**ECE 385**

Spring 2023

Experiment #2

**A Logic Processor**

Noah DuVal, Jason Wright

JZ / Friday 3:00 pm

Jason Zhu

**Introduction**

Over the course of this lab we have touched upon previous concepts to expand and deeper understand the way that we use transistor-transistor logic to implement finite state machines. During the first week of the experiment we designed the FSM and created it using a series of 7400 chips on a breadboard to understand the inner workings of the circuit. Then the following week we used a software program called Quartus in which we implemented the same circuit, but this time on a Field-Programmable Gate Array board. Also referred to as a FPGA board.

**Operation of The Logic Processor**

The primary purpose of this circuit is to compute a series of seventeen different functions and display the contents of the two four bit shift registers to show the calculation. To use the circuit the values are initially selected using dip switch inputs D0, D1, D2, and D3. Once the load switch for either register A or B is then flipped, on the following clock cycle the input values will load into the chosen register. When both registers are loaded the function is then selected using the dip switch inputs F2, F1, and F0. With the destination of the output chosen from the routing unit using inputs R1 and R0.

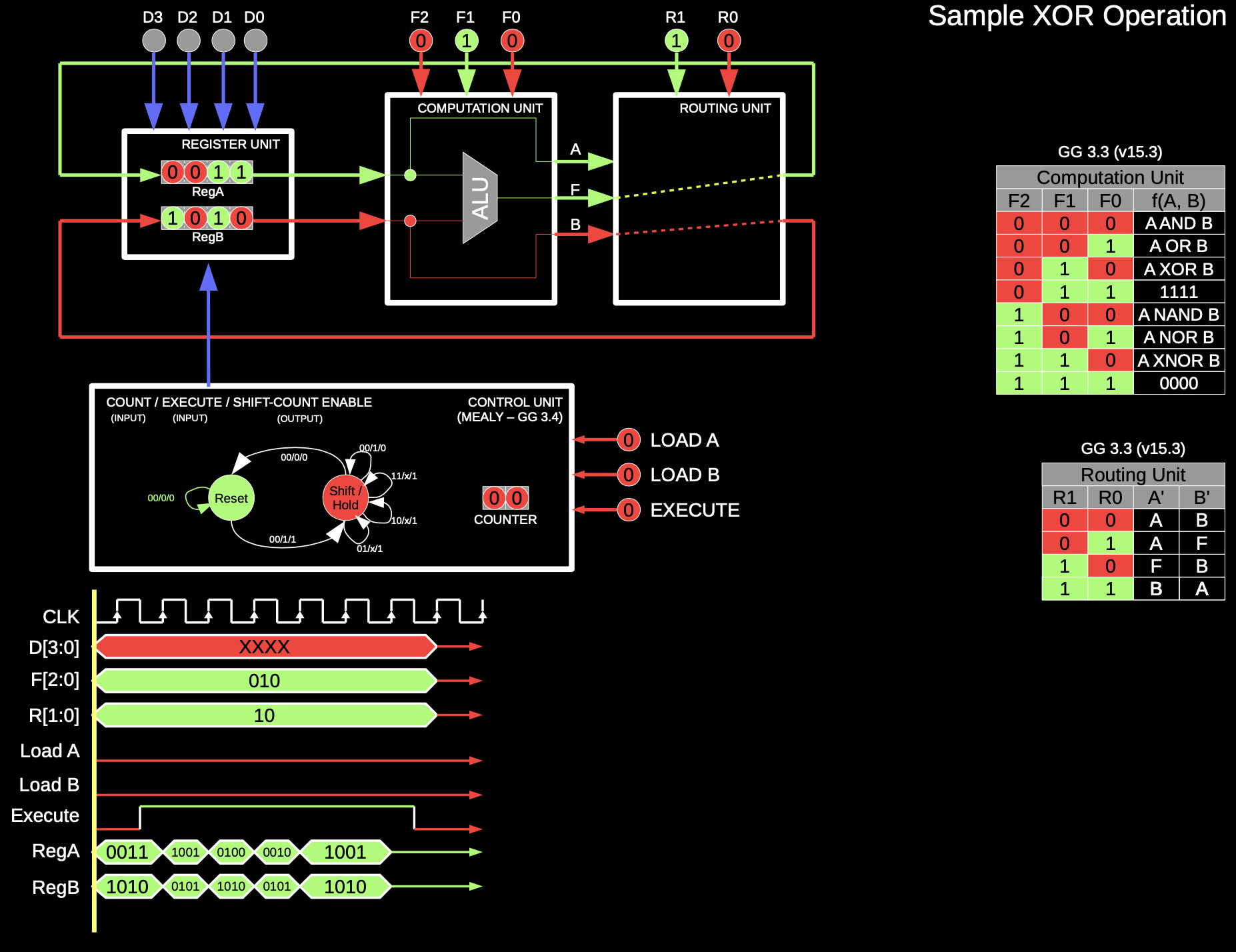


Figure 1: Block Diagram and State Machine

Finally, once the values are loaded, function selected, and routing chosen you press the execute button once and after four short clock cycles the change will be seen on the LED’s to represent the values in both of the shift registers. In figure 1 above is the combination of functions the user can choose from. As seen in the routing table on the right hand side of figure 1 the (R1,R0) inputs set to (0,0) will ignore the computation and simply retain the values held in the register after the four clock cycles it takes to completely shift through the registers. In the case of (1,1) the values of registers A and B are simply swapped as the calculator is ignored again. However, with either of the two remaining combinations of the routing unit inputs, select which register to place the F(A, B) output in as the unchosen register will retain the original value.

**Written Description**

The circuit consists of four key units. The first one being the Register Unit as this is the part that takes the input values from the dip switches and loads them into the register. This portion takes inputs from the control unit to tell it when to shift and when to load the registers.

