University of Illinois at Urbana-Champaign

ECE 385

LAB 2

LAB REPORT

By Shubham Gupta & Devul Nahar (sg49 & danahar2)

2/12/2022

1. Introduction:

In this lab we will design and build a 4-bit serial logic operation processor. It can perform 8 different bitwise logical operations: AND, OR, XOR, FILL 1, NAND, NOR, XNOR, and FILL 0. It operates on 4-bits and utilizes two 4-bit shift registers. The result can then be routed to 4 different locations based on the user's input. We will use a Mealy finite state machine to manage the states as the control unit, the states will be stored in a flip flop.

2. Operation of Logic Processor:

a) Following are the steps that must be used to carry out an operation, first let us parallelly load data into registers A and B:

- Set Load A switch to 1 and Load B switch to 0.
- Set values for switches D0, D1, D2, and D3. (4-bit data for A).
- Set Load A switch to 0 and Load B switch to 1.
- Set values for switches D0, D1, D2, and D3. (4-bit data for B).
- Set Load A switch to 0 and Load B to 0.

b) Now to compute and start the finite state machine, the user must perform the following:

- Set values for switches F2, F1, and F0 to carry out the desired bitwise operation as described in the table below:

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

Figure 1 Function select input and outputs

- Set values for switches R0 and R1 to carry out the desired routing operation as described in the table below:

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

Figure 2 Routing selection table

- Set Execute switch to 1 to begin the operation.

3. Description, Block diagram, & Finite State Machine of Logic Processor:

a) Description:

1. **Register Unit:** We have two 4-bit shift registers called RegA to store data for A and RegB to store data for B. These are being used as right shift registers, i.e., for every rising edge of the clock cycle they output the least significant bit and the most significant bit is updated. Note that for this lab we chose to use the 4-bit BIDIR Shift Register (194E) because it has the hold state, which is preferred because of the mealy state machine implementation of our design.
2. **Computation Unit:** We noticed that this unit needs to carry out bitwise logical operations of 8 different kinds, in which the first four operations are the inverted inputs of the remaining four operations, i.e., AND, OR, XOR, and FILL 1 are the inverted outputs of NAND, NOR, XNOR, and FILL 0 respectively. Therefore, this

unit consists of a 4 to 1 MUX in which the 4 inputs are connected to AND, OR, XOR, and FILL 1. The two select bits of the 4 to 1 MUX are connected to F1 and F0. To implement the remaining four operations, we used a XOR gate with inputs F2 and the output of the 4 to 1 MUX to compute the final. Note that the reason we did this is because the XOR gate inverts F whenever F2 is 1 (0 XOR 1 = 0, 1 XOR 1 = 0). AND, OR, and XOR are implemented using TTL chips that are provided. Fill 1 was implemented by wiring the output to high voltage.

3. **Routing Unit:** We simply use two 4 to 1 MUX to route f(A) and f(B) based on the selection bits R1 and R0. Note that in implementation, we only use one Dual 4:1 MUX.
    - **a.** The inputs of MUX for A are A, B, and f(A), the outputs are A' (new A).
    - **b.** The inputs of MUX for B are B, f(A), B, and A, the outputs are B' (new B).
4. **Control Unit:** The control unit consists of:
    - **a.** 4-bit counter – keeps track of the number of shifts the register has gone through. Note we only use 2 bits of these 4 bits as we only need to count to 4.
    - **b.** Flip flop – stores the current state of the FSM and helps determine the next state (along with the inputs).
    - **c.** Combinational logic – implements the Mealy state machine for this lab. It does this by using combinational logic for Q^+ and S. Upon execution, the unit completes the complete cycle and only then comes to halt/rest state.

b) High Level Block Diagram:

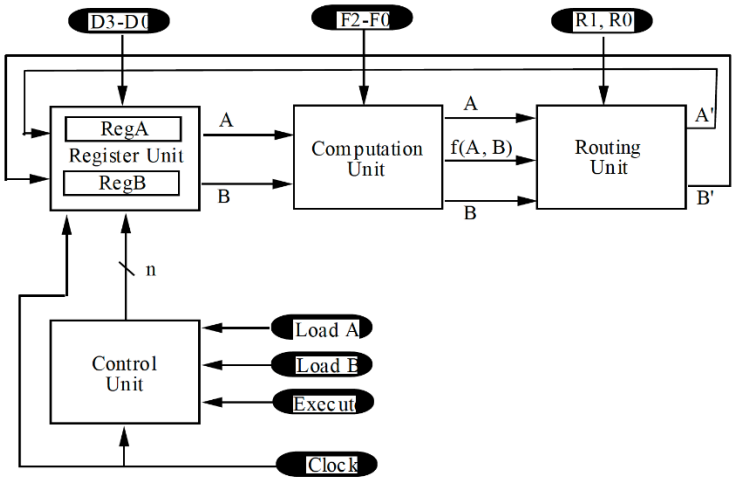

Figure 3 High level block diagram

c) State Machine Diagram:

i. In this implementation we have used the **Mealy State Machine** because Mealy State machine only used one flip flop as there are only two states, 0 and 1, while the most efficient Moore state machine used 3 states 0, 1, and 2 (for hold). Having only two states meant that we only needed to use one D Flip Flop (as it can store 0 and 1) instead of two.

ii. The following state diagram depicts our Mealy State Diagram which has two states, Rest (0) and Shift/Hold (1). Since Mealy machine's next states are determined by current states and

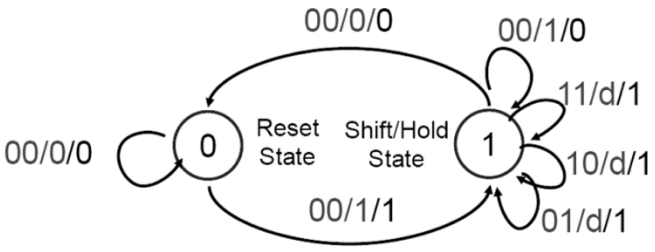

Figure 4 Mealy State Diagram

current inputs, the FSM therefore changes based on C1, C0, E, and Q, i.e., C1C0/E/Q.

