

ECE 385

Spring 2023

Experiment 2

A Logic Processor

Kevin Huang (kuanwei2), Steven Chang (sychang5)

TY 10:15 A.M.

Tianhao Yu

Introduction

This lab intends to be an introduction to the design of a bit-serial logic operation processor. This lab introduces the use of 4-bit shift registers, multiplexers, binary counters, and finite state machines in the first part of the experiment. The goal is to design a logic processor that will perform one of the eight logic operations(AND, OR, XOR, NAND, NOR, XNOR, Set to 1, and Set to 0) to the 4 bits value in A and B registers and then restore the value in A and B registers depending on the routing unit. The second part of the lab is an introduction to System Verilog, which allows designers to assign logic and components to the FPGA board in hardware coding language and simulate the result by using test benches.

Operation of the Logic Processor

The operation of the logic processor starts with loading A and B into the registers. To begin with, the 4-bit value has to be set either with switches or hardwiring. Then, the switch for the Load A signal has to be flipped in order to parallel load the 4-bit value into register A. Similarly, to load B into the register, the 4-bit value has to be set, and the Load B switch has to be flipped. Afterward, the selection inputs F and R have to be set by flipping switches or hardwiring in order to perform the computation shown in Figures 1 and 2. Finally, to start the computation, the execute switch must be flipped, and then the processor will go through a cycle computing the operation on each bit of the 4-bit value. In the end, when the computation is done and the execute switch is flipped back to the original position, the processor will then return back to the original state where it can perform another operation when execute signal is on again.

Written Description of Circuit and Lab Documentations

The circuit of the prelab is to design a 4-bit logic processor that will perform one of eight logic operations to the bits in Register A and B depending on the F signal in the computation unit. Afterward, the R signal in the routing unit will determine whether to store the computation result or A or B back in the register. Not to mention, the control unit will use a finite state machine with input signals, including execute, load A, and load B, and a binary counter to determine the number of bits the processor has processed and whether the processor should start or stop the operation.

Design Steps

I. Computation Unit

A. Computation Unit Logic Signals

As indicated in the computation unit description, there are 8 possible computation unit outputs, which require the right-shift output bits A and B, of the registers A and B, respectively, to generate. The following signals need to be fed into a 8-to-1 multiplexer detailed further below before the proper output of the computation unit is selected by the 3-bit selection input F.

1. $A \cdot B$ (AND)
2. $A + B$ (OR)
3. $A \oplus B$ (XOR)
4. 1111 (V_{cc})
5. $\overline{A \cdot B}$ (NAND)
6. $\overline{A + B}$ (NOR)
7. $A \odot B$ (XNOR)
8. 0000 (Ground)

To implement the above logic, we needed to rewrite a few of the logic expressions for the above possible computation outputs into the following form in order to implement it in our circuit using NAND, NOR, NOT, and XOR gates.

1. $\overline{\overline{A \cdot B}}$ (AND implemented using a NAND gate and a NOT gate)
2. $\overline{\overline{A + B}}$ (OR implemented using a NOR gate and a NOT gate)
3. $A \oplus B$ (XOR implemented using an XOR gate)
4. 1111 (connected to V_{cc})
5. $\overline{A \cdot B}$ (NAND implemented using a NAND gate)
6. $\overline{A + B}$ (NOR implemented using a NOR gate)
7. $A \odot B$ (XNOR implemented using an XOR gate and a NOT gate)
8. 0000 (connected to Ground)

These generated possible computation unit outputs are then connected to the 8-to-1 multiplexer as described below.

B. 8-to-1 Multiplexer Selecting Computation Unit Output

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 1. Computation Unit Logic Table

First of all, in the computation unit, the 3-bit selection input F2-F0 will determine the output of the computation unit, in other words, the logic operations to perform on A and B as shown in Figure 1. To implement this input and output relation, the computation unit is designed with a single 8-to-1 MUX in order to choose one of the eight functions only using a 3-bit selection signal. The selection input F2-F0 will be manually set using a switch in our circuit.

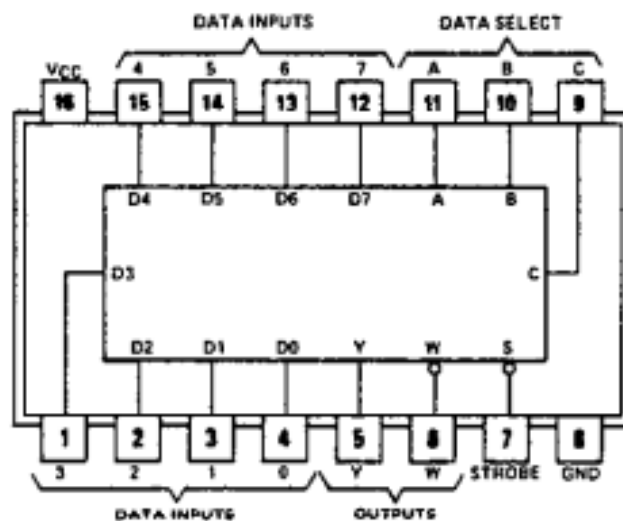


Figure 2. 8-to-1 Multiplexer Datasheet

We selected the 74151N 8-to-1 multiplexer with the above schematic. Selection input signals F2, F1, F0 are connected to pins 11 (A), 10 (B), and 9 (C), respectively of the chip, while A

AND B is connected to pin 7 (D7), A OR B is connected to pin 6 (D6), and so on, with pin 4 (D0) connected to ground to represent 0000. Pin 5 of the chip, or the output Y, then becomes our computation unit output $f(A, B)$. As with all IC chips, Vcc and GND pins need to be connected, to 5V input and Ground, respectively. Strobe, or pin 7 of the chip represents the enable signal of the multiplexer, which needs to be set to Ground as well as the the chip is active-low.

II. Routing Unit

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

Figure 3. Routing Unit Logic Table

With the computation unit output generated successfully generated by our computation unit, our signals A, B, and $f(A, B)$ are then passed into the routing unit, where the 2-bit selection input R1-R0 will determine which signals A* and B* to pass back into registers A and B, respectively, according to the truth table above. To implement this input and output relation, the routing unit is designed with two 4-to-1 MUXes in order to choose one of the four outputs to feed back into each register only using a 2-bit selection signal. The selection input R1-R0 will be manually set using a switch in our circuit.

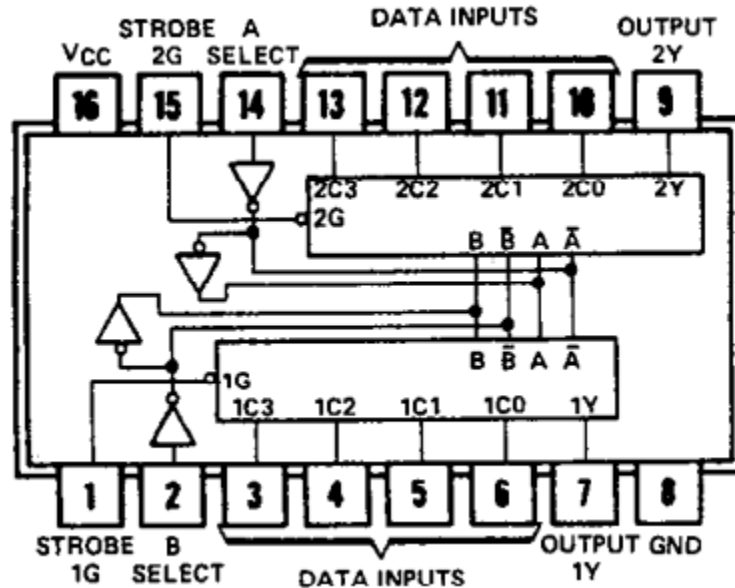


Figure 4. Dual 4-to-1 Multiplexer Datasheet

We selected the 74153N dual 4-to-1 multiplexer with the above schematic. Selection input signals R1, R0 are connected to pin 2 (B Select) and pin 14 (A Select) respectively of the chip. The computation unit output $f(A, B)$, as well as A and B outputs from the right-shift of registers A and B, respectively, are utilized in the routing unit. B, $f(A, B)$, A, and \bar{A} are connected to pins 3, 4, 5, and 6, respectively, meaning the bottom 4-to-1 multiplexer output from pin 7 represents the output A^* of the routing unit that will be the right-shift input to register A of our register unit. A, B, $f(A, B)$, and \bar{B} are connected to pins 13, 12, 11, 10, respectively, meaning the top 4-to-1 multiplexer output from pin 9 represents the output B^* of the routing unit that will be the right-shift input to register B of our register unit. As with all IC chips, Vcc and GND pins need to be connected, to 5V input and Ground, respectively. The Strobe inputs, or pins 1 and 15, represent the enable signal of the multiplexer, which needs to be connected to Ground as well as the chip is active low.

