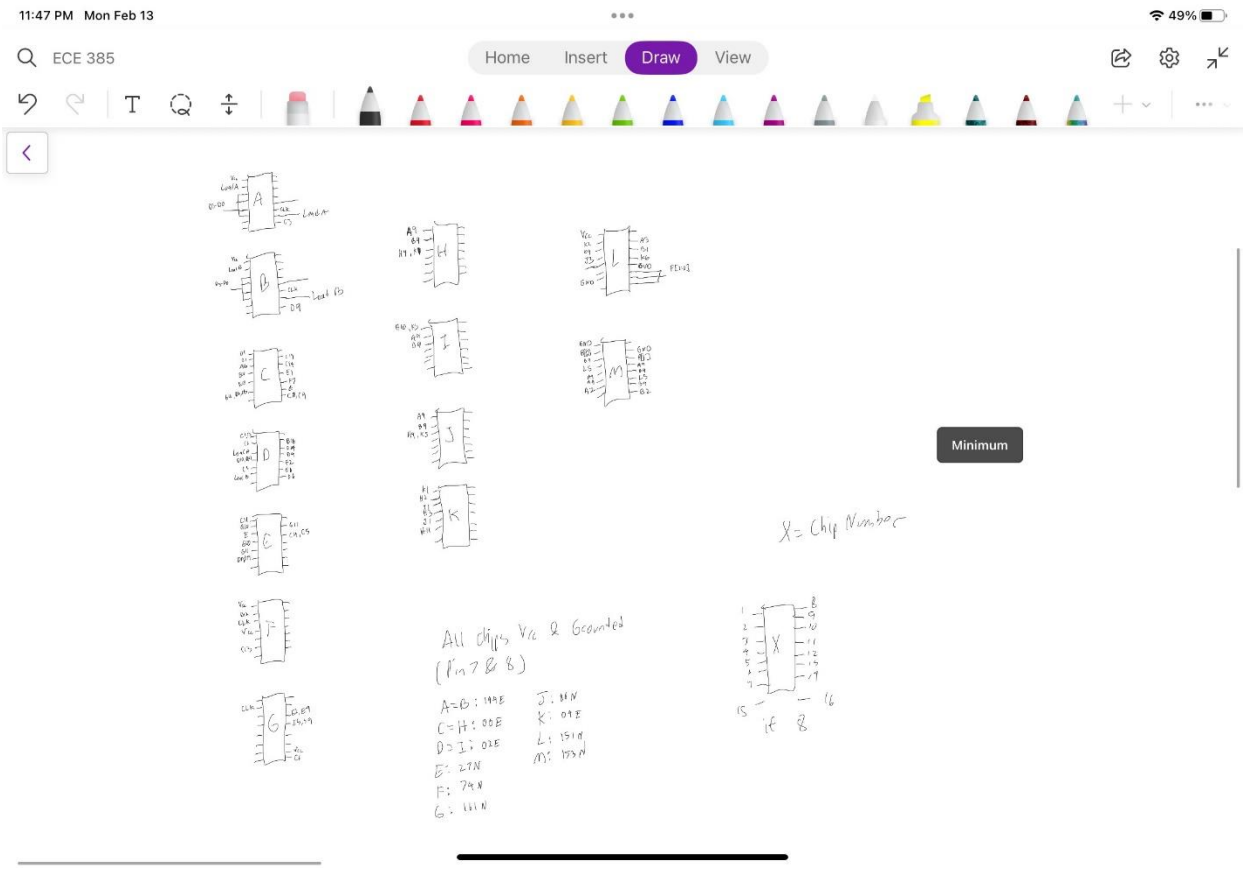


Figure: Breadboard Layout

VI. 8-bit Logic Processor in FPGA:

Summary of SystemVerilog Modules:

Changes to the code were relatively simple. Most of the code changed was just changing any occurrence of [3:0] to [7:0] to account for 8-bits instead of 4.

compute.sv

- The compute block (or computational unit) is the unit that does the Boolean math for the inputs and outputs their corresponding values
- No changes were made to this block because the unit essentially calculates each bit one at a time

control.sv

- The control block controls pretty much the whole operation of the design. It tells the other modules when to load inputs, when to execute operations, and it keeps track of when to stop executing
- Changes were made to this module as this is where the state machine resides. More states had to be added to account for the extra 4 bits, so that the control unit can go through all 8 bits. Additional emun logic was added for the extra states and the state machine further in the code was also modified to account for this. This was simply done by squeezing the extra states in-between what was already written and changing some values to account for the new states.

