Written Description:

Part A: Register Unit

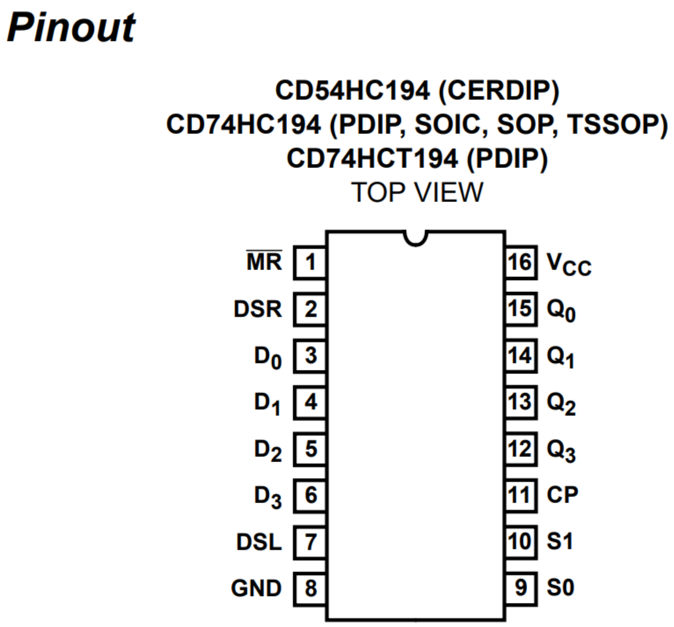


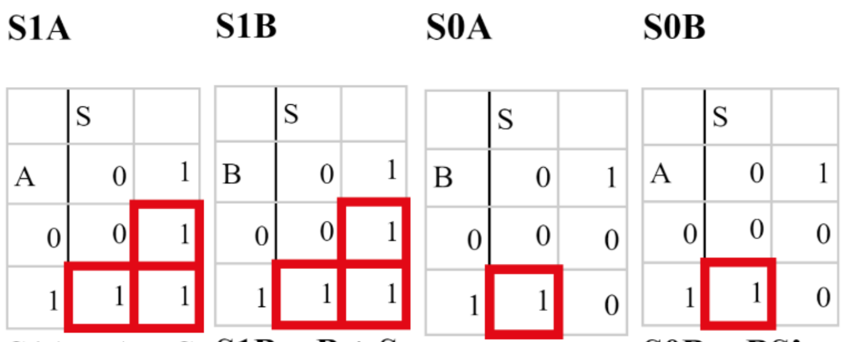
Figure 2 - CD74HC194E Pin Layout (from datasheet)

The register unit is built using a combination of two 74194 chips and specific logic to control chip behavior. The chip pin layout is above in *Figure 2.* Pin one is the master reset and should be connected to the Reset switch. The reset switch is always high and only set to low when a reset is desired. Pin two is the DSR pin. This pin receives the next A or B value from the routing unit and this value will be shifted into the register upon the next rising edge of the clock. Pins three through six are the input pins. These are connected to the input switches that are used to load the register. Pin seven was not used in our implementation of the circuit because we did not require the register to support shifting left. Pin eight is connected to ground. Pins nine through ten are the S1 and S0 pins. These pins determine if the register will shift left, shift right, hold, or parallel load. In our logic we determined the S1 and S0 values using the truth table below in *Figure 3*.

A	B	S	S1A	S0A	S1B	S0B
0	0	0	0	0	0	0
0	0	1	0	1	0	1
0	1	0	0	0	1	1
0	1	1	0	1	0	1
1	0	0	1	1	0	0
1	0	1	0	1	0	1
1	1	0	1	1	1	1
1	1	1	0	1	0	1

Figure 3 - Truth Table Logic for S1 and S0 in Register Unit

In *Figure 3*, A and B represent the Load A and Load B switches and S is one of the outputs of the control unit. When both Load and S are set to 0 then the register is in a hold state. When S high and both Loads are off then the register is in a shift right state. If Load B is high while Load A and S are low then register B is in a parallel load state and register A is in a hold state. The opposite is true for when Load A is high while Load B and S are low. If both Loads are high and S is low then both registers will be in a parallel load state. If S is high then the register will be in a shift right state regardless of the Load switches. The K-maps and Boolean expressions for S1A, S1B, S0A, and S0B are displayed in *Figure 4* below.

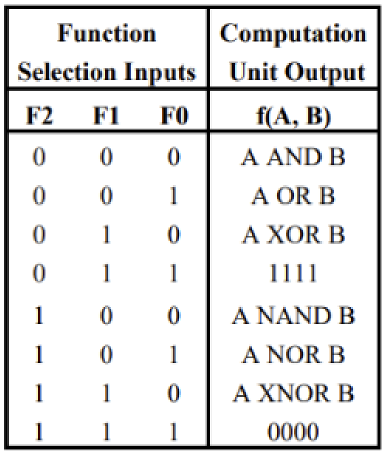


S1A = A + S S1B = B + S S0A = AS' S0B = BS'

Figure 4 - K-maps and Boolean Equations for S1 and S0 Logic

The next pin on the diagram is pin eleven which is the clock pin. Then pins twelve through fifteen represent the output of the register unit. Because we are shifting right, our Q3 or final output bit at pin twelve is connected directly to the computation unit and is the bit used when calculating the inputted function. Lastly, pin sixteen is connected to VCC.

Part B: 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

Figure 5 - Function Value (Experiment #2, 2.3)

Our computation unit design created logic for an AND, OR, XOR, and high output functions. We controlled the selection and exportation of these values with the combination of one 4 to 1 mux and one 2 to 1 mux. The mapping of F bit inputs to logical operations is shown above in *Figure 5.* When all input bits are low, the computation unit will perform an AND operation. Since we decided to shift right, the inputs to all of our logical operations come from the furthest right bits stored in the registers. This corresponds to the twelfth pin on the diagram. The input from pin twelve on register A will be referred to as input A and the input from pin twelve on register B will be referred to as input B. We used one two input NAND chip and one inverter chip to implement the AND logic. First, we passed input A and B into a NAND gate. Then, we passed the output through a NOT gate to achieve a functional AND following DeMorgan's law. The next function that we implemented is OR. To set the computation unit to an OR, F2 and F1 are set low while F0 is set high. The OR is implemented with one NOR chip and an inverter chip. Similarly to the AND logic, inputs A and B are first passed into a NOR gate, and then inverted via a NOT to achieve a logical OR. The next function implemented is XOR. To set the computation unit to a XOR, F2 and F0 are set low while F1 is set high. The XOR is implemented with one XOR chip. Simply input A and B once more and take the output as the result of the function.. The last function that we implemented was the high output function. To implement this, we can draw the high output straight from VCC.