III. Control Unit

The control unit forms the crux of our circuit design. It helps control and dictate the shifting of the registers. The following inputs are utilized in the control unit: Load (L) signals of the registers A and B, respectively, Execute (E) signal to indicate that we are ready to enter the next computation cycle, as well as the Clock (CLK) signal generated by our M2K. The Load and Execute signals will be generated manually using switches. Additional intermediate input and output signals within the control unit include the state representation Q and its next state Q^+ , as well as the counter count C1C0 to keep track of the progress of our computation cycle (i.e. the number of shifts performed). These signals will be used to control the state of control unit and generate the critical Shift (S) signal that tells the register unit whether the registers should perform right shifts. Hence, the output of the control unit is the register Shift (S) signal.

As described in the lab manual, we wish for our circuit to perform 4 right shift operations that is initiated by a high Execute (E) signal. This execute signal will transition us from a reset state into a shifting state, in which regardless of the execute signal being high or low, will send out a high Shift (S) signal to our registers to indicate that we need to right shift four times. This signal will continue for four clock cycles or four rights shifts. We will then transition into a halt state if our execute signal still remains high. This is to prevent us from entering a second, third, or fourth, and so on computation cycle. Only once the execute signal drops low will we transition back to the original reset state thus completing one complete cycle of our logic operation.

The following state diagram is derived based on the stated requirements. We form a Mealy Finite State Machine diagram as shown below, with inputs Execute (E), C1C0 of the counter, and state bit Q. The machines reliance on inputs to dictate output makes our machine a Mealy machine rather than a Moore machine, which would have output only depend on current state.

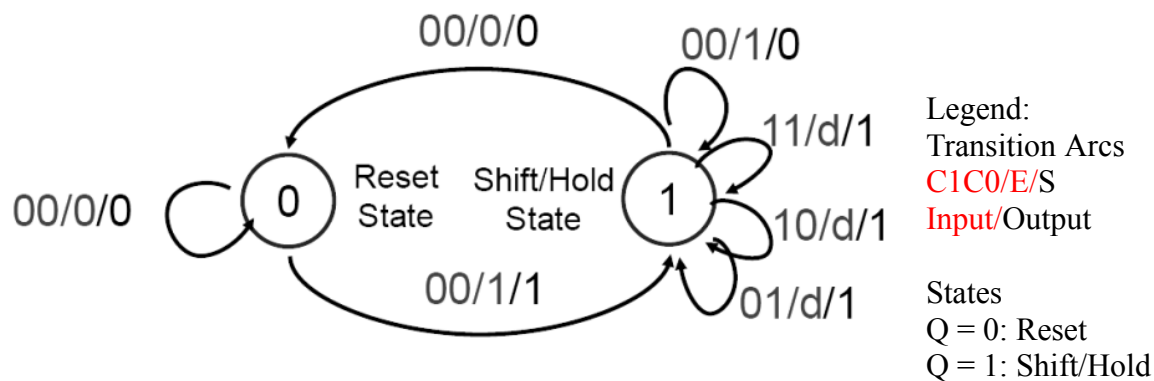


Figure 5. Mealy State Machine Design

The above Mealy machine diagram has two possible states, the reset state, characterized by the flip-flop state representation value $Q = 0$, and the shift/hold state, characterized by the flip-flop state representation value $Q = 1$. The arcs denote the inputs and the outputs of our state machine, the first two bits represent the count, or C1C0 of the counter, the third bit represents the execute signal, and the last bit represents the output Shift (S) signal. As can be seen, we transition from the reset state to the shift/hold state as Execute (E) is switched to high. This results in our output Shift (S) signal being 1. The bottom three self-connecting arcs then represent the second, third, and fourth shift of our shift registers, as our execute signal can be disregarded and the counter counts from 01 to 10 to 11. During these transitions, the Shift (S) signal will be high to indicate to our shift register to continue right shifting. Once four right shifts are complete, then our counter we will go into the fourth self-transition arc at the very top, where our counter should go to 00 and halt and Shift (S) signal drops down low to indicate to the register that we have completed four right shifts and should halt. This transition will continue to occur while our Execute (E) signal is high. Finally, once our Execute (E) signal is dropped down low, we will then transition to the reset state, thus completing one full computational cycle.

As can be seen, the important signal that needs to be generated each clock cycle by our finite state machine, the Shift (S) signal. As aforementioned, Shift (S) signal instructs the registers to

perform a right shift when it is logic high. On the other hand, the Q+ signal that helps keep track of the current state/next state of our finite state machine, which is important in deriving the Shift (S) signal can be derived based on the current input output combinations.

Combining the counter count C1C0 that keeps track of the number of shifts performed and our input signal Execute (E) that indicates a desire to enter a computation cycle, as well as current state of our FSM, the following truth table can be derived to help us transform our machine into a physical circuit design. For invalid or don't care cases such as when our counter counts without the execute switch being on while remaining in the reset state, we can use don't care values 'd' to simplify our truth table and subsequent k-maps.

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

Figure 6. S/Q+ Signal Truth Table

Q+ Signal K-Map		EQ			
		00	01	11	10
C1C0	00	0	0	1	1
	01	d	1	1	d
	11	d	1	1	d
	10	d	1	1	d

Figure 7. Q+ Signal K-Map

Shift (S) Signal K-Map		EQ			
		00	01	11	10
C1C0	00	0	0	0	1
	01	d	1	1	d
	11	d	1	1	d
	10	d	1	1	d

Figure 8. S Signal K-Map

Based on the truth table and subsequently generated K-Maps above, the logic expressions for the signals Shift (S) and Q+ can be derived as follows.

$$S = C0 + C1 + E \cdot \overline{Q} \quad Q+ = E + C0 + C1$$

To implement the above logic, we needed to rewrite our logic expressions for S and Q+ into the following form in order to implement it in our circuit using NAND, NOR, and NOT gates.

$$S = \overline{\overline{C0 + C1 + E} + Q} \quad Q+ = \overline{\overline{E + C0 + C1}}$$

In other words, the Shift (S) signal will be generated as follows: invert the Execute (E) signal and NOR the result with Q. The signal will then be triple NOR with signals C0 and C1 of the counter, before being inverted to generate Shift (S). On the other hand, next-state Q+ will be generated as follows: triple NOR the Execute (E) signal as well as the signals C0 and C1 of the counter, then inverting the result to get Q+. As will be mentioned in the flip-flop section of the report, the Q+ signal will then be passed as input into the positive-edge-triggered D flip-flop, while the Shift (S) signal will be used alongside the Load signals of registers A and B, respectively, to generate the register control signals S1 and S0, which will also be detailed further below.

A. 74163E 4-Bit Synchronous Binary Counter

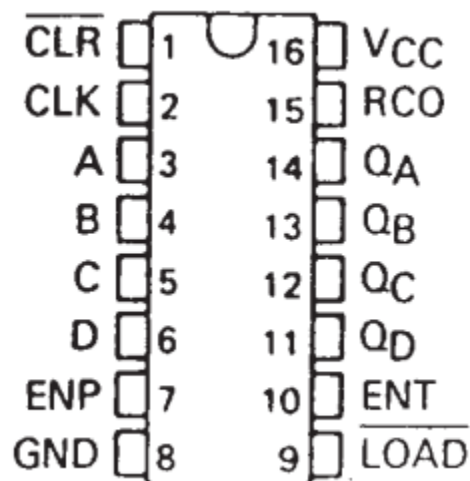


Figure 9. 74163E 4-Bit Synchronous Binary Counter Datasheet

For the purpose of our design to limit each computation cycle to perform 4 right shifts in our registers, we incorporate the use of a binary counter to keep track of the number of right shifts performed that will help us decide whether to keep shifting our registers. We selected the 74163E 4-bit synchronous binary counter for our design, although alternative counters may be used. The important features required of our chosen counter is the ability to retrieve the lower 2 bits of our current count, since we want to track 4 cycles (00, 01, 10, 11), and the ability to halt our count when needed (i.e. when we are in our reset or halt states). Based on the schematic above, we want to connect CLR (pin 1) to Vcc, or 5V, as we do not need to clear the counter at any point in time and the clear signal is active low, for reasons that will be discussed further below. CLK or (pin 2) will be connected to the clock signal generated from our M2K. Load (pin 9) will be connected to Vcc, or 5V, as we do not need to load any data into the counter and the load signal is active low. Based on the datasheet, the counter will count while ENP and ENT are both high. Since the purpose of our counter is to keep track of the number of shifts performed, we want to wire ENP and/or ENT to the Shift (S) signal. For slight ease of implementation, we simply connected ENP to the Shift (S) signal and had ENT fixed at Vcc, or 5V. The important signals generated by our counter is Q_A and Q_B , which represent C0 and C1, respectively, the signals that keep track of the current count of our computation cycle. C1C0 should cycle through 00, 01, 10, 11, repeatedly in consecutive computation cycles without the need to clear the counter during our halt or reset states because with a 4-bit counter (able to halt) and 4-shift computation cycle, $Q_DQ_CQ_BQ_A$ will become 0100 once we exceed 0011, and 1000 the next time around after another four counts and so on until overflow and reset to 0000, which means that we can continue the count in the next computation cycle without needing to worry clearing the counter. As with all IC chips, Vcc and GND pins need to be connected, to 5V input and Ground, respectively. The remaining pins can be left floating, as they are unused in our circuit design.

