ECE 385
Lab 2: Logic Processor

Partners: Akshat Singh and Adnan Challawala
TA: Shitao Liu
Lab section: SL

Introduction:

Summarise what high-level function your circuit performs. What operations can the processor do? How many bits can it operate on? Etc. The introduction should be approximately 3 - 5 sentences.

- Through the course of this lab report we aim to design and build a logic processor on a breadboard and on the Quartus Prime software. The logic processor is divided into 4 main components which comprise the: logical unit, Control Unit, Routing unit, and the Arithmetic unit. We were responsible for the design and implementation of the above-mentioned units along with the documentation that was provided to us. We used a variety of IC chips like Hex inverter, Registers, Flip Flops, NAND, NOR, XOR and many more to implement the functioning of the logic processor. Depending on the user input, the logic processor was able to take in and store user values into the register and then using the user inputs it was able to perform arithmetic functions on those values and store it on to the different register, again depending on the user value. We were to expand these operations from 8 bit to 16 bits on the Quartus prime software by making various changes onto the code provided to us.

Operation:

A. Describe the sequence of switches the user must flip to load data into the A and B registers.

- We had a variety of switches that were used to take in user values, which determined which operation each unit would be performing. The clock was constant and the same for all the units. The switches would take in the bit of either 1 or 0, and then store those values into the register. There were sets of bits of D0,D1,D2,D3, one for register A and one for register B. They were labelled as inputs by the user. Then we had two switches called LOAD A and LOAD B, which determined if the values would be loaded into the register or not. So, the sequence was as follow: execute would be set to low (0) and then the user must flip the switch to make LOAD A or LOAD B high, and flip the switches to set D0, D1, D2, and D3 to the value that the user wants to store into the register. Once we are done, we will switch the LOAD A or LOAD B to low and then proceed with EXECUTE, if we wish to.

B. Describe the sequence of switches the user must flip to initiate a computation and routing operation.

- The routing unit and computation unit allows the user to perform different operations of their choice on the logical unit. Both of these units have input bits which determine

what operation to perform on these units. The computation unit has 3 sets of input bits, which leads to 8 possible combinations of operations that can be performed by the computation unit. Each combination of 0 and 1 represents an operation like NAND, NOR, XOR, XNOR and more. Similarly, the routing unit had 2 sets of input which leads to 4 possible combinations of where the output from the routing unit will be routed to. So first the user will press EXECUTE and then the user must specify the desired logical operation to be performed on the binary values stored within the A and B registers using the F0, F1, and F2 switches on the computation unit. Subsequently, the user must decide on the storage and display of the computation.

Written description, block diagram and state machine diagram:

A. Written description: describe in words each block in the high-level diagram.

- **The logical processor into 4 parts and here is the information about each of the four different types of units:**

1. **Routing unit:** The routing unit is responsible for routing the result from the computation unit back to register, depending on the type of input given by the user. The routing unit comprises two muxes, one for each register. Each of the muxes have 4 inputs, 1 output and a 2 bit select pin, which is entered by the user. The output of the muxes go as input back into the register to display the result, depending on the input of the user.

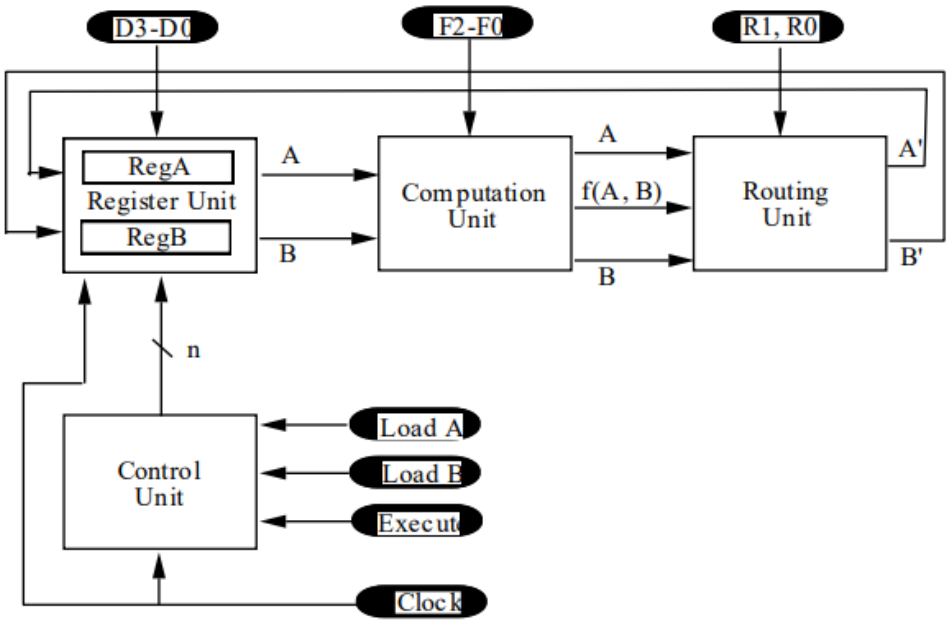
2. **Control unit:** The Logical unit is responsible for the FSM cycle of the system which controls the number of different states the system must be in to complete one full cycle of operation. For the application of the system we went forward with the Mealy state machine and we implemented the FSM using positive edge flip flops, logic gates and input switches such as Execute(E), Count bits(C1C0) and single bit state(Q). Each of these bits are responsible for a state and the cycle of the whole logic processor.

