ECE 385

Spring 2023

Experiment #2

# Lab #2

Leonardo Gonzalez and Cain Gonzalez
Section AL1 / 4:00pm-5:00pm
Dohun Jeong

## Purpose of Circuit

The purpose of this circuit was to create and design a 4 bit-serial logic operation processor using Transistor-Transistor Logic. The design utilizes two 4-bit shift registers, several multiplexers, and a 4-bit binary counter. The circuit is capable of calculating eight different functions (AND, OR, XOR, NAND, NOR, XNOR, CLEAR (all zeroes), and SET (all ones)) and routing the results of those operations in four different ways. A finite state machine will be implemented to serve as the control unit of the circuit. Additionally, we expanded our circuit to an 8-bit serial processor using Quartus and Modelsim software. For our 8-bit design, we used Resistor-Transistor Logic that was implemented on our DE10-Lite FPGA.

## Operation of the Circuit

In order to perform operations on the circuit, there are various switches to take into consideration. The first are LDA, LDB, and a 4-bit input signal we will call D[3:0]. If the user wants to load a value into a register, they must put those values into D and then switch LDA or LDB to high to load Register A or Register B, respectively. Once the registers are loaded, set F to the appropriate value to perform the desired operation, as seen in Table 1, and R to the desired values as seen in Table 2. Finally, press execute and watch as the circuit does its thing.

## Written Description of Circuit

Our circuit takes switch inputs Execute, LDA, LDB, F[2:0], R[1:0], as well as a 4 bit input signal D[3:0] for both Register A and Register B.

### Control Unit and Finite State Machine

The topmost component that needed to be created for our logic processor was a Finite State Machine (FSM) to control its operation. For the 4-bit Serial Logic Processor we opted to use a Mealy Machine to reduce the number of states and thus number of gates required to operate it, which left us at two states: Reset and Shift/Hold.

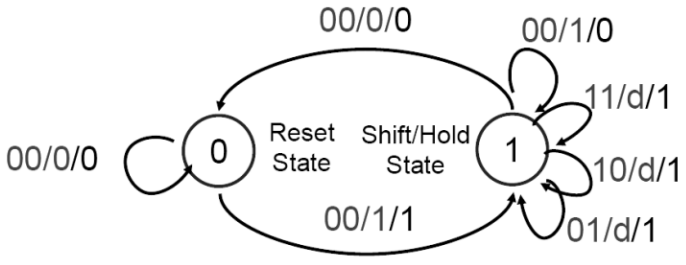


Figure 1. The Mealy Finite State Machine used to control operation of the processor
The state transition variables are in the order C1C0/E/S and the bit inside the circles is Q

The idea behind our state machine was that before the Execute signal went high, we would remain in the Reset (Q = 0) State and just wait for it to switch.

Once Execute went high, S (our output) would switch to 1 to begin the first shift of the register units and it would move to the Shift/Hold (Q = 1) State. Then S and Q would remain 1 for 3 more clock cycles, which was kept track of by the counter chip to ensure that the registers shifted 3 more times before stopping. Finally, we included one more condition to remain in the Shift/Hold state while Execute is 1, but C1C0 are 0. This condition is to account for the case where Execute is held high for longer than time required for all 4 shifts because in this case we wouldn't want to begin shifting again, we would just want to hold off until Execute returns to 0 before resetting. Additionally, in order for our circuit to remember what state it was in, we used a flip flop to store the value of Q.

Using the inputs and outputs we described above, we created the truth table and K-maps in Figures 2 and 3. The reason we only have a K-map for S and Q+ is because C1+ and C0+ are taken care of by the counter, so we do not need logic for them.

| 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 |

Figure 2. Control unit state transition table using Mealy state machine

**S K-map**

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

**Q+ K-map**

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

Figure 3. K-maps for both S and Q+

After setting up the K-maps, we found the min-states and converted them to use NAND, NOR, and NOT gates.
The only user controlled input to the control unit was Execute, whereas Q, C1 and C0 were managed within the circuit.

**Counter**

To manage C1+ and C0+ for the control unit, we used a 74161A chip instead of manually creating the logic using gates for simplicity. It ran on the same clock as the register unit and took S as its input to continue counting, so it ran completely in sync with the FSM and only counted from 0 to 4 per press of Execute.

