ECE 385

Spring 2023

Lab 2

Experiment #2

Aleena Majeed & Ali Chaudry
17:50 - 17:55

Wei Ren

1. **Introduction**
2. **Operation of the Logic Processor**
3. **Description and diagrams**
4. **Design steps and schematic diagram**
5. **Breadboard view**
6. **8-bit logical processor on FPGA**
7. **Description of all Bugs and corrections**
8. **Conclusion**

1. Introduction

Summarize 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.

The circuit built has the capability of carrying out bitwise logical operations, but to avoid duplicated processing of each bit of data, it needed to prevent multiple circulations. The circuit will perform operations like the bitwise logical functions in machine language programming. The processor incorporates a register unit, computational unit, routing unit, and control unit to execute logical operations on 8-bit inputs. It is capable of performing the logical operations of AND, OR, XOR, NAND, NOR, and XNOR, as well as swapping the values stored in each register

2. Operation of the Logic Processor

a. *Describe the sequence of switches the user must flip to load data into the A and B registers. 3.10*

Given the constraint of a restricted quantity of switch inputs, the A and B registers leveraged a common input switch mechanism for the purpose of value loading into their respective storage locations. To ensure distinct identification of the target register, supplementary switches labeled as "Load A" and "Load B" were incorporated and employed as activators for the loading process, being deactivated upon completion. The actual loading of the binary values into the registers was accomplished via the use of the D0, D1, D2, and D3 labeled switches. With D0 representing the least significant bit and D3 representing the most significant bit. As an example, to load the binary values of 0001 into register A and 1000 into register B, the following procedure must be followed: activate the Load A switch, then switch on D0 while preserving the other switches in an inactive state, subsequently deactivating the Load A switch. Then activate Load B switch and switch on D3, maintaining the other switches in an inactive state, before deactivating Load B switch. Upon completion, the values of 0001 and 1000 are effectively loaded into registers A and B respectively, thus enabling the system to proceed with computation and routing operations.

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

The computation unit is tasked with executing bitwise logical operations on the binary values stored within registers A and B. After the values have been successfully loaded into their respective registers, the user must engage the F0, F1, and F2 switches to specify the desired operation. With F0 serving as the least significant bit and F2 as the most significant, the circuit has been engineered to perform AND, OR, NOR, 1111, NAND, XNOR, and 0000 operations when the switches are set to 000, 001, 010, 011, 100, 101, 110, and 111, respectively. The routing unit, equipped with switches R0 and R1 (R0 being the least significant bit and R1 being the most significant), then provides the means by which the user may dictate the storage and display of the output generated by the computation unit. The routing unit switches, set to 00, allow the values within both registers to remain unchanged. A routing selection of 01 retains the values within register A while displaying the computation output in register B. Setting the routing switches to 10 stores and displays the computation output in register A, while maintaining the value within register B. Finally, a routing selection of 11 facilitates a value swap between registers A and B. In summary, the user must first 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 make a determination on the storage and display of the computation output through the utilization of the R0 and R1 switches on the routing unit.

3. Description and diagrams

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.)

REGISTER UNIT:

The purpose of the register unit for this circuit is to take the 4-bit values that we input into the register through switches, and store them in their respective registers. This unit is also responsible for shifting the bits as we do our logic operations, so that each register will end with their respective value once the bitwise shifting is over. The shifting of this unit is dependent on the control unit.

COMPUTATION UNIT

The computation unit takes the 4-bit input of each register unit and does the computation that is set by a set of switches. Then the circuit performs the specified operations bitwise. The computation unit is capable of performing the

logical operations of AND, OR, XOR, NAND, NOR, and XOR. The answer can be stored back in the registers depending on the settings of the switches in the routing unit.

ROUTING UNIT

The routing unit determines what new values will be stored in the registers A and B. The routing unit uses a 4-to-1 muxes, which has 2 select bits controlled by switches, to decide what values should be stored and which register the values should be stored in. Register A and B can either hold their original values, the answer to the computation, or the value of the other register again, this is decided by the select bits of the 4-to-1 mux.

CONTROL UNIT

The control unit is the most complex portion of the circuit. This unit controls when the computation of the 4-bit values stored in A and B begins, as well as controlling the shifting that takes place in the register unit. The control unit was built based on a Mealy FSM, where the outputs depended 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.

