ECE 385

Spring 2023
Experiment # 2

A Logic Processor

Max Ma & Dhruv Kulgod
NL / Friday 9:15 AM - 9:30 AM
Nick Lu

Purpose of Circuit:
This was a two-part lab. In the first part, we designed a 4-bit serial logic processor using TTL chips on a breadboard. The reasons for this are twofold: 1) we got firsthand experience going through the hardware design process of making decisions based on efficiency and hardware availability and 2) we realized exactly how difficult physical design and testing is, setting us up for the second part of the lab.

The second part of the lab had us replicate and extend the first part of the lab to 8 bits using our FPGA boards. We received an introduction to SystemVerilog, our MAX 10 boards and Quartus Lite software.

Operation of the Logic Processor:
Looking at the breadboard layout on Fritzing (Fig 15), we see two rows of DIP switches, 3 buttons and a row of LEDs. These are the inputs and outputs of our system. The processor has two 4 bit registers A and B which can be loaded by flicking switches 1:4 (the lowest 4) to high or low, and then pressing the 'Load A' or 'Load B' button. Since both registers use the same set of switches as inputs, they must be set up one at a time to hold unique values. LEDs 9:6 show the value in register A (MSB is LED 9), while LEDs 3:0 show register B (MSB is LED 3).

Next we must pick the logic operation we wish to perform. We use switches 7:9 as input F to our Computation Unit, which lets us choose a logic operation according to the figure below. We also use two switches 5:6 as inputs to our Routing Unit, allowing us to decide where to store our function after computation.

Once we have set up our inputs, operation and routing, we just need to hit the 'Execute' button located on the left side of the breadboard. The circuit will then step through its state machine to perform the correct number of operations and store the result as the user intended.

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

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 1: Operation Truth Table

Written Description of Circuit:

The overall flow of the circuit is as follows: the Control Unit handles when and how many operations occur on each bit in the register units. It does this through the implementation of a 4 bit finite-state machine. As we will see further, despite the 4 bits, we only required 2 flip flops to implement this. The Computation Unit takes one bit being shifted out from each register and applies a logical operation on these bits. The Routing Unit decides which bits are fed back to each register to be shifted in. All this happens synchronously. The only asynchronous processes are the loading of the two registers.

We began our design process by focusing on the state machine and register units, as we considered them the most complex component in our design. Fig 12 describes the 4-bit Mealy machine we had to implement, which counted 4 clock cycles after the execute button was pressed to ensure we shift bits in and out of our shift registers the correct number of times. Looking closely at the C1 and C0 bits, we can see that they are simply counting up in binary from 00 to 11 while S = 1. Thus, we could use the binary counter present in our chip kit to handle the logic for C1 and C0 for us. We used the shift signal S as input to the (CLR)' pin on the counter. Thus, when the registers were not shifting, the counter was constantly cleared, keeping it at 00.

The logic for S and (CLR)' was implemented by creating a K-map for it, and then optimizing the design to be implemented on a single chip. See Fig 7. The logic function for S and (CLR)' are $CLR' / S = (E + Q)(Q' + C_1 + C_0)$. S was then stored in a flip flop. Q+ was also implemented on a single chip using an MUX to create a lookup table of its values (See Fig 8), controlled by C1 and C0. The logic function for Q+ is $Q^+ = E + C_1 + C_0$. Q+ was also stored in a flip flop.

E	Q	C1	C0	~CLR	S
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	0
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	1	1
1	0	1	0	1	1
1	0	1	1	1	1
1	1	0	0	0	0
1	1	0	1	1	1
1	1	1	0	1	1
1	1	1	1	1	1