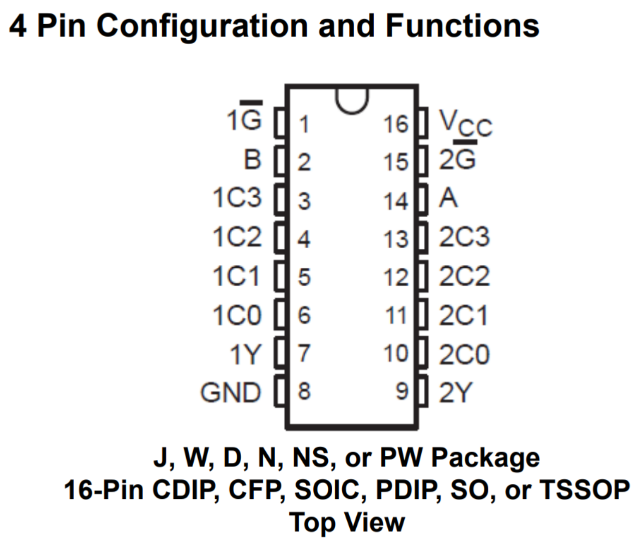

4 Pin Configuration and Functions
1G̅
B
1C3
1C2
1C1
1C0
1Y
GND
1
2
3
4
5
6
7
8
16
15
14
13
12
11
10
9
VCC
2G̅
A
2C3
2C2
2C1
2C0
2Y
J, W, D, N, NS, or PW Package
16-Pin CDIP, CFP, SOIC, PDIP, SO, or TSSOP
Top View

Figure 9 - SN74HC153 (4 to 1 mux) Pin Layout (from Datasheet)

The output of the first part of the computation unit was decided based on a 4 to 1 mux. The pin diagram for the 4 to 1 mux is displayed above. Pin two is connected to the F1 switch and Pin fourteen is connected to the F0 switch. Together, these two bits act as the select bits and are used to choose the desired function. Then pins three through seven are connected to the outputs of the four functions described above. Pin three is connected straight to VCC for the high output function. Additionally, pin four is connected to the output of the XOR logic, pin five is connected to the output of the OR logic, and pin six is connected to the output of the AND logic. One of the function outputs is chosen by the select bits and is the output seen on pin seven. The output of pin seven goes to the final chip in the computation unit, the 2 to 1 mux.

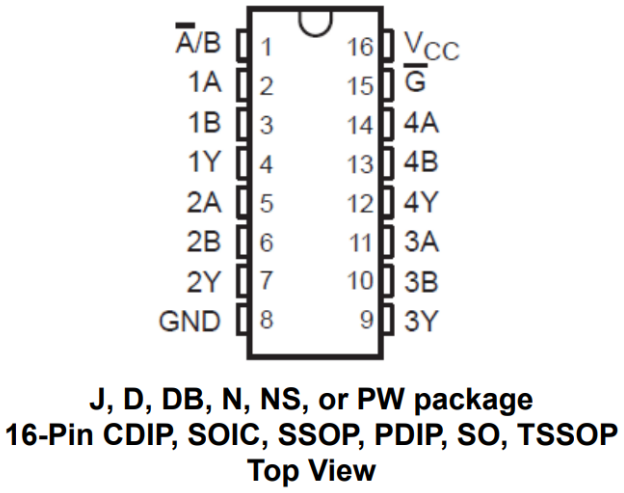


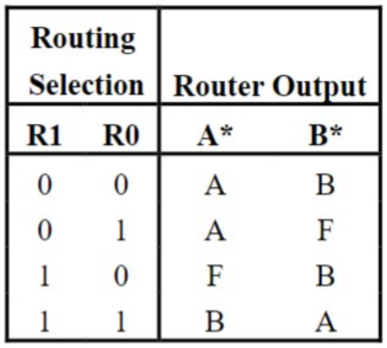
Figure 9 - SN74HC157 (2 to 1 mux) Pin Layout (from Datasheet)

When F2 is high the output is flipped from the output of the 4 to 1 mux. This is because functions that share F1 and F0 bits are simply inverses of each other. For example, 000 is AND and 100 is NAND, the inverse function. The pin diagram for the 2 to 1 mux is shown above in

Figure 9. The value from the F2 switch is the input to pin one. Pin two receives the output of the 4 to 1 mux.Pin three is the inverted output of the 4 to 1 mux. Depending on the value of the F2 switch, either the output of the computation unit is inverted or maintained. The resulting value is taken from pin four and is then connected to the corresponding inputs of the routing unit.

Part C: Routing Unit

For the routing unit, we elected to use one 4 to 1 mux chip. Each chip contains two separate 4 to 1 muxes which can be used to determine the next outputs that will be sent to registers A and B. *Figure 10* shows the truth table that determined the behavior of the routing muxes.



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 10 - Routing Unit Truth Table

With this information, it is clear to see that the next value of A and B each can take on 3 possible values: A, B, and F, the function value. A and B represent the current values of those registers, and the value of F is obtained directly from the output of the selected function from the computational unit. Both muxes share the same R1 and R0 bits which act as the select bits, allowing us to design a simple mux system shown in *Figure 11*.