b. High-Level Block Diagram

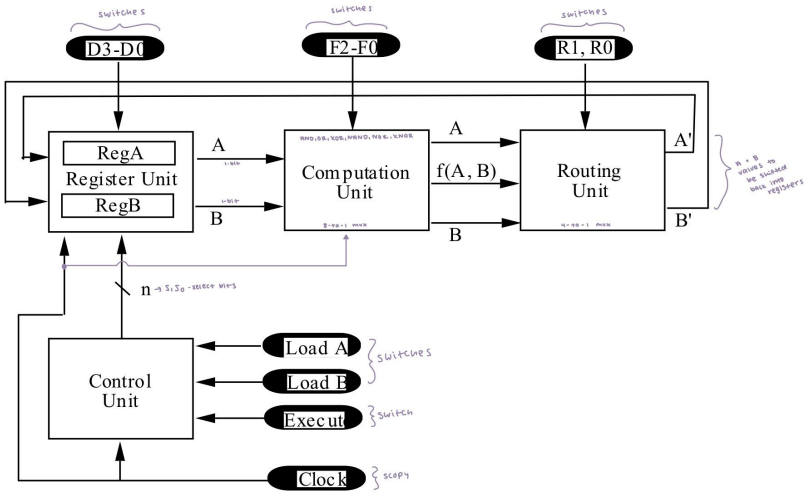


Figure 1: Block Diagram

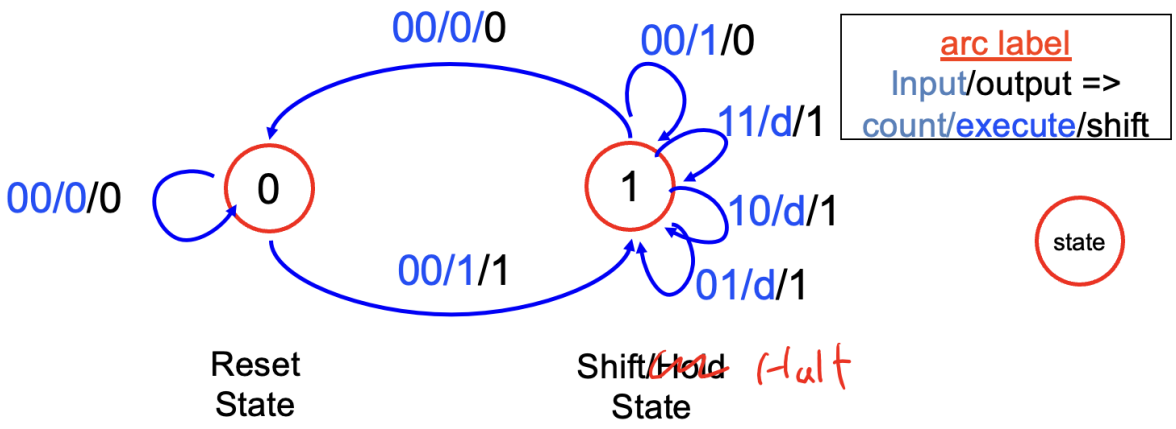
High level block diagram for the logic processor - includes the register unit, computation unit, routing unit, and control unit

In this high-level block diagram, we start with the register unit where we input our D0-D3 values through manual switches. Then, when we enable our ‘load A’ and ‘load B’ switches, the 4-bit input values are stored in their respective registers. Then, the most significant bit of each register is then sent to the computation unit, where a logical operation is performed on the bits. The operation to be performed is determined by the select bits F0-F2, which are also manual switches. Then, the answer to the computation is then passed to the routing unit, which consists of a 4-to-1 mux. This unit determines what values will be stored in the registers A and B, and this is determined by the select bits, R1 and R0. Finally, there is the control unit. This unit determines when the shifting and computing will begin based on an enable switch. It also counts the number of shifts that need to be done (in this lab it’s 4), and this works with the clock. The same clock is connected to the control unit, register unit, and computation unit. This clock comes from a square wave generated by our ADALM2000.

c. State Machine Diagram

i. *Did you use a Mealy or Moore machine?*

To create the logic processor for this lab, we used a Mealy finite state machine to control the operation of the register unit. We used a Mealy machine, which relies on both the current state and current inputs, over a Moore Machine, which solely depends on the current state, because it required fewer states while still being able to correctly implement the logic needed for the control unit to work properly. 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.



Mealy State machine used to create our control unit

Above is the Mealy state machine that we implemented in the control unit. The FSM depends on Q, E and C1C0. This machine implements the ideas that we start in their reset state, and will only start the bit shifting once the execute flip is switched (00/0/0 -> 00/1/1). Then once we start shifting, we will count 4 shifts to shift the existing values in the registers out and replace them with their new and respective values (this increases the count bits to count from 0 to 3, and the shift bit will always be high while shifting). Additionally, once the shifting begins, we want our execution bit of the machine to be "don't care" values, because no matter if execution is high or low, we don't want to stop in the middle of shifting. Finally, once our shifting is complete, we must turn off our execution switch to go back to the reset state so we can shift again if the execution switch is flipped to high (00/1/0 -> 00/0/0).

4. Design steps and schematic diagram

a. Written Procedure

COMPUTATION UNIT

For this lab, the first unit that we completed was the computation unit. Our approach to this was to start with each logical function that the processor could compute. Since our lab kit only contained NANDs and NOR gates, we used hex inverters to create the NAND and NOR operations. Knowing that multiple operations were possible, but that we only wanted 1 to be computed at a time, this fact indicated that we should use a mux to have only 1 of the solutions of our

computation to be the answer. We were also able to determine this from the truth table provided in the lab documentation, shown 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

In order to check if this unit worked, we set the inputs of A and B to switches and LEDs, as well as the output of the 8-to-1 mux to LED. Additionally, we set the select bits, F2-F0 to switches and LEDs as well. This way, we were able to test each possible combination of the inputs A and B with each possible logical operation.

ROUTING UNIT

The next unit that we were able to complete was the routing unit. In the lab manual, we were provided with the information that we have 3 possible options for what can be stored in registers A and B once the computation has been finished: value of A, value of B, or value of the solution to the computation (f(A,B)). Select bits were also provided in that their purpose is to determine what value will end in which register after the computation of each bit, shown in the truth table provided in the experiment shown below:

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

Truth table for routing unit

This truth table indicates that we need to use two 4-to-1 mux for the routing unit, where R1 and R0 are the select inputs that will determine the values of next state A and B registers. To implement this we use a 4-to-1 mux with R1 and R0 as switches, and the inputs being A, B, and f(A,B).

REGISTER UNIT

The register unit was implemented using 4-bit BIDIR shift register ships, one designated for register A and the other for register B. The designing of this unit did not require any truth tables or k-maps, as the building of it only required the connection of inputs and outputs from other parts of the circuit. The D0-D3 pins are connected to switches for both registers, and the output Q3 becomes the input to the computation unit. The output of the routing unit was the input for the DSR pin on this chip because these would control the new values that would be shifted into the registers. Finally, the S1 and S0, the select pins that decide if the registers will shift or hold, are determined from logic from the control unit.

CONTROL UNIT

The control unit required the most planning out of all the units, as it depended on both the current state and inputs, as well as needed to take the next state logic into consideration with our design as well. The control unit was also the key component in managing the functions of the circuit. It was tasked with directing the shifting, loading, and computation of 4-bit values stored in register A and B. 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.

