ECE 385

Fall 2022

Experiment #2

A Logic Processor

Ravi Thakkar, Siraj Khogeer

TA: Gavin Wu

9-10-2022

Introduction

In the first part of this lab, we created a bit-serial logic operation processor on the breadboard using TTL ICs. The design takes in two four-bit registers A and B, and executes one of eight bitwise logic operations (eg. AND, OR, XOR), then routes the inputs/outputs in four different ways. A control unit was designed to drive the compute, routing, and register units. In the second part of this lab, we were given SystemVerilog code for the same four bit-serial processor and we were tasked to convert the processor into an eight bit one. Both labs are synchronous.

Operation of Logic Processor

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

To load data into the A and B registers, the user must first have both Load A, and Load B on low. Then set the desired data inputs to the register. To load a specific register, flip the Load A, or Load B to load the data into that register. Then turn the load switch off so it doesn't keep loading during further operation.

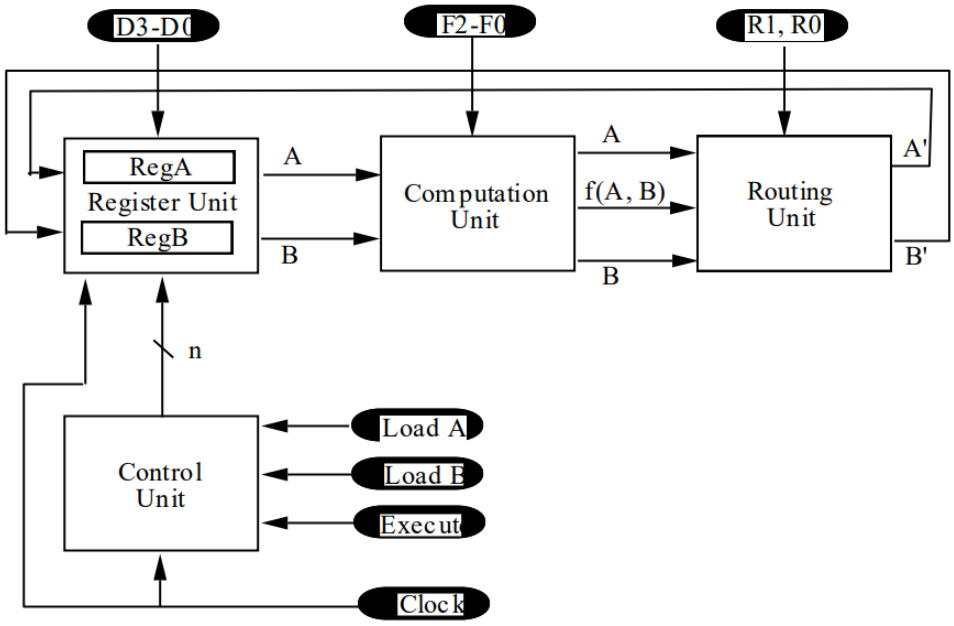
b. Describe the sequence of switches the user must flip to initiate a computation and routing operation.

Before executing the processor, set the F2F1F0, and R1R0, switches according to the desired operations and routing conditions. These can be seen in the figure above. After the routing and computation conditions are set, make sure the load switches are off then press the execute button to run the processor.

Written Description of Circuit

a. Written description: describe in words each block in the high-level diagram (a short paragraph for at least the register unit, computation unit, routing unit, and control unit).

Shown below is the block diagram for the four bit-serial logic operation processor.



Register Unit:

The register unit consists of two 4-bit shift registers that hold the data of A and B. The unit has three modes of operation; parallel load, hold, and right shift. When the Load A or B switches are high the registers parallel load the data in (D3-D0) until the switches are low again. If the load switches are low and the Shift Enable (S0) is also low, then the registers hold their value. Whenever the Shift Enable is high the unit feeds its output into the computation unit, then right shifts the bits in the registers. At the same time the router gives two output values which are feeded into the DSR pin of each register which dictates what the left-most bit is, essentially storing the output of the routing unit serially. The shifting occurs bit-by-bit at each rising clock pulse. To implement the register unit we used the CD74HC194E Bidirectional shift registers and we had to use the datasheet for this IC to understand how to load A and B and shift bits right. The truth table below shows how the shift right and load actions are executed through the S1, S0 and DSR inputs.

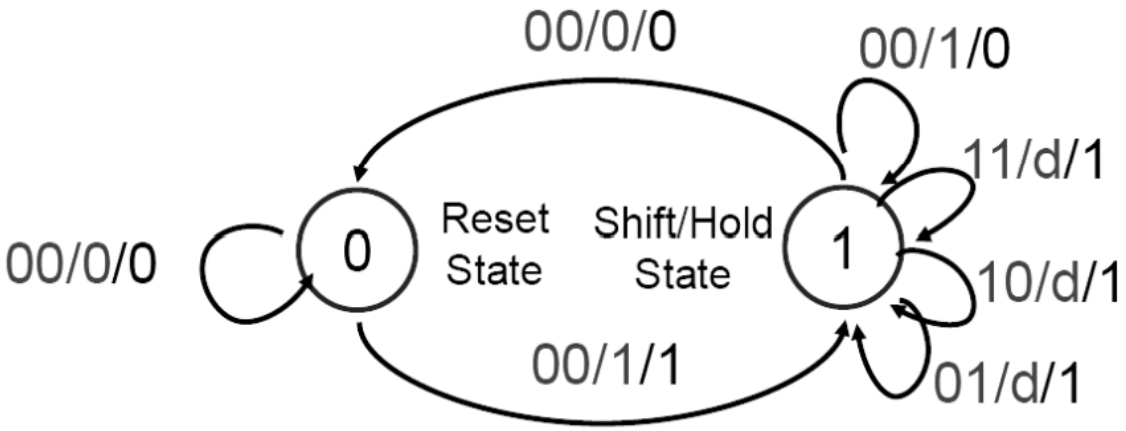
TRUTH TABLE

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

H = High Voltage Level,
h = High Voltage Level One Set-up Time Prior To The Low to High Clock Transition,
L = Low Voltage Level,
l = Low Voltage Level One Set-up Time Prior to the Low to High Clock Transition,
d_n (q_n) = Lower Case Letters Indicate the State of the Referenced Input (or output) One Set-up Time Prior to the Low To High Clock Transition,
X = Don't Care,
↑ = Transition from Low to High Level