***Note that d in this diagram denotes don't care.**

4. Design and Circuit Schematic:

We used truth tables for the control unit and for some combinational logic in the control unit to implement loadA and loadB. First let us look at the implementation of our Mealy state machine based on the the truth table as follows:

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 5 Mealy FSM Truth Table

The input Execute (E) is for starting the the execution process. Q denotes the states of the Mealy machine which are Reset (Q = 0) and Shift/Hold (Q = 1). C1 and C0 are used to keep track of the counter. Since we have 2-bits for counting, we can count to 4 states (C1C0 = 00, 01, 10, 11). The output S is fed into the register unit and determines whether a shift should take place. Q^+ is the

next state for the machine, meanwhile $C1^+$ and $C0^+$ are the next state for the counters. **Note that we don't draw the K-maps for $C1^+$ and $C0^+$ as they are handled by the counter unit itself.**

The K-Map for Mealy State Machine is as follows:

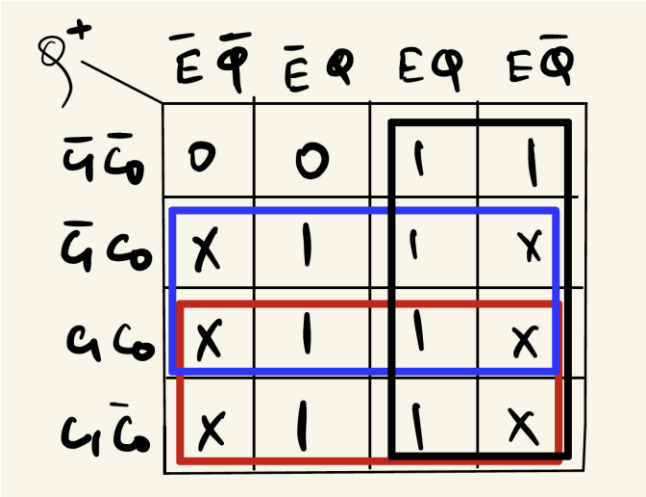

Figure 6 K-Map for Q+

Therefore, the expression for Q^+ is:

$$Q^+ = C0 + C1 + E$$

Similarly, the K-Map for S (Shift Enable) is as follows:

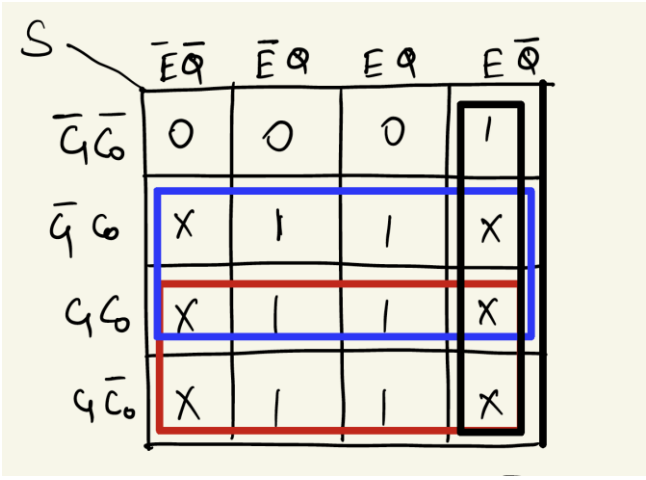

Figure 7 K-Map for S

Therefore, the expression for S is:

$$S = C0 + C1 + E\bar{Q}$$

Now we need a truth table for implementing LoadA and LoadB. Before we move on to the truth table, we need to check the datasheet for CD74HC194E which is the TTL chip for RegA & RegB:

OPERATING MODE	INPUTS							OUTPUT			
	CP	$\overline{MR}$	S1	S0	DSR	DSL	D_n	Q_0	Q_1	Q_2	Q_3
Reset (Clear)	X	L	X	X	X	X	X	L	L	L	L
Hold (Do Nothing)	X	H	l	l	X	X	X	q_0	q_1	q_2	q_3
Shift Left	↑	H	h	l	X	l	X	q_1	q_2	q_3	L
	↑	H	h	l	X	h	X	q_1	q_2	q_3	H
Shift Right	↑	H	l	h	l	X	X	L	q_0	q_1	q_2
	↑	H	l	h	h	X	X	H	q_0	q_1	q_2
Parallel Load	↑	H	h	h	X	X	d_n	d_0	d_1	d_2	d_3

Figure 8 Datasheet for the shift register

Therefore, LoadA and LoadB will utilize the same combinational logic as they operate on the same TTL chip. The only difference is that the Load signal will be LoadA for RegA and LoadB for RegB. S1 and S0 select between the parallel load and shift right modes in the TTL chip. These bits can be generated using the Load (L) and Shift (S) signals. The following is the truth table:

L	S	S0	S1
0	0	0	0
0	1	1	0
1	0	1	1
1	1	1	1

The following are the K-Maps for S0 and S1:

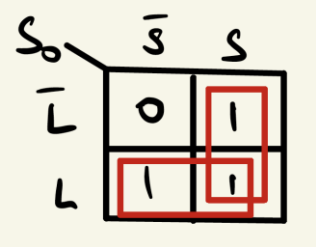

Figure 9 K-Map for S0

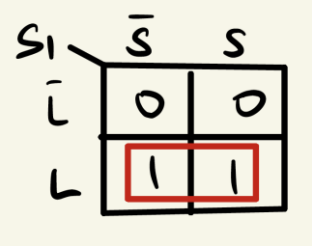

Figure 10 K-Map for S1

The expression for S0:

$$S0 = S + L$$

The expression for S1:

$$S1 = L$$

Other considerations: We could have used a Moore FSM instead of a Mealy FSM. The Moore FSM would take a total of six states. On the other hand, a Mealy FSM takes only two states which can be represented by a single flip flop. Therefore, we implemented Mealy FSM as it takes less hardware and is easy to put on circuit. In addition, we also initially thought about implementing the ALU using an 8 to 1 MUX but ended up using an XOR gate after realizing that the bottom 4 functions are simply inverted outputs of the top 4 functions. Finally, we also

thought about implementing the XOR and NOT gates using NOR gates, but then ended up using the XOR and NOT chips instead as we realized that we used those functions multiple times in the circuit, and it will be more convenient using those gates instead.