**Computation Unit**

Afterwards, we started planning the design for our computation unit. With our computation unit, we took many items into consideration, as it was the most essential part of the calculation functionality of our processor. We wanted our computation unit to select from 8 different functions, so it was easy for us to decide to utilize an 8 to 1 multiplexer. These 8 functions are XOR, XNOR, OR, NOR, AND, NAND, SET (sets bit to 1), CLEAR (sets bit to 0). The truth table for this is shown in Table 1. For selecting our function when doing the 4-bit processor with

TTL logic, we simply used 3 switches. However, when transitioning our circuit into an 8-bit processor with RTL logic, we had to hardwire our function selection signal into our testbench file simply due to the lack of switches on the FPGA board. With that being said, we simply recompiled each time we wanted to try a different function in order to ensure that the intended signal was being tested. For the design process, we simply designed by inspection as the connections were coming either directly from the register unit or via the chips used to calculate the logical output of the values for each function we wanted to simulate. Table 1 displays the selections for the computation unit depending on the F[2:0]. Furthermore, the computation units also provide the A and B signals unchanged in order to be fed to the routing unit.

| **Function Selection Inputs** | | | **Computation Unit Output** |
|---|---|---|---|
| **F2** | **F1** | **F0** | **f(A, B)** |
| 0 | 0 | 0 | A AND B |
| 0 | 0 | 1 | A OR B |
| 0 | 1 | 0 | A XOR B |
| 0 | 1 | 1 | 1111 |
| 1 | 0 | 0 | A NAND B |
| 1 | 0 | 1 | A NOR B |
| 1 | 1 | 0 | A XNOR B |
| 1 | 1 | 1 | 0000 |

Table 1. Function Selection Inputs and Corresponding Computation Unit Output

**Routing Unit**

Following the design and implementation of our computation unit, we began to design our routing unit using two 4 to 1 multiplexers, with R[1:0] signals serving as the select bits for each multiplexer. One multiplexer served to select our A' signal, while the other multiplexer served to deliver our B' signal. These signals were routed to the inputs of the registers, from which they would later be displayed on LEDs in order to verify everything worked. The purpose of the routing unit was to switch how the output of the calculation performed by the computation unit was displayed on the FPGA. Again, this design was simply done by inspection as no logic was needed to alter the signal in any way that came from the registers or the output of the computation unit. Below, we have included Table 2, which shows each signal and their corresponding outputs.

| Routing Selection | | Router Output | |
|---|---|---|---|
| R1 | R0 | A* | B* |
| 0 | 0 | A | B |
| 0 | 1 | A | F |
| 1 | 0 | F | B |
| 1 | 1 | B | A |

Table 2. Routing Selection Inputs and Corresponding Router Output for Router A and Router B

**Register Unit**

As shown in Figure 5, the computation unit required 1 XNOR chip, 1 HEX Inverter, 1 NAND chip, and 1 NOR Chip. Inputs to the 8 to 1 multiplexer were supplied by the Q3 value of each register unit, with Register A and Register B providing the signal. The purpose of the register unit was to store two 4-bit values before and after execution was committed on the circuit.