Control Unit: When triggered by an execute button allowing the execute signal to go high, the control unit would begin the computation cycle. Once Loads A and B were inserted into the shift registers and the desired computation and routing select bits (F2-F0 and R1,R0) were set, the control unit would allow the register unit to shift exactly four times. After four clock pulses, the register unit halts until the execute button is pressed and the execute signal goes high again. For the control unit, we implemented a state machine to keep track of the shifts and for the halting of the shifting of the registers. The input to the control unit is execute, Load A, and Load B. Logic was used to supply the two outputs, Shift Enable (S0), and load enable. The load enable was done using logic to set S1S0 to 1 to load the specified register. This was only done due to issues with floating and shorted input/outputs, this is not necessary in theory. The load operation takes priority over the execution. The control unit contains three main parts, a counter, a state register, and logic to supply the outputs.

State Machine Diagram:



The State Machine Diagram shown above is the one we implemented, which is a Mealy machine. We went with the Mealy machine because it uses two states, and that is due to the fact that the output of the Mealy Machine depends on the current state and the input. This is different from the Moore machine whose outputs only depend on the current state.

Computation Unit: The computation unit performs bitwise operations on each bit from the A and B registers to generate an output F(A,B). The computation unit also takes in three selection bits F2-F0 that correspond to the desired operation to be performed on the set of four bits.

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

Since all of the logic in the computation unit is combinational, we decided to implement the unit using NAND and NOR gates. We also recognized that the F2 bit acts to negate the operation indicated by F1 and F0. For example, if F1 and F0 are 0, then switching F2 from 0 to 1 changes the indicated operation to NAND, which is just the negated form of AND. Using this idea allowed us to simplify the K-map we used from 5 inputs to 4. The last four operations were implemented using a NAND, NOR, and XOR chip. These outputs were then fed into a mux that

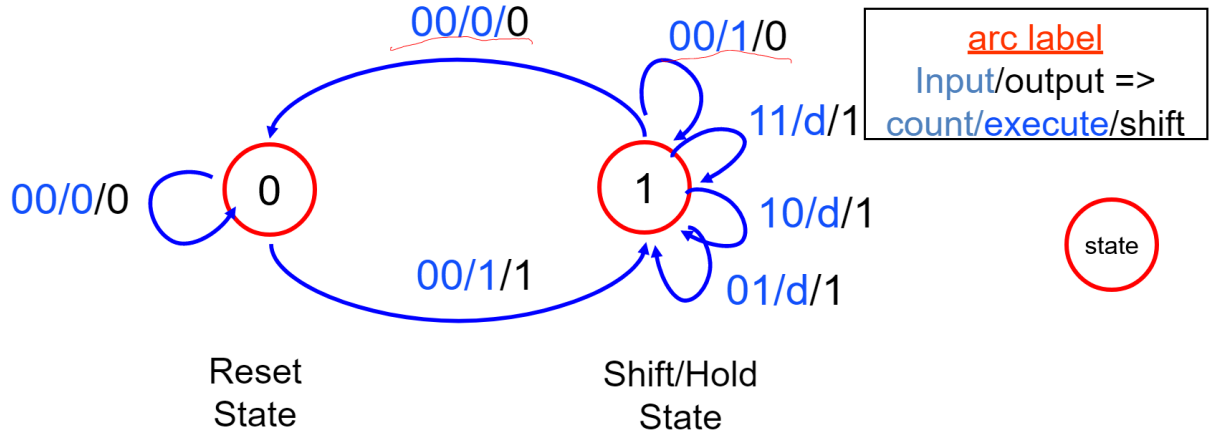
had the selects F1 and F0 to choose which operation. Finally the output is fed into an XOR gate with F(A,B) and F2, which inverts F(A,B) if F2 is one. The computation input and outputs are 1 bit only which come from the register unit serially at each clock pulse.

Routing Unit: The routing unit takes in bits from A, B, and F(A,B) and routes these bits back to the shift register, according to the two routing select bits R1-R0.

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 implement the routing unit, we used two 4:1 muxes for each shift register. The routing select bits would select which of the four routing options would be executed: If both R1 and R0 were 0 then the shift registers would not change their inputs, but if they were both 1 then the muxes would load A into B and B into A. If R1 was 0 and R2 was 1, the output of the computation unit would be sent to Shift Register B, and if R1 was 1 and R2 was 0, the output of the computation unit would be sent to Shift Register A. This was directly implemented using two muxes and wiring the input/outputs following the truth table.

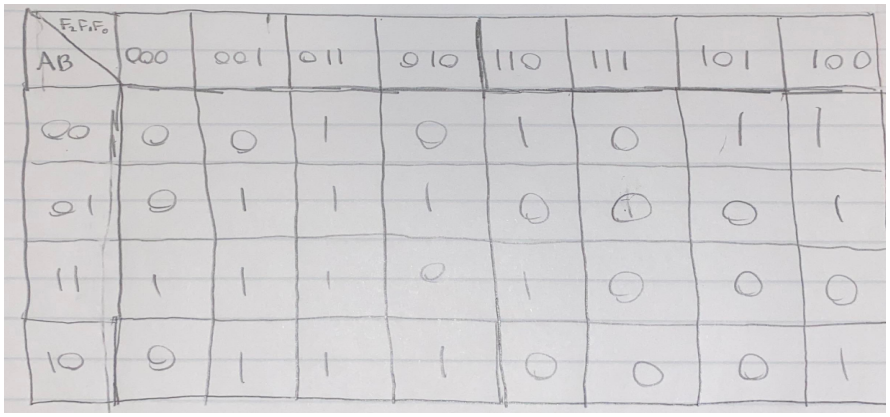
State Machine Diagram: Mealy Machine (Annotated from Lecture)