B. 7474 Dual D Positive Edge Triggered Flip Flop

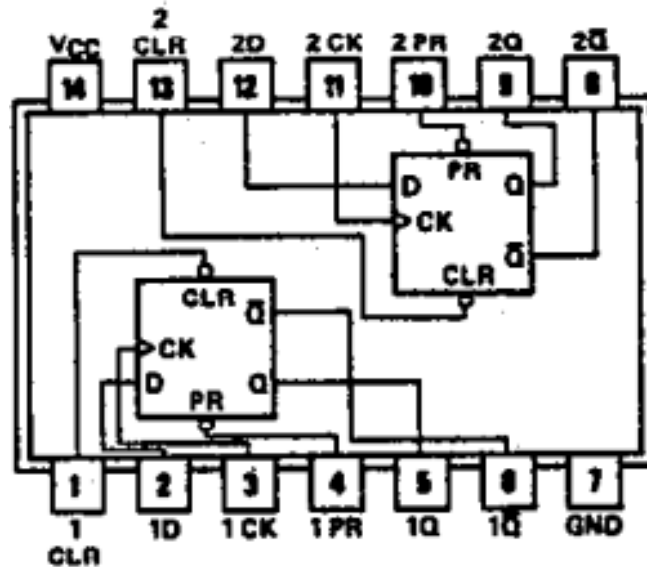


Figure 10. 7474 Dual D Positive Edge Triggered Flip Flop Datasheet

For the purpose of our design to store and calculate Q and its next state Q^+ , we only need to utilize one D-Type positive-edge-triggered flip-flop in order to update Q at the positive edge of each clock cycle. Per our board implementation, we utilize the bottom flip-flop in the schematic. Based on the schematic above, since clear is active low, we need to set the pin 1 clear signal to V_{cc} , or 5V, to avoid clearing our flip-flop at any point in time. Pin 3 (1CK) will need to be connected to our clock signal generated by the M2K. Pin 4 (1PR) will also need to be connected to V_{cc} , or 5V, as it is an active low preset, which will set our flip-flop into a state, which we do not need. Pin 2 will be connected to the Q^+ signal that we generated above. Q^+ represents the next-state Q signal, which will be passed through the flip-flop to replace the current Q value at the next positive edge of the clock. Pin 5 outputs the current value of Q , which is used above to generate the correct Shift (S) and next-state Q^+ signals of our control unit. As with all IC chips, V_{cc} and GND pins need to be connected, to 5V input and Ground, respectively. The remaining pins can be left floating, as they are unused in our circuit design.

IV. Register Unit

A. 74194A/E 4-Bit Bidirectional Universal Shift Register

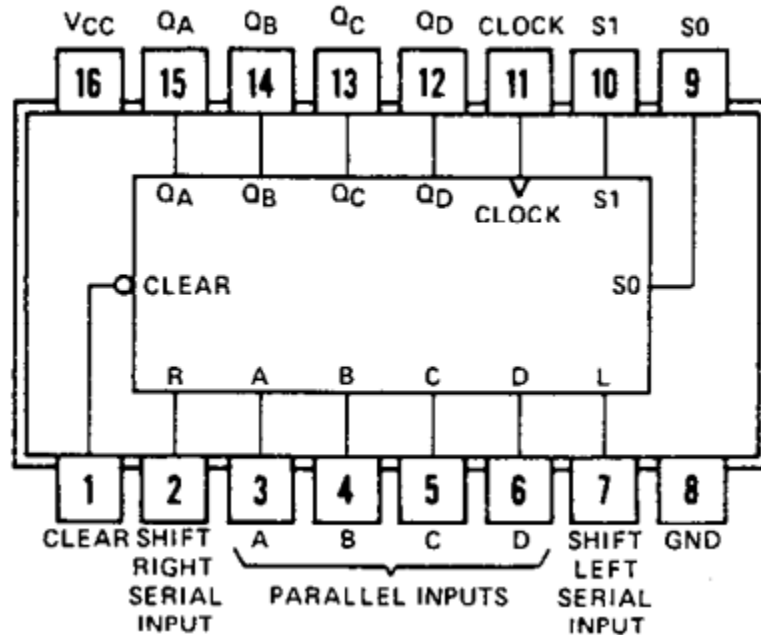


Figure 11. 74194A 4-Bit Bidirectional Universal Shift Register Datasheet

For our design, we utilized the 74194A/E 4-Bit Bidirectional Universal Shift Register. Important features of this register that make it an optimal choice for our design is its ability to perform parallel loads as well as halt the shifting of its contents when required. This is necessary because we want to parallel load data from input switches into the registers prior to performing the 4-shift computation cycle. The halt feature is crucial to us because we do not want our register to perform an infinite number of shifts. For the purpose of our design, we simply want to complete one computation cycle, which requires 4 right shifts to be performed once the Execute (E) signal is generated by our switch. The register will then halt after completion until the Execute (E) signal is lowered to enter the reset state. Another Execute (E) signal high will then restart a computation cycle.

The A, B, C, and D parallel inputs will be connected to switches representing the parallel load data into the registers, D0-D3. The shift right serial input (pin 2) of registers A and B will be connected to A* and B*, respectively. The CLOCK signal (pin 11) will be connected to our clock signal generated by the M2K. The register control signals S1 and S0 will be connected to the output of the combinational logic discussed below utilizing the Shift (S) signal and Load signals of each register. Q_D (pin 12) of the register represents the signals A and B of their respective registers, or the right-shift outputs that will be used by the computation and routing units as discussed above. The CLEAR signal will be connected to Vcc, or 5V, as it is an active low and we do not need to clear our register of its contents at any point in time, as we can simply parallel load new data into the registers without clearing the original contents. As with all IC chips, Vcc and GND pins need to be connected, to 5V input and Ground, respectively. The remaining pins can be left floating, as they are unused in our circuit design.

With general characteristics of the shift register in mind, we note the following behaviors of the control signal S1S0. When S1S0 are both low (00), the register is in its halt state, not shifting and not parallel loading. When S1S0 are low-high (01), the register performs a right shift. When S1S0 are high-low (10), the register performs a left shift, and when S1S0 are both high (11), the register performs parallel load. With this in mind, we derive the following truth table. We want the register to be in its halt state when neither the Shift (S) signal or the Load (L) signal are high. We want the register to perform a right shift when its Shift (S) signal is high and Load (L) signal is low. We also want our register to perform a parallel load when our Shift (S) signal is low and Load (L) signal is high. A point to note here is that we made the design choice of having the register control signals as don't cares when both our Shift (S) signal and Load (L) signal is high, as we made the logical assumption that we will not purposely load new data into the registers during a computation cycle. However, if needed, additional logic can be used to prevent errors from occurring if the Shift (S) signal and Load (L) signal were to ever be high at the same time.

Shift (S)	Load (L)	S1	S0
0	0	0	0
0	1	1	1
1	0	0	1
1	1	d	d

Figure 12. S1/S0 Truth Table

Register S1 Signal K-Map		Shift (S) Signal	
Load (L) Signal		0	1
	0	0	0
	1	1	d
Register S0 Signal K-Map		Shift (S) Signal	
Load (L) Signal		0	1
	0	0	1
	1	1	d

Figure 13. S1/S0 Signal K-Map

Based on the truth table and subsequently generated K-Maps above, the logic expressions for the signals S₁ and S₀ can be derived as follows.

