ECE 385

Spring 2020

Experiment #3

Logic Processor

Freddy Zhang and Mehul Dugar

Section: ABO (Thu. 3PM)

Gene Shiue

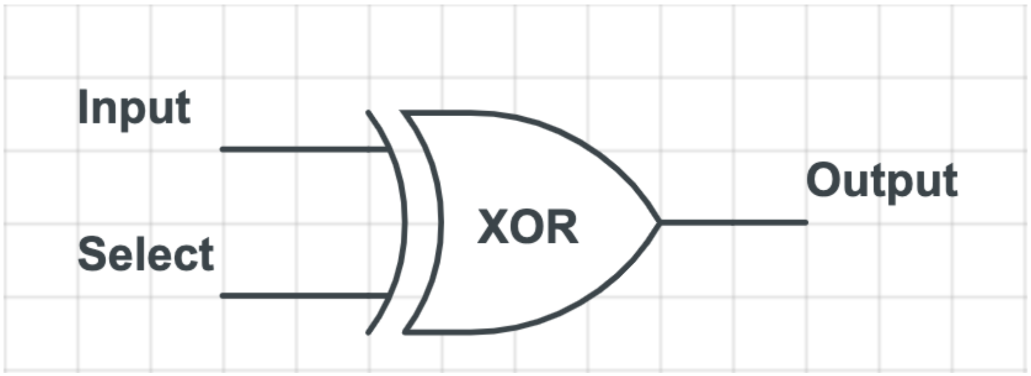
Introduction:

Our implementation of the circuit that behaves like a processor that is capable of working on 4-bits of data. The processor circuit performs AND, OR, XOR and their inverse functions (NAND, NOR, XNOR) bitwise on data of four bits.

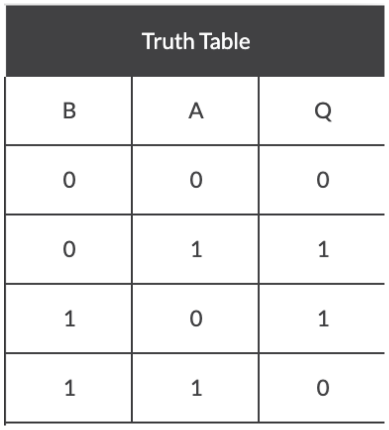
Answers to Prelab Questions:

a. 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). Sketch your circuit.

   The simplest circuit that can optionally invert a signal would consist of a XOR gate. It would have two inputs, the signal to be transmitted and the select signal where the select signal is 1 when an inverted output is required.



   With the following truth table it is clear that this is exactly the function we require.



Truth Table		
B	A	Q
0	0	0
0	1	1
1	0	1
1	1	0

   Here, Q is the output, A is the select signal and B is the input signal.

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

    A modular design such as the one described above helps in reducing the amount of logic required to implement functions and this in-turn reduces the time taken to build the element of the circuit that performs the inverse of the function. It reduces the development time because it saves us the time needed to implement all the functions individually. Additionally, because the number of functions implemented are halved, we only need to test half the number of things versus having to test all the separate implementations in the case where we do not use such a modular design.

Operation of Logic Processor:

Prior to any computation, there must be data loaded into the registers (A and B). For this purpose, we make use of the four Din switches that hold the value of the data to be stored in either of the registers. With our design, we make use of the Din switches to input data into both the registers, thus to choose between the two registers we have two more switches, Load A and Load B which when switched on would allow data in Din switches to be stored in their corresponding registers.

Moving onto the computation aspect, the user is given three input switches, F2, F1 and F0, whose configuration is used to determine which function to perform. These functions are performed only when Execute is switched on. The functions are given by the following table -

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

After the operation is performed, the user is given an option to store the result in various ways. This choice is implemented by using two switches R1 and R0 whose functions are given in the following table -

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

Here, the router output of F refers to the result obtained after performing the operation and A and B correspond to the data stored in the respective registers. Note that when R1 and R0 both are high, the values stored in each register are swapped.