3. **Register unit:** The Register Unit is responsible for storing the bits that user enters using the input switches D3, D2, D1, D0. Further, the switch Load A and Load B determines which register to load the value. The unit comprises of two 4-bit shift register IC chips. There is another input into this shift register. This is from the routing unit which refills the value of these registers after the logic has been performed. The output from these registers goes to the computation unit for the computation between the bits from register A and B.

4. **Computation unit:** The Computation unit is responsible for the computation between the bits of the two registers. We implemented the computation unit using a combination of IC chips which perform specific functions like NAND,

NOR, XOR and a hex inverter to invert the output to perform functions like AND, OR, XOR. The unit also comprises a 4-1 MUX along with some logic implemented with a XOR gate to choose from the various operations possible. The user is allowed to send in a 3(F2, F1, F0) bit select input to choose the various operations. But we use only two of these inputs for the mux and then use the left out select bit as an input to XOR gate. The output from this unit goes into the routing unit, to store the result back into the register.

B. Include a high-level block diagram. It is acceptable to use the one in the lab manual, provided it is modified as necessary to reflect what you implemented.



➢ The combination of the above 4 units gives rise to the above shown logic processor. Each of the units have a set of user-controlled switches along with connections between the units which pass on the information from one unit to another. The clock is common for all the units and it is generated by the ADALM 2000. The execution starts after parallel loading the values into the register. Once that is done, we press execute which then starts the FSM cycle and starts sending a signal to start the computation and routing the process. The input of the routing unit is sent in as the MSB into the registers and keeps on shifting right until we get the desired output as required by the user.

C. State Machine Diagram

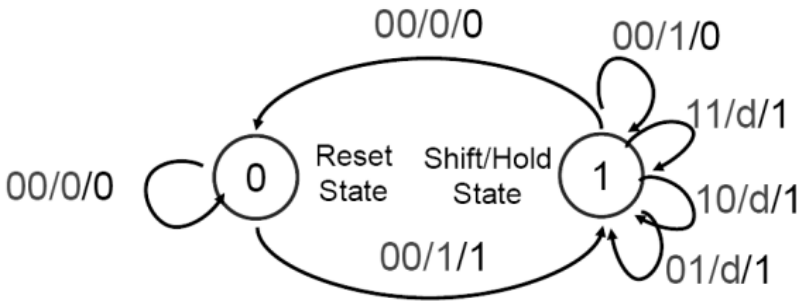


Figure 2: State Diagram

➢ We implemented the Mealy state machine for this lab. The biggest difference between the two state machines is that the outputs of the Moore machine depend solely on the current state, each serving a specific output configuration, while the outputs of the Mealy machine depend on a combination of the current state and the current inputs. From this point of view, it is apparent that the Mealy machine will be able to achieve the same level of control by using fewer states than what's required by the Moore machine, which also makes the circuit implementation a little bit easier. For our Mealy machine, the outputs depend on 'E', the execute switch, 'Q', the single bit reset/halt state, and 'C1C0', the counting bits that keep track of the number of shifts to be executed. The next state outputs are 'S', which is the register shift select, 'Q+', and 'C1+C0+', the next state counts.We decided to divide our states into two main sections: Shift and Hold. As we look into the truth table below we can see there are outputs called "d/D" as they all are don't care. We have don't cares in our lab report because once we have 1 the shifting should happen regardless.