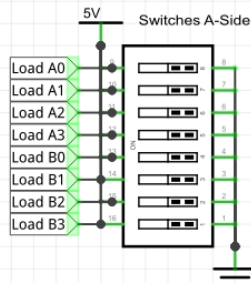
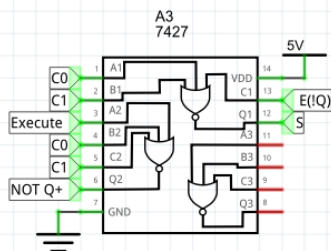
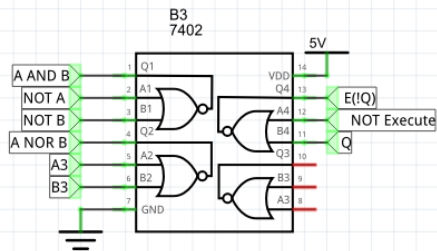
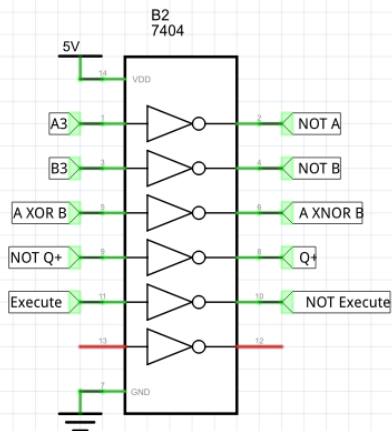
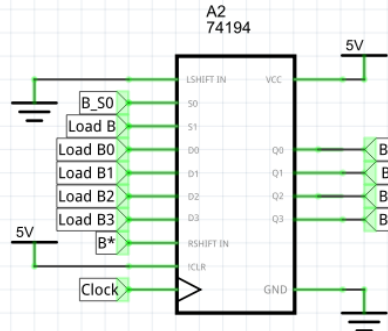
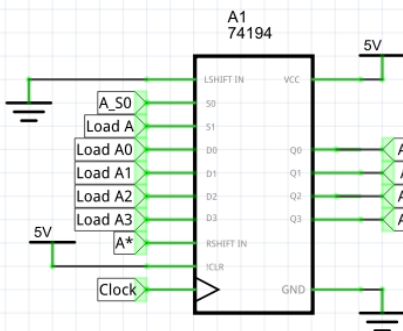
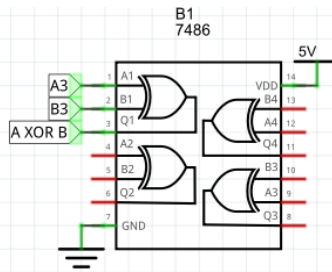
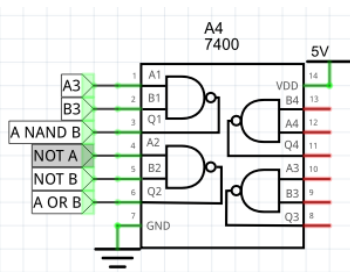
$$S_1 = L \quad S_0 = S + L$$

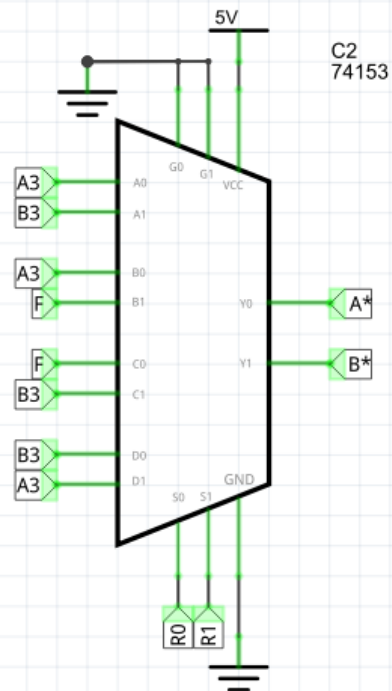
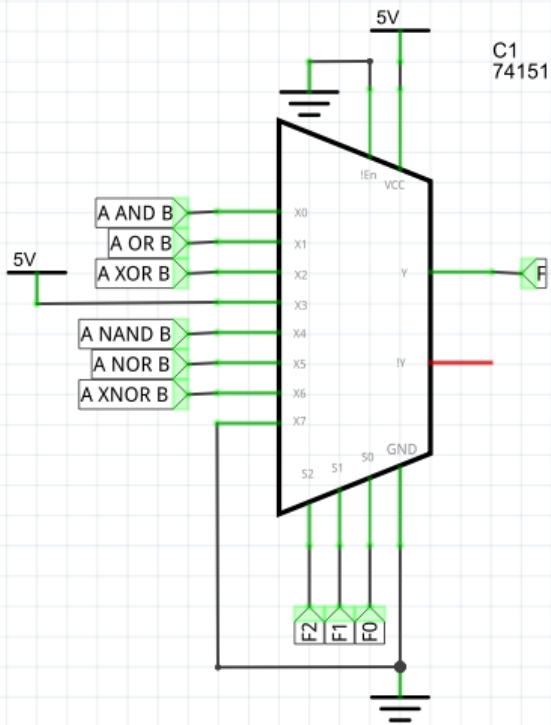
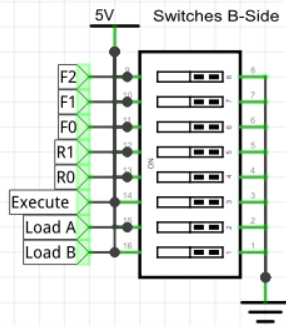
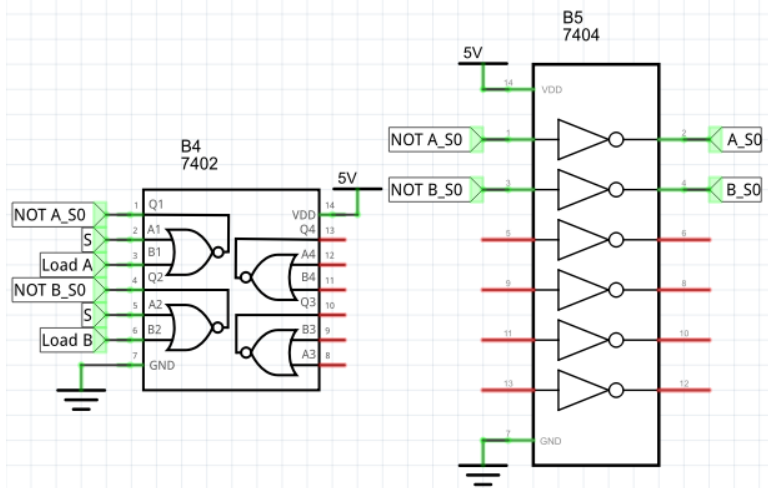
To implement the above logic, we needed to rewrite our logic expressions for S₁ and S₀ into the following form in order to implement it in our circuit using NOR and NOT gates.

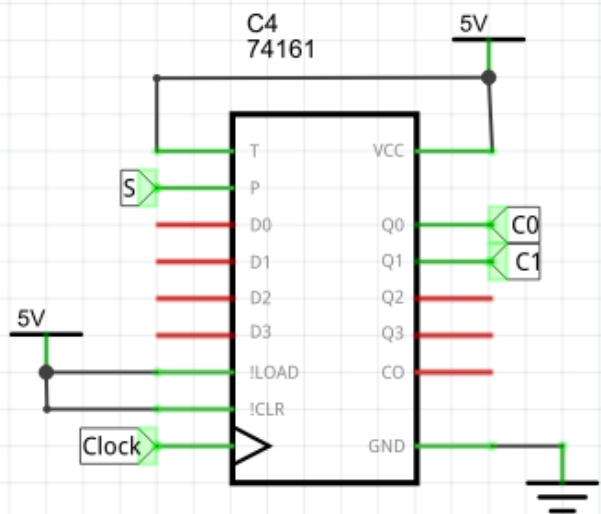
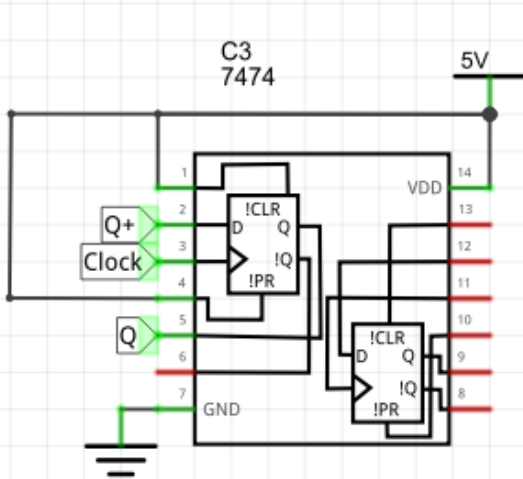
$$S_1 = L \quad S_0 = \overline{\overline{S + L}}$$

In other words, the S_1 signal of the shift register will simply be the Load (L) signal; Load A for register A and Load B for register B. The S_0 signal of the shift register will be the NOR of the corresponding Load (L) signal of the register along with the Shift (S) signal generated by the control unit, then inverted by passing it through a NOT gate.