To accomplish this, the control unit utilized a mealy finite state machine, as defined by a truth table in the lab documentation, shown below:

TABLE 1: Control unit state transition table using the Mealy state machine

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 truth table for Mealy machine that depends on current state and current inputs

The circuit operated using a 4-bit synchronous counter with an execute switch as inputs, the interconnection of which was determined through the use of Karnaugh maps based on the given mealy finite state machine (illustration of interconnections shown in part b of this section). The current state logic was obtained by employing a flip flop, which extracted the information from the Q value and generated the output S. Below is the K-map for the next state Q and its minimal SOP expression:

Q+					
	C1C0/EQ	00	01	11	10
	00	0	0	1	1
	01	X	1	1	X
	11	X	1	1	X
	10	X	1	1	X

K-map for next state Q

$$Q^+ = C_1 + C_0 + E'$$

The output S, determines whether the circuit should shift (1) or hold (0) and influence the values of S1 and S0 for both register A and B. These values decide the priority between parallel loading or shifting. By using the counting bits 'C0', 'C1' and 'Q' and the given truth table, we generated expressions from the Karnaugh maps, producing the output logic S. Below is the k-map and boolean expression for how to implement the S:

S					
	C1C0/EQ	00	01	11	10
	00	0	0	0	1
	01	X	1	1	X
	11	X	1	1	X
	10	X	1	1	X

K-map for S, the register shift

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

This allowed the circuit to determine whether to shift the bits in the register unit or stay in a hold state. With the S logic combined with the Load A and Load B switches, we further derived expressions to help the circuit decide whether to prioritize parallel loading of values or shifting the values within the register units. The dependency on S1 and S0 for the shift registers come from the datasheet of the chip. shown below:

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

Truth table from data sheet of shift register chips - shows the expected inputs to the S1 and S0 chips depending on if they are to start parallel loading or shifting

Below are the truth tables and k-maps for the S1 and S0 inputs of registers A and B, where these inputs are dependent on the load of each register as well as the shifting bit S.

A				
	S	LDA	S1	S0
	0	0	0	0
	0	1	1	1
	1	0	0	1
	1	1	1	1

truth table for shift select bits of register unit - register A

A_S1			
	LDA	0	1
	0	0	0
	1	1	1

K-map for register A's shift select chip - input S1

$$A_{S1} = LDA$$

A_S0			
	LDA	0	1
	0	0	1
	1	1	1

K-map for register A's shift select chip - input S0

$$A_{S0} = LDA + S$$

B				
	S	LDA	S1	S0
	0	0	0	0
	0	1	1	1
	1	0	0	1
	1	1	1	1

truth table for shift select bits of register unit - register B

$$B_{S1} = LDB$$

B_S1			
	LDA	0	1
	0	0	0
	1	1	1

K-map for register B's shift select chip - input S1

$$B_{S1} = LDB$$

B_S0			
	LDA	0	1
	0	0	1
	1	1	1

K-map for register B's shift select chip - input S0

$$B_{S0} = LDB + S$$

Written description of the design considerations taken (did you consider multiple implementations of the same circuit and the tradeoffs of each?

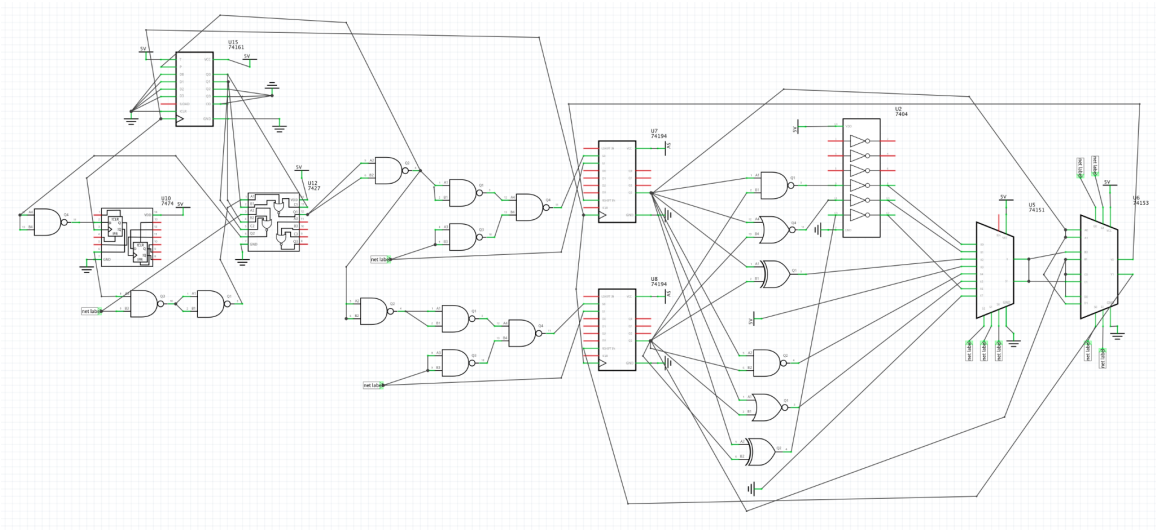
For this circuit, most of our initial design decisions were executed, though for some units there were multiple options on how to implement the logic. For our group specifically, we had to decide between using a 4-to-1 mux or an 8-to-1 mux for our computation unit. If we used a 4-to-1 mux, it would've required less wiring because we didn't need to consider the inverted versions of each logical operation. However, we would've needed to add an extra logical operation between F2 and the output of the 4-to-1 mux to use all select bits and account for the inverted operations. If you decide to use the 8-to-1 mux, you must compute all logical operations to be the inputs of the mux.

We decided to use the 8-to-1 mux approach for our computation unit. When we were thinking of our design choices it conceptually made more sense to us to use an approach where the select bits were used directly in the mux. However, looking back, the 4-to-1 mux approach would've been the better option, and it would've required less wiring and could've made debugging easier.

a. Detailed Circuit Schematic

Draw a gate level schematic of your circuit

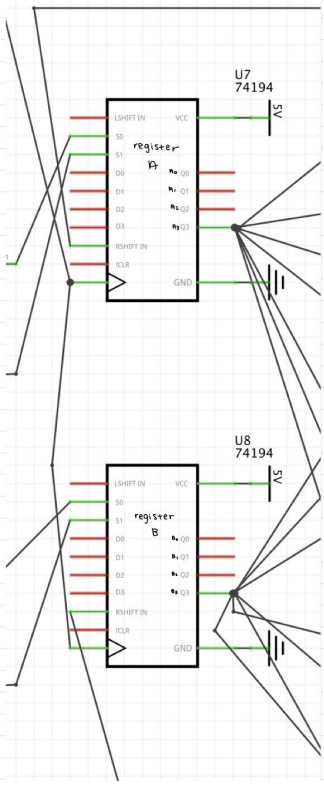
FULL CIRCUIT SCHEMATIC



Gate level schematic of logic processor

Above is the schematic for the entire logic processor circuit. From left to right, the circuit contains the control unit, register unit, computation unit, and routing unit. In this picture, we can see that the register unit, control unit, and computation unit all share the same clock, and each different unit has inputs/outputs that connect it to another unit. Let's go more into detail of each of these units and connections below.