## High Level Block Diagram

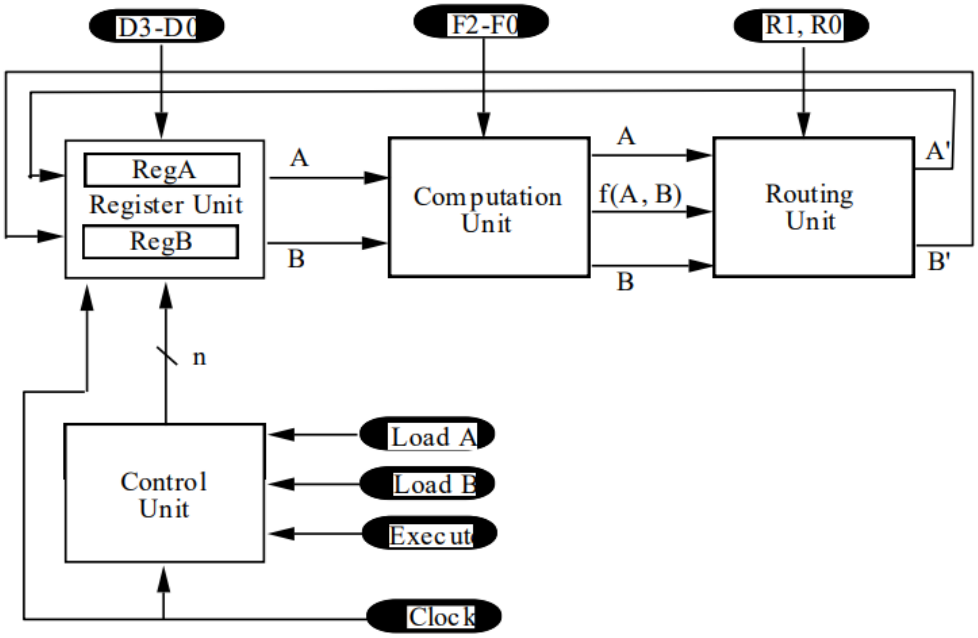


Figure 4. High Level Block Diagram of Our Circuit. This diagram is displaying the connections made between human inputs (Load A, Load B, Execute), the Control Unit, Register Unit, Computation Unit, and Routing Unit.

## Logic Diagrams

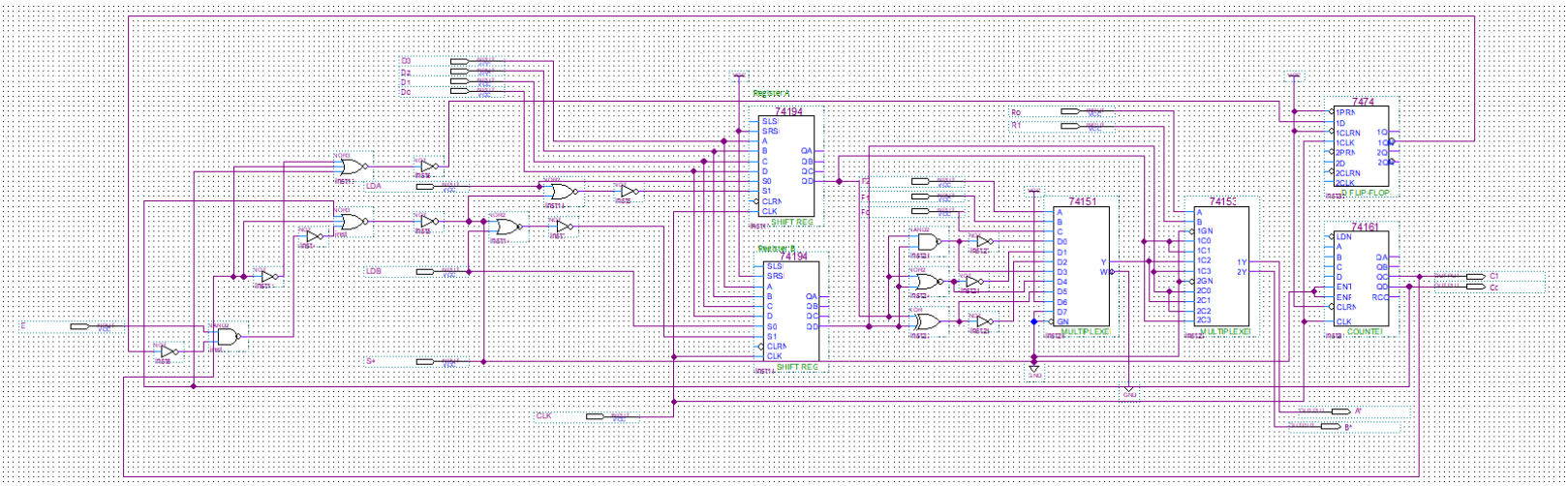


Figure 5. Logic Diagram of Our Circuit. The diagram displays the Routing, Control, Computation, and Register Units as how they are connected logically. Inputs E, F[2:0], R[1:0], and D[3:0] are displayed, as well as LDA (load A signal) and LDB (load B signal) for signaling which register is being loaded.