Thus, the next key stage is the control unit. As seen in the block diagram shown in figure 1 the control unit is responsible for tracking the system. In other words, it houses the clock and counter which is used to limit the system to only four shifts of the register. Not only that, but it holds the system in either the Reset or Shift/Hold states. While in the reset state the control unit will load the values upon request from the load switches and wait for the execute signal to start the operation and change the state. When in the Shift/Hold state the control unit will count and shift the registers four times and then wait until the execute signal goes back to low before it goes back to the reset state.

In between each of these counts and clock cycles is where the Computation and Routing unit do their work. The computation unit consists of an 8:1 mux which takes in eight possible function values that are created using boolean logic operations. The computation unit thus chooses one of the functions to send to the routing unit based on the inputs F2, F1, and F0 as stated above.

The routing unit then decides what to store and where to store it. The possibilities are shown in the table on the right hand side of Figure 1. As you can either keep the registers, swap the registers, or place the function output from the computation unit in either register A or B.

**Design Steps and Circuit Diagram**

For our design, we essentially started at the end with the routing and ALU then worked our way backwards to the registers and control unit. For the routing, we realized quickly that it could be accomplished with two 4:1 muxes, one for each register. The A mux was given inputs A, A, F, B while the B mux was given inputs B, F, B, A. This was determined to be correct based on the provided routing truth table, and did not require any K-map work to determine.

To select the correct operation, we used the 3 F inputs as selecting bits on an 8:1 mux. The input to the first 4 operations were simple combinational logics that implemented the functions in accordance with the provided truth table (000 = A AND B, 001 = A OR B, etc.) This logic was implemented with the corresponding gates when available, or gates were made using a combination of other gates, mostly NOR and NAND when needed. The second 4 operations are just the inversions of the first four, so we just used NAND and NOT gates to invert the other inputs.