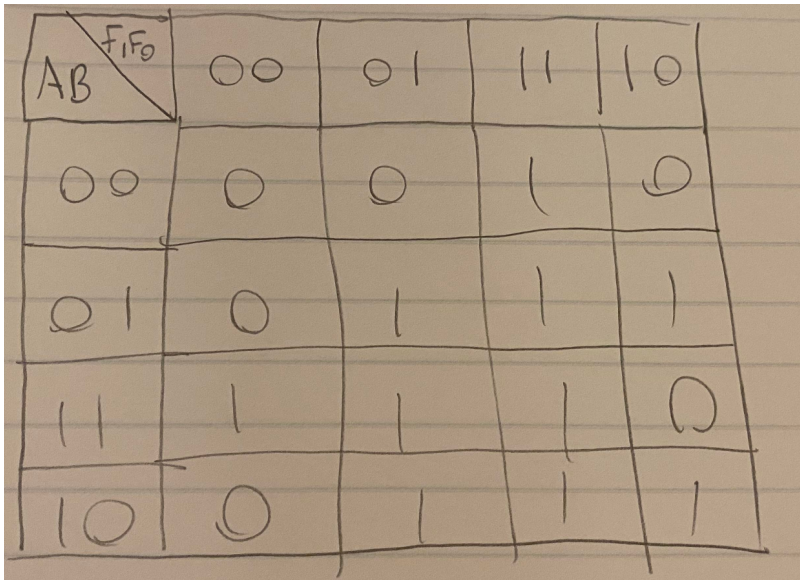
Design Steps/Circuit Schematic

Part 1:

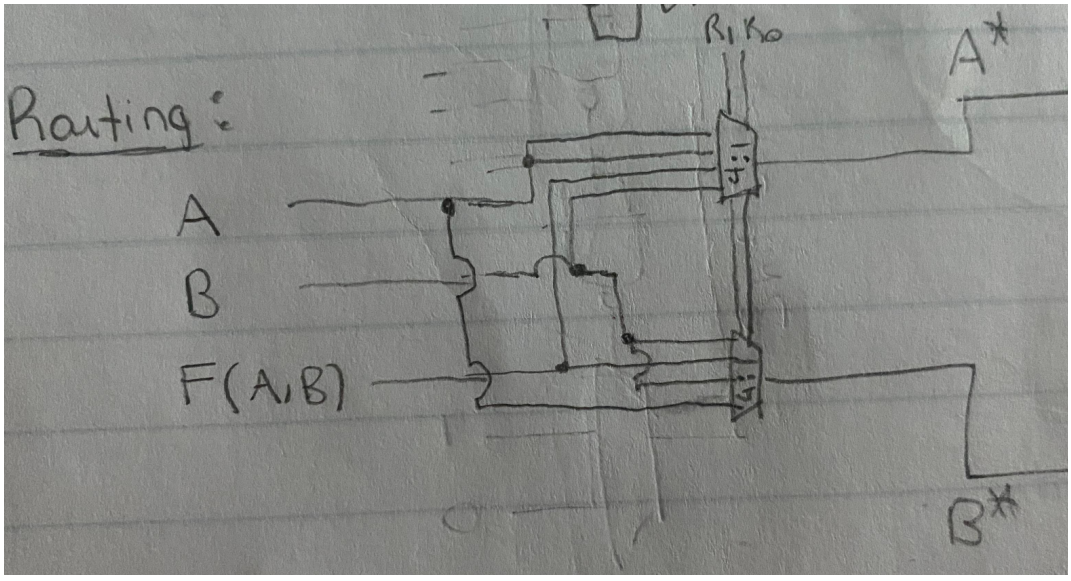
1. The first unit we designed was the computational unit. We did this because it was the easiest to implement and debug. We decided to implement this using a NAND, NOR, XOR, and a 4 to 1 mux for the unit. We found out that the F2 select bit would only invert the bit from the first four computations, and this is the same as a XOR operation. When we found this we could then implement the last four operations (NAND, XNOR, NOR 0000) and use one or two logic gates for each operation. We then fed the output into a 4 to 1 mux according to the truth table specified. Finally we inverted the result and fed it into an XOR gate with F2. Whenever F2 was 1, it would flip the intermittent result. Initially we wanted to implement the compute unit using only NAND and NOR gates but we decided not two because it was more complicated and was harder to debug. We made k-maps and found min-terms first but didn't need them in the end.
    a. Below is the first iteration of the K-map that we were using to find minterms for our combinational logic.



    b. Using the idea of the F2 bit negating operations when pulled high, we were able to simplify the K-Map and use a NAND gate with F2 and the output of the boolean function represented by the following K-Map:



2. Next, we constructed the routing unit because we anticipated that it would be easier to debug the routing unit with the computation unit already constructed. The routing unit takes input bits from A, B, and the output from the computational unit, and routes these inputs as selected by the routing select bits. We decided to use two 4:1 muxes using one SN74HC153N IC. R1 and R0 were the two select bits for the muxes. The way that we wired the muxes was according to the table of routings shown in the written description of the circuit. Below shows how we wired the routing unit. The routing unit was very straight forward and there weren't other implementations we thought of.



3. We worked on the register unit next because it was the only part left that needed to be connected to the compute and routing units. The registers did not have much logic. The chips used were two “”. The register unit does two things, hold and shift. After looking

at the data sheet for some time, we found the pins that we needed to use as inputs. We found that D3-D0 inputs from the switches needed to be connected to d3-d0. Other pins were connected such as the clear and unused pins such as DSL. We found that there were three inputs that controlled the behavior of the registers. S1S0, and DSR. DSR was the easiest which basically says what the left-most bit after the shift is. We connected the output of the routing unit directly to the DSR pin of each register. The harder part was figuring out S1S0. In our case, when S1S0 was 1, the register would load d3-d0, and when they were both 0, the register would hold its value. Finally, if S1 was 0, and S0 was 1 then the register would right shift at each clock pulse. Knowing this, we connected S1 to the Load A or Load B switches depending on the register. S0 then needed to be connected to Shift Enable, but because we hadn't built the control unit yet, we left it at 1. This meant that when Shift Enable and Load was 0, the register would hold, when they were both 1, it would load, and when only Shift Enable was 1, it would right shift with data dictated by DSR which was connected to the router output. We decided to implement it this way because it seemed the most intuitive and simple. An issue we faced was how to connect the S0 pin, as it would have two inputs, Shift Enable and Load A/B. We solved this by using an OR gate (NOR-NOR) to isolate Shift Enable and Load A/B.