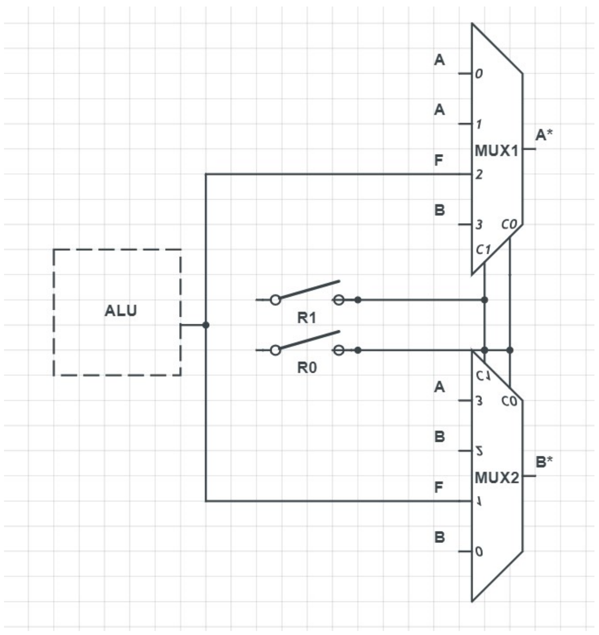
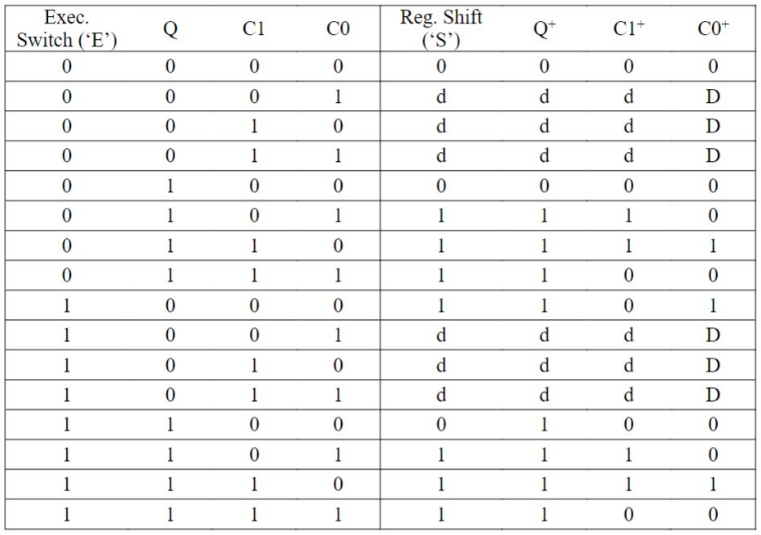


Figure 11 - Routing Unit Design

These R1 and R0 bits are controlled directly by switches and allow the user to easily change the desired routing. The top mux produces the next value which will be shifted back into registerA, and the bottom mux produces the next B value, denoted A* and B* respectively. The A and B inputs to these muxes are pulled directly from the shifting registers, and the F value is the result of the computation unit as shown. Overall, this dual-mux system allows us to correctly determine the next values of A and B and route them accurately based on user input in the R1 and R0 bits.

Part D: Control Unit

The control unit is the group of circuit components that make up the finite state machine and allow us to control the behavior of the processor. The actual behavior, transitioning, and possible states of the designed FSM are discussed in detail in the next section. For now, it is significant to know that we selected a Mealy machine and that the control unit makes use of five values: C1, C0, E, Q+, and S. These values are calculated from *Figure 12* below.



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 12 - Truth Table for Next State Calculations S+, Q+, C1+, and C0+

From this table, it is clear to see that the value of the execute switch (E) does not impact the next state output except for when it is initially flipped. What this means is that regardless of whether E is high or low at the end of one 4 clock cycle computation, the processor must return to the reset state. Therefore, the values of C1+ and C0+ can be abstracted into a counter chip. The counter will take the clock as an input and cycle through values from 00 to 11 in binary on its two output pins. The values of C1 and C0 can be directly taken from these output pins and used to signal when the registers should stop shifting by implementing the logic shown in *Figure 13*.

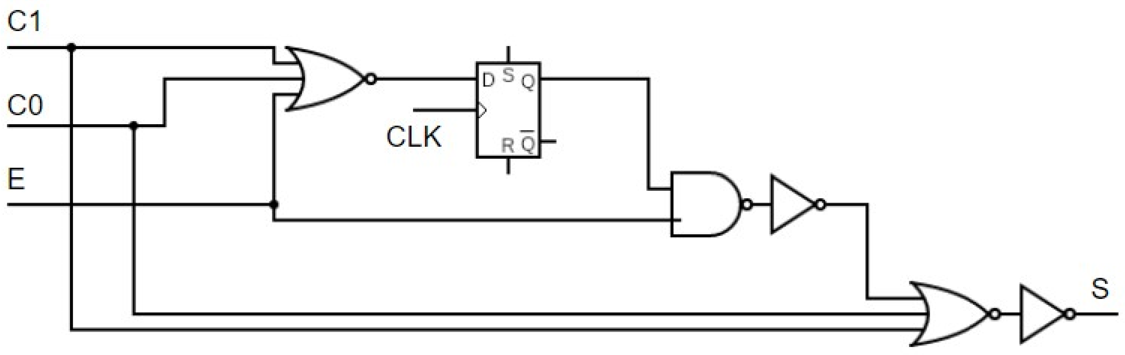


Figure 13 - Circuit Schematic for the Control Unit

To begin breaking down this schematic, it is essential to look at the next state K-maps created from the overall truth table in *Figure 12.*