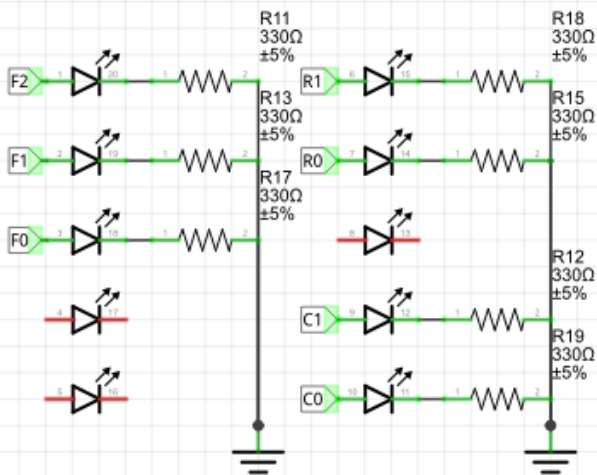
Logic Diagrams



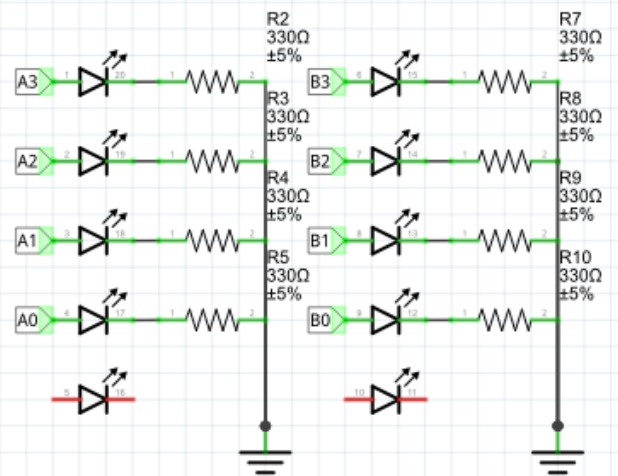




LEDs B-Side

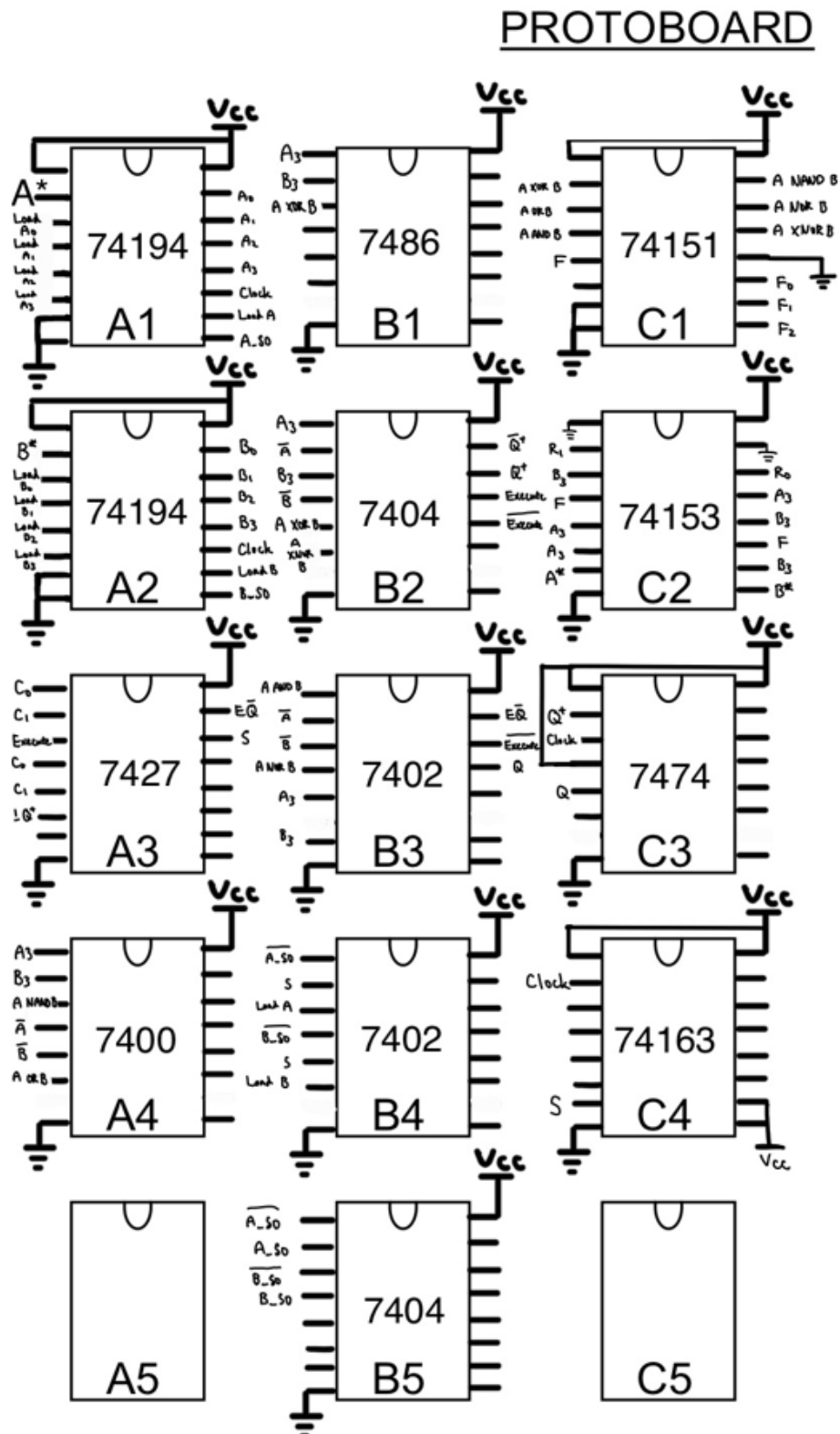


LEDs A-Side



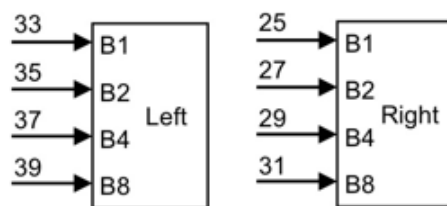
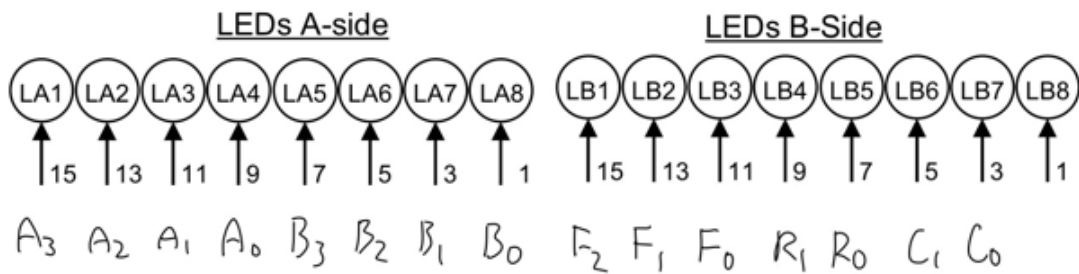
Component Layout

Protoboard:

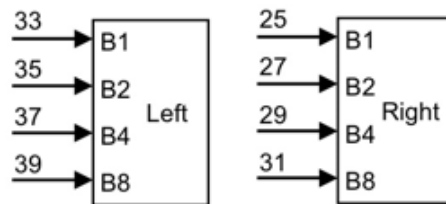


I/O Board:

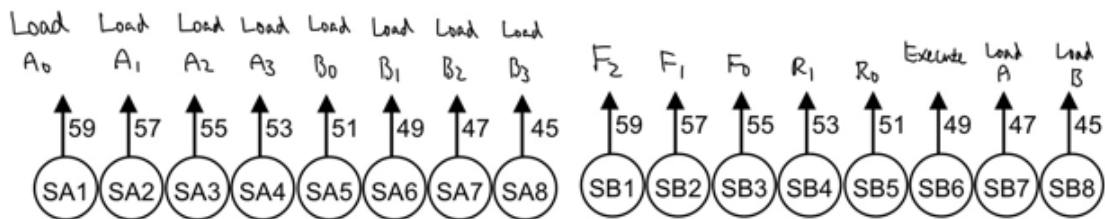
16-bit I/O BOARD



HEX A-side



HEX B-side



SWITCHES A-side

SWITCHES B-side

8-Bit Logic Processor on FPGA

SV Modules Summary

Module: reg8.sv

Inputs: [7:0] D, Clk, Load, Reset, Shift_In, Shift_En

Outputs: [7:0] Data_out, Shift_Out

Description: This is a positive-edge triggered 8-bit register with synchronous reset and synchronous load. When the Load is high, data is loaded from Din into the register on the positive edge of Clk. Otherwise, if Shift_En is high, the left-most 7 bits will be shifted to the right with Shift_In as the most significant bit.