Written Description:

Control Unit -
The control unit consists of a four-bit shift register, some logic gates (NAND, NOT), and the LoadA, LoadB, and Execute switches. LoadA and LoadB are concerned with parallel loading the shift registers so there are some logic gates that go through LoadA and LoadB to S1 and S0 of the storage shift registers, nothing too exciting there. The control unit relies on a finite state machine to operate. When the execute switch is switched to high, the shift register in the control unit will start shifting to keep track of the number of shifts. The circuit was set up so that when the shift register in the control unit starts shifting, the two storage shift registers in the register unit will start shifting as well. Likewise, when the shift register in the control unit stops shifting, the shift registers in the register unit will stop shifting. While the shift registers are shifting, the values of A and B will be going through the logic gates and MUXs in the computation unit. After four shifts have passed, the shift register in the control unit will contain "1111" which means the all of the registers will stop shifting. A more elaborate explanation of the state machine will be available later in the lab report.

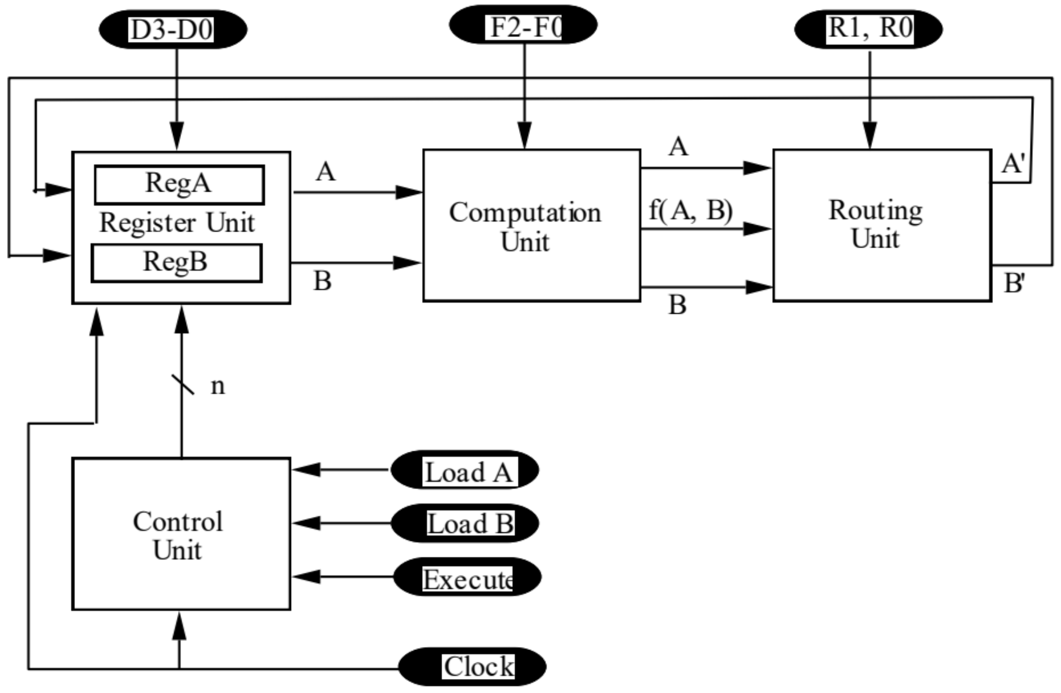
Register Unit -
The register unit is simply two four-bit shift registers. One will hold the values of A while the other register will hold the values of B. When the registers will shift is determined by the control

unit and the right input serial bit is determined by the routing unit. The parallel load inputs are four switches and the parallel load is controlled by the control unit.

Computation Unit -
The computation unit consists of a 4:1 MUX, a 2:1 MUX, and a few logic gates (NAND, NOR, XOR, NOT). The last bit of the shift registers go into the logic gates to make NAND, NOR, and XNOR. The output of those logic gates go into the corresponding 4:1 MUX inputs based on the function selection input table. After that the output goes into the 1's bit of the 2:1 MUX and the inversion of the output goes into the 0's bit of the 2:1 MUX. The select bit of the 2:1 MUX is the F2 switch. The reason why this implementation works is because F2 acts like an inverter where the logic of 110 is just 010 inverted. This applies to all of the function logic.