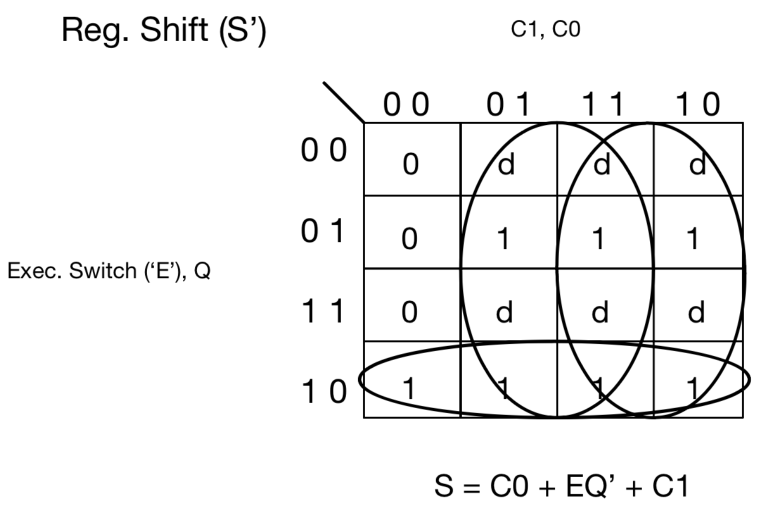
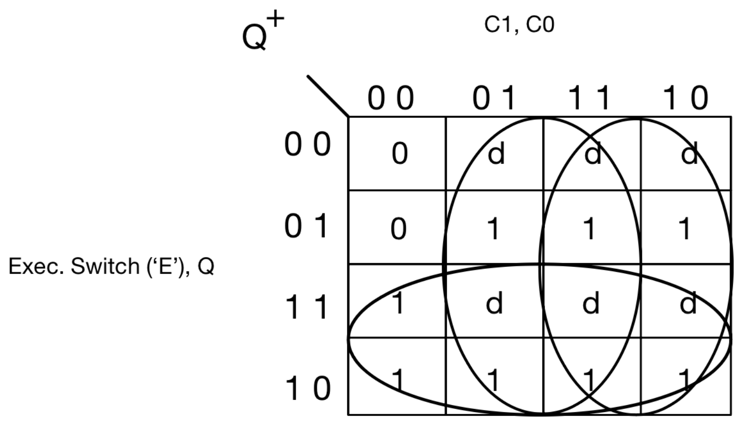


Figure 14 - Kmap for S+

The critical output from our control unit is the S bit, which after being passed through simple register logic described in the Register Unit section determines whether the registers should continue shifting or hold. From the K-map, it is known that S depends on both counter bits and also EQ'. This is logical as if neither of the counter bits are active (if the counter is at 0), no shifting should be done. The only exception to this rule is when E is high and Q' is low. This is intuitive because the only way to start the system when the counter is in reset state (00) is by triggering the execute switch. However, this logic is not perfectly clear due to the value of Q' being unknown. *Figure 15* shows how the value of Q+ can be determined.



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

Figure 15 - Kmap for Q+

Q+ depends on both counter bits as well as the execute switch. In *Figure 13*, these three inputs are first put into a NOR gate which will output the value of Q+'. This is necessary because the S bit depends on Q' rather than Q. Next, the value of Q' is passed through a flip-flop. This is critical, as the flip-flop stores the value of Q+ and only shifts on the rising edge of the clock. This allows us to store the next state value for use in the calculation of S, rather than using the current value of Q. So, the Q output from the flip flop represents which state the circuit is currently in, and the Q input represents which state the circuit should transition to next. Next, the flip-flop output Q' is taken and passed through logic which computes C0 + C1 + EQ', resulting in the final output being S. Overall, by combining logic to control Q+ and S+ with the counter chip to control C0+ and C1+, we designed a Mealy machine which is responsible for controlling all behavior in the processor.

Block Diagram:

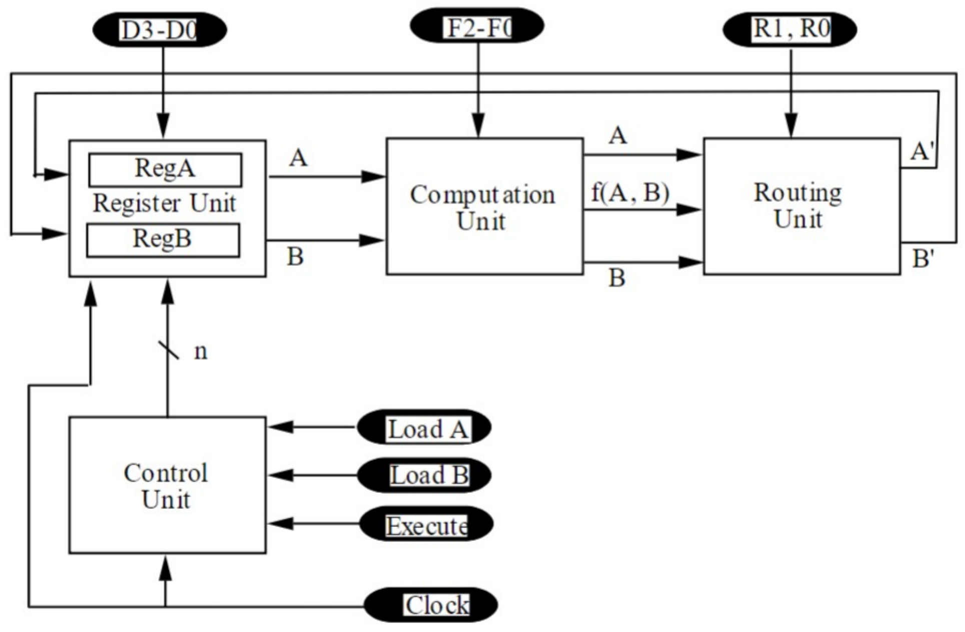


Figure 16 - Block Diagram of Processor

The block diagram shown in *Figure 16* models our processor. It begins with the register unit where registers A and B each take in 4 bits of data to store while their respective load switches from the control unit are high before execution occurs. When execute is high, the registers shift serially into the computation unit. This unit uses the A and B bits to perform a function designated by the 3 input F2-F0 bits. Then, the result of this operation in combination with the previous A and B values are routed through the routing unit based on the input value of the R1 and R0 bits. Finally, the control unit stops processor execution until the execute switch is disabled and enabled again.

State Machine:

We chose to use a Mealy machine because it was convenient to build a system with only two states. Instead of a Moore machine, Mealy machines depend on both current state and inputs. On the other hand, Moore machines depend solely on the current state. In this case, integrating a counter chip makes it easy to cycle through one state, the shift state, based on the counter value rather than needing to cycle through four completely separate states in a Moore machine design. This implementation allowed us to create a machine with just a shift/halt and reset statewhich simplified the circuitry greatly. Our design followed the design established in the lab handbook and is shown in *Figure 17*.