The logic for the control unit required more work to implement. By reading the datasheets for the flip flop, counter, and register units, we realized that we needed logic to control when the register shifts, when the flip flop changes states, and when the counter starts counting. Deciding to avoid a Moore machine to reduce the number of needed states, we only wanted 2 states on our flip flop, one for rest / reset and the other state for when processing is happening or just finished. Using the provided truth tables, we were able to determine an equation for the next flip flop state without using a K-map by determining that the state should always be 1 while the count bits are not zero (i.e. counter is at 1, 2, or 3 on the count) or when execute is 1. Our final equation was Q+ = E + C1 + C0 .

Our counter had to be toggled on and off using the PE pin. We realized that this should be counting at the exact same times that the register should be shifting, so we figured we can use the same logic to control both. The register had two shift bits, S1 and S0 used to control how it shifts. However, for our purposes S1 could be permanently set to zero at all times other than loading, which could be hardwired to both inputs by connecting our load switches to the pin inputs directly. S0 logic was found with the following K-map:

| EQ \ C1C0 | 00 | 01 | 11 | 10 |
| --- | --- | --- | --- | --- |
| 00 | 0 | x | x | x |
| 01 | 0 | 1 | 1 | x |
| 11 | 0 | 1 | 1 | x |
| 10 | 1 | 1 | 1 | 1 |

Table 1: K-map for Shift Enable

Using the pairings defined by green highlight, blue text, and underline as three separate circles, we get shift enable S = C0 + C1 + EQ’ . This, as previously stated, can be used as both the S0 and PE’ inputs to simultaneously control the counter and the registers.

Note: Because the Load input signals were hard wired in with simple gates and not put into this logic, problems may arise if the load switches were enabled during operation. However, this was ignored for the purposes of this lab since that situation should not occur during operation.