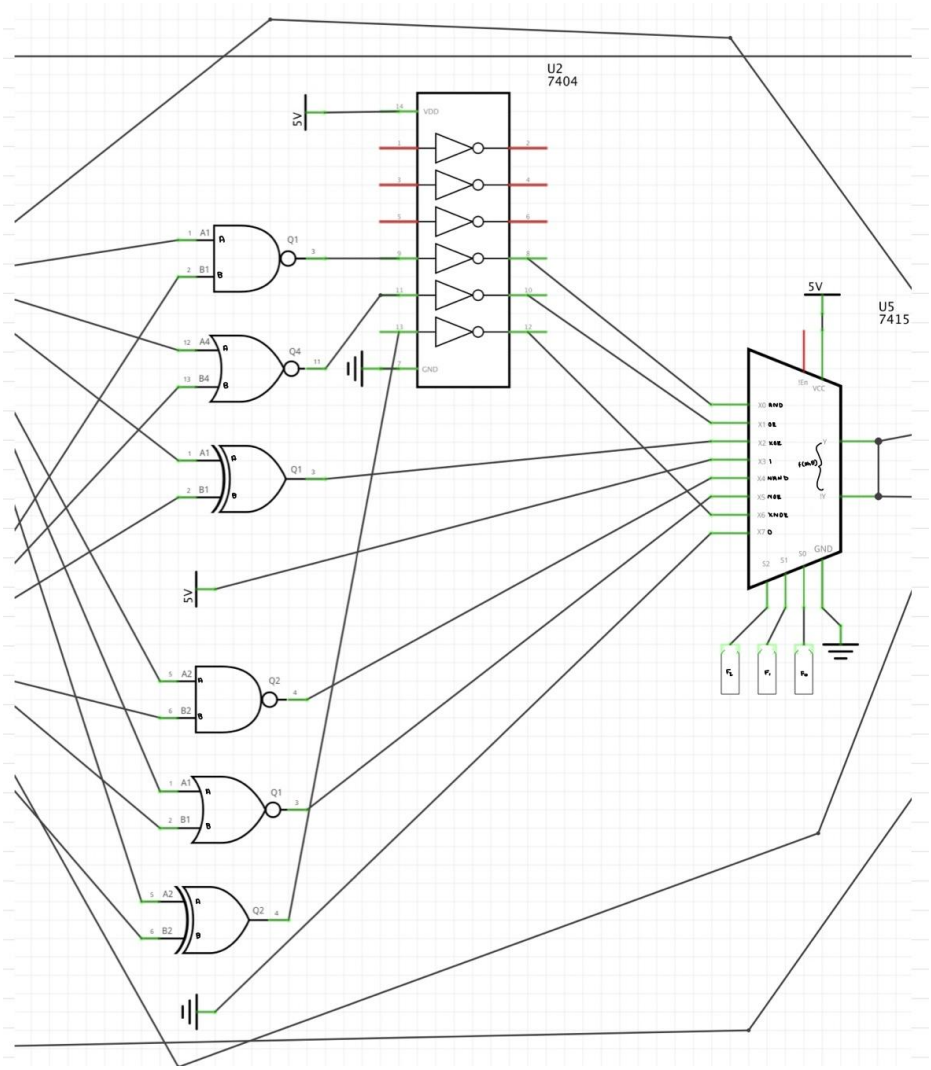
REGISTER UNIT



Register unit portion of the circuit - the unit that handles the shifting and values of the registers - (used the 194 chip provided in the Fritzing software, but the 195 chip was used in the actual hardware design)

This 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.