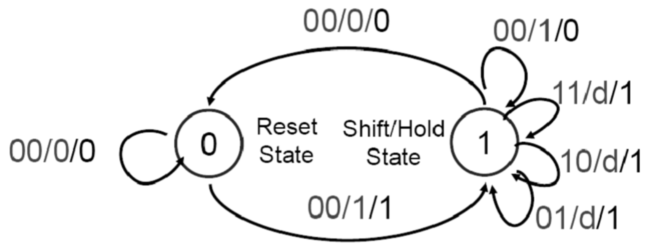


Figure 17 - Mealy Machine State Transition Diagram

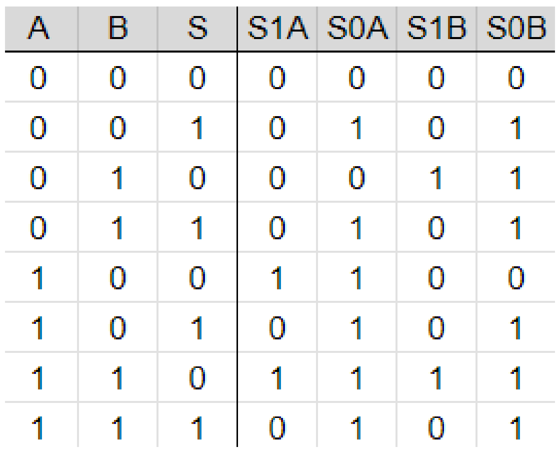
In this diagram, the arcs are labeled with four values. The first 3, XX/X, are the inputs to the system corresponding to C1C0/E. These are the three values which determine Q, the state, and then sequentially determine if the registers should shift. While the machine is in the shifting state, it is critical that the execute bit is a "don't care" to ensure that even if it is disabled in the middle of a computation, the registers continue shifting for four clock cycles. The value furthest to the right represents the output, S. This is the value that is passed through logic to the registers to control the shifting behavior. As the diagram shows, this value is 1 even before we are in the "shift state", which allows our machine to function synchronously with the clock and only shift four times. This is a product of the value Q being stored in a flip-flop before it is used to calculate S. In general, utilizing this design of a Mealy machine simplified our design greatly.

Design Steps Taken:

Note: K-maps in this section have already been included in the Written Description

Part A: Register Unit

The first step in our design process was creating the register unit. Since all of the computations handled by our processor require serially shifted data from both A and B registers, we decided to implement our register functionality first. The first challenge was choosing the correct chip. Our register needed to have the ability to parallel load, shift serially, and also most importantly hold 4 bits of information. Initially, we attempted to use the 74195 chip, but changed to the 74194 depicted in *Figure 2* because the 74195 did not have the ability to hold data. The next design step was to determine the direction in which our register shifted. Logically, shifting left and right are equivalent because the bits are cycled back to their original locations after the computation is performed and four clock cycles have passed. As long as the choice is consistent, either works. We chose to shift right. This means that the result from the routing unit would go into the DSR pin and the register would shift that value into the stored bits at the next clock rising edge. Next, the logic to control register shifting was implemented. This logic is depicted in *Figure 3* below.



A	B	S	S1A	S0A	S1B	S0B
0	0	0	0	0	0	0
0	0	1	0	1	0	1
0	1	0	0	0	1	1
0	1	1	0	1	0	1
1	0	0	1	1	0	0
1	0	1	0	1	0	1
1	1	0	1	1	1	1
1	1	1	0	1	0	1

Figure 3 - Truth Table Logic for S1 and S0 in Register Unit

In this table, S1 and S0 can have 3 possible values: load (11), shift right (01), or hold (00). In this case, the registers are instructed to shift any time that the S bit is high. In other words, any time the control unit is in the shifting state, our registers will shift, regardless of whether one is attempting to load them. To load either register, its switch must be the only high input bit. This allowed us to create the K-maps and the logic found in *Figure 4*.

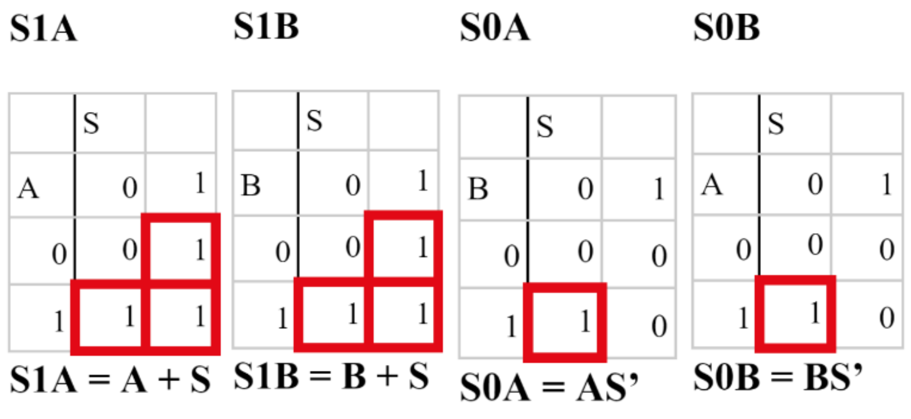
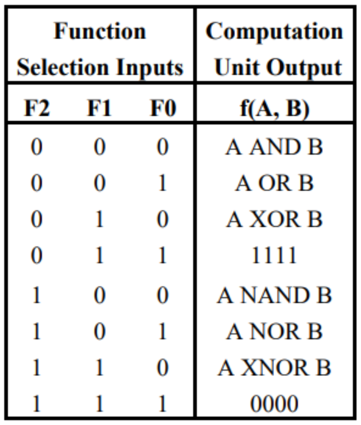


Figure 4 - K-maps and Boolean Equations for S1 and S0 Logic

Once this logic was devised, the last step was translating it into NAND, NOR, and NOT gates. This can be easily done by simply using NAND instead of AND, and NOR instead of ORas long as the output is also inverted with the use of a NOT gate. Now, our registers are set up to shift depending on the S bit, and they are equipped to store information as desired.

Part B: Computation Unit

The next step in the design process was the computation unit. Now that our registers could accurately shift data serially, we needed to create the computation unit to operate on one bit at a time and perform the correct function according to the F2-F0 bits, as seen in the table below.