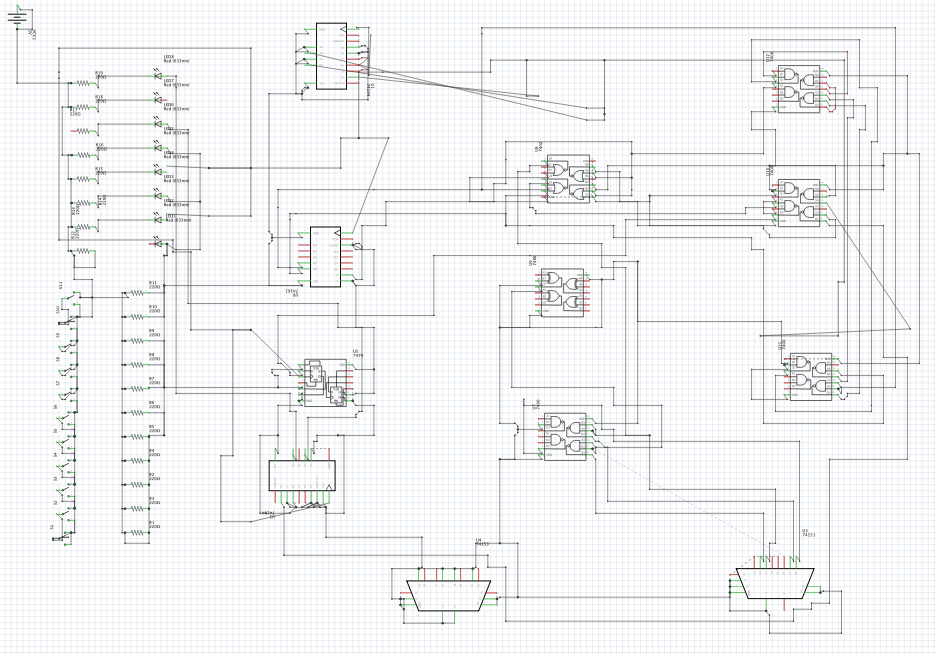
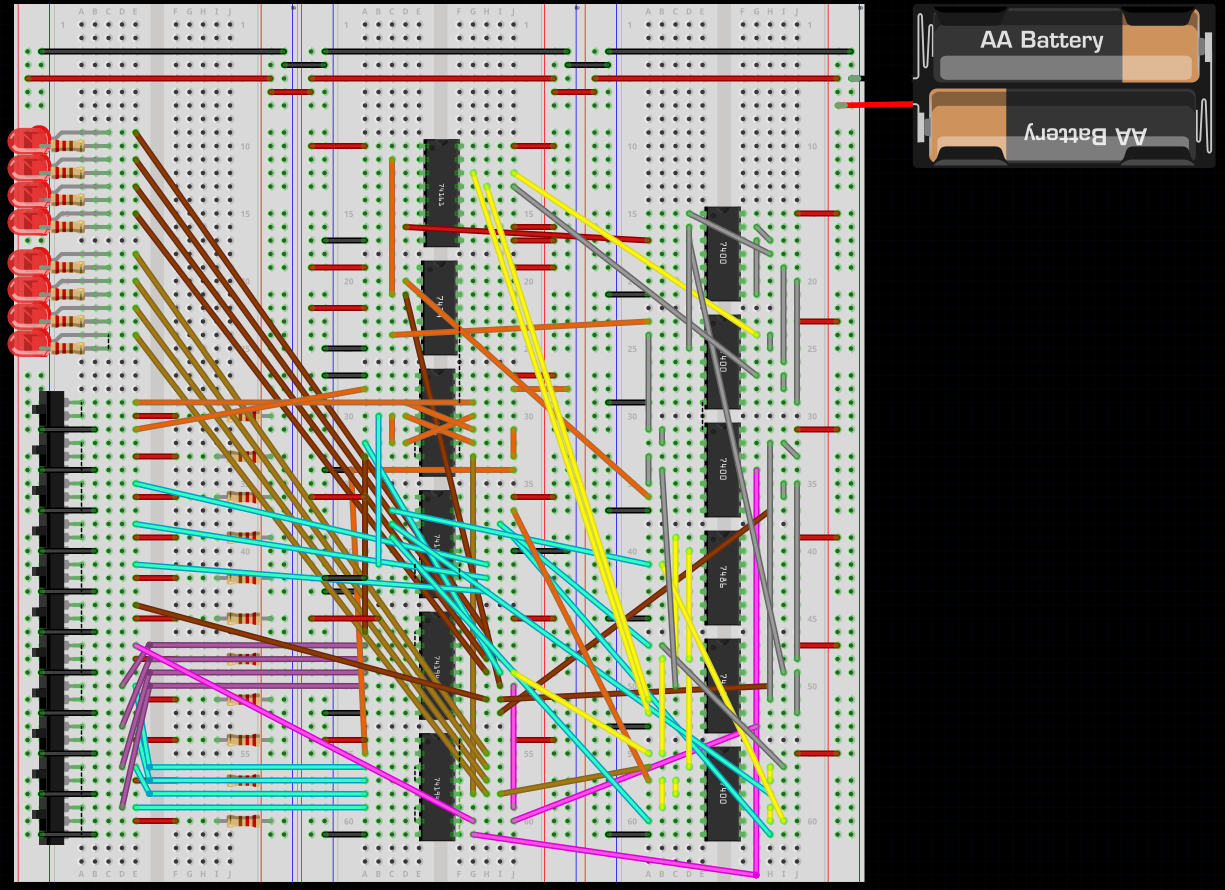
Figure 2: Circuit Schematic

Figure 3: Breadboard Layout

**8-Bit Logic Processor on FPGA**

We were tasked with extending the provided 4-bit logic processor to 8-bits by editing several .SV modules. Several of these files were designed in a way that required no editing for their functionality to be untouched by the move from 4 to 8 bits. This includes “Synchronizers.sv” which allows asynchronous signals from the board (switches and buttons) to correctly interact with the synchronous design of the circuit. “Router.sv” is a multiplexer unit that guides single bits to their destination according to the input signals R1 and R0. “HexDriver.sv” allows our register contents to be output to the hex display, but it is always a function of 4 bits only. “Compute.sv” processes single bits according to the provided F signals, it is essentially just a multiplexer with various combinational logic for the different operations. “Testbench\_8.sv” is the provided test bench used to ensure correct response and easily analyze.