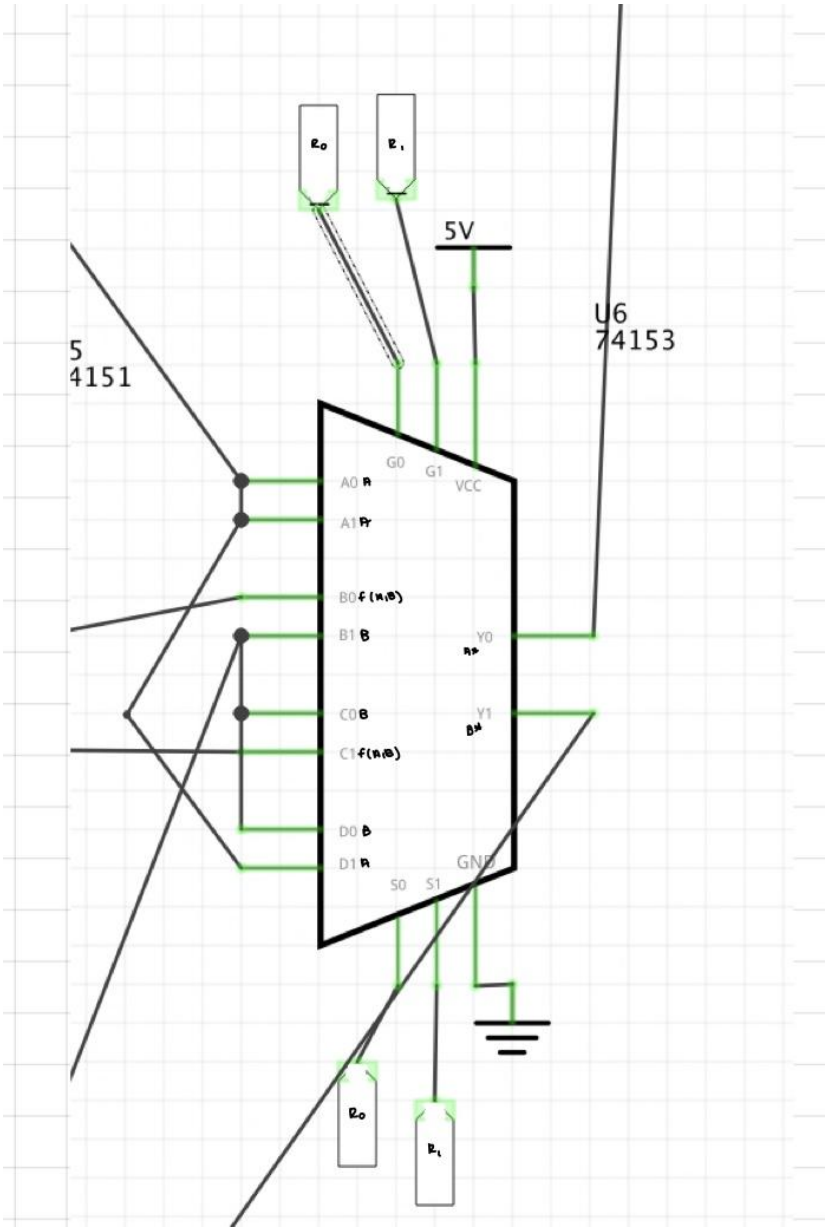
COMPUTATION UNIT



Computation Unit - does logic operations

This unit performs logical operations and the solution to each operation corresponds to the input of an 8-to-1 mux. The left side of this unit is connected to the register unit, and the right side is connected to the routing unit.

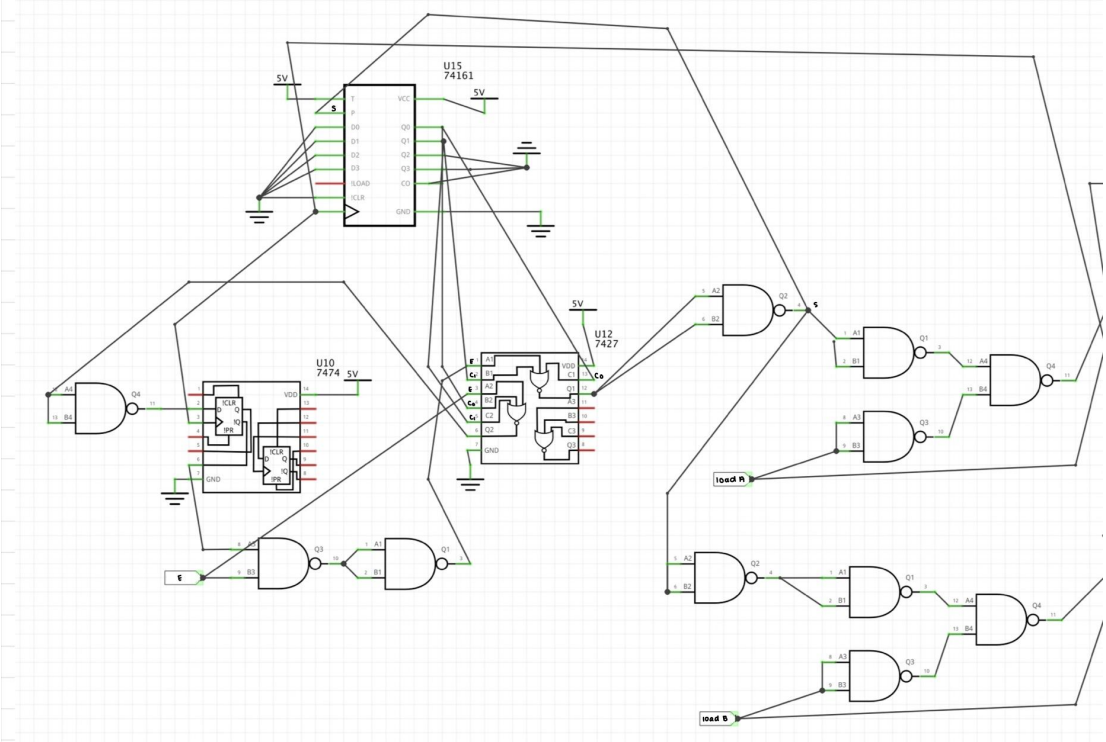
ROUTING UNIT



Routing Unit - chooses next state A and B and location of next sate value

The routing unit outputs the next A and B values that are to be shifted into each respective register based on the shared select values, R1 and R0, for the two 4-to-1 muxes. The left side of the routing unit is connected to the computation unit and the right connects back to the register unit.