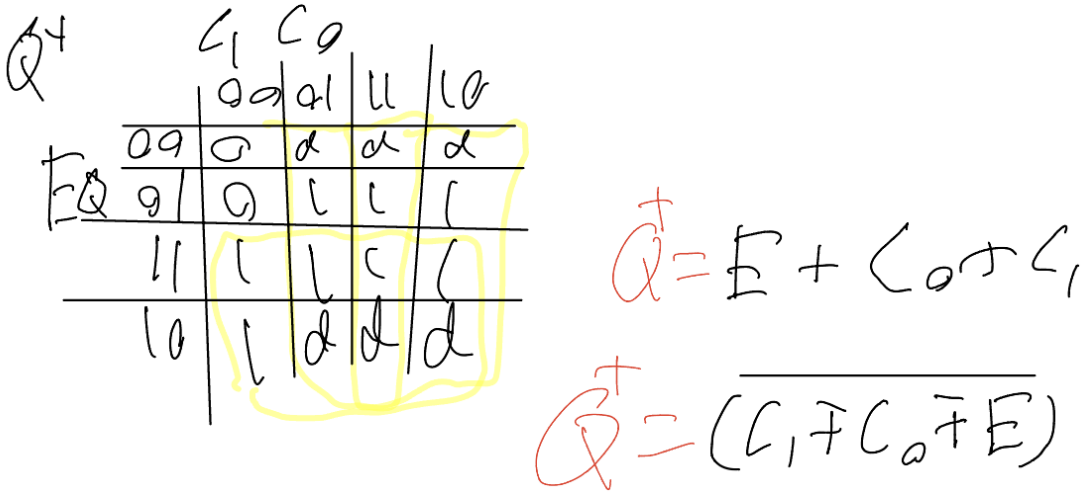
TRUTH TABLE

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

H = High Voltage Level,
h = High Voltage Level One Set-up Time Prior To The Low to High Clock Transition,
L = Low Voltage Level,
l = Low Voltage Level One Set-up Time Prior to the Low to High Clock Transition,
d_n (q_n) = Lower Case Letters Indicate the State of the Referenced Input (or output) One Set-up Time Prior to the Low To High Clock Transition,
X = Don't Care,
↑ = Transition from Low to High Level

4. The final unit we designed was the control unit. The design and truth table was mainly taken from lectures. We used the truth table and Mealy state machine to design and build the unit. These can be seen below. The control unit has one input, and one output. The input is the execute button, and the output is the Shift Enable. The control unit also has another restriction, it must only shift for 4 clock cycles. We then used the truth table below, which was derived from the state diagram, to design the circuit. The easiest thing

to implement was the state register Q. Using the truth table we found that Q+ = E + C1 + C0. We then used the truth table to make a k-map to find the SOP for Shift Enable (S0). We found S = EQ' + C1 + C0. Finally, for C1+ and C0+, we found that we just needed to stop the count whenever 4 cycles finished, and the SOP was identical to the SOP for Shift Enable. We however, did not use much logic for count enable. We first began with collecting the chips we needed. This included a Flip-Flop, a counter, and some combinational logic. We decided to convert all of our SOP to NOR only implementations. We did this because we only had NAND and NOR chips and there were less clusters when we did it this way. The updated algebra is shown below. The invertesions either came from a hex inverter or were built in, such as the case for Q'.



Shift Enable:

EQ	C_1C_0 = 00	01	11	10
00	0	d	d	d
01	0	1	1	1
11	0	1	1	1
10	1	d	d	d

$$S = EQ' + C_1 + C_0$$

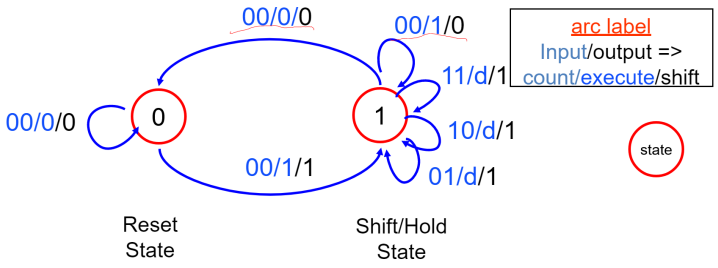
$$\overline{EQ' + C_1 + C_0} = \overline{(E' + Q) \cap \overline{C_1} \cap \overline{C_0}}$$

$$= \overline{(E' + Q)} \cap \overline{(C_1 + C_1)} \cap \overline{(C_0 + C_0)} =$$

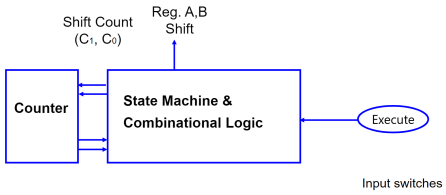
$$\overline{C_1' C_0' E'} + \overline{C_1 C_0 Q}$$