## Component Layout

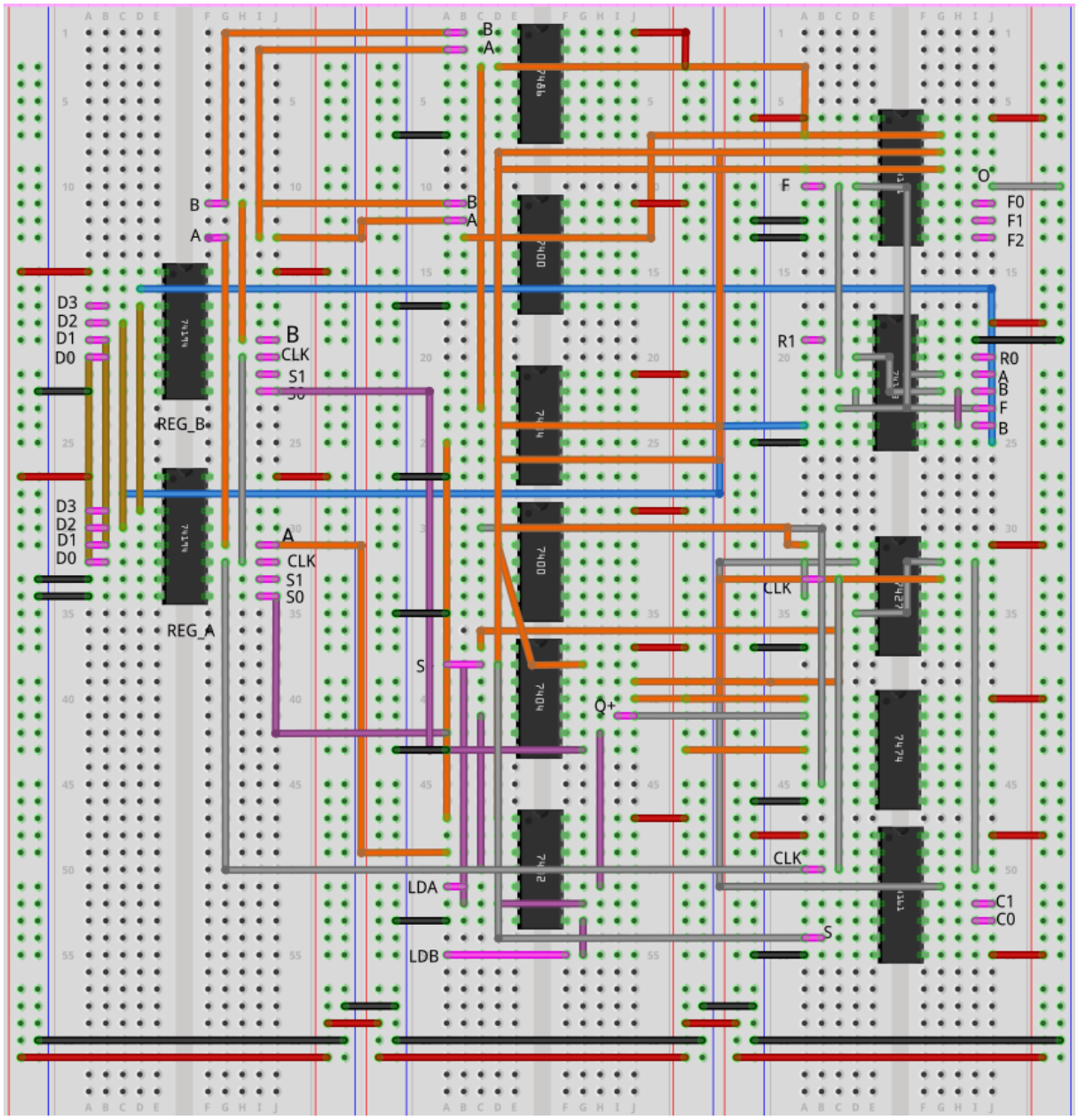


Figure 6. Component Layout of Our Circuit.

## Extension to 8-bit on the FPGA

### Description of changes to .sv files

In order to extend our 4 bit logic processor to an 8-bit logic processor, we needed to make a variety of changes to the provided code, which are outlined below.

Module name: **Processor.sv**
Inputs: Clk, Reset, LoadA, LoadB, Execute, [7:0] Din, [2:0] F, [1:0] R
Outputs: [3:0] LED, [7:0] Aval and Bval, ([6:0] AhexL, AhexU, BhexL, BhexU)
Description: This module controls all of the functions of our circuit. It instantiates all four of our components and wires the contents of the registers to the Hex displays on the FPGA. When used on the FPGA, the F and R values are no longer made to be inputs and are hardcoded, since the FPGA only has 10 switches that need to be used to load the registers.
Purpose: The purpose of this module is to control the operation of our circuit and put it all in one place so that aspects of it can be changed easily.
Changes: The changes that we made to this module were changing any signal that interacted with the values of the registers, such as Din, A, and B, to 8 bits long instead of 4.