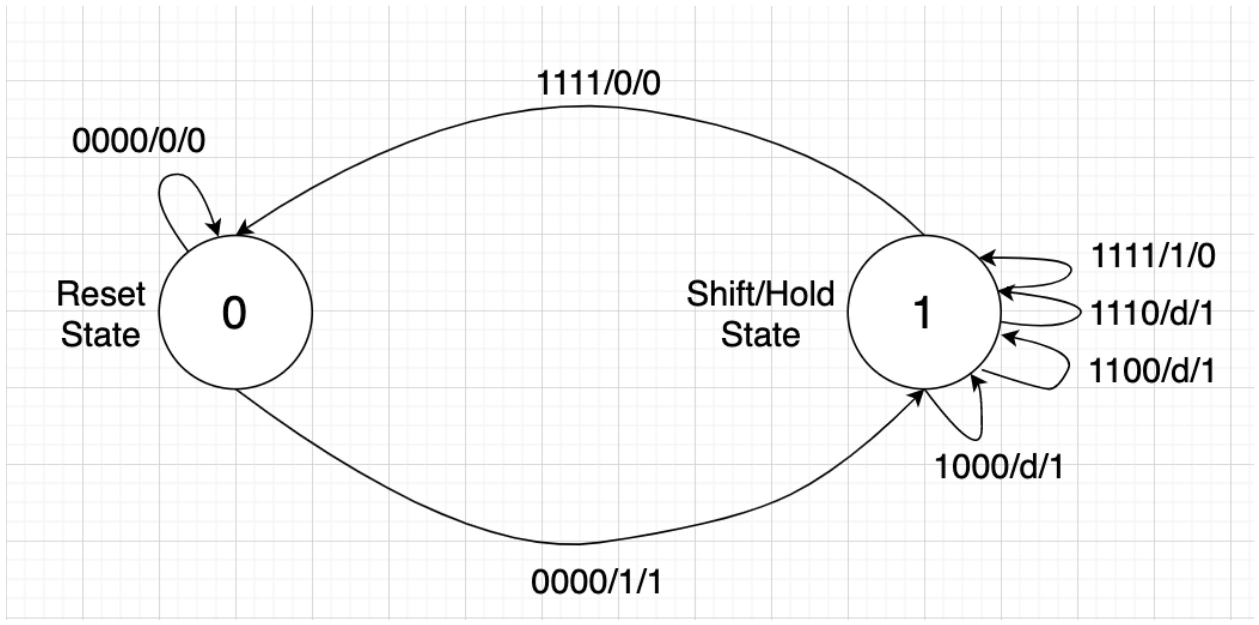
Routing Unit -
The routing unit determines what bits will be stored in the shift registers. It uses a 4:1 MUX to get the options of the router output ready and the R1 and R0 switches are the select bits that determine the output of the MUX. The output of the MUX is inserted into the right serial input bit of the shift registers.

Block Diagram:



State Machine Diagram:

We made use of a Mealy state machine as was recommended in class. This required less states and was easier to implement.



In our implementation of the Mealy state machine, the least significant bit represents the shift bit (determines whether the data is being shifted), the second least significant bit is representative of the execute signal, and the next 4 bits are representative of the shift register states. We made use of the shift register with the help of a variation of hot one encoding to implement a simpler circuit to perform all the operations required.

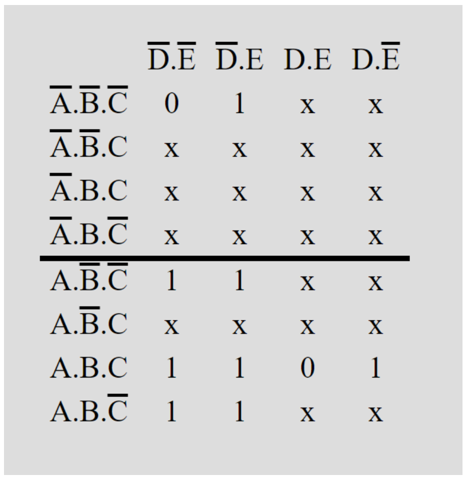
Design Steps:

While designing the Mealy state machine, we tried using a counter and a comparator. We were not very confident in the design, but we started implementing it because it was the first thing we thought of. Once we started writing the K-maps and logic to implement it on the breadboard, we decided to scrap it because it was too complicated.