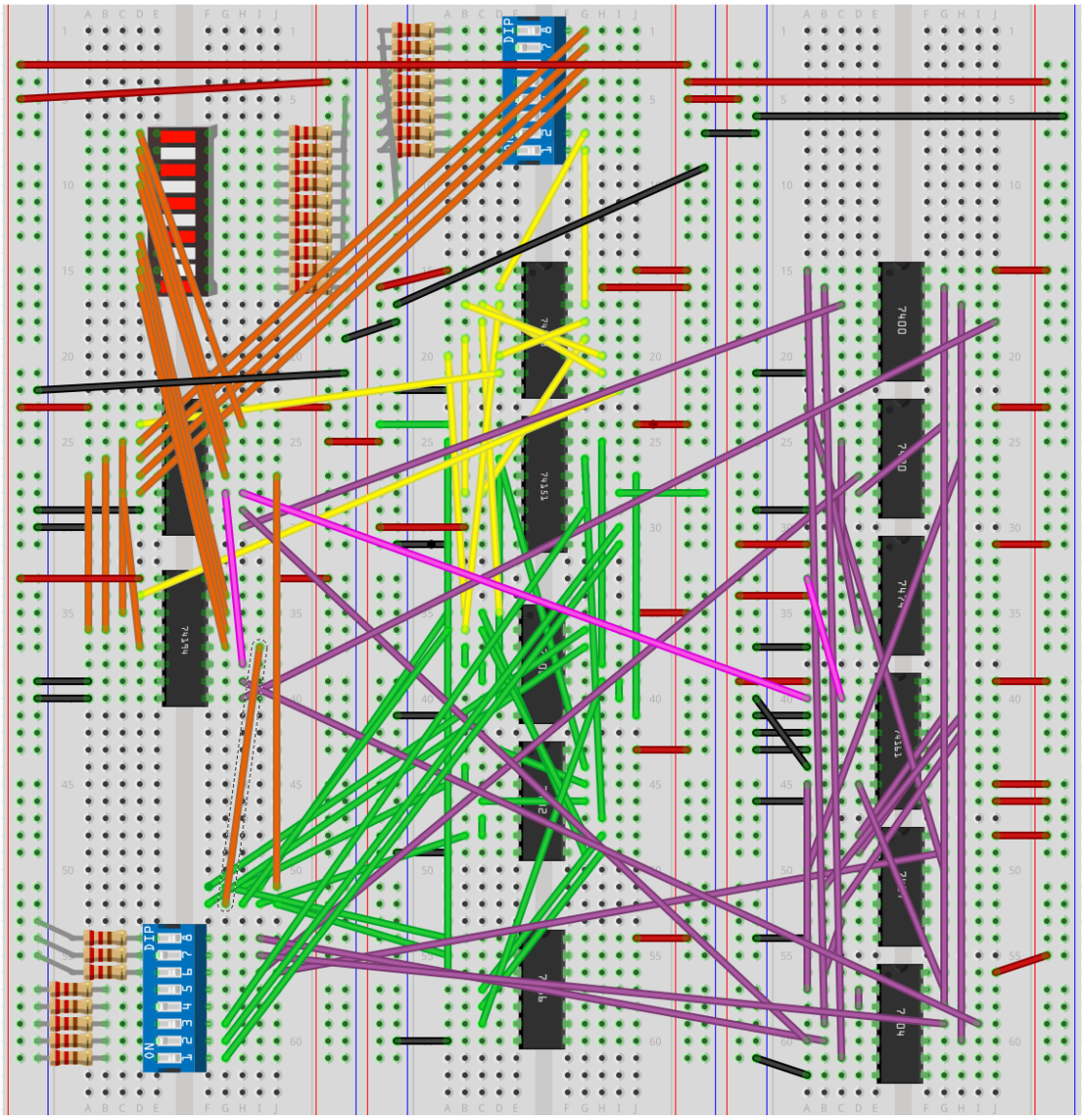
CONTROL UNIT



Control unit - initiates the execution of the shifting and computation unit - (used a 161 counter chip in the schematic but used the 163 when building circuit)

The control unit uses the execute switch, loads A and B, and a clock signal to control the start of the 4-bit shifting of each register as well as the computation of those bits. The right side of the unit is connected to the register unit by sharing the Load A and Load B inputs. The control unit also provides the S1 and S0 for each register, either through loads or through the combinational logic shown above to have S0 depend on both S and the respective load.

5. Breadboard view/Layout Sheet



Breadboard layout of logic processor: orange section: register unit, green section: computation unit, yellow section: routing unit, and purple section: control unit

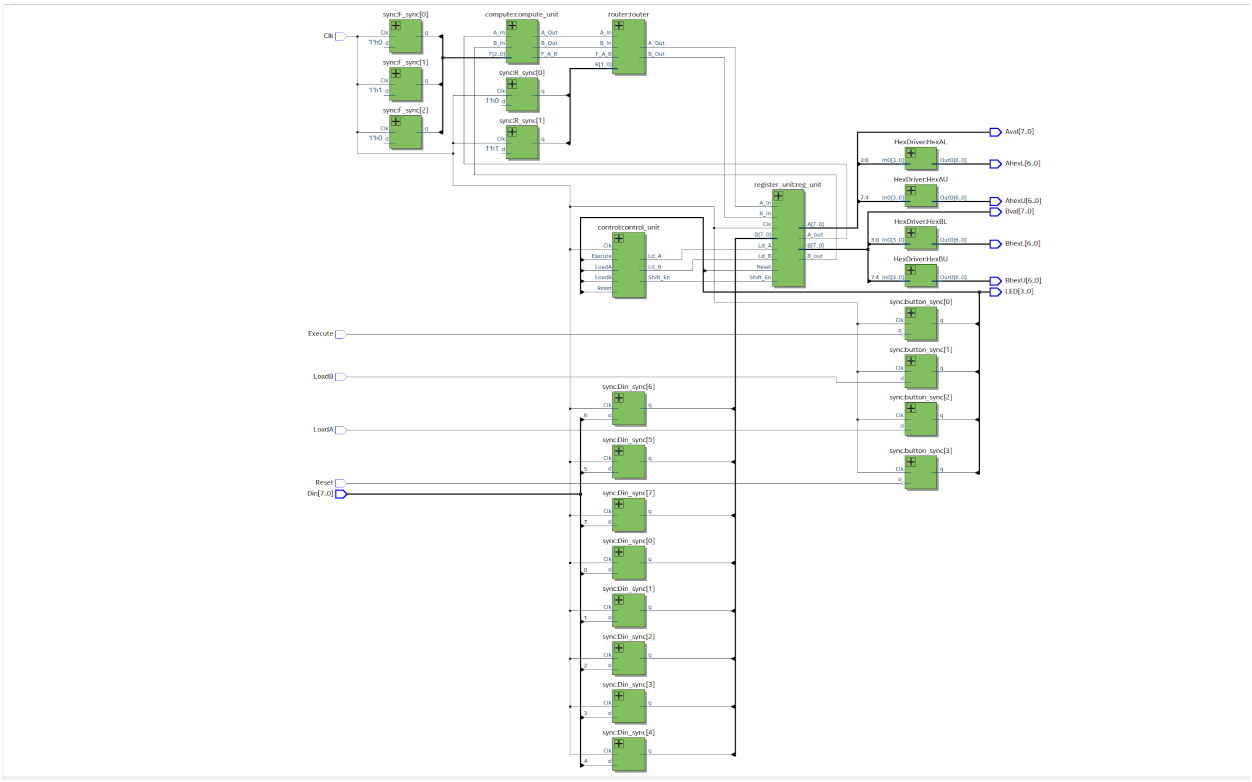
6. 8-bit logical processor on FPGA

a. *Summary of all .SV modules and the changes you made to extend the processor to 8-bits.*

   i. synchronizers.sv - no changes were made as this file's purpose is to create asynchronous signals for the FPGA and does not use any of the 8-bits used in the logic processor