Design steps taken and detailed circuit schematic diagram:

A. Written procedure:

- **Computation 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

➢ We started out by building the Computation unit. Our computation unit consisted of 1 NAND, 1 NOR, 1 XOR, 1 4-1 MUX, 1 HEX inverter. The main reason behind this design was because of the truth table that was provided to us. So instead of using an 8-1 MUX IC chip we thought we could combine the

output from a 4-1 MUX and XOR with F2 input from the user to get the required computation that needs to be performed. Our inputs to the 4-1 MUX were: A NAND B, ANOR B, A XNOR B, 0000 with select input pins as F1 and F0. The output of the MUX was XNORed with F2 select pin. If F2 was 1, then the values from the MUX would have passed on as output but if F2 was 0, then the values would be changed to A AND B, A OR B, A XOR B and 1111. The output of the computation unit was then passed forward along with values of A and B to the routing unit.

- **Routing unit:**

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

➢ To design the routing unit we decided to implement it using two 4:1 MUXes, one for each register. Our inputs to the Muxes were: A, B, F(A,B) and 0. The select inputs were R1 and R0, which were input manual switches controlled by the user. The design implementation was pretty standard as we couldn't have thought to design it any differently. The outputs from the muxes were feeded back into the registers as MSB on every clock cycle and were pushed in every clock cycle to replace the values that were initially stored in the registers.

- **Control unit:**

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

- The control unit was the toughest unit to implement in terms of design and debugging as we had to convert the truth table stated above into possible logic and put the outputs of that logic into flip flops to implement and store the future values. For this lab, the control unit worked by flipping an execute switch, which indicated that the cycle should start the desired computation, as well as the shifting of the 4 bits in each register to have the desired value in the desired place after the cycle was completed. From the truth table we were able to formulate the following K-maps:

EQ/C1C0	00	01	11	10
00	0	d	d	d
01	0	1	1	1
11	0	1	1	1
10	1	d	d	d

S = C0 + C1 + EQ'

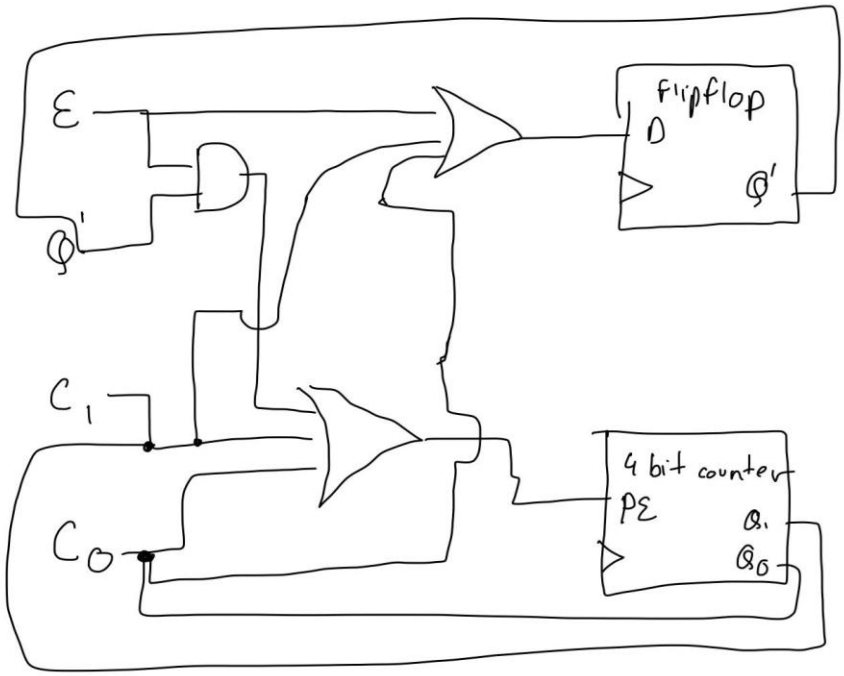
The logic above would decide which state the logic processor would stay in.

EQ/C1C0	00	01	11	10
00	0	D	D	D
01	0	1	1	1
11	1	1	1	1
10	1	D	D	D

Q+ = C0 + C1 + E

- With the use of these logic and flip flops we were able to compile and implement the FSM diagram. We can use these equations along with flip flops to implement the Mealy FSM. There were two possible implementations that we could have gone with, first the one where we use flip flops and the above logic or we could have implemented the logic for all the possible future states and implement the FSM. But we felt that could affect the counter of the and further increase the number of IC chips that we might have to use on the breadboard.So instead we decided to use the 4-bit counter that was provided to us. In this way, only one flip-flop and one counter would

be used. In addition, the amount of gates reduces since we would no longer need logic for the counter. We only needed the two LSB from the counter and the other two outputs were floating. Following is the schematic for the control un