Next, we tried using the one-hot method to implement the state machine using the shift registers. We designed the states where there would be one '1' at all times. This ended up failing because there were not enough states to shift through the whole register.

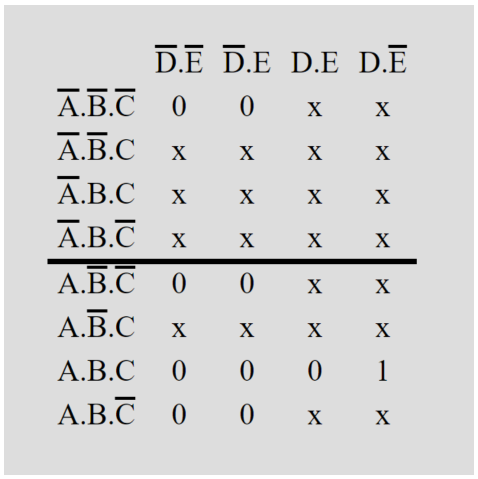
The final iteration of our state machine was based on the one-hot method. The default state had the register be 0000, and the right serial input bit was connected to power so that it would always be a 1. When the execute switch was switched to high, the S1 and S0 bits would change so that the registers would shift right. Because of the right serial input bit always being high, it would always input a 1 into the shift register. The shift registers would keep shifting until all of the values in the shift register in the control unit were 1's. Then, it'll wait until the execute switch is switched to low for the circuit to go back to the default stage and the parallel load will reset the register to 0000. The truth table below shows when the shift registers will shift. The output, S1 and S0, correspond to the S1 and S0 of the shift registers. The states not shown are "don't cares" that should not occur during the operation of the circuit.

Q_a	Q_b	Q_c	Q_d	Execute	S1	S0
0	0	0	0	0	0	0
0	0	0	0	1	0	1
1	0	0	0	x	0	1
1	1	0	0	x	0	1
1	1	1	0	x	0	1
1	1	1	1	1	0	0
1	1	1	1	0	1	1



	$\overline{D}.\overline{E}$	$\overline{D}.E$	$D.E$	$D.\overline{E}$
$\overline{A}.\overline{B}.\overline{C}$	0	1	x	x
$\overline{A}.\overline{B}.C$	x	x	x	x
$\overline{A}.B.C$	x	x	x	x
$\overline{A}.B.\overline{C}$	x	x	x	x
$A.\overline{B}.\overline{C}$	1	1	x	x
$A.\overline{B}.C$	x	x	x	x
$A.B.C$	1	1	0	1
$A.B.\overline{C}$	1	1	x	x