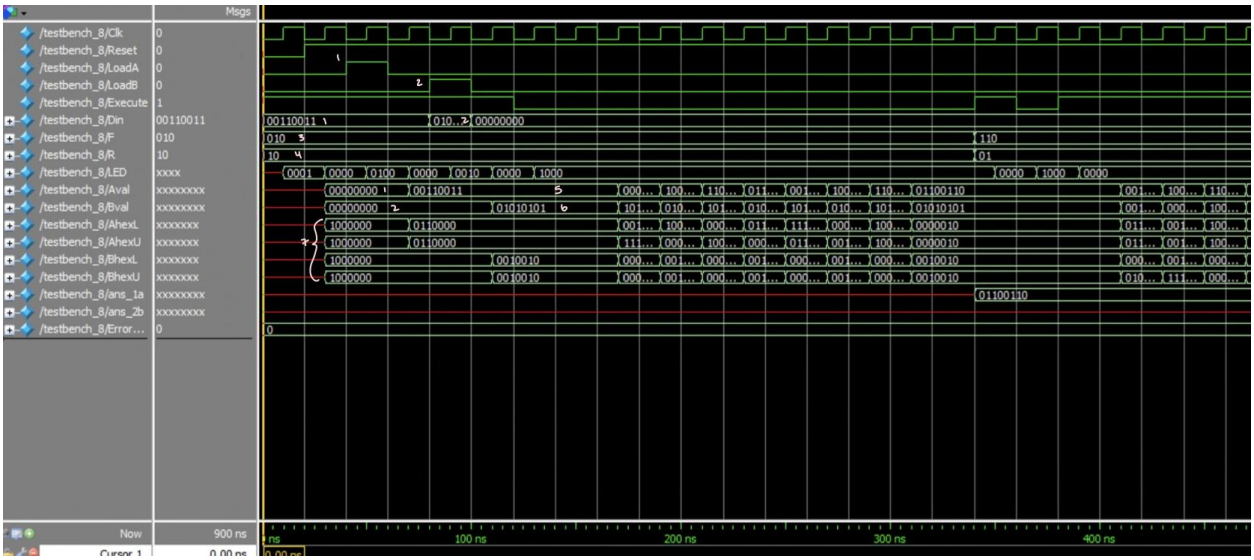
ii. router.sv - no changes were made in this file because this file is programming the specifics of the selects for the routing unit: determining what values will be put back into registers A and B
iii. register_unit.sv - for this file, the number of bits allocated for D, A, and B were all changed from 4 bits to now 8 bits. This is because ‘D’ is dealing with the input values, which is now 8 bits, and ‘A’ and ‘B’ are both outputs to an 8-bit input.
iv. Reg_4.sv - 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 ‘Data_Out’ assignment for the reset, making it have a size of 2 hexadecimal numbers, or in other words, have a size of 8 bits as well. Finally, we changed the size of the ‘Data_Out’ for when we are shifting our bits. We changed this to a size of 7 because as one bit shifts out, a new bit will be shifted in, meaning we need 1 bit less than the original number of bits in the register
v. processor.sv - ‘Din’ and ‘Aval’/’Bval’ were changed to operate on 8-bits rather than 4. Additionally, 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.
vi. Hex_Driver.sv - no changes were made to this file
vii. Control.sv - in this file we had to adjust the number of states for our state machine because we’re now dealing with 8 bits instead of 4 bits, so we had to add 4 more states to account for the additional 4 bits.
viii. compute.sv - no changes were made in this file because it’s programming the specifics of the selects for the computation unit: determining what logic operation should be performed

b. RTL block diagram



RTL block diagram generated in Quartus

c. *Include a simulation of a processor that has notes (annotations) that give information such as what operation is being performed, where the result was stored, etc.*



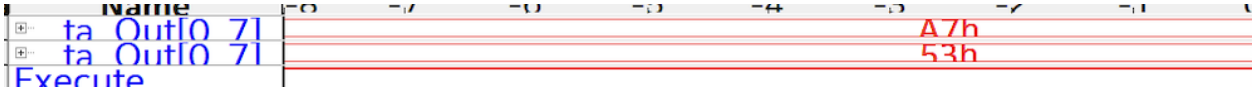
1. When load A is high, this allows for the value at Din to be loaded into register A. This will happen at the positive edge of the clock, but will happen one clock cycle after load A is high.
2. When load B is high, this allows for the value at Din to be loaded into register B. This will happen at the positive edge of the clock, but this will happen one clock cycle after load B is high.
3. This is our select value for the computation unit. This 3 bit value determines what logical operation will be performed. In this case, the XNOR operation has been selected.
4. This is our select value for the routing unit. This 2 bit value determines what values will be stored in the registers A and B once the operation is complete. In this case, A will hold its original input value and B will hold the output of the operation.
5. Aval is register A, holding the value of Din that was loaded in when load A was enabled. As you can see in this line, the value was not loaded until a clock cycle after load A was enabled. As you continue to move to the right, you can see the values of A changing as the bits start to shift right. At the end after all 7 shifts are completed, we can see that the value stored in A is its original value, 11001100.
6. Bval is register B, holding the value of Din that was loaded in when load B was enabled. As you can see in this line, the value was not loaded until a clock cycle after load B was enabled. As you continue to move to the right, you can see the values of B changing as the bits start to shift right. At the end after all 7 shifts are completed, we can see that the value stored in B the answer to the XNOR of A and B, 01010101
7. These contain the upper and lower hex values for the lights for each register to be displayed on the FPGA.

d. *Include procedure to generate SignalTap ILA trace, as well as the result of such trace executing the operation.*

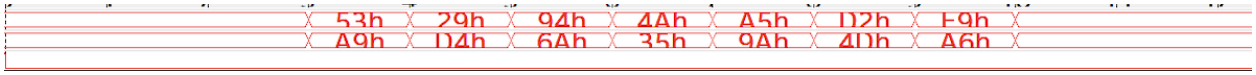
1. Step 1: hardcode the values for F and R in the processor.sv file because there are not enough switches on the FPGA to set these select values. In the example shown below, we have F = 3'b010, which means we want to perform the XOR function, and R = 2'b10, which means that the answer to the computation should be stored in register A and the value in register B should stay the same after execution.
2. Step 2: Next, we are going to go to Tools -> Signal Tap Logic Analyzer. Once there, we must set up the clock and then must add 3 nodes to our analyzer: reg_A|Data_Out[0:7], reg_B|Data_Out[0:7], and execute.