Detailed Circuit Schematic:

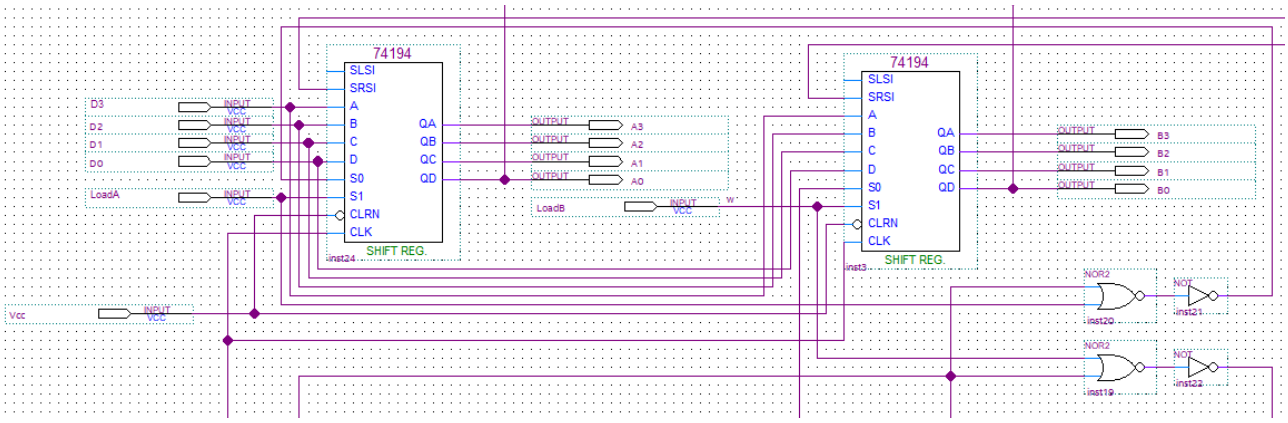

Figure 11 Register Unit

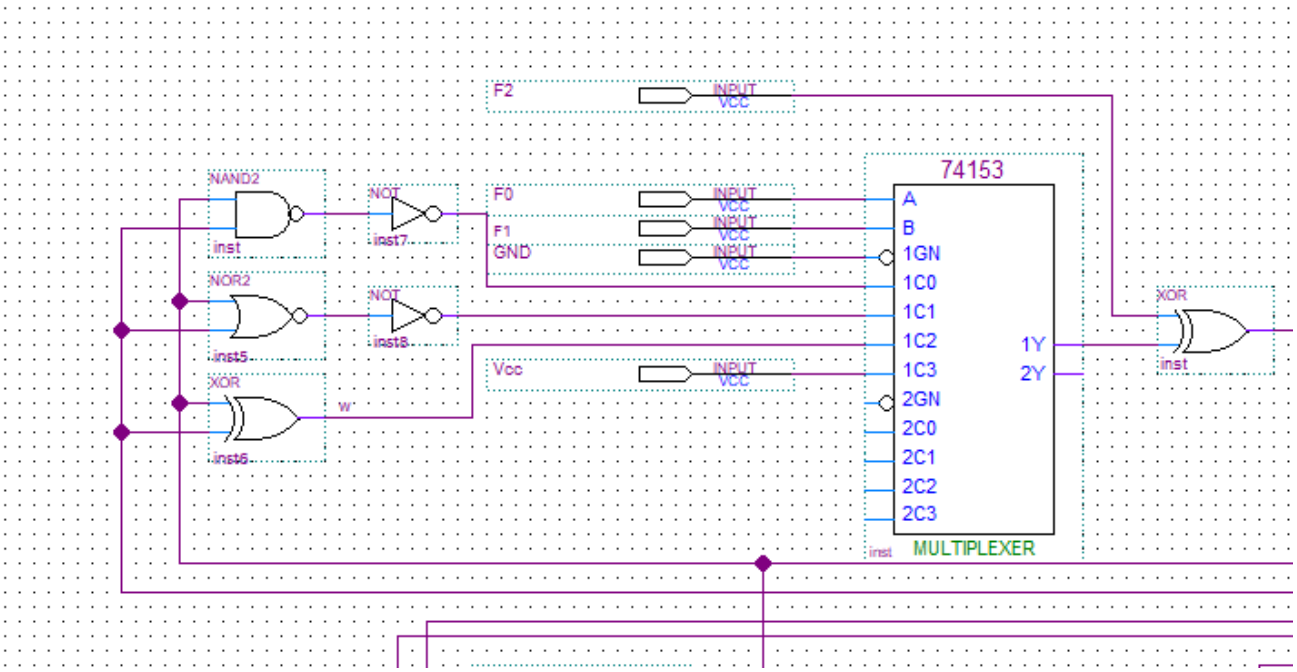

Figure 12 ALU

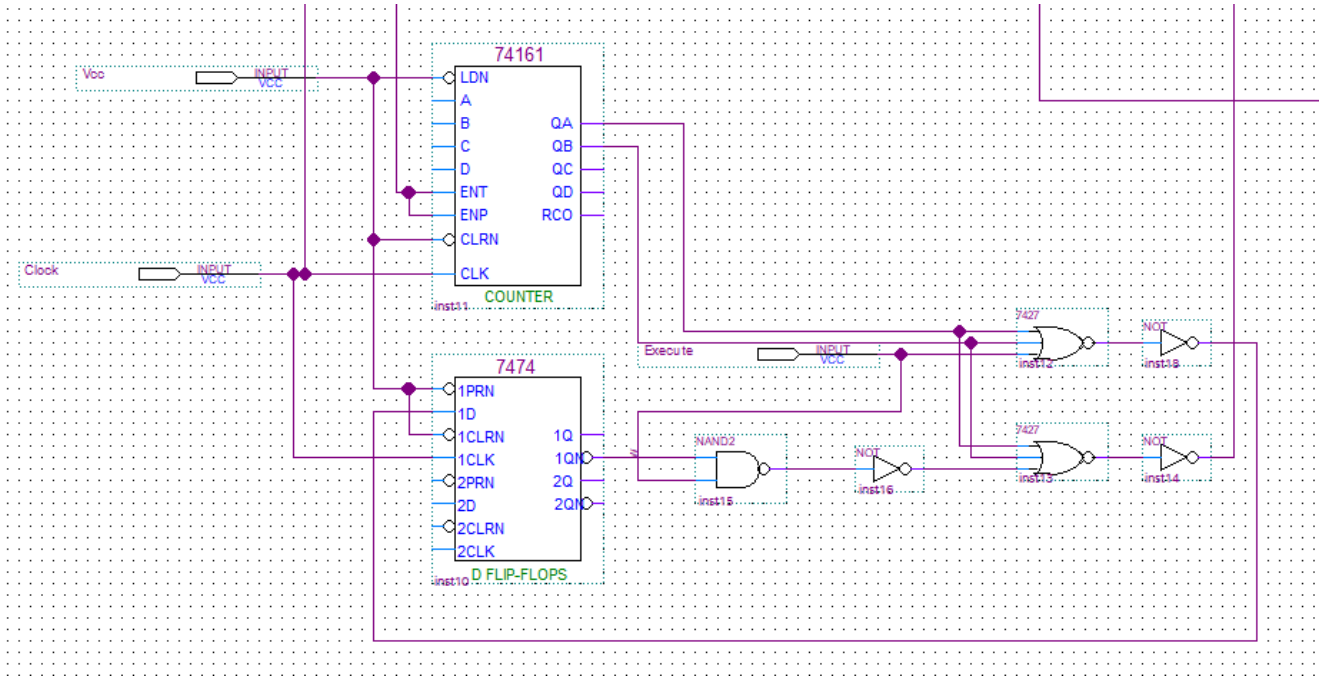

Figure 13 Control Unit

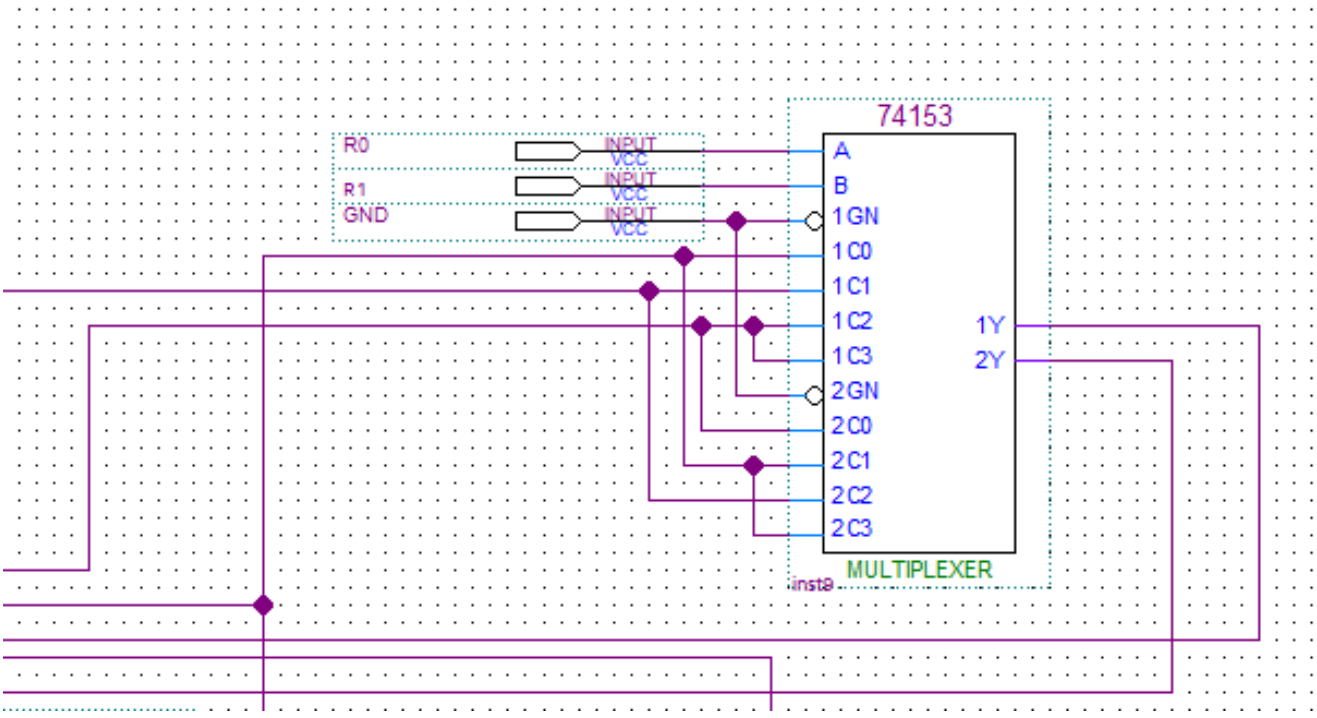