Truth table for S0



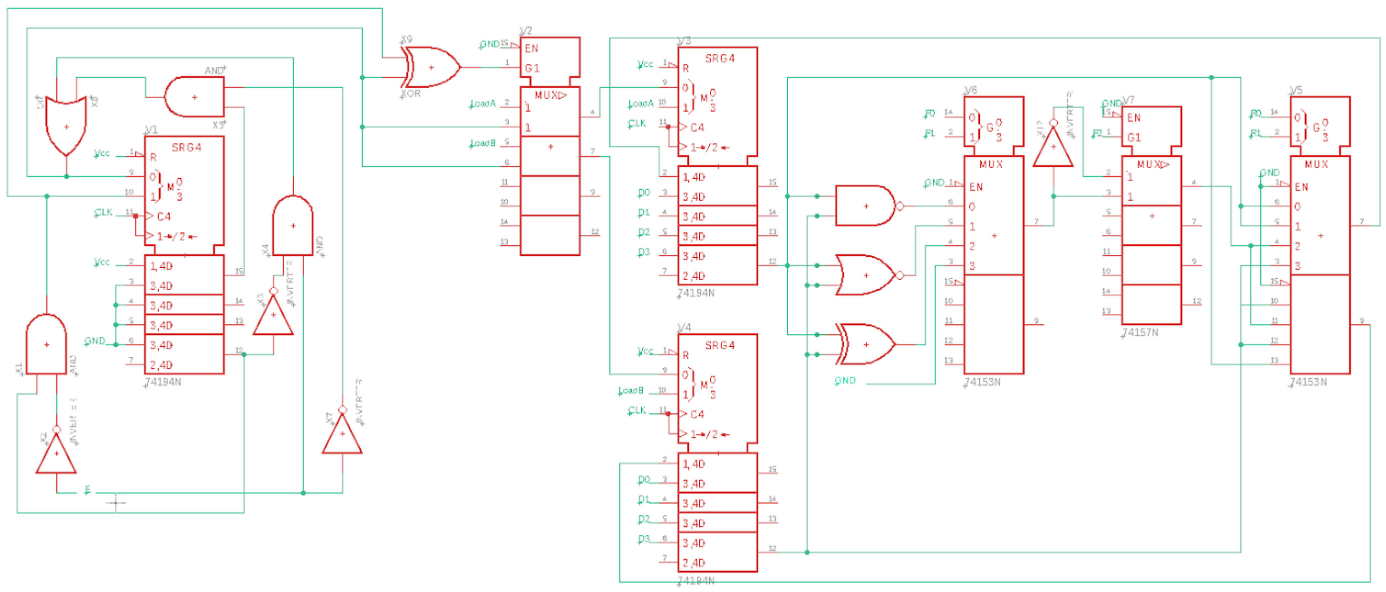
	$\overline{D}.\overline{E}$	$\overline{D}.E$	$D.E$	$D.\overline{E}$
$\overline{A}.\overline{B}.\overline{C}$	0	0	x	x
$\overline{A}.\overline{B}.C$	x	x	x	x
$\overline{A}.B.C$	x	x	x	x
$\overline{A}.B.\overline{C}$	x	x	x	x
$A.\overline{B}.\overline{C}$	0	0	x	x
$A.\overline{B}.C$	x	x	x	x
$A.B.C$	0	0	0	1
$A.B.\overline{C}$	0	0	x	x

Truth Table for S1

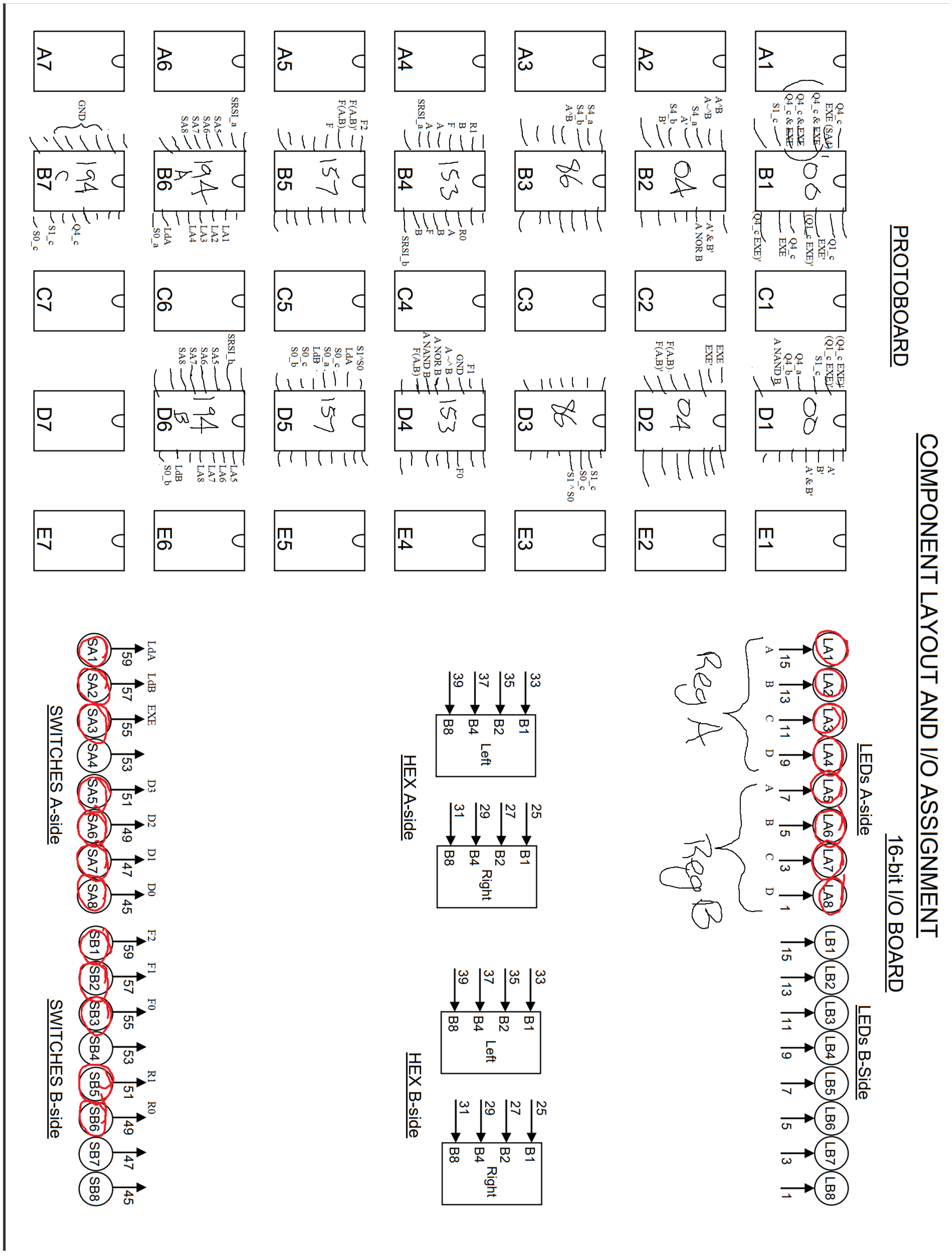
S1 = Q_d & E’