For the files that did need to change, the largest was “Processor.sv.” Many logic variables had to be extended from 4 to 8 bits, including Din, Aval, and Bval. Additional HexDriver units needed to be defined here to display the additional bits. Hardcoded values for F and R were also defined here for the signal tap to work properly on the FPGA because of the limited number of switches. “Reg\_8.sv” was created to replace “Reg\_4.sv,” and it is essentially the same function but with 8 bit inputs and outputs. The separate file was created just to be able to reference the original 4 bit unedited. “Register\_unit.sv” had to have the logic variables extended to 8 bits. The module calls were also switched to “Reg\_8.sv.” “Control.sv” had 4 more states added for a total of 10 states. This change is also reflected in the shift logic to properly move through the states.

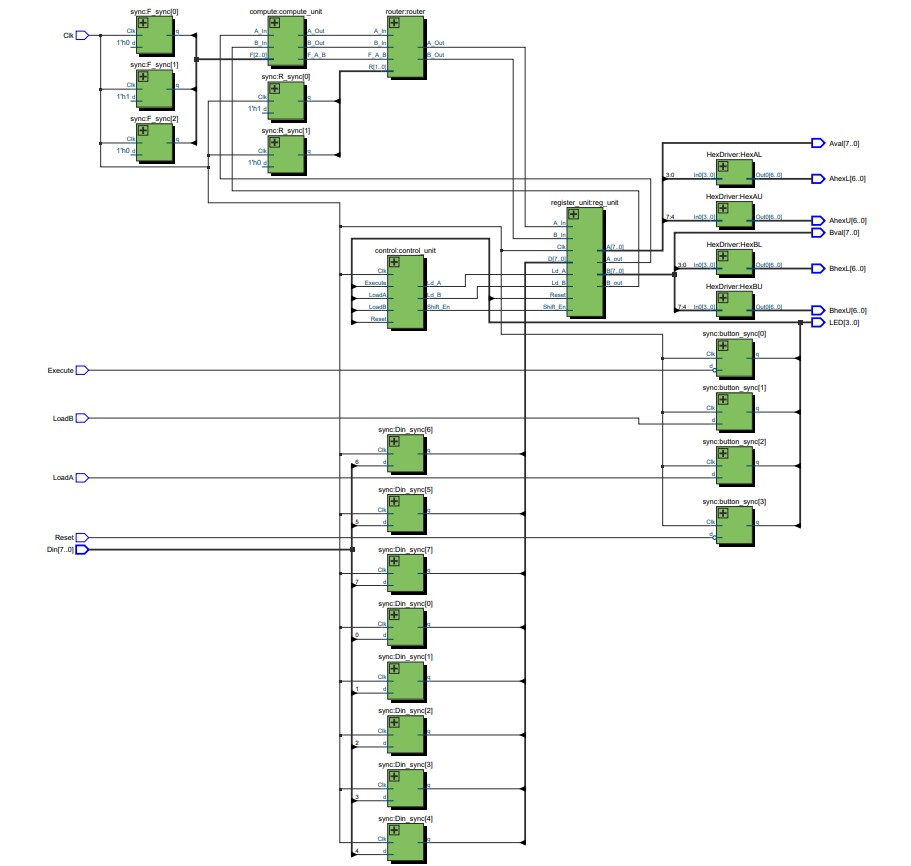


Figure 4: RTL Block Diagram

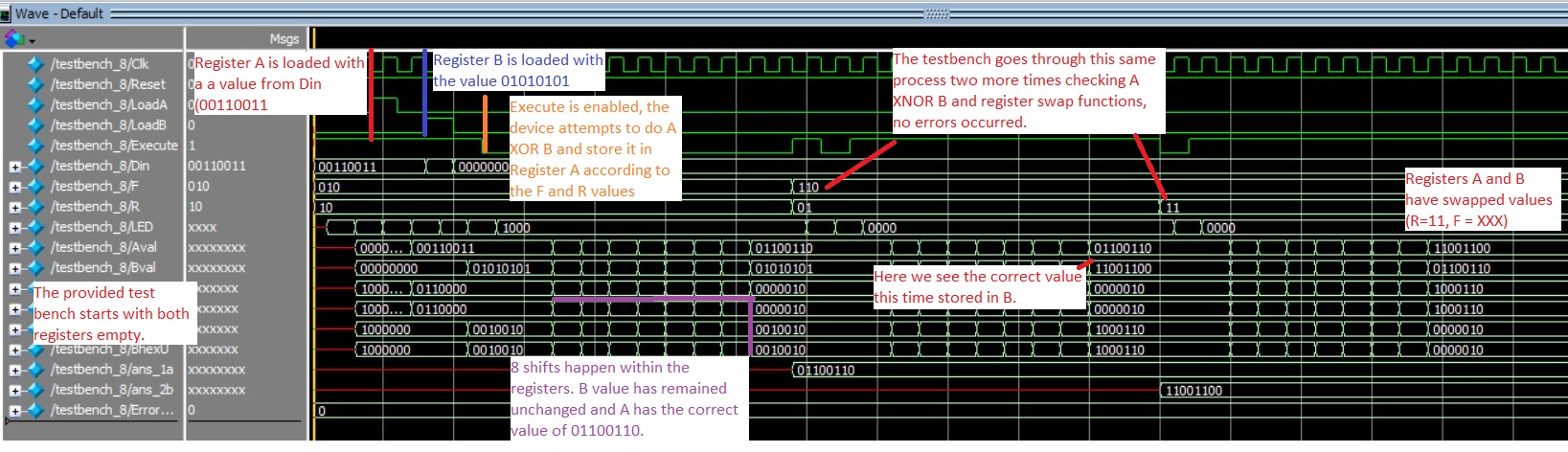


Figure 5: Modelsim simulation with annotation

To generate a SignalTap, we performed the following procedure:

Step 1, Choose the clock in the setup menu and select sample depth of 64.

Step 2, Set the displayed signals as Reg A and B contents and the execute button.

Step 3, Set the trigger to be either edge of execute.

Step 4, set Din to 8h’A7 on the FPGA. Press load A switch.

Step 5, set Din to 8h’53 on the FPGA. Press load B switch.

Step 6, enable data collection in the SignalTap software.

Step 7, press execute on the hardware.

The results of this process are shown below in Figure 6. The value of F was hardcoded beforehand to be 010, and the value of R was hardcoded to 01. This results in the function A XOR B being performed and stored in register B.