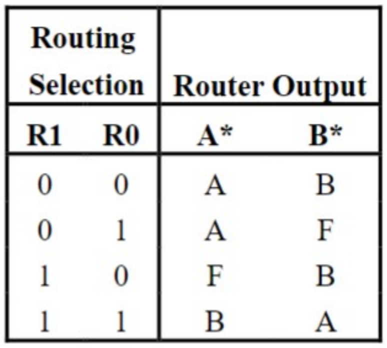
Function Selection Inputs			Computation Unit Output
F2	**F1**	**F0**	**f(A, B)**
0	0	0	A AND B
0	0	1	A OR B
0	1	0	A XOR B
0	1	1	1111
1	0	0	A NAND B
1	0	1	A NOR B
1	1	0	A XNOR B
1	1	1	0000

Figure 18 - Map of F Bits to Desired Function

From this table, we observed that the last four functions which correspond to an F2 value of 1 are the inverse of the first four functions which correspond to an F2 value of 0. This allowed the design to feature solely one 4 to 1 mux and one 2 to one mux. The former would use the F1 and F0 bits as select bits, and the latter would only require the F2 bit to select whether to maintain the output from the first mux or invert it. We considered following the table exactly and manufacturing logic for all eight functions, but this would have required almost twice the gate usage. In hindsight, since the final four functions correlate precisely with the gates available in our given chip set, it is possible to use less NOT gates. For example, creating the AND function required one NAND gate and then a NOT gate to invert the output. Optimally, the NAND function could have been created with just a NAND gate and behavior would have been the same because this output could then be inverted once to achieve a logical AND. This would be preferred rather than the original NAND being inverted to get an AND and then potentially being inverted once more to return to a NAND. However, our circuit features the first four functions. To create these, a similar method to the routing unit was used; where an AND or OR gate was desired, a NAND or NOR was used and the output was inverted. For XOR, the 74086n chip contained XOR gates which could be utilized directly to create the logic. Finally, to return a high value, VCC was connected to the 11 mux input. Overall, our design was slightly inefficient but nevertheless successful in performing the required functions with only NAND, NOR, XOR, and mux chips.

Part C: Routing Unit

Now that the registers can shift data into the computation unit and the computation unit can compute the result of the eight required functions, the next step in the design process is to determine where these new values should be routed. To do so, our implementation depends on two input bits which act as the select bits to two separate 4 to 1 muxes: R1 and 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

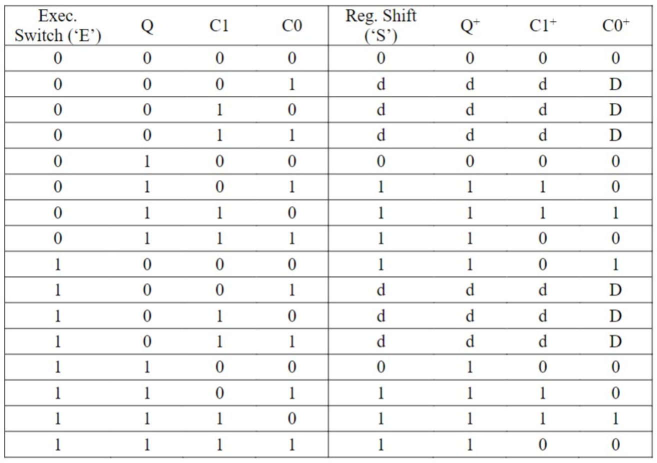
Figure 10 - Routing Unit Truth Table

From this table, it is clear that both registers could receive three possible values. First, the register could simply get its current bit back. Or, it could be swapped, receiving the partner register's bit. Finally, either register could get the computed function value from the computational unit. So, with three possible choices and two select bits the 4 to 1 mux is the clear choice. Technically, creating a mux with only 3 inputs utilizing logic to determine when these inputs would be selected is possible since for both registers two possible routing inputs result in the same value being chosen. However, this design is complicated and inefficient compared to the 74153 chip we are given. Because of this, we elected to route the duplicate value into its corresponding slot so that our mux had four inputs. Since we elected to shift right, the last step was to connect the output of the muxes to their respective register's DSR pin. This completes the design of the routing unit with just two 4 to 1 muxes.

Part D: Control Unit

Also Addressed: Question 3: "Discuss the design process of your state machine, what are the tradeoffs of a Mealy machine vs a Moore machine?" (Lab 2, 2.9).

Designing the control unit was the last step in our design process. At this point, our registers could shift, the computation unit accurately calculated the inputted function, and the routing unit cycled the correct bits back into the registers. The final task is to create a finite state machine which will control the value of our S bit and tell our registers when to shift. The desired machine behavior is seen below in *Figure 12*.



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 12 - Truth Table for Next State Calculations S+, Q+, C1+, and C0+

This table represents the first decision that was made with regard to the finite state machine. Achieving the desired functionality is possible with either a Mealy or a Moore machine. For this lab, we determined that a Mealy machine was optimal. This system determines the next state by taking into account the current state as well as any sequence of input values. On the contrary, a Moore machine determines the next state based exclusively on the current state. First, we will discuss a possible implementation with a Moore machine and why this path was not chosen. Notoriously, Moore machines require more states than Mealy machines in general. In this lab, one possible design for a Moore machine would cycle through states in this pattern: reset -> shift (1) -> shift (2) -> shift (3) -> shift (4) -> halt. This results in a total of six states. However, the advantage to this system is clear in that no inputs are required. No other chips are necessary; once the execute switch is flipped, the sequence of states will continue in exactly the same manner every time. However, creating the logic to transfer between these states is not trivial and can often be difficult because each state still needs to output the correct S value. This system would require next state truth tables and K-maps for six states, which cumulatively results in complex logic. Additionally, even simple logic like AND or OR operations take up double the gates due to the necessity to form these gates out of their inverted counterparts, NAND and

NOR. So, we determined that this logic could be bulky and inefficient compared to the Mealy machine. This is especially the case because two of the Mealy machine inputs, C1 and C0, can be easily obtained from the 74161 counter chip. Additionally, the Mealy machine would only require two states; shift/halt to control the shifting and waiting at the end of computation, and reset to end the computation and prepare for the next. Moreover, the logic in *Figure 14* and *Figure 15* show that calculating Q+ and S are relatively simple in the two state machine. These figures can be seen below.