Figure 14 Routing unit

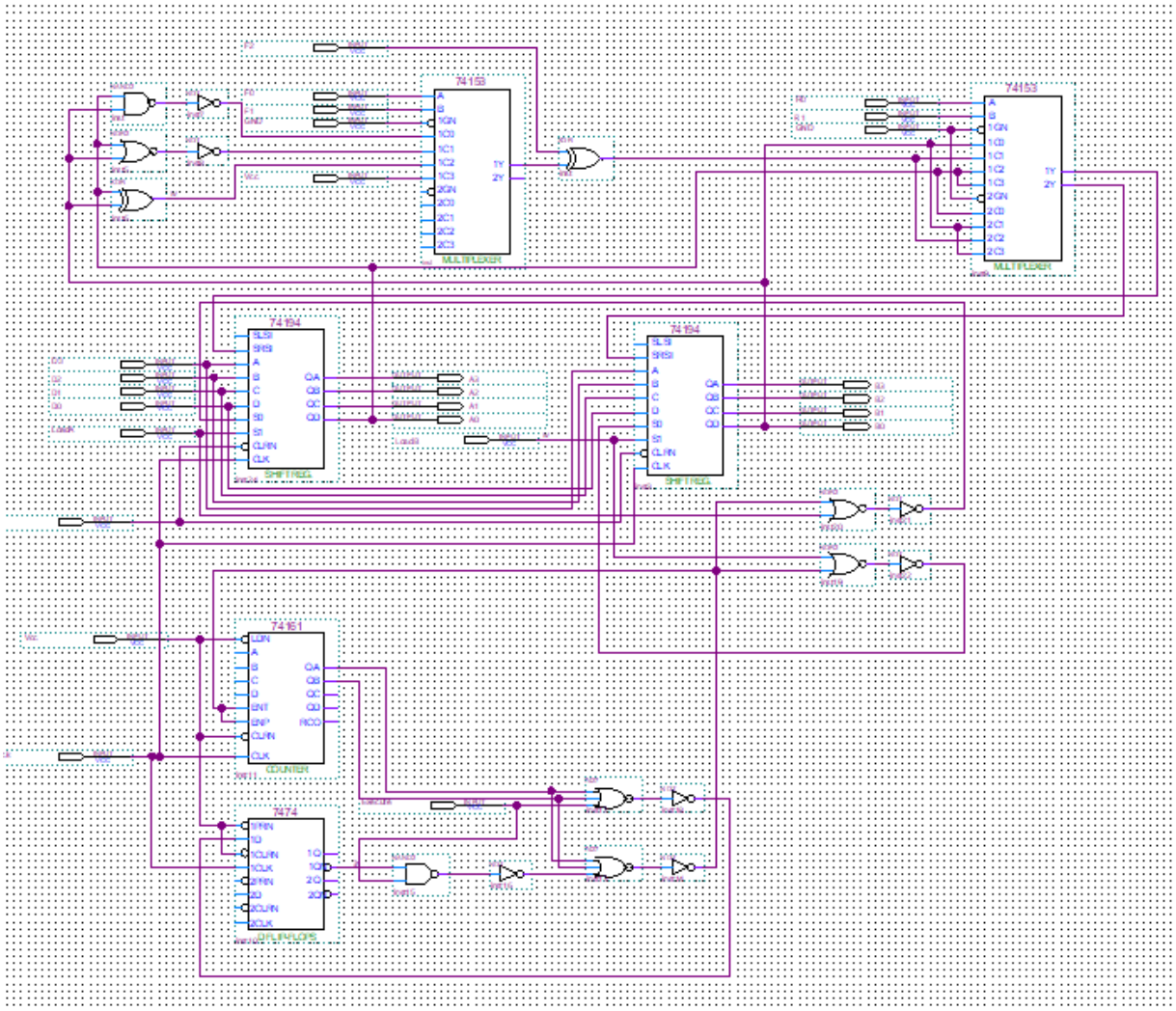

Figure 15 Complete circuit schematic

Now let us see the breadboard view…

5. Breadboard View:

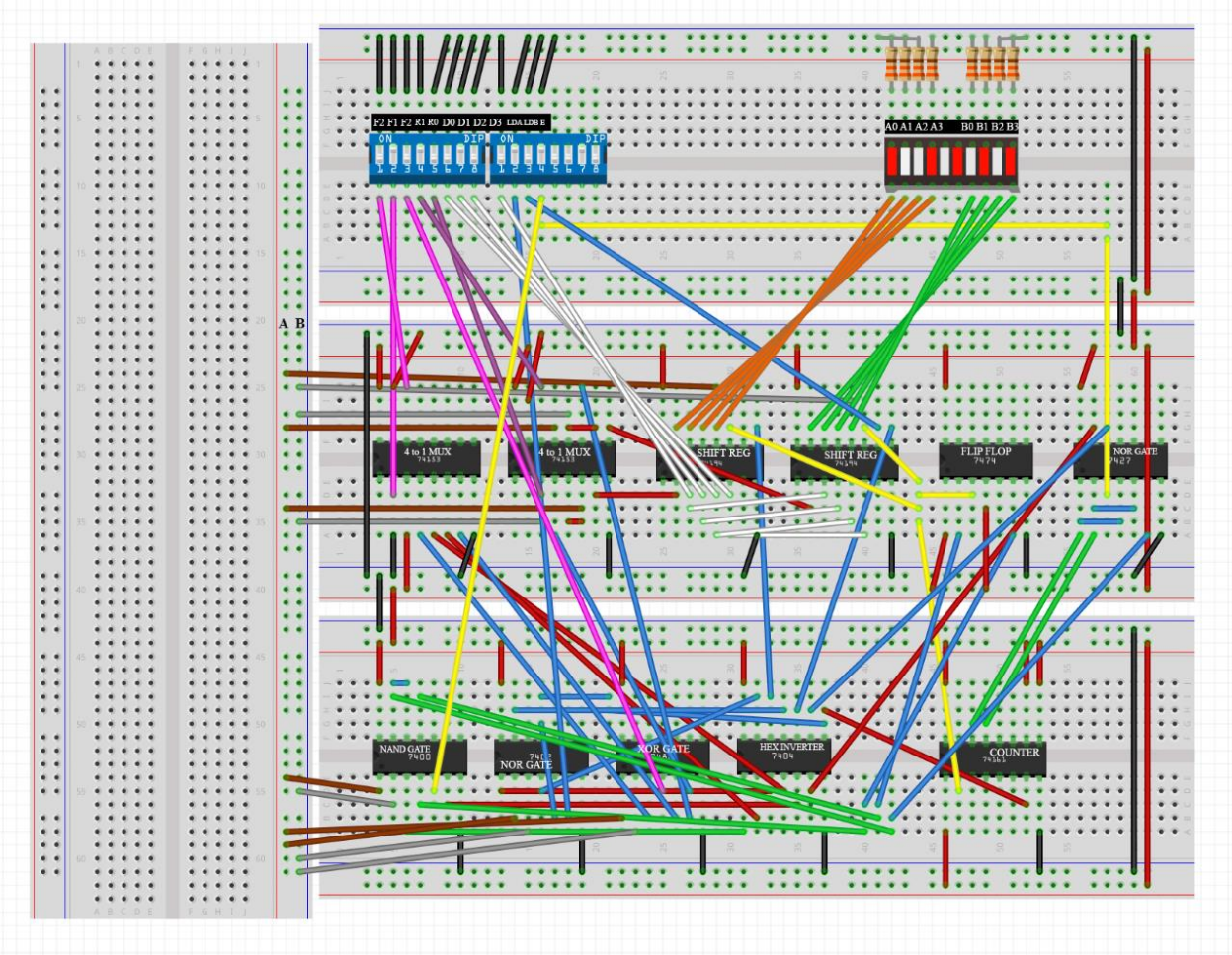

Figure 16 Breadboard layout diagram

6. 8-bit Logic Processor on FPGA:

a. Description:

Module: compute.sv

Inputs: [2:0] F, A_in, B_in

Outputs: A_out, B_out, F_A_B

Description: It is a one-bit ALU. It can calculate 8 different logical operations. We used UNIQUE to create a MUX.