Module name: **Control.sv**
Inputs: Clk, Reset, LoadA, LoadB, Execute
Outputs: Shift_En, Ld_A, Ld_B
Description: This module controls all of the logic for the control unit. In this module, a Moore Machine is implemented as opposed to a Mealy Machine for simplicity's sake and while the process is running it just switches from one state to the next every clock cycle. The point of the states is to determine the outputs of the module and feed them to the rest of the circuit. If we are in the reset state, S is 0 and Ld_A and Ld_B are equal to LoadA and LoadB, respectively. If we are in the hold state S, Ld_A and Ld_B are 0, and otherwise S is 1 and Ld_A and Ld_B are 0.
Purpose: The purpose of this module is to generate outputs that dictate whether or not the rest of the circuit actually runs through its operations. S controls whether or not we shift the registers and Ld_A and Ld_B control whether or not we parallel load into them.
Changes: The changes that we made to this module were adding four more states to the Moore machine so that S would stay high for four more cycles, making the registers shift 4 more times since they now contain 8 bits instead of 4.

Module name: **Reg_4.sv**
Inputs: Clk, Reset, Shift_In, Load, Shift_En, [7:0] D
Outputs: Shift_Out, [7:0] Data_Out
Description: This module describes the outputs of the register units we use to store our 8 bits. It is positive edge triggered and parallel loads D into the register at the positive edge of Clk. When Shift_En is high, the register right shifts the whole register right by one bit and loads Shift_In as the new leftmost bit. The bit shifted out is stored in Shift_Out and the contents of the register are stored in Data_Out
Purpose: The purpose of this module is to store the 8 bits we manipulate and provide the capabilities to parallel load or shift them.
Changes: The changes that we made to this module were extending D and Data_Out to be 8 bits instead of 4. Other than that the actual function of the registers were maintained.

Module name: **Register_unit.sv**
Inputs: Clk, Reset, A_In, B_In, Ld_A, Ld_B, Shift_En, [7:0] D
Outputs: A_out, B_out, [7:0] A, [7:0], B
Description: This module instantiates a reg_4 unit for both the A register and B register and passes them the inputs that they require to function. All it really does is put the two registers in one place to centralize them and make it easier to interact with them.
Purpose: The purpose of this module is to keep both registers in one place and make it easier to interact with them.
Changes: The changes that we made to this module were extending D, A and B to be 8 bits instead of 4. Other than that the actual function of the registers were maintained.

The above .sv files are the only ones we made changes to. The rest were kept as they were provided to us and are described below:

Module name: **Synchronizers.sv**
Inputs: Clk, d
Outputs: q
Description: This module synchronizes the received signals to the clock
Purpose: The purpose of this module is to keep signals synchronized since it is impossible to manually synchronize every input to the clock.

Module name: **Router.sv**
Inputs: A_In, B_In, F_A_B, [1:0] R
Outputs: A_Out, B_Out
Description: This module takes in three different inputs and depending on the value of R1R0, outputs them in different ways through A_Out and B_Out. If R = 00, A_Out gets A_In and B_Out gets B_In. If R = 01, A_Out gets A_In and B_Out gets F_A_B. If R = 10, A_Out gets F_A_B and B_Out gets B_In. If R = 00, A_Out gets B_In and B_Out gets A_In.
Purpose: The purpose of this module is to properly route the output of our computation unit back to the registers according to the directions given to it.

Module name: **HexDriver.sv**
Inputs: [3:0] In0
Outputs: [6:0] Out0
Description: This module converts a 4 bit input to a 7 bit output encoded to display the input in a human readable way on a Hex display.
Purpose: The purpose of this module is to aid with debugging since it allows us to see the values stored inside the registers

Module name: **Compute.sv**
Inputs: [2:0] F, A_In, B_In
Outputs: A_Out, B_Out, F_A_B
Description: This module takes inputs A and B and performs a bitwise operation on them according to F. The operations are AND, OR, XOR, SET, NAND, NOR, XNOR, and CLEAR, corresponding to F = 0-7 respectively. It then outputs the original values of A and B, as well as the result of the operation.
Purpose: The purpose of this module is to do the actual logic processing and output the desired result bit by bit.

**Modelsim**
Figure 8 shows the ModelSim simulation of our circuit. At 70ns, register A is loaded with value 0x33 and at 110ns, register B is loaded with 0x55. Then, at 170ns, the process is executed and registers A and B are shifted 8 times each, with B getting its original value and A getting A XOR B. Then, at 410ns the process is executed again, but this time with F = 110 and R = 01. This is telling the circuit to store A XNOR B in register B and A in A, which it does. Then, finally R is set to 3, which translates to swapping the bits in Register A and B, which happens after the third execution occurs.