The output from the logical unit is then combined with the input from the LOAD switches to feed in the select input of the registers A and B. Following are the truth tables for combination (same logic for both Load A / S and Load B / S):

A_S1			
	Load(A/B) / S	0	1
	0	0	0
	1	1	1

$S1_{A/B}$ = Load(A/B)

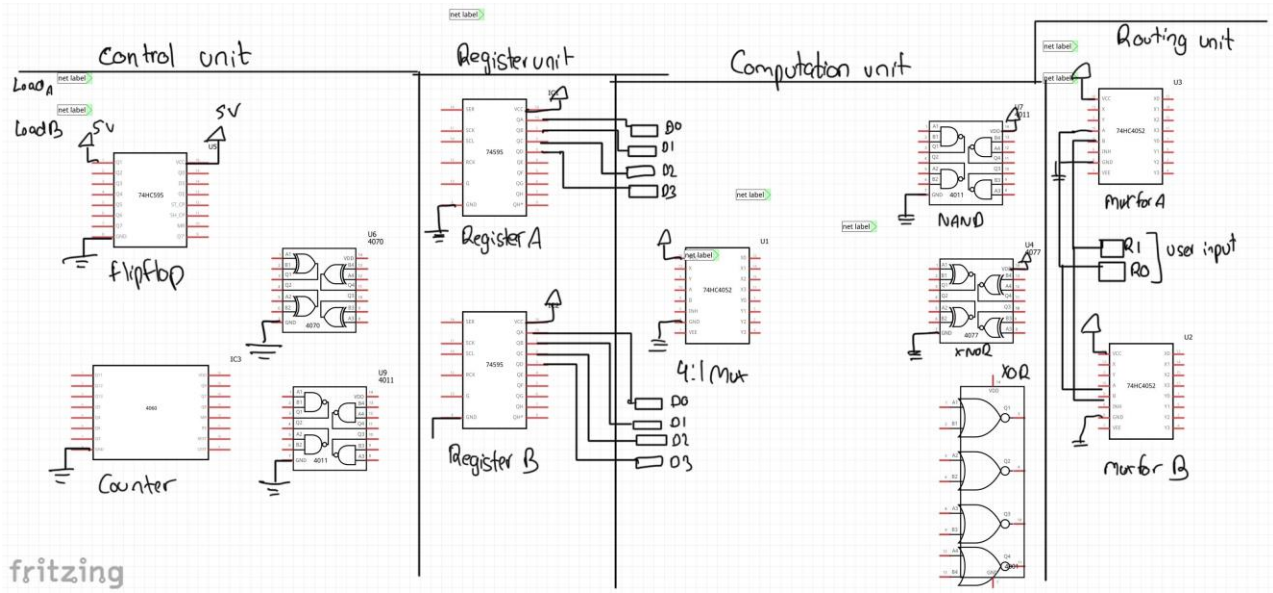
A_S0			
	Load(A/B) / S	0	1
	0	0	1
	1	1	1