$$= (C_1 + C_0 + E) + (C_1 + C_0 + Q')$$

$$= (C_1 + C_0 + E) + (C_1 + C_0 + Q')$$

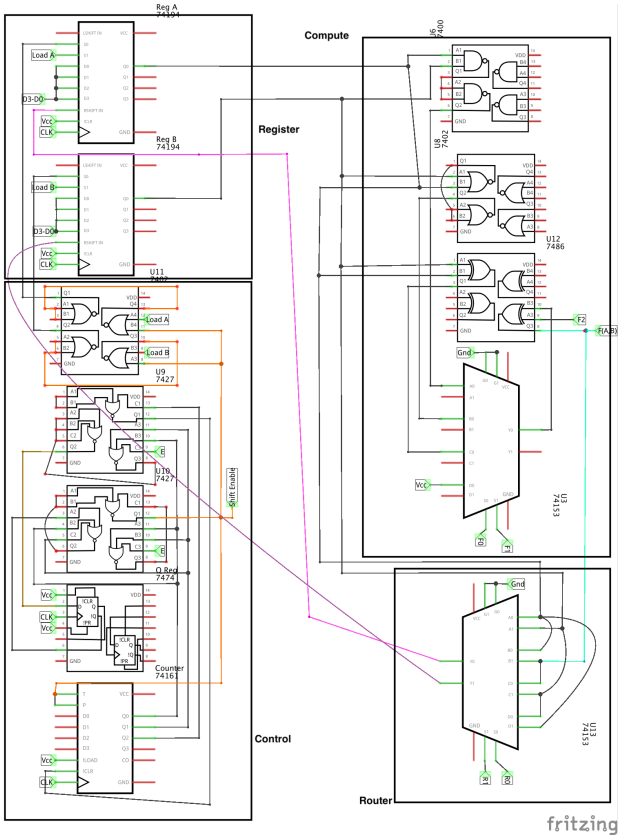


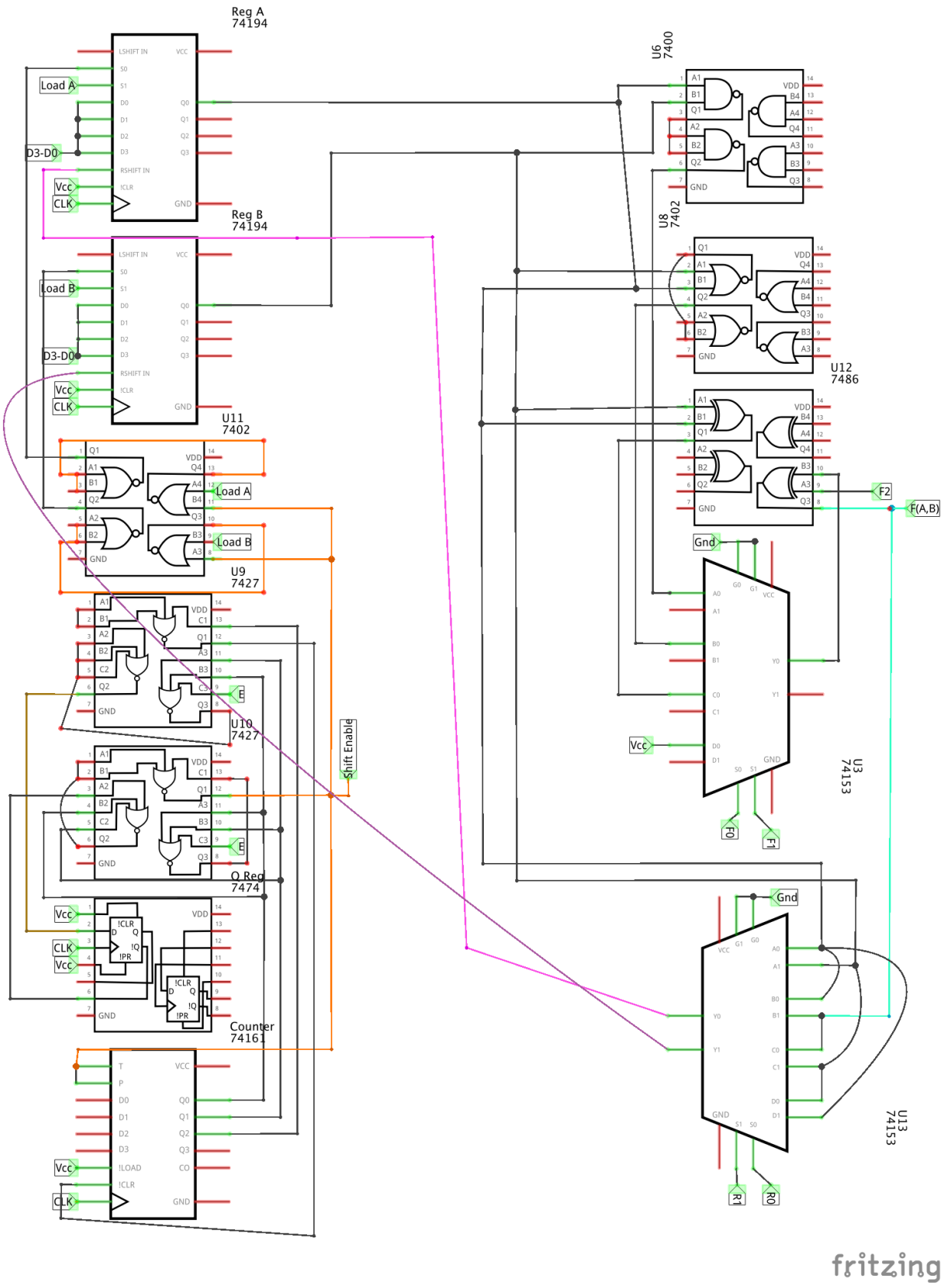
Experiment 2.1 –State Machine (Mealy)