Purpose: This module is used to create the registers that store operands A and B in the register unit and allow the right shift during the process.

Module: compute.sv

Inputs: [2:0] F, A_In, B_In

Outputs: A_Out, B_Out, F_A_B

Description: This is the computation unit that takes in 3-bit selection input F to compute one of the eight operations(AND, OR, XOR, NOR, NAND, XNOR, 1, 0) to A_In and B_In. Then the result will be stored in F_A_B while A_Out and B_Out remain the same value as A_In and B_In.

Purpose: This module is used as the computation unit of the logic processor to perform the specific operation on A and B determined by the F selection input.

Module: Router.sv

Inputs: [1:0] R, A_In, B_In, F_A_B

Outputs: A_Out, B_Out

Description: This is the routing unit that takes in 2-bit selection input R to determine which value of the inputs A_In, B_In, and F_A_B to be stored in the outputs A_Out and B_Out.

Purpose: This module is used as the routing unit to determine which input values have to be stored in the shift in bits(A_In and B_In) of A and B.

Module: Register_unit.sv

Inputs: clk, Reset, A_In, B_In, Ld_A, Ld_B, Shift_En, [7:0] D

Outputs: A_Out, B_Out, [7:0] A, [7:0] B

Description: This is the register unit that creates register A and B based on the input values in order for the user to shift, load values, and reset the registers.

Purpose: This module is used as the register unit to create registers A and B.

Module: Control.sv

Inputs: clk, Reset, LoadA, LoadB, Execute

Outputs: Shift_En, Ld_A, Ld_B

Description: This is a positive-edge triggered flip-flop that has 10 states, including reset state (A), 8 shifting states (B, C, D, ..., I) and halt state (J). When the Reset is high, the flip-flop will return to the reset state. Otherwise, it will continue to the next state. Also, it will assign the next state to the current state based on the condition. When Execute is high and the current state is the reset state (A), it will make the next state B. While it is in shifting states, it will unconditionally make it move on to the next state. Last but not least, when it is in the halt state (J), it will make it

return back to A only when Execute is set back to low. Not to mention, to determine the Ld_A, Ld_B, and Shift_En, it will only allow LoadA and LoadB to pass through when it is in the reset state. Otherwise, it will have Ld_A and Ld_B be 0. Similarly, Shift_En will only be high when the current state is in the shifting states.

Purpose: This module is used as a control unit to determine when to shift the bit, when to reset the registers, when to load A and B into the registers, and when to start executing the logic operations.

Module: HexDriver.sv

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: This is the Hex driver that takes in 4-bit value In0 from the register in order to determine which segment of the 7-segment LED has to be light up and store the value into 7-bit output Out0 with 0 meaning turning light on and 1 meaning turning light off.

Purpose: This module is used to show the value in registers A and B in the LED on the FPGA board by having 7-bit output indicating which segments of the LED have to be turned on.

Module: Synchronizers.sv

Inputs: clk, Reset, d

Outputs: q

Description: These are the synchronizers that use positive-edge-triggered flip-flops to set the output value q to the input value d at every rising edge of the clock. During the rising edge of Reset signal, the synchronizer with reset to 0 will turn q to 0 while the synchronizer with reset to 1 will turn q to 1.

Purpose: This module is used as the synchronizer for bringing asynchronous signals into the FPGA board.

Module: Processor.sv

Inputs: clk, Reset, LoadA, LoadB, Execute, [7:0] Din

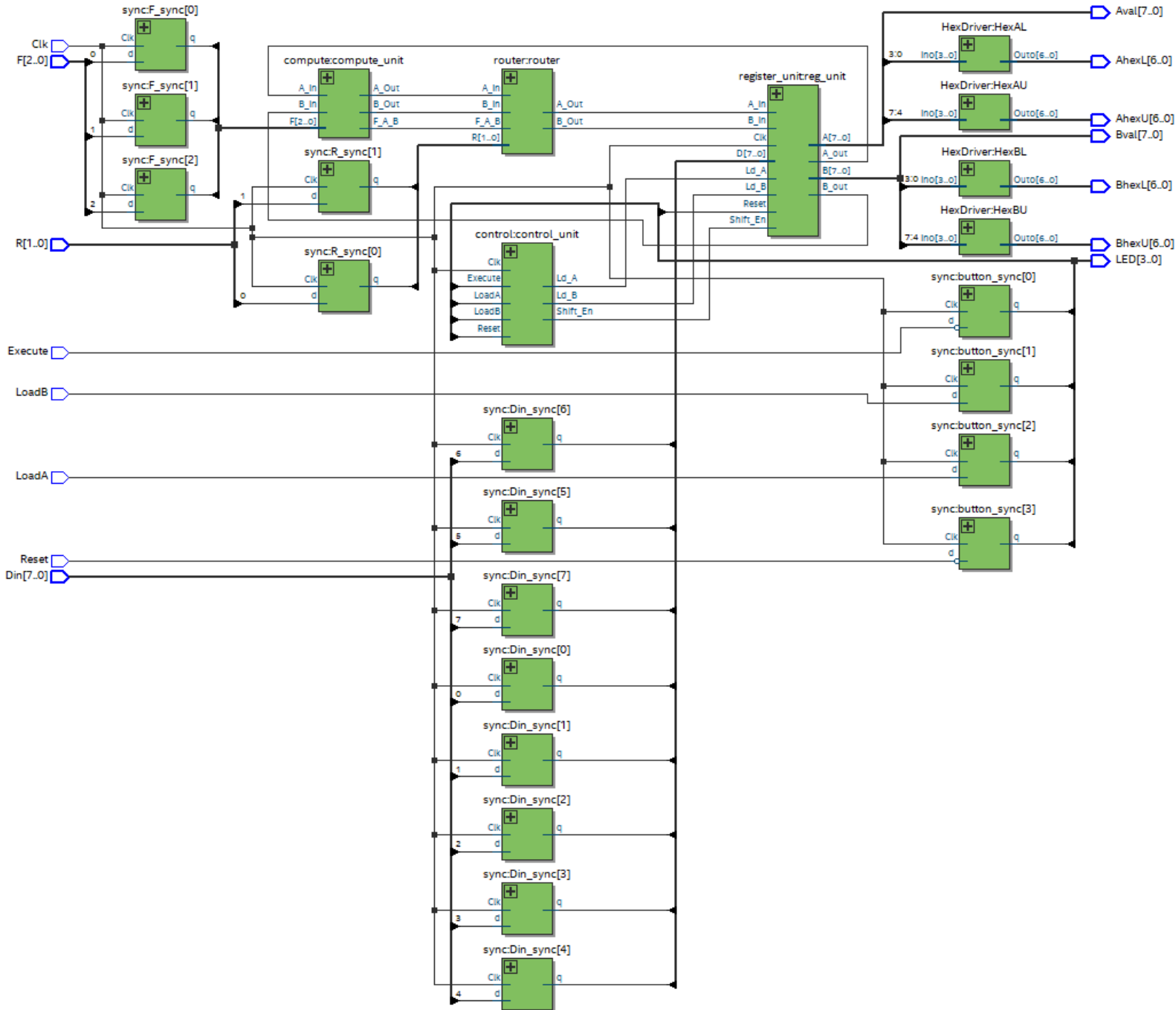
Outputs: [7:0] Aval, [7:0] Bval, [3:0] LED, [6:0] AhexL, [6:0] AhexU, [6:0] BhexL, [6:0] BhexU

Description: This is the entire logical processor where each input values are used to initiate all the modules that are created and described above and organize all the modules into a single logic processor. Not to mention, this is also where F and R values are hardwired.