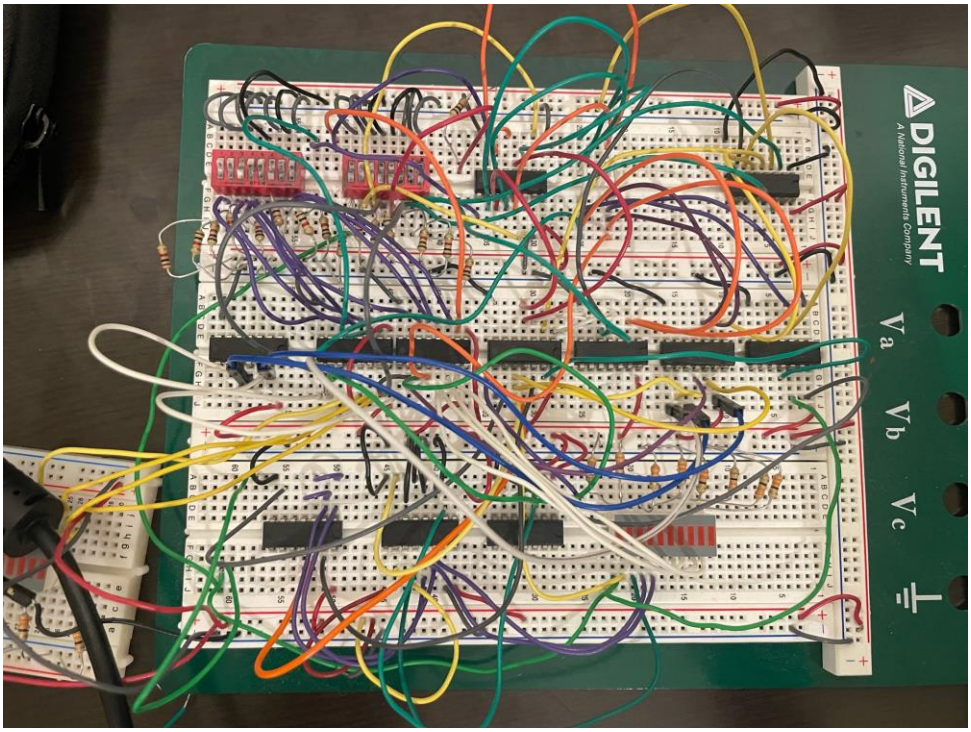
$S0_{A/B}$ = Load(A/B) + S

- **Register Unit:** The register unit contains two registers, A and B. Each register takes in 4-bit, and each register is capable of shifting the bits, left or right depending on the input of the DSL and DSR pins, as well as storing new values upon computation of the bits of A and B. It has pins D0-D3 which are determined manually through switches, the current values and Q0-Q3, which are the current register values. The current values are simply just connected to LEDs to show the shifts and values. The left side of the unit is connected to the control unit, and the right side is connected to the computation unit.

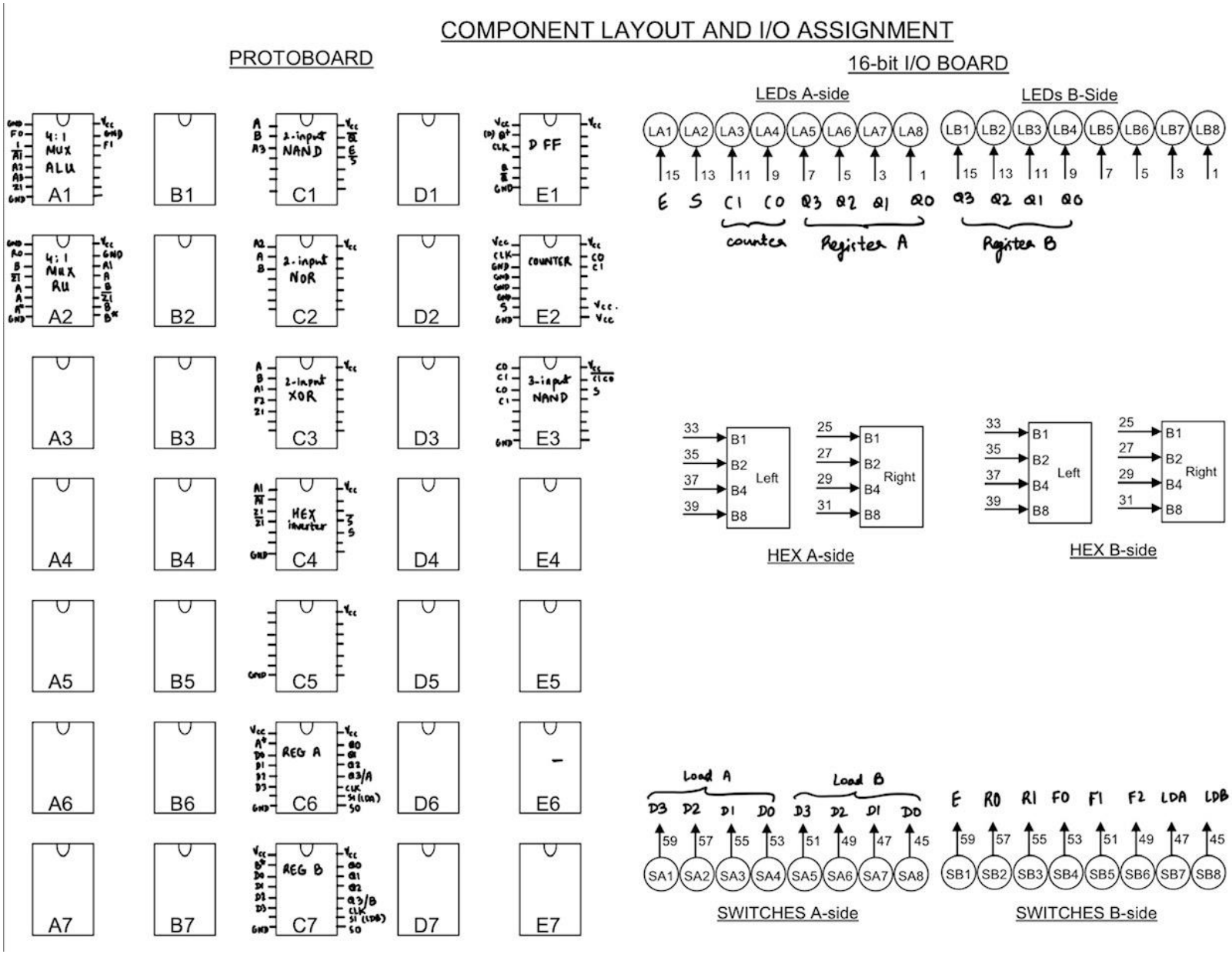
B. Fritzing gate schematic and chip layout:

- **Fritzing gate schematic alongside physical circuit**





- **Chip layout diagram:**



8-bit logic processor on an FPGA:

- In the second part of the lab, we were made to implement the lab we had built in 2.1 and expand that to 8-bit inputs for each register and do this all on the Quartus Prime software. We were able to make the following changes in the files that was provided to us and implement those logic onto a 16-bit processor. For ModelSim, there needed to be a top-level file and we made the processor the top-level file.

A. Summary of all .SV modules and the changes made to extend the processor to 8-bits:

- ➢ **Synchroniser.sv :** File's purpose is to create signals for FPGA that are synchronous. No changes to the file.
- ➢ **Router.sv:** The specifics of the routing unit were put into this file. No changes were made to this file.
- ➢ **Register_unit.sv:** We had to expand the register unit capacity from 4 bits to 8 bits as we were dealing with 8 inputs instead of the original 4 bits. Further, A and B both are 8-bit outputs now.
- ➢ **Reg_4.sv:** The 'Data_out' assignments were changed to accommodate 8 bits. So they were changed to 2 hexadecimal numbers. The number of bits for D and Data_out were both changed from 4 to 8 in order for the processor to be able to work with the desired number of bits. We also changed the size of 'Data_out' for shifting our bits.
- ➢ **Processor.sv:** For this file we needed to assign (hardcode) values to F and R (the select bits) for when we use the FPGA because there are not enough switches on the board for these selects. Further, 'Din', 'Aval' and 'Bval' were changed to accommodate 8 bits.
- ➢ **Hex_driver.sv:** No changes were made to this file.
- ➢ **Control.sv:** We had to change the number of states in this file by adding 4 new states to accommodate the FSM.
- ➢ **Compute.sv:** No changes were made in this file.