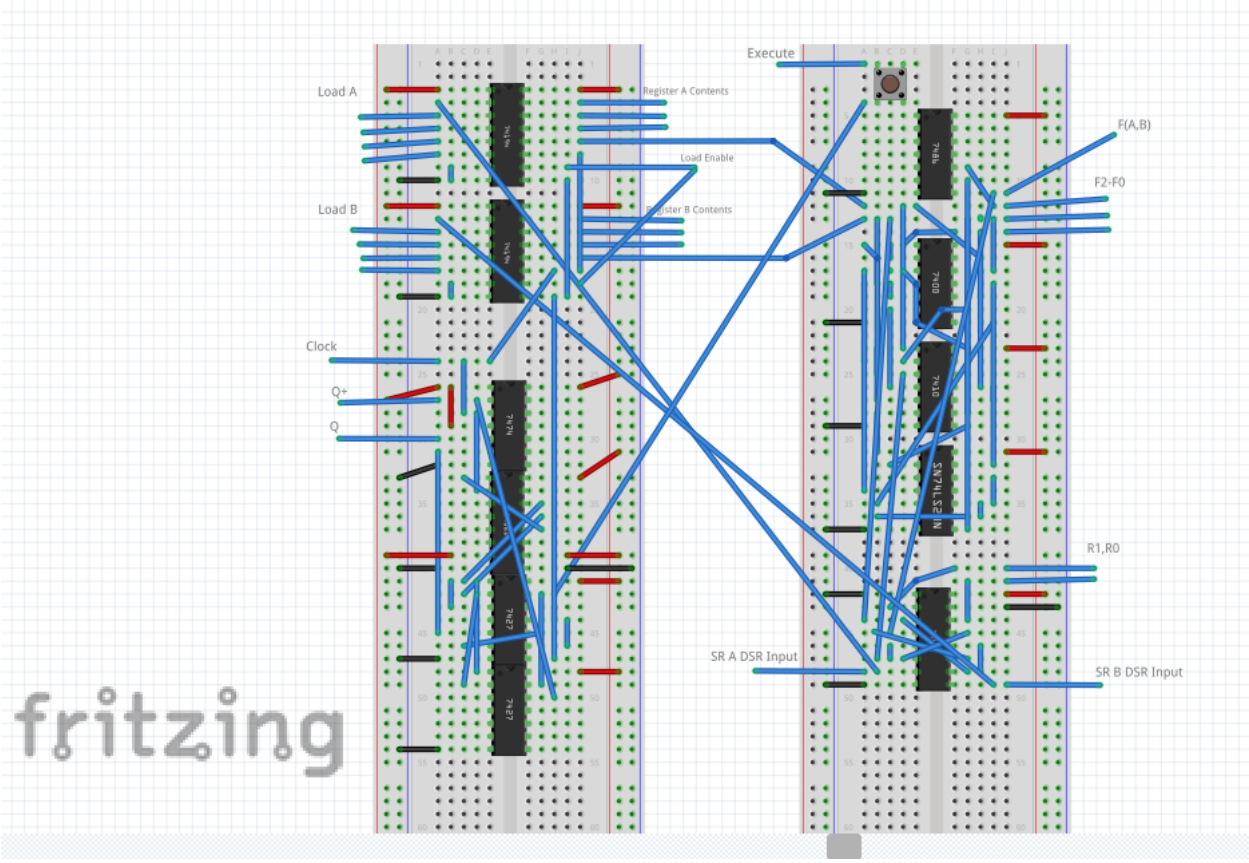
Exec. Switch ('E')	Q	C1	C0	Reg. Shift ('S')	Q+	C1+	C0+
0	0	0	0	0	0	0	0
0	0	0	1	d	d	d	d
0	0	1	0	d	d	d	d
0	0	1	1	d	d	d	d
0	1	0	0	0	0	0	0
0	1	0	1	1	1	1	0
0	1	1	0	1	1	1	1
0	1	1	1	1	1	0	0
1	0	0	0	1	1	0	1
1	0	0	1	d	d	d	d
1	0	1	0	d	d	d	d
1	0	1	1	d	d	d	d
1	1	0	0	0	1	0	0
1	1	0	1	1	1	1	0
1	1	1	0	1	1	1	1
1	1	1	1	1	1	0	0

Schematics:




Reg A
74194
Load A
D3-D0
Vcc
CLK
Reg B
74194
Load B
D3-D0
Vcc
CLK
U11
7402
Load A
Load B
U9
7427
U10
7427
E
E
Shift Enable
Q Reg
7474
Vcc
CLK
Vcc
Counter
74161
Vcc
CLK
U6
7400
U8
7402
U12
7486
F2
F(A,B)
Gnd
Vcc
F0
F1
U3
74153
Gnd
R1
R0
U13
74153
fritzing

Breadboard View



Part 2

Since the SystemVerilog code for the same four bit-serial logic operation processor was already given to us, our task was to change the existing code to create an eight bit-serial logic operation processor. The harder part of this part of the lab was to figure out how to use Quartus and understand how to use Modelsim and Signaltap, and then program and use the FPGA to demonstrate the functionality of the processor.

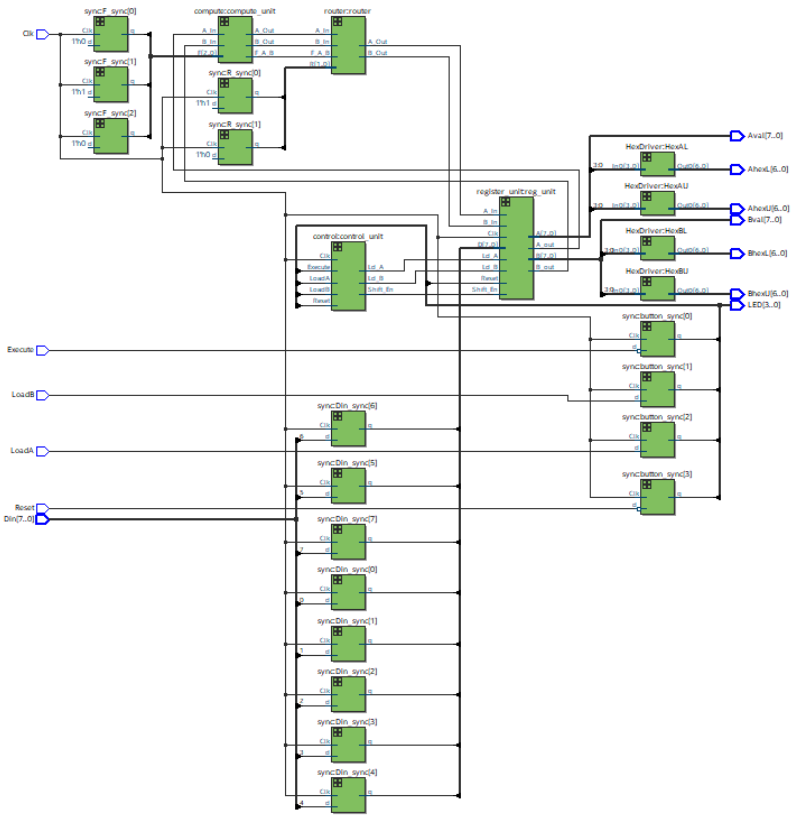
After taking some time to understand the code for the four bit-serial processor, it became more evident how to extend it to eight bits. This entailed editing values that only made up four bits into eight bits, essentially going through the code and changing some of the values that went from [3:0] into [7:0]. We also had to add extra lines/extra letters in the control.sv file to allow for eight clock pulses.