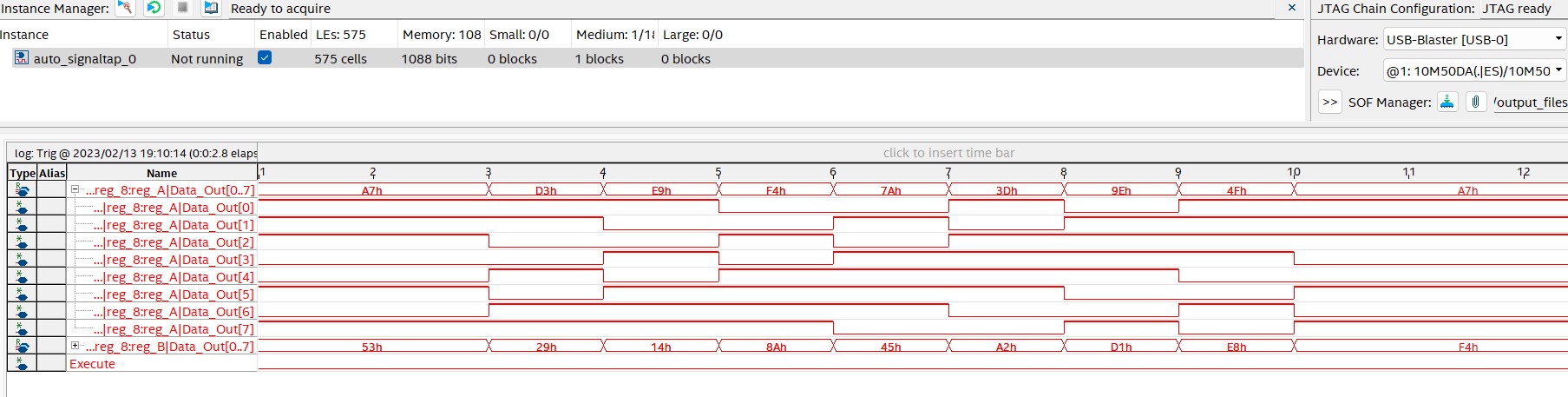


Figure 6: SignalTap Results

**Bugs Encountered**

Week 1 bugs encountered:

1. Counter chip did not seem to be stopping and started at the correct times according to the logic. Solution: datasheet was studied and we found that we previously misread it. Correctly wiring the reset pin fixed this.
2. Flip Flop changed states almost randomly, this resulted in our design sometimes working, but most of the time caused unnecessary shifts and incorrect responses. Solution: similar to the counter trip, we discovered one of the pins needed to be wired to Vdd and the connection was left out. Adding this in fixed the issue.
3. When implementing the control logic we were short on logic chips because we forgot to bring our lab kit on this particular day. Solution: rather than postponing our implementation and testing until retrieving the chips, we managed to come up with ways to implement the remaining state logic using the chips we did have. This resulted in some inefficient logic using leftover NAND and NOR gates on the chips that were being used for the ALU.

Week 2 bugs encountered:

1. Our only bugs this week came about because of weird settings on the quartus software that needed to be corrected. The main one was that our compiled code did not generate the .sof file needed to program the FPGA. Solution: After fiddling with various settings to no avail, we discovered by chance that selecting “start compilation” rather than “start analysis and synthesis” generated this file. This issue did not persist and ever since then “start analysis and synthesis” also generates the .sof file.

**Post-Lab Questions**

To optionally invert a signal with the simplest implementation you would simply need one inverter and a 2 to 1 multiplexer (or the equivalent made from 1 NOT, 2 ANDs, and 1 OR gate). The input signal will go to one mux input, and be inverted for the second mux input. The select bit can then choose if the inverted or normal signal makes it to the output.

A modular design increases testability because each module can be tested individually and independently of the other modules. For example, in the first section of this lab, we were not confident in our control unit design after our first day of designing and building. However, we first tested our ALU and routing unit when they were finished so that any issues after that point could be attributed to the control unit.

For our state machine, we focused on keeping it to the minimum number of flip flops and states. This resulted in us creating a Mealy machine utilizing a counter chip so that we only needed a single flip flop with two states: rest/reset and counting. We designed combinational logic that took the current count, state, and the execute button’s status to tell the registers when to shift and told the counter when to keep counting. If we used a Moore machine, our outputs would be determined only by the current state, not the other inputs, therefore we would have needed extra flip flops to create more individual states.

ModelSim and SignalTap are both useful tools for analyzing and debugging. ModelSim is a functional simulation of the compile software. In contrast, SignalTap generates a waveform by reading the actual results of the programmed hardware. In many situations, they can be used interchangeably. ModelSim might be preferred if you are designing a system that cannot be run on your available hardware. SignalTap is very useful for ensuring your design transferred to the hardware without any issues.

**Conclusion**

This lab tested our logic designing and debugging skills by having us build a 4 bit shifting register logic processor on a breadboard. Our machine was able to properly shift, process, and route the bits to perform 8 different operations with 4 different routing options. We then successfully extended this design to 8 bits using Quartus and implemented it on an FPGA board as an introduction to working with SystemVerilog.