B. RTL Viewer from Quartus Prime (netlist viewer):

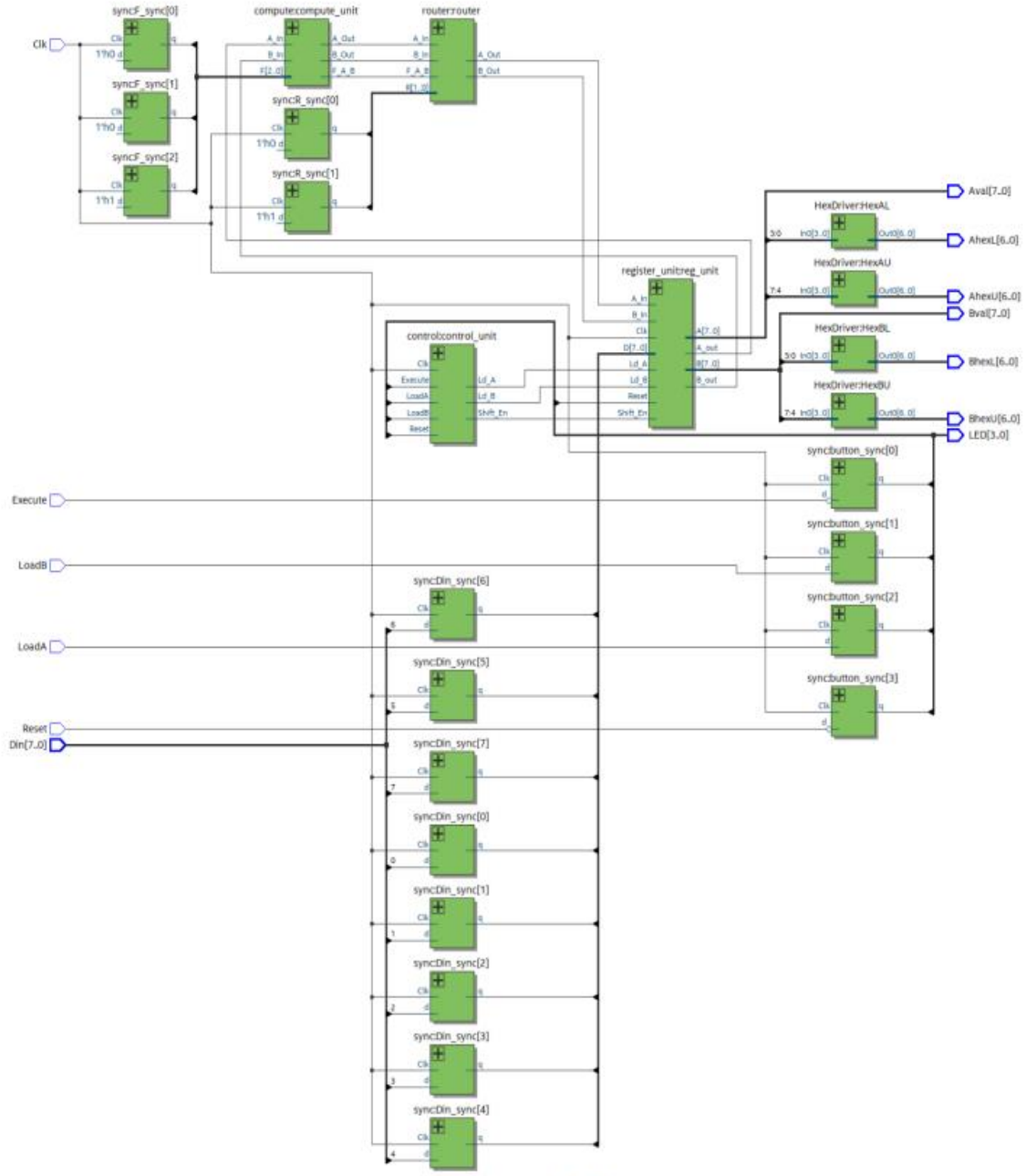
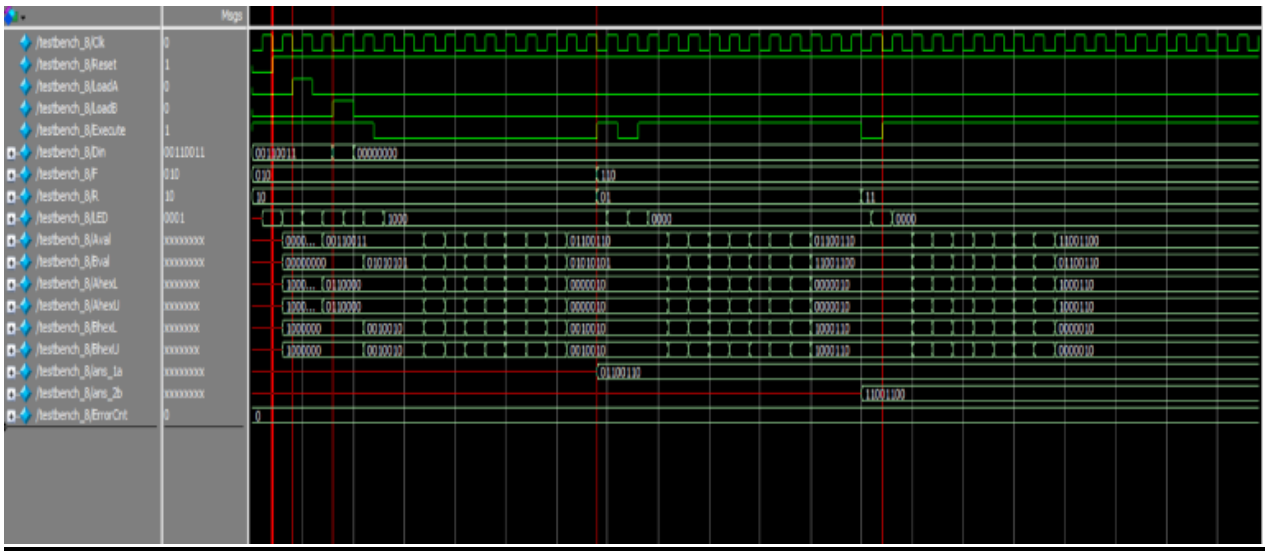