Changes made:

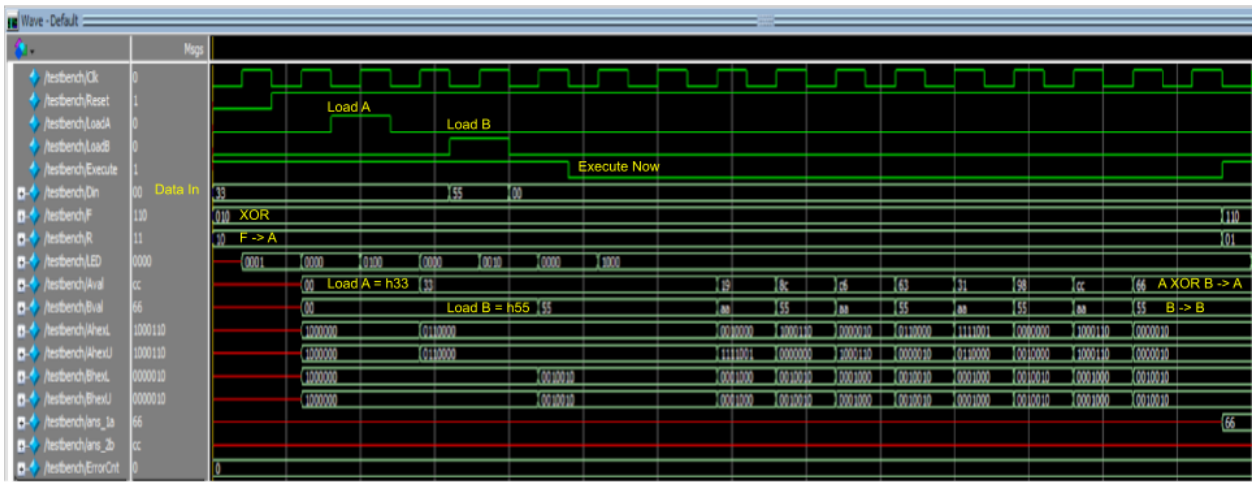
- Compute: Computes F(A,B) depending on F2-F0
    - No changes
- Control: Controls the state, shift enable, and counter.

    - Extended the number of states from 6 to 10. Changed end state to state F (state 10, after 8 bits of computation).
- Processor: Top level module used to connect all the files together and is where the physical switches are connected to.
    - Extended Din from 4 to 8 bits. Extended A and B to 8 bits. Extended Aval and Bval to 8 bits. Extended other variables from 4 to 8 bits. This was mainly done by changing the declaration from [3:0] to [7:0].
- Reg_4: Internal register that holds A and B
    - Changed input D, and Data out to 8 bits. Just changed the module I/O from [3:0] to [7:0].
- Register Unit: Register unit that inputs data in, load, and other signals to Reg_4
    - Changed D, A, B, A_out, B_out from 4 to 8 bits. Used previous method
- Router: Used to decide how to store the output of the computation unit
    - No Changes
- Overall, the changes were minimal as the processor was serial and main change was extending the register then fixing the input/output declarations in the other files.

Below is the RTL diagram of the eight bit-serial logic operation processor.



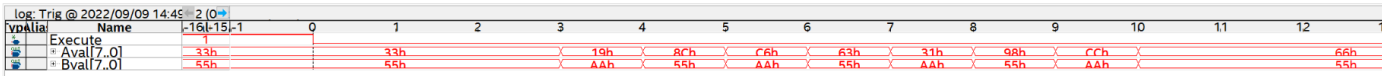
Simulation:



- XOR operation of A = h'33 and B = h'55, output is stored in A

Signal Tap:

To generate the signal tap signal below, first the FPGA and settings needed to be connected properly. Second, we shorted the F2-F0 and R1R0 signal, as there werent enough switches to drive them. The lab stated to do an XOR and store in A. We then opened signal tap, connected the clk in the settings, and added execute, Aval, and Bval as the nodes.We turned off the trigger for Aval and Bval and had the trigger for execute as either edge. We then made sure the pin assignment was connected properly and then loaded the program on the FPGA. We then loaded A and B using the switches. This can be found in experiment 2 handout. We then pressed execute and the signal tap produced the signal shown below. The result of the operation of 33XOR55 was 66 which matched simulation and theory. The eight shifts and operations can also be clearly seen in the trace.



Bugs Encountered

- Floating inputs
  - An issue we encountered was that when we initially built the circuit, some inputs were left floating which caused unexpected behavior. The issue got very severe to the point where we had to completely disassemble and then re-assemble the circuit to make sure everything is connected properly. Another major issue we had was that the back-plate of the breadboard was acting as an antenna/capacitor

which kept on inducing voltages on the power lines. We solved this by grounding the back-plate which reduced the effect of noise.

- Isolating inputs and outputs
    - Another issue we had is that when two inputs needed to be connected to an output, such as S0, a short would occur which would break the circuit. The way we solved this is by utilizing OR gates to isolate the inputs from each other.
- Connecting switches and LEDs
    - A bug that occurred in our development was that when we tried connecting LEDs to debug the circuit, we accidentally connected them in parallel to the switches which clamped the voltage of the switch outputs to around 1.5V. We had to remove the LEDs and debug using the voltmeter instead. Another issue is that the switches were floating when we initially connected them, but this was solved using a pull up resistor.
- Learning the syntax of system verilog
    - We had some difficulties learning the syntax of Verilog and how to use Quartus, but with time and reading the manuals we figured it out.

Postlab Questions:

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.

Make sure you report discusses the following:

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 way to implement this is by using an XOR. If A XOR B = F, then whenever B is equal to 1, the value of A will invert, according to the XOR truth table. This was useful for our lab as it made it possible to use a 4 to 1 MUX and an XOR gate instead of a 8 to 1 MUX to make the compute unit, which simplified the circuit by a lot.