HexDriver.sv

- The Hexdriver file contains the code required to display digits onto the 7-digit displays on the FPGA board
- No changes were made to this file because having more bits doesn't change the display at all

Processor.sv

- The Processor file connects all of the individual modules together and causes them to work.

- As mentioned above, any time [3:0] was present, the value was changed to [7:0] to account for 9 bits. Also, the code mentions to change the input values of HexAU and HexBU to [7:4] for the upper four bits.

Reg_4.sv

- The Reg_4 module is essentially the shift-register on the hardware board. It takes in values and shifts them accordingly
- As mentioned above, any occurrence of [3:0] had to be changed to [7:0] so 8-bits could be used. Also, [3:1] had to be changed to [7:1] with the same reasoning

Register_unit.sv

- The Register_unit acts as a combination of the two registers A and B. Having them as one unit simplifies some of the code in the other files as A and B how act as one unit instead of two. This can be thought of as two shift-register chips next to each other
- As mentioned above, [3:0] was changed to [7:0] for 8-bits as they were changed in the Reg_4.sv file

Router.sv

- The Router tells the design how the outputs from the Boolean operation done should be stored
- No changes were made to the router as the router still has only 4 ways to route the output

Synchronizers.sv

- The Synchronizer acts as a way to interpret the user-input signals so that the SystemVerilog code can work accordingly with the FPGA board

- No changes were made as our input signals didn’t change

RTL Block Diagram:

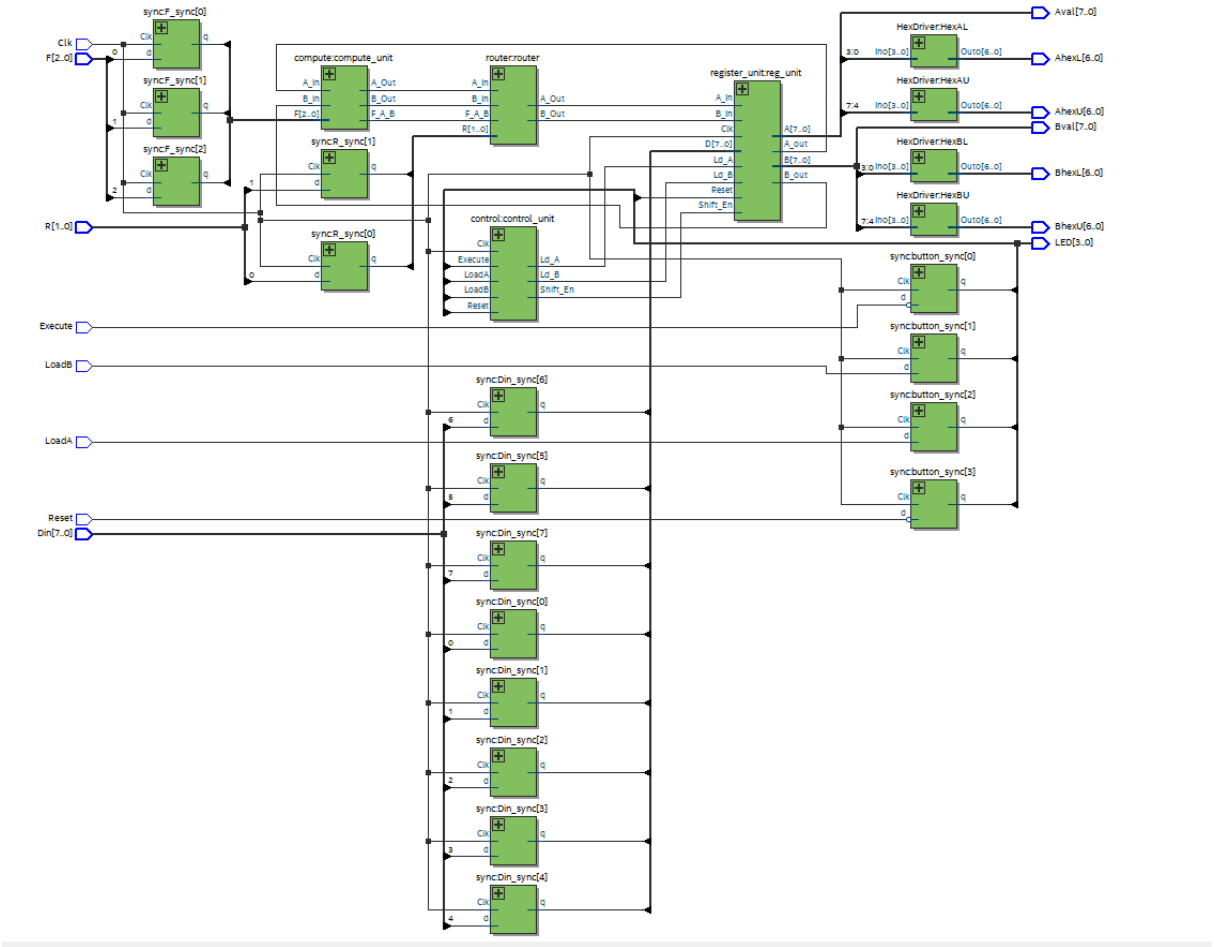


Figure: RTL Block Diagram

Simulation Results:

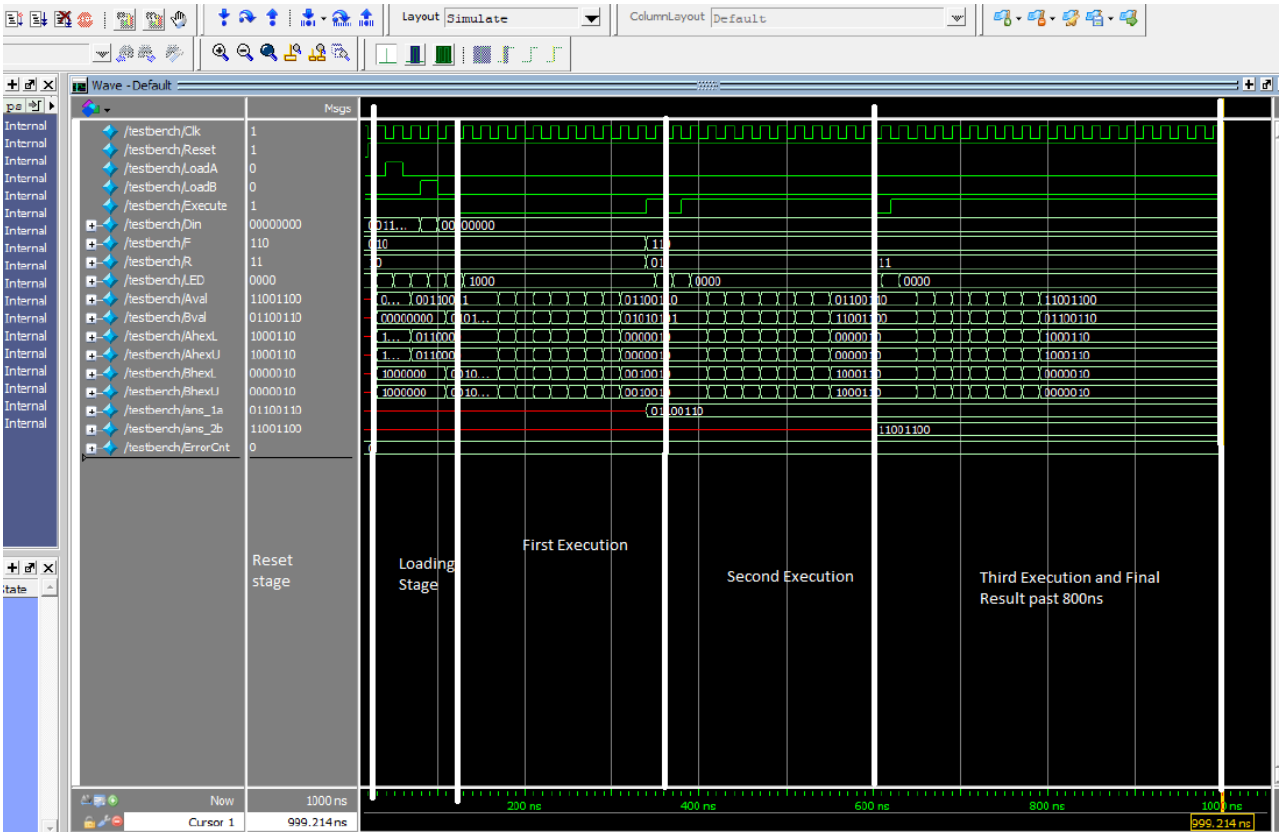


Figure: Annotated Simulation Results

SignalTap Trace Generation:

1. Assign all the required pins on the FPGA board using the chip assigner in Quartus. Also hardcode the values of F and R as there aren't enough switches on the board for every input.

2. Open up SignalTap by going to Tools->Signaltap Logic Analyzer

3. Check to make sure the USB-Blaster is selected on the top-right hand side as this is where the program will be programmed to.

4. Setup the clock so there will be a clock signal. Click on the three dots next to clock input on the very right side of the window. Then, press list and click on CLK to set it up. Click ok and exit out that window.

5. Double click on the left side of the screen where it says “Double click to add node” to add the nodes you want to view. In our case, it would be Execute and Registers A and B.

6. Once the window pops up, click list again and select the nodes we want to view. For both registers, select Data_Out[0]-[7] for registers A and B and select execute for executre. Press the > button to bring the nodes over and then Insert to confirm it.

7. Group the Data_Out values together for easier viewing. Do this by highlighting the Data_Out values, right clicking it, and select group.

8. Set up execute so it can signal our FPGA board. Do this by right clicking it and selecting the either edge option.

9. Compile the SignalTap and SystemVerilog code by clicking on the Start Compilation button on the top left of the screen. This will take a while to compile.

10. Once compiled, took to the top-right of the SignalTap screen and select the SOF file which you just compiled and import it.

11. Finally, click on the Run Analysis button next to the instance manager. Now, the results of Registers A, B, and the signal execute should show up whenever you trigger it on the FPGA board.