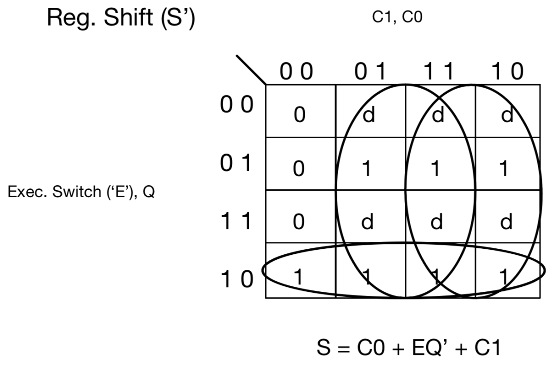


Figure 14 - Kmap for S+

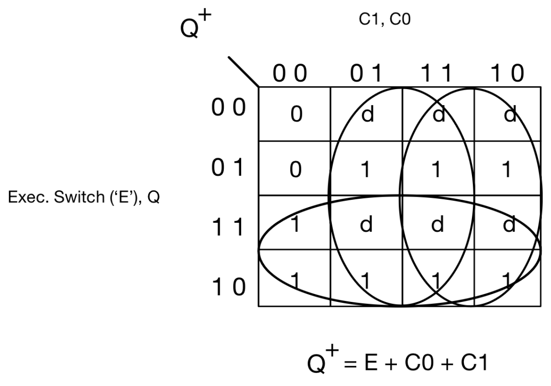


Figure 15 - Kmap for Q+

With this logic in place a Mealy machine with the circuit schematic seen in *Figure 13* can be designed. The design choice to utilize a flip-flop to store the value of the state bit, Q, is explained in more depth in the 'Written Description: Part D: Control Unit' Section. Finally, the logic above once again needed to be translated into the available NAND/NOR/NOT gates, and the result of this transformation is seen in *Figure 13*. The design required the use of the given 3 input NOR gates, as well as inverting outputs to achieve the desired output following DeMorgan's law. The output of the control unit, S, is routed into the logic designed to control the shifting of the registers described in the above Register Unit design segment. At this point, our Mealy machine can properly control the shifting of the registers, meaning our circuit is complete.

Detailed Circuit Schematic:

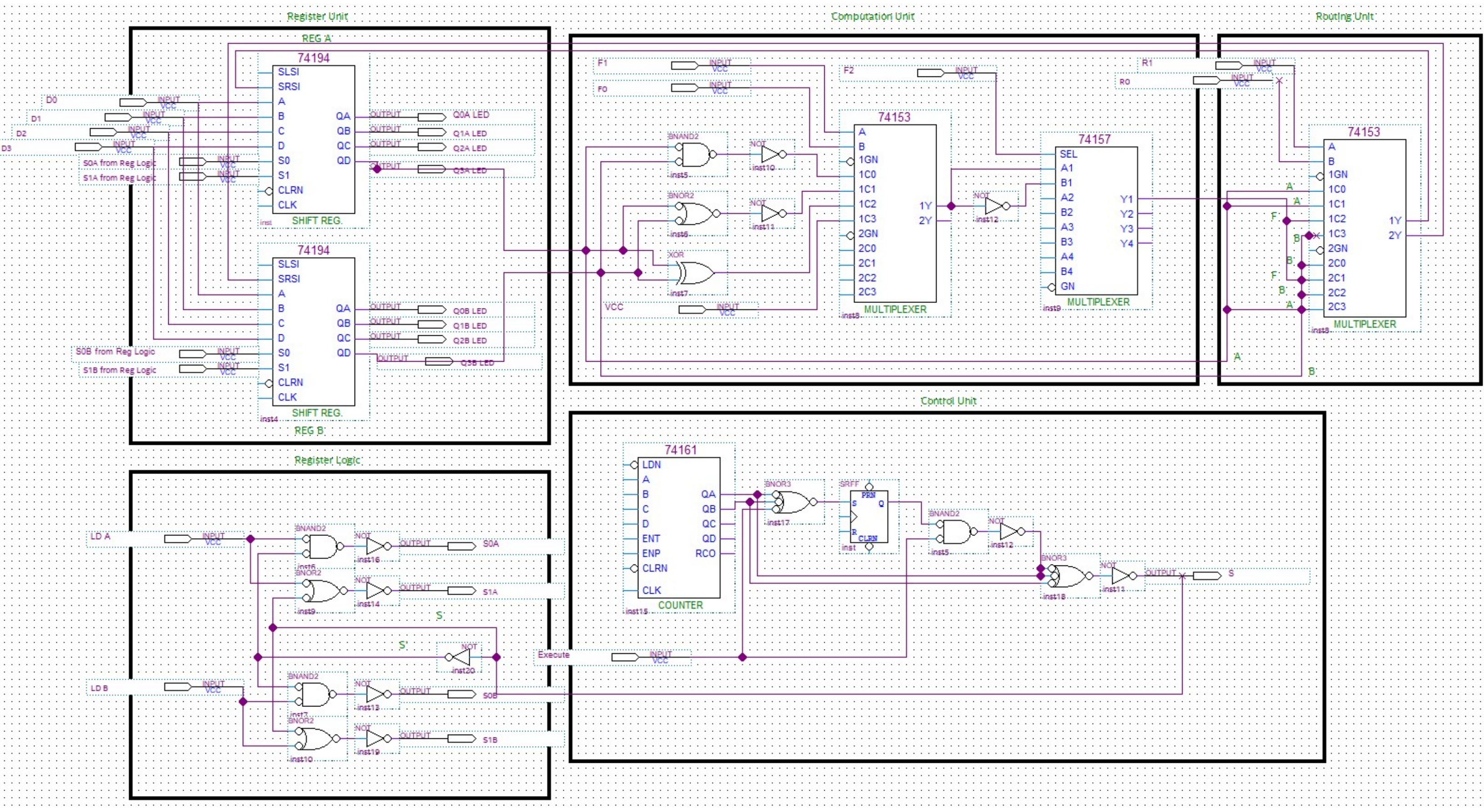


Figure 18 - Overall Circuit Schematic

Breadboard View / Layout Sheet:

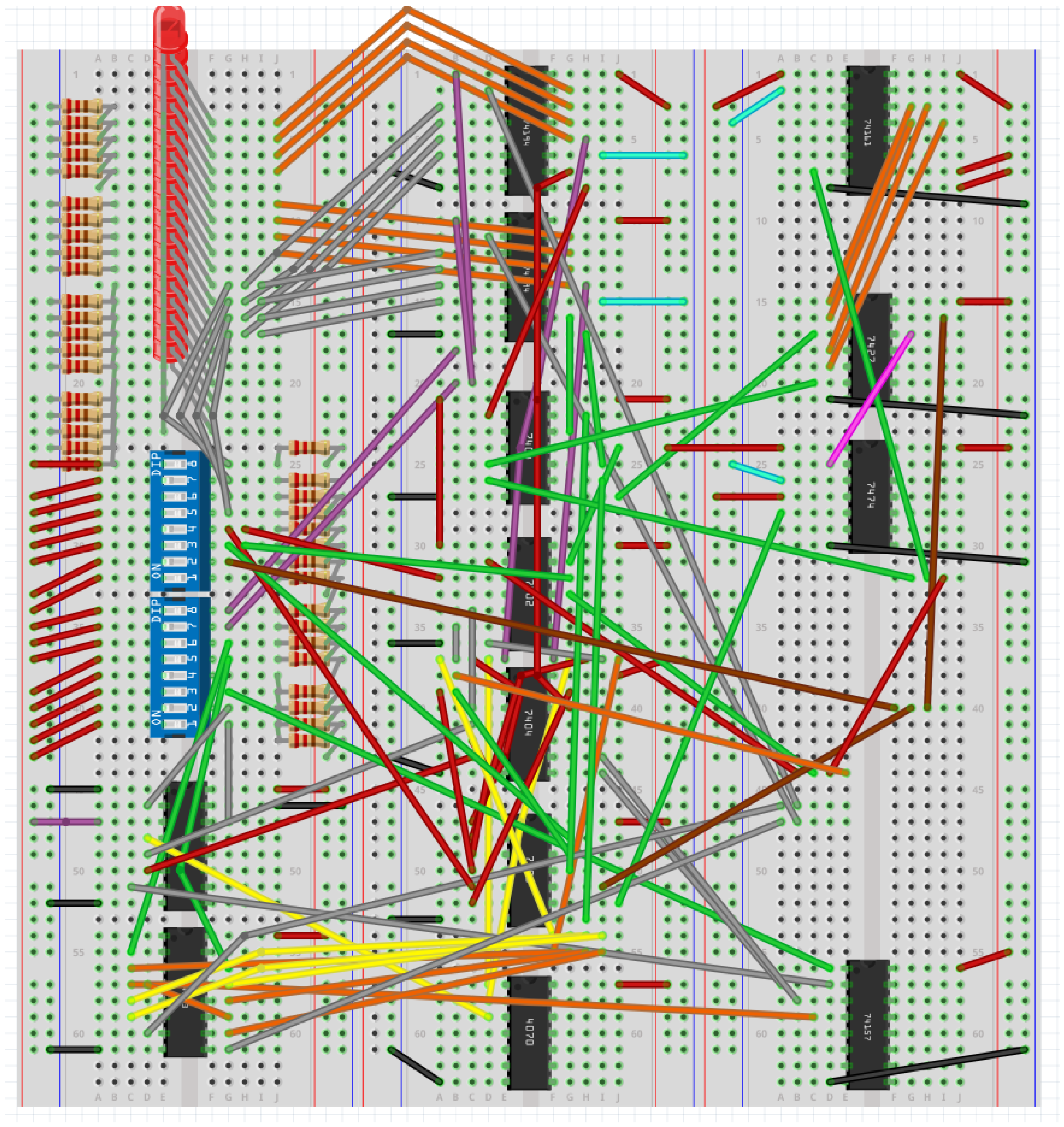


Figure 19 - Breadboard View of Complete Circuit

Bugs/Corrective Measures:

When implementing the circuit design for lab 2.1 we ran into many small errors. Frequently, we would find a wire off by one pin, forget to power switches, or flip the logic of some of the muxes. Resolving wire placement issues required careful circuit tracing. We would begin with an input and trace through all of the logic it was involved in to make sure every wire was plugged in correctly. The next error that became a major issue for us was unwired switches. We forgot to wire one of the switches which led to incorrect behavior. Because of this, we inspected elements like our implemented logic, wire placements, and chip selection to attempt to locate the error. However, the switch was the true cause of the error. The fix for this bug was ensuring that all of our switches are connected to power and ground. Another error that we encountered was flipped logic in the muxes. This required quite a bit of time to debug. The muxes present in our computation and routing units seemed to be returning the inverse values. For example, if the F1 and F0 bits were 000 (AND), the mux would select and return the output of the NAND function which should have corresponded to F bits 111. . The fix for this was to run the computation unit with all of the outputs from the 4 to 1 mux connected to LEDs. This made it so we could visibly confirm that the logic in the computation unit was correct and one of the muxes must be flipped. In the end, the logic going into our 2 -1 MUX in the computation unit was flipped resulting in an inverse output. A larger issue that we encountered was an inability to hold data in our registers when we initially utilized the 74195 register chip. . The solution to this was to switch to the 74194 chips for the register unit. The advantage of this chip was the DSR, hold state, parallel load, and shift right functionality controlled by the DSR and S1S0 pins respectively. Once we made the switch the logic was much easier to implement. While implementing the whole circuit, we avoided many potential errors by testing each unit by itself.

We also frequently used LEDs to ensure that the control unit and other outputs functioned correctly. As our circuit neared completion, we encountered one last hardware error causing our Reset switches to lose functionality. Eventually, we realized that the metal on a pair of neighboring resistors was touching which led to the Reset being set to high when it should be set to low and vice versa. Overall, we mostly encountered hardware errors, and handled any larger logic errors with LED debugging.