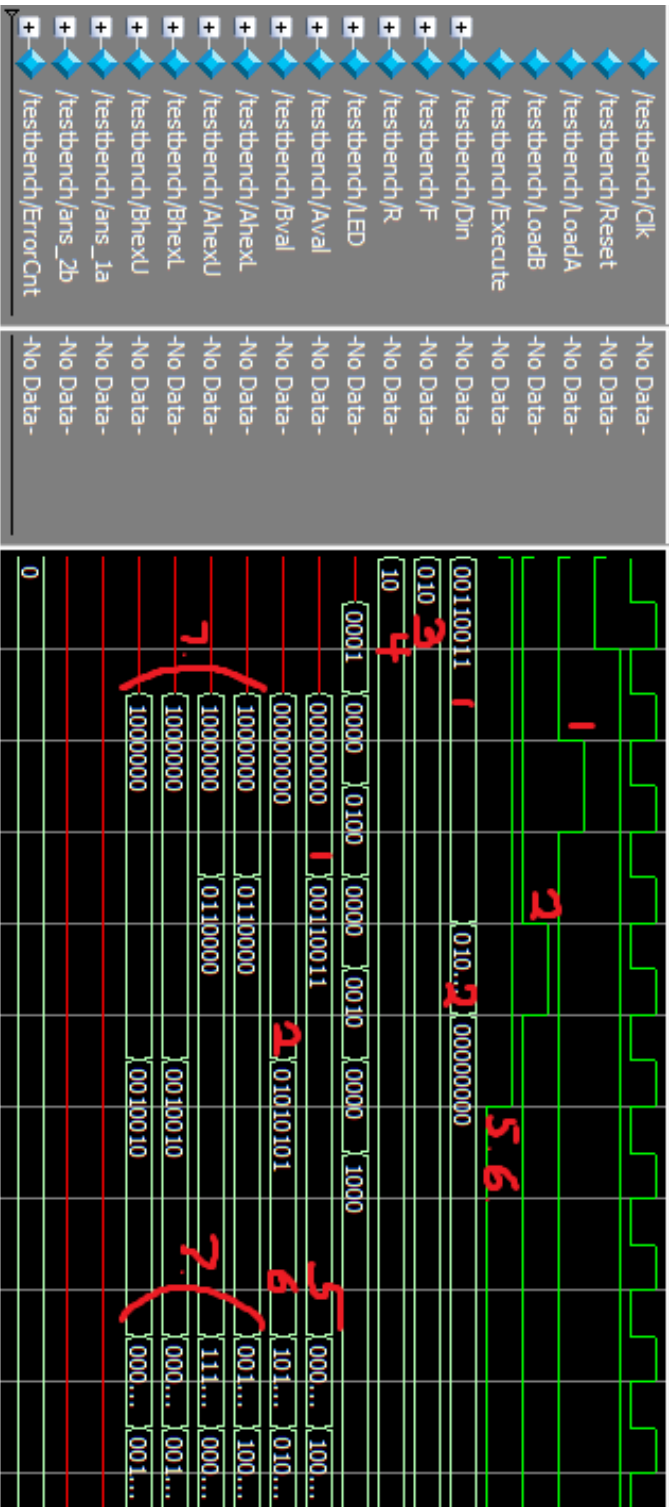
Purpose: This module is used to combine all the modules into a single logic processor.

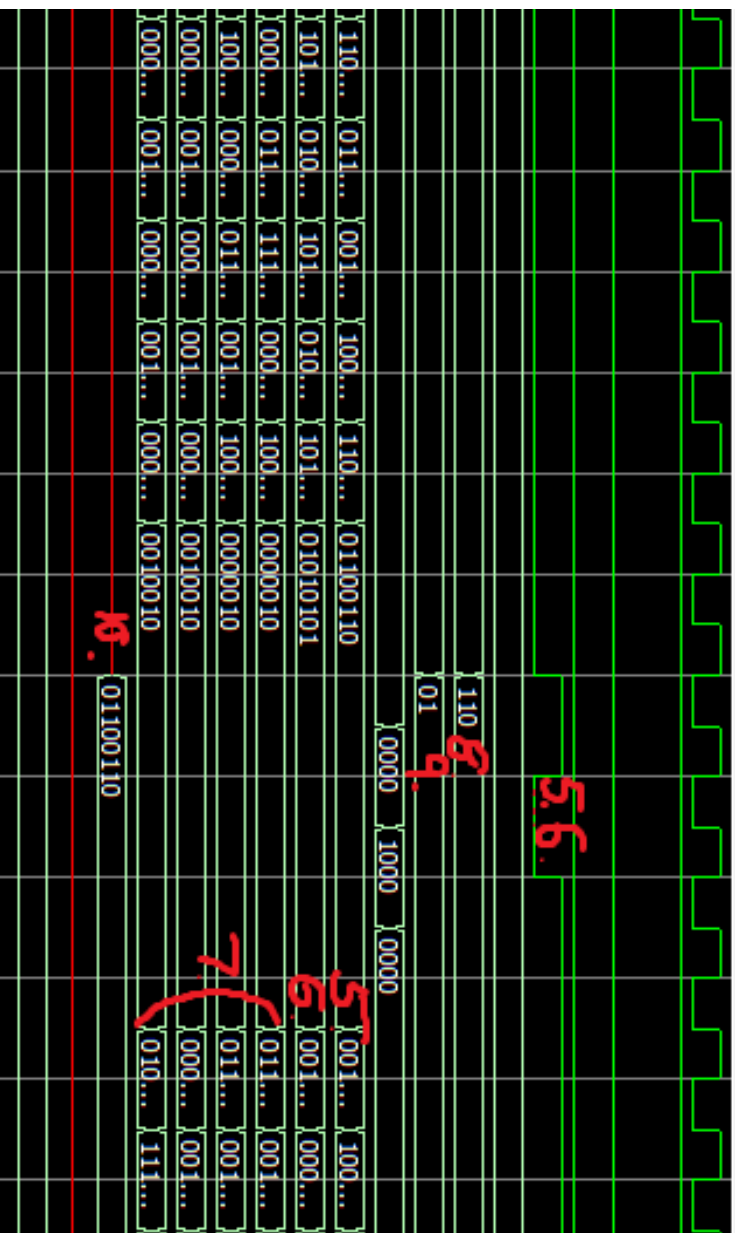
To extend the processor to 8 bits, the D and Data_Out in register.sv are changed from 4 bits into 8 bits. Also, the D, A, and B in the Register_unit.sv are changed from 4 bits into 8 bits. In addition, the number of shifting states in Control.sv are changed from 4 states into 8 states in order for the processor to perform operations for all 8 bits from A and B. Last but not least, the Aval, Bval, and Din in Processor.sv are modified to 8 bits in order to store 8-bit values in registers A and B.

RTL Block Diagram



Processor Simulation



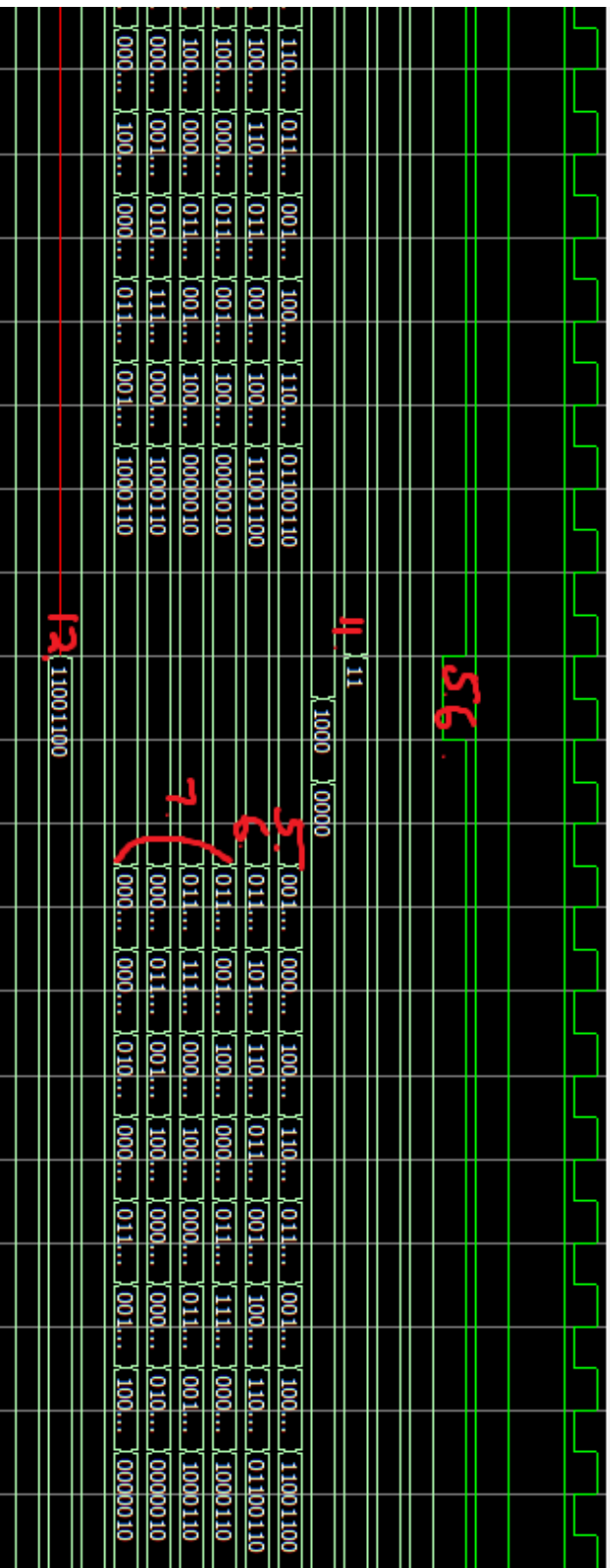


7. The AhexL, AhexU, BhexL, and BhexU values represent the hexadecimal values necessary to output the current values contained within the registers A and B to the seven-segment display.