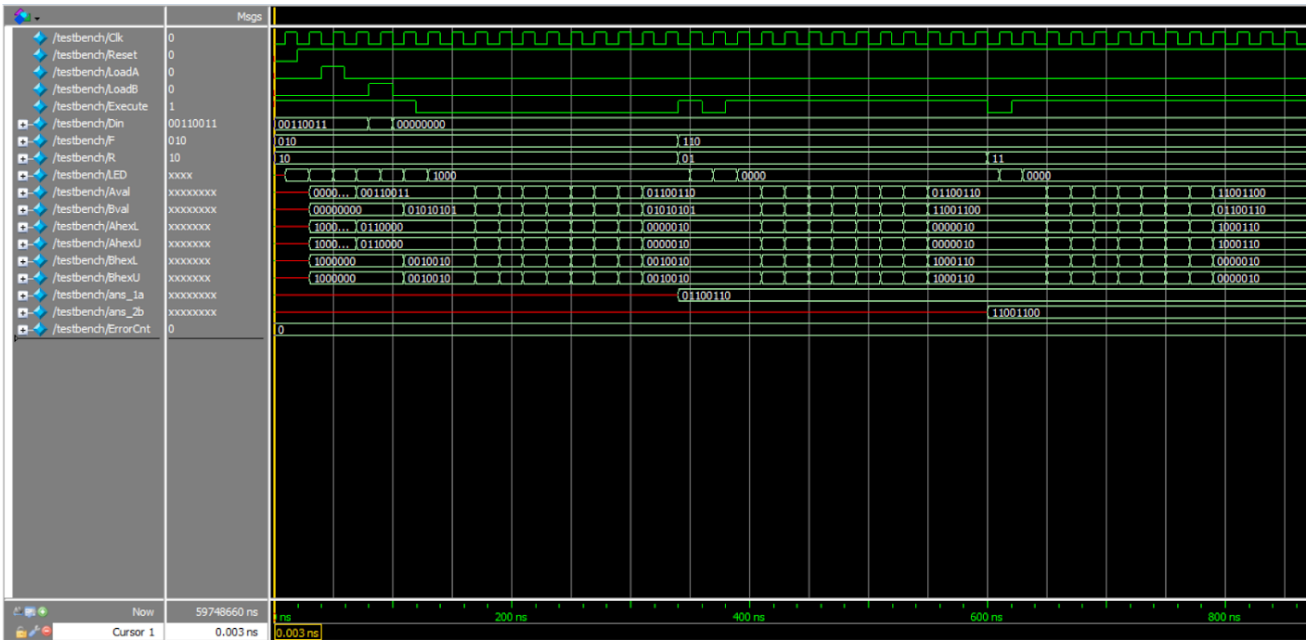


Figure 8. Modelsim simulation

**SignalTap**

Figure 9 displays the result of running 0xA7 XOR 0x53 with the result being stored in register B in SignalTap. To produce this result we stored the values of Register A and Register B in RegA and RegB respectively and set our trigger to be the falling edge of Execute. The reason we have to set it to the falling edge of Execute is because the FPGA buttons are active low, so pressing it actually sets it to 0.

The way this is set up is that F and R are hardcoded to 010 and 01 respectively within the processor file since there are not enough switches on the FPGA, then after the files are loaded to the FPGA we load 0xA7 into RegA and 0x53 into register B. Finally, we hit Execute and at clock cycle 3 the process begins, shifting the registers 8 times and storing 0xA7 XOR 0x53 into register B.