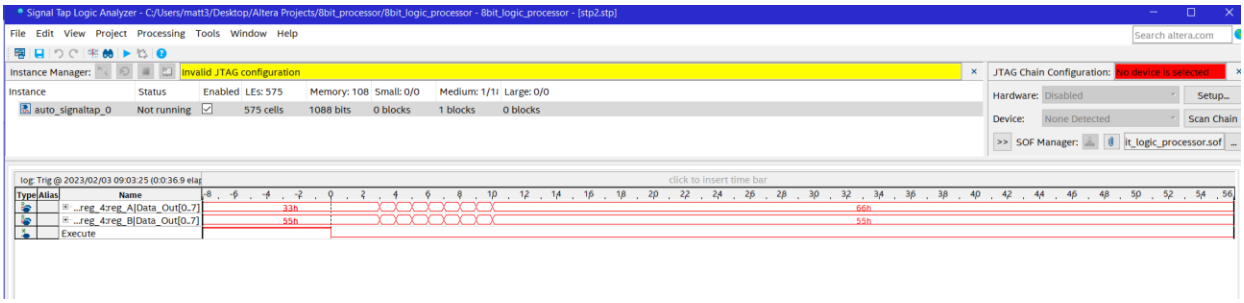


Figure: Results of 8'h33 XOR 8'h55 SignalTap trace

VII. Bugs:

There were quite a few bugs that we encountered. Our first and most significant bug that we encountered was that our shift-register wasn't properly loading in our values. Initially, we thought that this was intentional because the rest of the board wasn't hooked up, so we built the rest of the circuit. However, that ended up causing a lot of other things to not work (mostly anything that required a switch). This led us to looking at documentation of all the chips, checking wiring on all the chips, verifying logic for S and Q, and other stuff like unit testing individual units. After looking through old ECE 120 lab pictures, we found out we had wired the switches wrong. After fixing the wiring, most of the circuit worked fine.

Our second big bug was our multiplexer wasn't working properly. Using a voltmeter, we found it was taking inputs correctly, but was never outputting anything. After looking through documentation, we found out we didn't connect one of the pins that the chip needed to function properly. After fixing that, the multiplexer worked without a hitch.

Other minor bugs were present such as Vcc or ground not being connected, a wire being routed to the wrong place, and misunderstanding of documentation. These were quickly solved by looking at the circuit or rereading documentation.

VIII. Conclusion:

In the end, we got bot part 1 and part 2 of our lab working properly. This lab was interesting as it showed how simulation in SystemVerilog can be done and how manually building a circuit on a breadboard can be tedious and annoying. It was a good introduction to SystemVerilog and as our lab hardware class, was interesting as it allowed us to build a basic processor.

Post-Lab Questions

Describe the simplest (two-input one-output) circuit that can optionally invert a signal (i.e., one input determines if the output is equal to the other input or equal to the other input inverted). Explain why this is useful for the construction of this lab.

Using XNOR logic allows us to effectively selectively invert any signal. One input is zero, the result will be whatever value the other input is. However when one input is one, the output will be inverted. This is useful for the lab because for the computational unit, the inputs are directly mapped so that the 000-011 values are the inverted values of 100-111. This can cutdown the chips we use because we can use bit 2 (left-most bit) as the bit to selectively invert the output values.

Explain how a modular design such as that presented above improves testability and cuts down development time.

It allows us to group together specific components of the design so that the board can be more organized and easier to debug. Having them split up can allow for easier unit testing to determine if specific parts of the design are working as intended or not. This will help in the grand scheme of things so that you can narrow down which parts need to be fixed and which don't.

Discuss the design process of your state machine, what are the tradeoffs of a Mealy machine vs a Moore machine?

When designing the state machine, we grouped it into different states for which the design could be in. We did this first by finding out our inputs. In our case, it was S, Q, C1, and C0. From here, we decided which states and inputs were needed to correctly implement our design and after that, we drew a truth table for it and wired up the circuit. The main difference between a Mealy and Moore machine is that Mealy machines determine its next state by both its current state and inputs while a Moore only looks at current state. A trade off with using a Mealy machine is that it can reduce the number of states required for a design as if a design has multiple inputs for one state, you would need two states to represent the different inputs it can take while a Mealy would only need one.

What are the difference between ModelSim and SignalTap? Although both systems generate waveforms, what situations might ModelSim be preferred and where might SignalTap be more appropriate?

ModelSim and SignalTap are both ways to generate wave forms, but they do have differences. ModelSim simulates the code or design and can be simulated fully with its full range of inputs and outputs. SignalTap uses board simulation with the FPGA and it can be limited depending on how many input options are available. In case of our lab, some values had to be hardcoded to account for the missing inputs. ModelSim can be preferred when a full design wants to be simulated, so the full range of outputs can be verified by all its inputs. SignalTap may be more appropriate for any on-board testing for the FPGA in which ModelSim can't simulate. From our experience, it also is quicker to input values into the FPGA and get an output than it is to use ModelSim.