Figure 8: Top-level RTL viewer

C. Simulation Waveform:

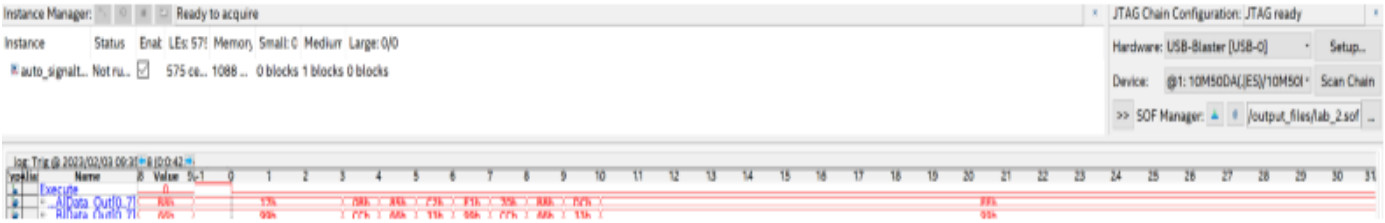


20ns: Both Registers A and B reset to 0
40ns: Load a changes to 1 and the value of 00110011 is loaded into A
80ns: Load B changes to 1 and the value 01010101 is loaded into B
340ns: Execute goes to 1 and as F is 110 it performs A XNOR B and the result is stored in A as R= 10
620ns: Execute goes to 1 again and the operations takes place again but now the values get swapped as R = 11

D. SignalTap ILA Trace:

The following steps were followed to generate a SignalTap:

1. Because there aren't enough switches on the FPGA to adjust these select values, step 1 is to hardcode the values for F and R in the processor.sv file. For example, if we want to perform the XOR and we have F = 3'b010 and R = 2'b10, indicating that the result of the calculation should be kept in register A and that register B's value ought to remain constant after execution.
2. The second step is to select Tools -> SignalTap Logic Analyzer. Once there, we must set up the clock before adding the three nodes reg A|Data Out[0:7], reg B|Data Out[0:7], and execute to our analyzer.
3. Next, we programmed the FPGA and clicked on “Run analysis” and waited till there was a message on the signal tap that said “waiting for trigger”. Once the execute was pressed we could see the intermediate and final values stored in each register:



E. Bugs and error:

- Firstly, as we were dealing with the software for the first time, we faced issues understanding the interface and the code. Further, this was our first time being introduced to SystemVerilog so it took us time to understand the concept and flow of the code that was written to us. We also faced issues while associated with interconnections of wires and the connections. We also faced an issue while uploading the code onto our FPGA as it would show invalid ports every time. Lastly, when using ModelSim and testbenches we had an error that said “error loading design”. This was because the name of the testbench file was not matching the name of the module. But we were able to fix the code and complete the lab in sufficient time.

Post-Lab Questions:

1) 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 do this in the most simple way by using a two input XOR gate. For example if it has two inputs, A and B, and a single output, Y. Depending on the state of input A, the output Y can either be the inverse of input B or be the same as input B. By setting input A to either "high" or "low", you can determine whether or not the signal at B will be inverted.This is very useful in our lab because it helps reduce the need of a hex inverter and further reduce the number of chips that are required.

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

- The circuit is more organised thanks to a modular architecture. Through physical separation of components on the breadboard, it becomes simpler to track the connections and assess input and output signals. We are able to test each module at each stage of construction and this makes it simpler to test and troubleshoot the circuit. It will be very challenging to locate the problem if we build the circuit from

the ground up. It also aids in circuit construction. Because each group member can focus on a different module, the group can operate more effectively.

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

➢ The difference between the two state machines is that the outputs of the Moore machine depend on the current state, while the outputs of the Mealy machine depend on both the current state and the current inputs. A Mealy machine can achieve the same output using fewer states than a Moore machine which simplifies the circuit implementation. A Mealy machine also uses lesser logic chips which makes it more cost and space effective.

4) 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?

➢ Both Modelsim and SignalTap accept inputs for registers A and B, perform a logical operation using the F select bits, and then return the appropriate values to registers A and B in accordance with the selects to the routing unit. If you want to be able to view all intermediate values and every processor input (such as clock, load A and load B, F selects, R selects, etc.), using Modelsim might be more advantageous. In other words, SignalTap only displays the input, output, and values while shifting, whereas Modelsim shows all the inputs and outputs, providing more information.

Conclusion:

In this lab, we built and designed a bitwise serial logic processor. The two inputs A and B were stored in separate shift registers and using the computation unit, we were able to compute eight different functions depending on what the values for F2-F0 were.We were able to expand this functionality to 8 bits in the Quartus prime and we saw how much easier it is to debug and build circuits on the software.All these components work in concert to store, display, and operate upon the values, ensuring a high degree of accuracy and precision in the computation process.