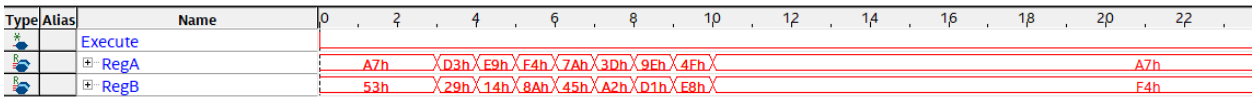


Figure 9. 0xA7 XOR 0x53 in SignalTap

**RTL Design**

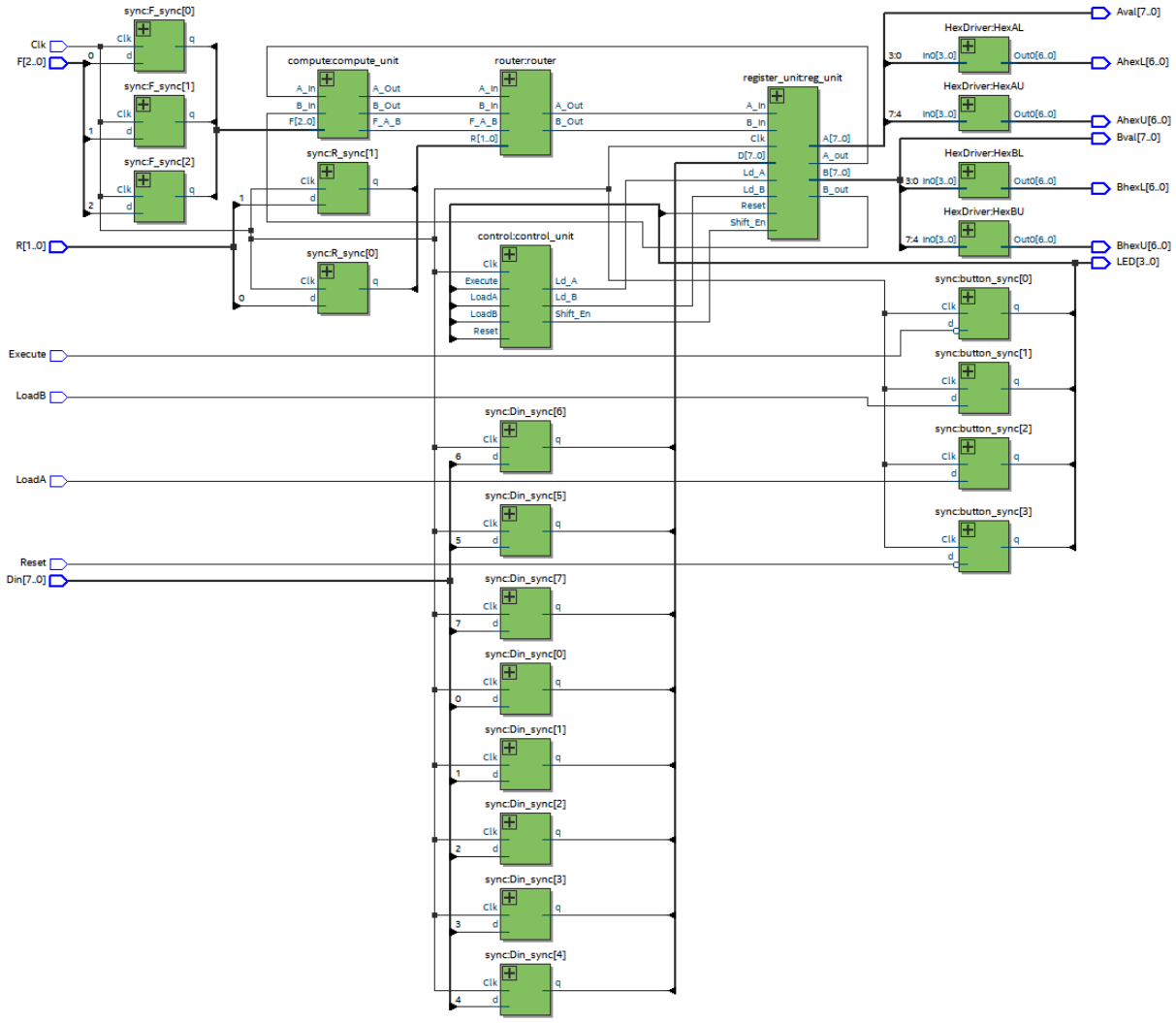


Figure 10. RTL Design of the 8-bit logic processor

## Answers to Post-Lab Questions

**Document changes to your design and correct your Pre-Lab write-up, explaining any difficulties you had in debugging your circuit.**

All of the difficulties we had with the lab came in part 1. We had an idea and diagrams for each of the components within the first day of working on the lab, but actually wiring and unit testing them all presented a big challenge for us. We ran into many difficulties, such as forgetting to connect ICs to power and ground, accidentally having a wire one row off from where it was supposed to be, and connecting the inputs for the computation unit to the wrong bit in the

register. Overall, most of the mistakes that we made were relatively simple, but when dealing with such a large circuit there are a lot of little things that can be overlooked and cause big issues with the execution of the circuit.