3. Step 3: Then, once you upload your lab file and run the analyzer, you can start to program your values into the FPGA. For this instance, you want to load value **8'hA7** into register A, and **8'h53** into register B. Once you hit execute, your signal tap should show the following:

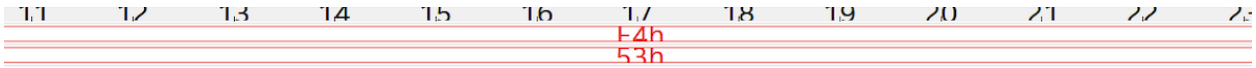
signal tap analyzer - 8'hA7 XOR 8'h53



Before execution: Register A holds 8'hA7 and register B holds 8'h53



The values in the registers while shifting



After execution: Register A holds 8'hF4 and register B holds 8'h53

We can conclude that our code works correctly from the output of the Signal Tap, as the XOR function was properly executed and the correct values were shifted into the registers once the execution was complete.

7. Description of all bugs encountered, and corrective measures taken.

During the fabrication of this 8-bit processor, several technical difficulties arose, primarily in the form of problems associated with the interconnections of the wiring. To ameliorate these complications and prevent further occurrences, a system of color-coded wiring was instituted to differentiate between the various units of the circuit. The Arithmetic Logic Unit was assigned a hue of green; the Routing Unit was allocated the color yellow, the Register Units were distinguished by orange wires, and the Control Unit was designated by the hue of purple.

Furthermore, issues arose in relation to the inactive ports present in the counter and flip-flop components that were integrated into the processor. To rectify this predicament, the utilization of color-coded wires was implemented, with red being utilized to denote power and black being designated for grounding purposes. Additionally, meticulous care was taken to ensure that any chip with an inverted bar labeled upon it was connected to a high-power source,

while any unused ports were either left floating or connected to the grounding source through a resistor via a black wire.

8. Conclusion:

a. Summarize the lab in a few sentences.

The 8-bit processor was engineered to execute a suite of six disparate bitwise operations on two separate 4-bit values. To accomplish this, the processor is equipped with a sophisticated configuration of components, including two registers, a routing unit, and an advanced Arithmetic Logic Unit. 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.

Furthermore, the processor incorporates a Control Unit, which acts as the central component of the circuit, coordinating and facilitating the execution of the computation process. This unit plays a crucial role in dictating the initiation of the computation process, as well as executing the various states defined by the implemented mealy state machine. The presence of this Control Unit results in a highly optimized and streamlined operation of the 8-bit processor.

b. Answer to all post-lab questions (they may be placed in conclusion or dispersed in more appropriate sections of the report).

Outline how the modular approach proposed in the pre-lab helps you isolate design and wiring faults, be specific and give examples from your actual lab experience:

The implementation of a modular approach in our circuit design facilitates the identification and isolation of design and wiring defects. This is achieved by dividing the circuit into its constituent components, namely the register units, the computation unit, the routing unit, and the control unit. By adopting a modular approach, we were able to physically separate these components on a breadboard, and to clearly differentiate their connections through the use of color-coded wires. Specifically, green wires were used for the computation unit, orange for the register unit, yellow for the routing unit, and purple for the control unit. Furthermore, light emitting diodes (LEDs) were utilized for debugging purposes, allowing us to verify the functioning of our chips by observing the inputs and outputs. The use of color-coded wires greatly aided in this process, particularly in the design and testing of the computation unit, where we could evaluate

the outputs of our bitwise operations by connecting them to the routing unit and observing the results displayed on the LEDs.

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 most basic circuit for controlling the inversion of a signal is an XOR gate. 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 was useful in the lab since the second half of our operations on the computation unit were NAND, NOR, XNOR, and 1111 which were the inverse operations of AND, NOR, XOR, and 0000. Using an XOR gate and plugging in the output of these gates into XOR gates would have taken less area on the breadboard and saved space, while inverting the output.

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

The modular design methodology promotes efficiency in testing by facilitating the isolation and examination of individual components within the circuit. Through physical separation of components on the breadboard, it becomes simpler to track the connections and assess input and output signals. The enhanced testing capabilities offered by the modular approach lead to reduced development time, increased component reusability, and improved project organization, resulting in a smoother overall development process.

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

The design process for our state machine came from the conceptual logic of the control unit. Knowing conceptual ideas like we only want the shifting to begin once the execute switch is flipped and that we want to only execute 4 shifts, we were able to start

implementing an FSM. For our project, we decided to implement a Mealy machine, as this machine was less complicated and required a lesser number of states to take into consideration when building the hardware design. The Mealy machine combines the shift and halt state, making the Mealy machine easier to implement.

Using a Mealy machine means that we would have to be able to make our outputs depend on both the current states and inputs, while the Moore machine outputs simply rely on the current state. Despite the conceptual simplicity of the Moore machine compared to the Mealy machine, for this circuit design, it was easier to implement the Mealy machine.

Both Modelsim and SignalTap take in inputs for registers A and B, and do a logical operation based on the F select bits, and store the correct values back into registers A and B depending on the selects to the routing unit. It may be more beneficial to use Modelsim if you want to be able to see all intermediate values and every input for the processor (ex. clock, load A and Load B, F selects, R selects, etc). In other words, Modelsim provides more information by showing all the inputs and outputs, whereas the SignalTap only shows the input, output and the values while shifting. This could be more beneficial because the SignalTap is easier to understand, and is useful if you only need to see the inputs to the operation and the outputs.

c. Make note of if there were any parts of the documentation which were unclear or otherwise need attention.

Not Applicable.