Purpose: It calculates 8 different logical operations AND, OR, XOR, FILL 1, NAND, NOR, XNOR, and FILL 0.

Module: Control.sv

Inputs: Clk, Reset, LoadA, LoadB, Execute

Outputs: Shift_En, Ld_A, Ld_B

Description: It is a Moore FSM with start, hold and shift states. We have a counter that shifts 8 times once the execute is pressed. Once it starts it doesn't stop until the shifting is completed irrespective of the input execute. Once execute is 0, it goes to hold state and after execute is 1, it begins execution again.

Purpose: It implements the states of the Moore FSM based on current state and calculates the next state.

Module: HexDriver.sv

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: We use UNIQUE switch case to map the binary values to the hexadecimal display.

Purpose: It maps the hexadecimal value to the LED display.

Module: Processor.sv

Inputs: Clk, Reset, LoadA, LoadB, Execute, [7:0] Din, [2:0] F, [1:0] R

Outputs: [3:0] LED, [7:0] Aval, [7:0] Bval, [6:0] AhexL, AhexU, BhexL,BhexU

Description: We instantiate the necessary values of the modules and perform all I/O connections between all the modules.

Purpose: This is the top-level entity in the Quartus. It uses all modules to assemble and map the entire circuit.

Module: Reg_8.sv

Inputs: Clk, Reset, Shift_In, Load, Shift_En, [7:0] D

Outputs: Shift_Out, [7:0] Data_Out

Description: We have implemented an 8-bit right shift register in this module. When reset is 1, Data_Out is set to 0. When Load is 1, D gets the value of Data_Out. When we enable Shift_En, the register shifts all the values, and the least significant bit is sent to Shift_Out.

Purpose: 8-bit shift register is implemented to be used as RegA and RegB.

Module: 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: We instantiate two instances of reg_8 as regA and regB. Then add all the required input and output values.

Purpose: Stores 8-bit values of RegA and RegB, acts like register unit as defined in design.

Module: Router.sv

Inputs: A_In, B_In, F_A_B, [1:0] R

Outputs: A_Out, B_Out

Description: We use two 4:1 MUXes to implement the routing. The select bits are R[0] and R[1]. The output of the MUX is set to A_Out and B_Out.

Purpose: This implements our routing unit in the design. It routes data based on the input R[0] and R[1].

Changes made: Since we are just increasing the number of bits to operate on, we basically change all inputs and outputs from 4 bits to 8 bits in the register unit (reg_4.sv). The next major change is in the control unit. We changed the number of states in the FSM from 6 to 10 states to account for the extra 4 bits (control.sv). The Processor.sv also had to be changed as the input and outputs have a new bit size of 8. Moreover, since the contents of the registers are now 8 bits, the hex drivers need to have 2 additional instances to display the upper and the lower 4 bits of the registers. Lastly, we had to use the 8-bit test bench to run simulation.

b. RTL Block Diagram:

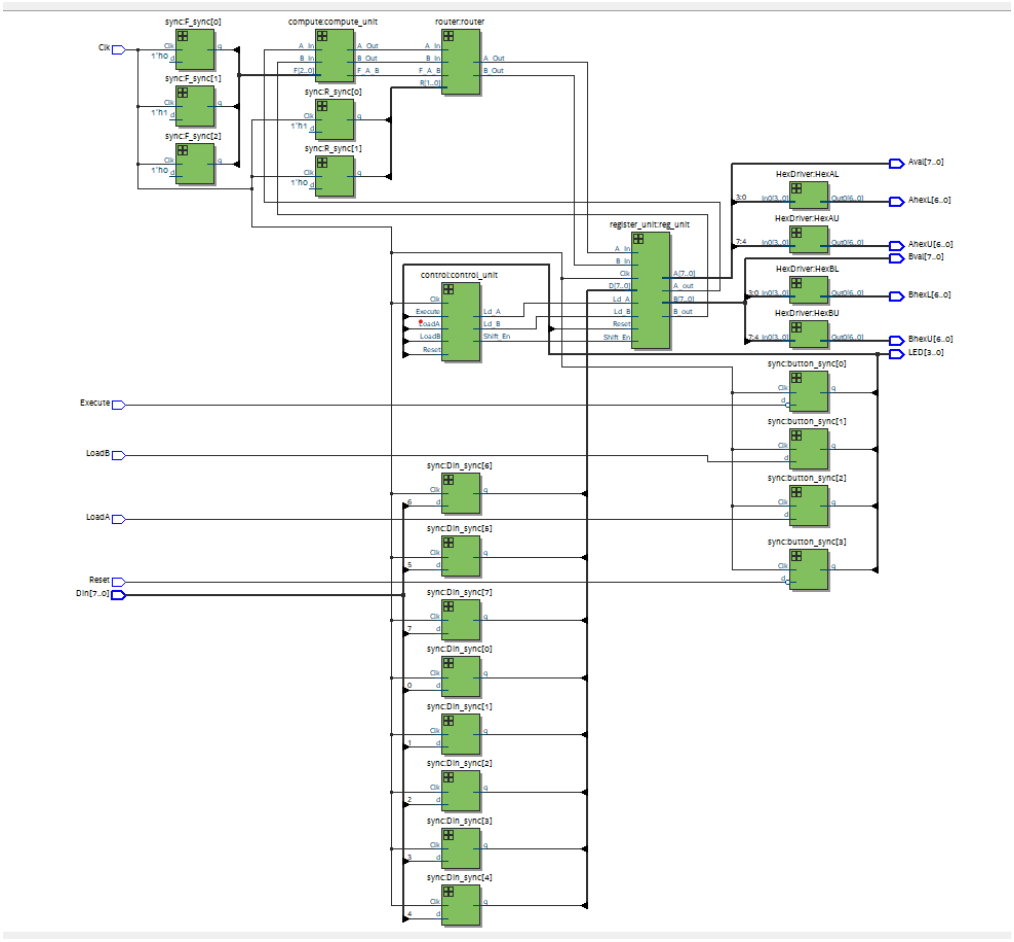

Figure 17 RTL Bloack Diagram

c. Annotated ModelSim Simulation:

Function 1:

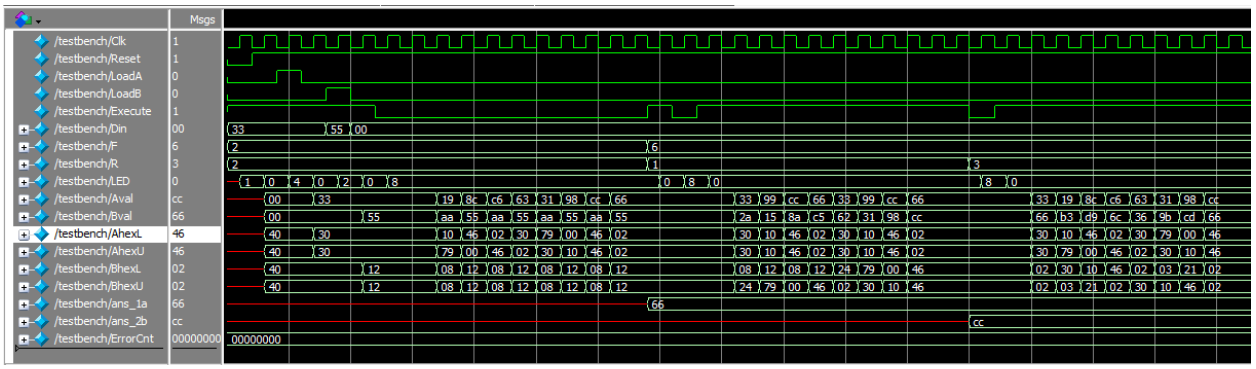

Figure 18 Function 1

Explanation:

- F = x2 meaning that we are doing the XOR function (F=010).
- R = x2 meaning that we store F in A (R=10).
- Din = x33 when LoadA is high. This means that Register A is loaded with value x33.
- Din = x55 when LoadB is high. This means that Register B is loaded with value x55.
- After 8 cycles, the output x66 (x33 XOR x66 = x66) is routed to Register A, while Register B retains its original value as expected.

Function 2:

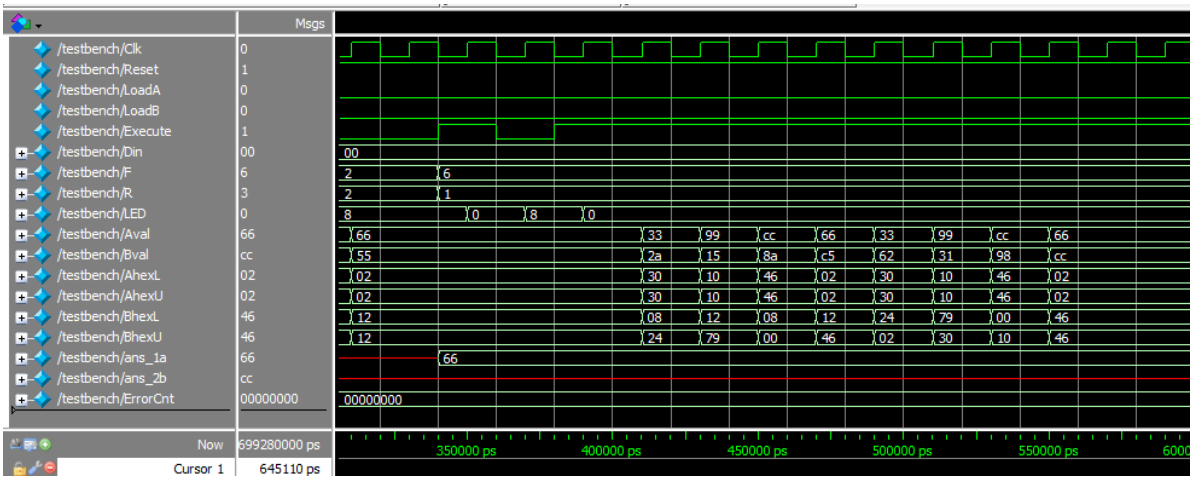

Figure 19 Function 2

Explanation:

- F = x6 meaning that we are doing the XNOR function (F=110).
- R = x1 meaning that we store F in B (R=01).
- Due to the last function, Register A currently stores a value of x66.
- Due to the last function Register B currently stores a value of x55.

- After 8 cycles, the output xCC (x66 XNOR x55 = x66) is routed to Register B, while Register A retains its original value as expected.

Function 3:

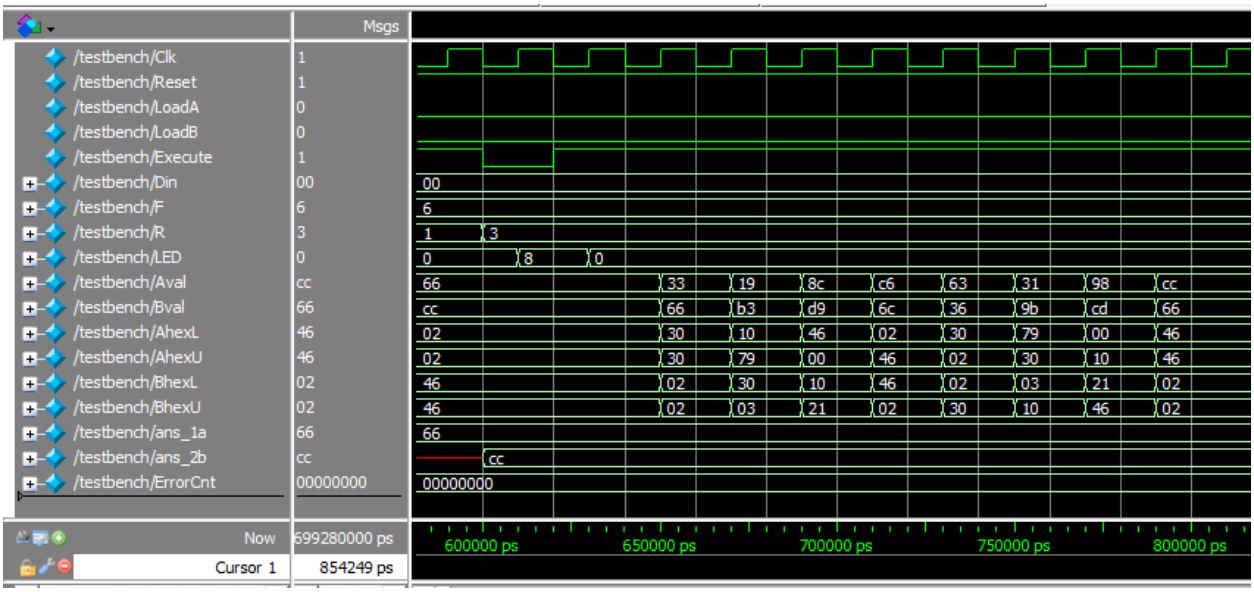

Figure 20 Function 3

Explanation:

- F = x6 meaning that we are doing the XNOR function (F=110).
- R = x3 meaning that we simply Route the value in A to B, and B to A (R=11).
- Due to the last function, Register A currently stores a value of x66.
- Due to the last function Register B currently stores a value of xCC.
- After 8 cycles, the output of the XNOR function becomes irrelevant as we are not storing the output in any of the registers, we are simply swapping the data bits from A to B.
- Therefore, Register B storing the value of x66 and Register A storing the value of xCC, which is to be expected.

d. Procedure used to generate SignalTap ILA trace:

1) Set all the pin assignments of inputs and outputs as per your requirement (in our case it was given to us in the lab manual).
2) Set the main file as the top-level entity (in our case it was the processor).
3) Compile and synthesize the code.
4) Connect FPGA to a USB port on the computer.
5) Navigate to Tools -> Signal Tap Logic Analyzer to open signal tap.
6) Now to set up the clock signal tap:
    a. Go to signal configuration -> clock (… symbol)
    b. Search for the clock node with an assigned pin, and then add it by pressing okay.
    c. Keep the sample depth to 64 (or as per requirement).
7) Now to set up the other signals that we want to analyze
    a. Double click in the region right below the node table.
    b. Look for the signals of interest and its nodes. In our case it was:
        i. reg_4:reg_A::Data_out[0:7]
        ii. reg_4:reg_B::Data_out[0:7]
        iii. Execute (triggered on either edge)
    c. Group any signals you want together.
    d. Start compilation of the whole design again.
8) Now to run signal tap:
    a. Inside signal tap, go to JTAG Chain Configuration.
    b. Select USB Blaster Hardware

c. Click Program Device

d. Click Run analysis

9) Now the FPGA has been programmed. To analyze the signal tap output, load values into register A and B, and then click execute. You can know analyze the output of the signal tap and see how selected signals have changed over time.

Bugs & Corrective Measures:

We faced minor bugs while building the actual circuit which were mainly due to human errors. We initially forgot to invert the output of a NOR gate which was producing incorrect values. This was debugged by checking the values of F, RegA, and RegB via LEDs. To prevent further such issues, we ensured the wiring of each TTL chip as per our breadboard design twice. Upon, completing the entire circuit we faced a bit error due to lose wiring, to fix this we double checked if every wire is tightly connected.

Conclusion:

In this lab we implemented a 4-bit serial logic processor which can carry out 8 different bitwise operations and can route them to 4 different locations. Moreover, we learnt about the Mealy state machine and how it differs from the Moore state machine. This lab utilized both sequential and combinational logic in different units. In addition, we learnt about TTL pin layouts and implemented the FSM with a 4-bit counter and a flip flop.

In the second half of the lab, we learnt about System Verilog Modules and how to implement them on an FPGA. We then used our understanding of the 4-bit logical processor to extend it to an 8-bit logical processor. In doing do, we used a Moore FSM for the FPGA design.

For debugging in System Verilog, we used ModelSim and Signal Tap while also understanding how they differ.

One major problem that we discovered as we were doing the lab was that as soon as you turned on Load A and Load B in middle of the execution cycle, the registers would stop shifting and would load whatever value is provided by our Data in bits. One way in which we could enhance the circuit to account for this fault is by adding additional combinational logic to S0 and S1 which decide the mode of the register.

Answer to 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.

We can perform this by using a 2 input XOR gate.

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

We have four main modules in our design: control unit, register unit, routing unit, and computing unit. We can design and unit test each of the parts by simulating them with a clock/test value. This greatly reduces the debug process as we do not need to debug the entire data path. Therefore, improves testability, development time is reduced by making the design part by part.

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

We preferred to go with Mealy machine primarily because:

1. Mealy machine implements only 2 states, which can be represent by a single 1-bit flip flop. Hence, we use less hardware in contrast to Moore machine which would have required 8 states that requires 3-bits of flip flop.
2. The combinational logic apart from flip flop hardware that is required, is comparatively more in a Moore machine as compared to a Mealy machine in this case. This is because Mealy machine depends on the current state and current inputs to decide next state.

In conclusion, a Moore machine is more intuitive in terms of design but can sometimes require a lot more states than a Mealy machine. The state diagram and implementation of a Mealy machine is harder, but in this case requires less hardware. These are the trade-offs.

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 is very different from SignalTap since it simulates a design, rather than collecting actual data points from the physical circuit. SignalTap cannot be used to manipulate signals, but it is useful in evaluating logic which can otherwise be very hard to view.

ModelSim is better in situations where we would want to partially debug our circuit. For example, each individual module like a full adder. This is helpful as in a full built circuit such a small error would be much harder to find.

SignalTap would be much more useful when we would want to find differences between the virtual model and the actual synthesized circuit on the FPGA. SignalTap is a real time representation of actual hardware.