8. Fis now set to 110, whichs mean that A XOR B will be the computation unit output once we enter a computation cycle.

9. R set to 01, which means that output A* of the router unit will be the right shifted out value A of the register A, while the output B* of the router unit will be the computation unit output. These two signals will be the right shift inputs of their respective registers in the computation

10. The result of the first computation cycle contained in register A is stored in answer_1a.



R set to 11, which means that output A^* of the router unit will be the right shifted out value B of register B, while the output B^* of the router unit will be the right shifted out value A of register A. These two signals will be the right shift inputs of their respective registers in the computation cycle.

The result of the second computation cycle contained in register B is stored in answer_2b.

Signal Tap Trace

Step 1: Hardwire '010' into F and '01' into R in Processor.sv in order to perform A XOR B and store the result in register B.

Step 2: Open the Signal Tap Logic Analyzer in Quartus

Step 3: Set Clk signal as the clock signal and set it with sample depth of 64

Step 4: Set 8 bits Data_Out from both registers A and B as triggers

Step 5: Group 8 bits Data_Out from each register in order to see 8-bit value together

Step 6: Set Execute signal as a trigger with the condition of either edge.

Step 7: Start compilation and program the FPGA board




Step 8: Click Run Analysis in Signal Tap Logic Analyzer

Step 9: Flip the Load A switch on the board and load 8'hA7 into A then flip the Load A switch again to stop loading A

Step 10: Flip the Load B switch on the board and load 8'h53 into B then flip the Load B switch again to stop loading B

Step 11: Push the Execute button on the board to start the computation

Step 12: Once the process is finished, the signal for each trigger will be available to see on Signal Tap Logic Analyzer

Type	Alias	Name	-8	-6	-4	-2	0	2	4	6	8	10	12	14	16	18	20		
		...g_unit/reg_8:reg_A Data_Out[0..7]	A7h												A7h				
		 ...g_unit/reg_8:reg_B Data_Out[0..7]	53h												F4h				
		Execute																	

Post-Lab Questions

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 circuit that we can build to optionally invert a signal is to simply use an XOR gate, assuming we can control a signal that determines whether to invert the other signal. Say we have signal 2 as the control signal. While signal 2 is 0, the XOR output will simply be what signal 1 is, as $1 \text{ XOR } 0 = 1$, and $0 \text{ XOR } 0 = 0$. On the other hand, say we have signal 2 as 1, the XOR output will be signal 1 inverted, as $1 \text{ XOR } 1 = 0$, and $0 \text{ XOR } 1 = 1$. This is useful for this lab because we can simplify our circuit design of the computation unit, reducing the need of an 8-to-1 mux down to a 4-to-1 mux with an XOR gate. We need to construct a 4-to-1 MUX that uses F1 and F0 as selection inputs to determine the operation to perform on A and B and produce an output F(A, B). Then, from the logic table of F, we can notice that F2 is the bit that determines whether to invert the operation determined by F1 and F0. Thus, we can simply XOR F2 and F(A, B). For example, if we have $F(A, B) = 1$ and $F2 = 0$, then $F(A, B) \text{ XOR } F2 = 1$, which is the original signal of the output F(A, B). Vice Versa, when $F(A, B) = 0$, the result of $F(A, B) \text{ XOR } F2 = 0$. On the other hand, when $F2 = 1$, the result of $F(A, B) \text{ XOR } F2$ will be the inverted value of F(A, B). Therefore, we can simplify the circuit by only building 4 logic operations instead of a total of 8 with F1 and F0 determining the operation and F2 determining whether to invert the output or not. The simplification of the circuit is useful for the construction of this lab because we can directly use Vcc, NAND, NOR, and XOR gate for the logic operations without implementing a series of logic operations to get AND, OR, XNOR, and ground, which is easier for debugging and constructing the circuit.

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

A modular design can efficiently improve testability by allowing the designer to test each module independently instead of testing the entire system. In this way, it is easier to debug and find out which modules operate incorrectly. For example, in this lab, when having an error output for our processor, we connected the output of each module separately to LEDs for us to figure out which module is not producing the desired output, which saves us a lot of time for debugging. In addition, modular design can also cut down the development time of the circuit. Instead of having a group of people working on the same part, modular design allows people to work on different modules and test the modules individually. When every module is finished, we only have to assemble all the modules to form the design we want. In large projects, modular design can effectively allow each person to work on different parts and reduce the development time of the circuit.

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

To design a state machine for the lab, we choose to design a Mealy machine for our circuit. As the Mealy machine output depends on the current state and the input, we can simply separate the

states into a reset state and a shift/hold state. Thus, when it is in the reset state, it will move on to the shift/hold state when the execute signal is high. While it is in the shift/hold state, it will only shift back to the reset state once the counter finished counting and execute signal is low. Otherwise, it will keep staying in this state no matter if execute signal is high or low. Compared with a Mealy machine, a Moore machine on the other hand requires more states in this lab as its output only depends on the current state and doesn't depend on the input. Thus, in this lab, making a Moore machine requires at least three states, including reset state, shifting state, and halt state. This is because if we had the halt and shifting state as a single state as we have in the Mealy machine, then it would have the same output Shift (S) signal, although that should not be the case since shifting and halting require the opposite operations on the register. Even though Mealy machines require fewer states to complete the design, it is more difficult to debug and test as its output depends on both input and the current state. Vice versa, a Moore machine will be easier to debug as it only depends on the current state. All in all, Mealy machines representation can be compacted further than Moore machines, which comes at the cost of increased complexity. A Mealy machine output's dependence on input also enables it to update its output more quickly as compared to a Moore machine, in which we will encounter a clock cycle delay as we transition to a different state that updates its output.

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 main differences between ModelSim and SignalTap are that ModelSim is completely a software simulator while SignalTap is the logic analyzer that is in the FPGA board and requires hardware operation. For example, when using the SignalTap, we manually input the value for registers A and B and also execute signal; however, ModelSim simulates those signals by only using the software. Although both systems generate waveforms, in situations where we have to test different waveforms and large numbers of inputs, such as testing all 8 operations to different combinations of 8-bit values in this lab, ModelSim is more preferable as it can simulate all these in software. However, when we would like to observe the real-time implementation of the circuit, SignalTap is more preferred as it is connected to the hardware component in the real environment, so the information might be more accurate about how the system works in the real world.

Conclusions

What have you learned? What worked? What didn't? What can you possibly do to enhance/simplify/fix your circuit to make it better?

In this lab experiment, we learned about how to design a circuit using modular designs. By using modular designs, we are able to debug and test each module independently and save a lot of time. In addition, we learned the importance of simplifying the circuit design. In our lab, we designed the computation unit by directly using NAND, NOR, NOT, and XOR gates to implement all 8 operations, which took us more time to debug as we have to test all 8 operations to determine if they produce the desired result or not. However, it was until a few days after we finished the circuit that we found out we could have simply produced 4 operations and used the F2 signal to determine whether to invert the operation result or not.

In this way, we could have saved a few gates in our design, which is what we can possibly simplify to make our circuit better. Furthermore, we learned to use System Verilog to simply create the logic processor and test the design using ModelSim and SignalTap. In this way, it is faster to debug as we can design each module individually in different files, and the entire project looks more organized compared to the hardware circuit. Overall, our circuit works properly as it can produce all 8 operations and store the result in the registers. To make our circuit better and more efficient, we should simplify our design in the future.