S0 = (Q_d’ & E) + (Q_a & E’)

Detailed Circuit Schematic:



Layout Sheet:



Bugs Encountered:
Most of the bugs encountered came from putting the wires in the wrong pin on the breadboard. Because of the large number of wires used and the small spaces between each pin, it was very easy to accidentally put a wire in the wrong pin because the view of the pins were bad. In order to prevent wires from being inserted into wrong pins, we just had to be extra careful when putting in the wires and have better foresight of the design of our circuit so that the wires wouldn't interfere with the other pins as much.

Answers to Post-Lab questions:

a. Document changes to your design and correct your Pre-Lab write-up, explaining any difficulties you had in debugging your circuit. Outline how the modular approach proposed in the pre-lab help you isolate design and wiring faults, be specific and give examples from your actual lab experience.

   Our original design involved a counter instead of a shift register, but while designing the logic for the control unit we found that it was significantly more complicated than the hot one encoding approach we ended up following.

   Debugging was a pain primarily because the number of wires we had used for the connections between all the components. However the modular approach described in the manual helped in reducing the confusion while debugging as it allowed us to split up the circuit into smaller chunks. This helped in debugging because the module of each output was easy to calculate so if the circuit had an error, we were able to easily and quickly narrow down what component was causing the error based on the expected outputs of each module.

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

   Our state machine was designed with the default state being the reset state, which the machine would hold until the execute signal was flipped. Once the execute switch was flipped, the state machine would move into the shift state until it was done processing 4 bits. Then it would loop back into the hold state until the execute switch was flipped back, after which it would return to the reset state.

   Mealy state machines depend on both the current state as well as inputs versus the Moore machines which only depend on the current state of the machine. Moore machines are synchronised with the clock whereas Mealy machines need to be synchronised with the clock and their outputs need to be read just before the clock edge.

Conclusion:

In this lab, we implemented our own version of the Mealy machine described in the lab manual. Our implementation made use of the simple one hot encoding method (or rather a variation of it). The errors we ran into was that the counter logic was too complicated to implement so we ended up using the extra serial shift register we had to implement the states. While implementing the shift register for the one-hot encoding method we found that our first attempt could only cycle through three states instead of four and we had to thus rethink our design to get four states. We got familiar with serial logic and modularization of the circuits in this lab as well.