**Outline how the modular approach proposed in the pre-lab helps you isolate design and wiring faults, be specific and give examples from your actual lab experience. Make sure you report discusses the following:**

The modular approach helps abstract our circuit into 4 main components that can each be unit tested respectively, allowing for a more systematic approach to circuit building as opposed to throwing everything together at once. Knowing the inputs and expected outputs for each component helped tremendously with unit testing because we could build a single component, feed it inputs and then probe its outputs to ensure it is working as we expect it to. Then, only after we are confident in the component can we connect it to the rest of the circuit and not have to worry about it having bugs. We did this process with each component we made and it helped a ton with debugging because instead of having to worry about every aspect of our circuit, we only had to focus on how we connected them to each other.

**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.**

The simplest circuit that can invert a signal is an XOR gate. If a 1 is fed into one of its inputs, the other input will be inverted (ie a 1 will produce a 0 and vice versa), whereas if the control input is a 0, the other input will not be inverted. This is useful for this lab because instead of needing to use additional NOT gates, we could use the unused XOR gates that were unused in the 7486 chip. In reality, we did not think of this at the time, but it would have proven useful. Additionally, this would have been very useful for debugging because we could confirm with an XOR gate that our circuit was producing the desired output, which will be explained more in the next question.

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

The utilization of XOR gates can improve testability by allowing for more effective testing. If you hook up the expected output to one of the inputs and the actual output to the other output, the gate will produce a 0 if the true output is correct, but 1 otherwise. These sorts of tests can be placed throughout your circuit to ensure that everything is working as expected and make debugging much faster. XOR gates can also cut down development time because they are the simplification the combination of a few simpler gates, so whenever the opportunity arises to use one it is almost always beneficial.

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

A Mealy machine is a FSM whose next state is determined using both the current state and current inputs, whereas a Moore machine's next state is determined solely using information about its current state. What's nice about the Mealy machine is that by using the current inputs and state, we can greatly reduce the total number of states and therefore the number of logic gates required to create an FSM. On the flip side, a Moore machine is nice because it is very easy to design. All you need to know is what you want your FSM to do and create a state for each step. In this sense, a Moore machine is faster to design conceptually, but a Mealy machine is more beneficial to implement if time is not a factor.

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

ModelSim utilizes a testbench to test the circuit made using your SystemVerilog file with your modules. It simulates a signal based on your code and what the inputs are, with the signal outputs displaying afterwards. No hardware is necessary to simulate a signal using ModelSim. SignalTap however requires physical hardware, such as the DE10-Lite FPGA that we currently use in the ECE 385 curriculum. SignalTap requires that certain pin assignments are made for all the input signal and LED displays that are to be utilized. SignalTap does not require a testbench file. All of the inputs are determined using the switches on the FPGA for the purposes of this lab, and Execute is set to high with a press of the button module on the FPGA. For ModelSim, Execute is active high once the simulation starts. Inputs for when simulating using ModelSim are set in the testbench, while the inputs for when simulating using SignalTap are set using the switches on the FPGA. Situations where more than 4 bits would be processed would probably be best to have simulated using ModelSim because you could alter your circuit to execute with larger numbers and then select the inputs to the multiplexers used for the computation unit and routing unit. Situations where 4 or less bits are used, it is better to use SignalTap so that you can select the inputs to the registers as well as declare the bits for which function is to be executed and where the values would be displayed.

## Conclusions

Overall, our circuit consists of a control unit, which takes Execute as an input and outputs S, which is an indicator to the other components to continue operation. When the shift registers receive this input, they send their rightmost bit to the computation unit, where the desired computation has been encoded with F[2:0]. The product and original A and B are then sent to the routing unit, where R[1:0] dictates where the signals are sent. Finally, the signals make it

back to the registers and are stored in the leftmost bit. This repeats for as many bits as are in the registers and then terminates. On the breadboard we made a 4-bit version of this and in Quartus we extended it to 8 bits.

Additionally, we have learned that when implementing a circuit on a breadboard, it is best to debug each unit separately in order to make sure that the design is being implemented correctly. This is known as unit testing, and it helps ensure that when the design is fully implemented, if there is any more debugging necessary it will not be as a result of the units not working properly themselves, but rather the connections made between the units themselves or the hardware being used. Additionally, there was a lot to be learned about how a circuit can be abstracted into parts, with each component being approached separately and appropriate to the functionality of that part. For example, when approaching the control unit where the most logic needed to be implemented, we made sure to use the Mealy FSM provided to us to think about how the circuit would shift bits with each clock cycle onto a different set of LEDs as well as how it would call on the necessary computations to be executed.