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

- A modular design allows for different modules to be built, tested, and developed independently. The main benefit is that each module can be tested independently, which is much easier than debugging when everything is connected together. Furthermore, if

you know that a module is working properly, then it reduces the possible places for error in the circuit. The overall ease and simplicity allows debugging and troubleshooting to go much faster. An example of this is that we built the circuit module by module, and made sure each unit was working separately before connecting them together. This allowed us to isolate if the issue occurred in which unit, and if it is a problem with theory or implementation.

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

- We started the development of the state machine by first creating a Moore machine with three states. Reset, Shift, and Hold. We got the states and outputs by thinking about how the circuit operates, the inputs and outputs, and how many cycles there needs to be. After we had the Moore machine ready, we were able to compress it into a Mealy Machine with only two states. The benefits of a Mealy machine is that requires less states and usually requires less chips to implement. Mealy is more advanced than Moore, and in larger systems, the efficiency can really add up. The trade off of the Mealy machine is that it is more complex to implement, and that the next state depends on both the inputs and the current state, in comparison to a Moore machine that only depends on its current state. Moore is more simple and easy to implement, while Mealy is more efficient but more complex to design.

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?

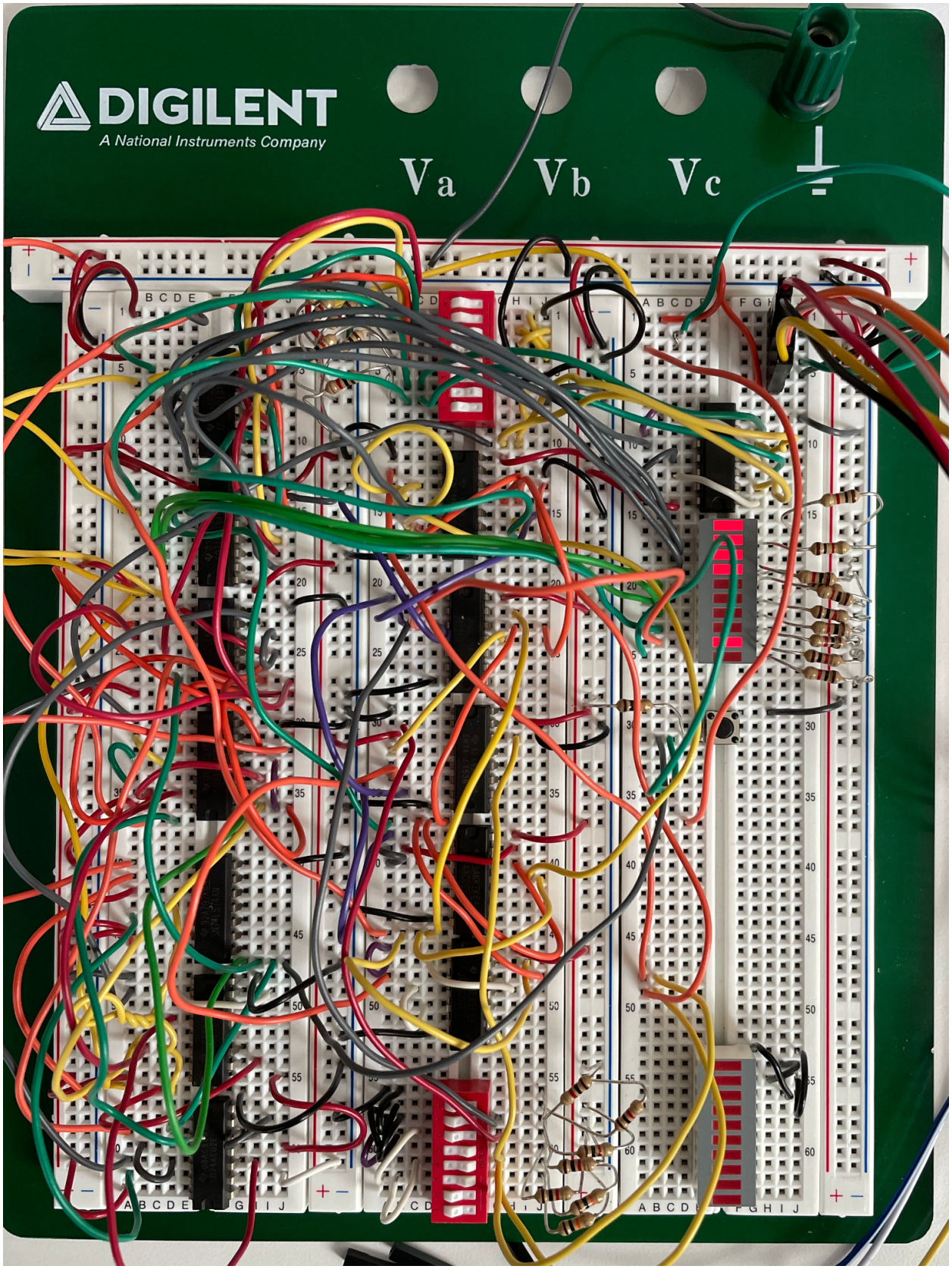
- ModelSim is a simulation on the computer which simulates the code. On the other hand, signal tap actually programs and runs the program on the FPGA. ModelSim is useful for debugging and making sure the code is working properly. ModelSim is also good when creating test benches which can test many things at one. SignalTap is more useful before deploying the program as it actually runs on hardware which might show waveforms or results that might not be captured on the simulator. SignalTap can sometimes show issues that might not appear on the simulator such as interference, hardware limitations, and mechanical issues.

Conclusion

In this lab, we physically constructed a four bit-serial logic operation processor on a breadboard using TTL chips, then used SystemVerilog code to extend an already existing program of the same processor into an eight bit version. Overall, this lab was not too complicated but it was tedious to create the physical circuit due to the large amount of components and wiring necessary. It was a great introduction to the rest of the content in the class because it exposed us

to digital circuits again after ECE 120. It also allowed us to use Intel Quartus, make sense of and edit SystemVerilog code, and understand how to use tools such as Modelsim and Signaltap.

Physical Breadboard Picture


DIGILENT
A National Instruments Company
Va
Vb
Vc