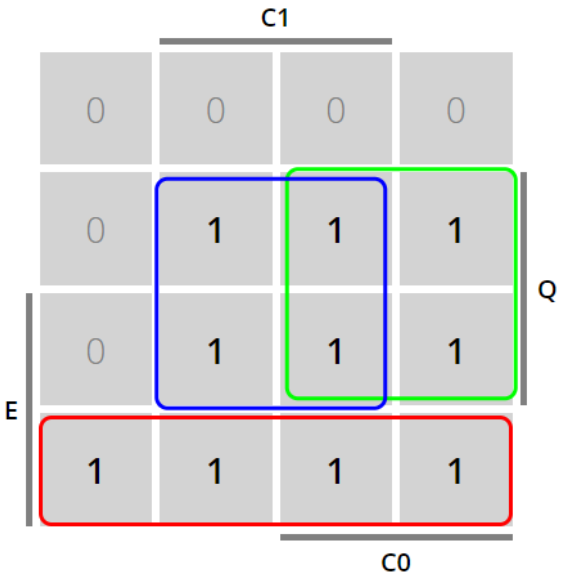


Figure 2: Truth Table and K-map for CLR' and S Logic

E	C1	C0	Q+
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

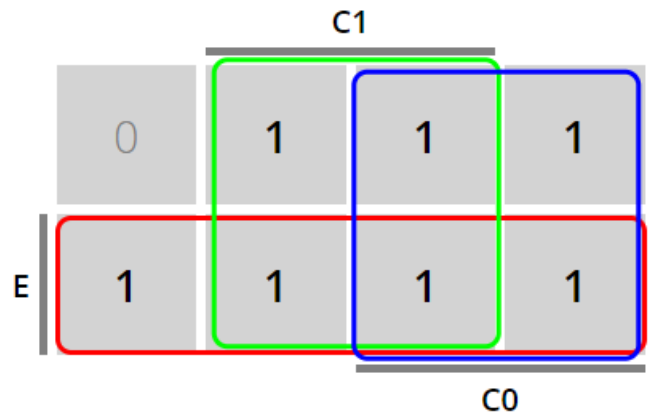


Figure 3: Truth Table and K-map for Q Logic

We needed some logic to control the shift registers. The registers take two control bits, S1 and S0. The logic function are provided: $S_1 = S' \bullet A$ and $S_0 = S \oplus A$. We decided on functionality (see Fig 9) based on what we thought would be a safe design. We handled conflicting inputs (loading and shifting at the same time) by stalling the system to prevent errors.

Ld A	S	S1	S0
0	0	0	0
0	1	0	1
1	0	1	1
1	1	0	0

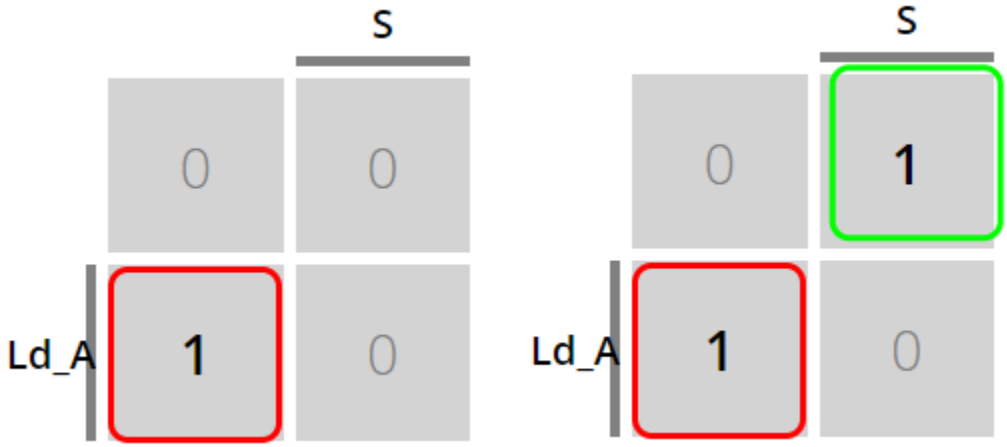


Figure 4: Truth Table and K-map for S1 (Left) and S0 (Right) Logic

For the Computation Unit, we were able to optimize a large, 8-to-1 MUX based design (a component we did not have available) to one requiring just 3 chips to implement all the operations as well as the routing. Looking carefully at the the truth table of the Computation Unit (Fig 10), the functions are arranged in an order that allows us to use a single 4-to-1 MUX and simply XOR the output to get the inverse of each function (eg. NOR instead of OR). Thus, we only needed to implement 4 operations (AND, OR, XOR and 1), which could be accomplished using the same XOR as above in addition to a quad 2-input NAND chip.

Finally, the Routing Unit took advantage of a unique quirk of our dual 4-to-1 MUX chips - the shared selection bits. Since both outputs A and B are calculated based on the same inputs R1 and R0 (see Fig 11), we could use a single chip to implement this routing mechanism.

High Level Block Diagram:

The high level block diagram shown in Figure 5 describes the breadboard level implementation of the 4-bit logic processor. Figure 6 describes the FPGA module level implementation of the 8-bit logic processor.

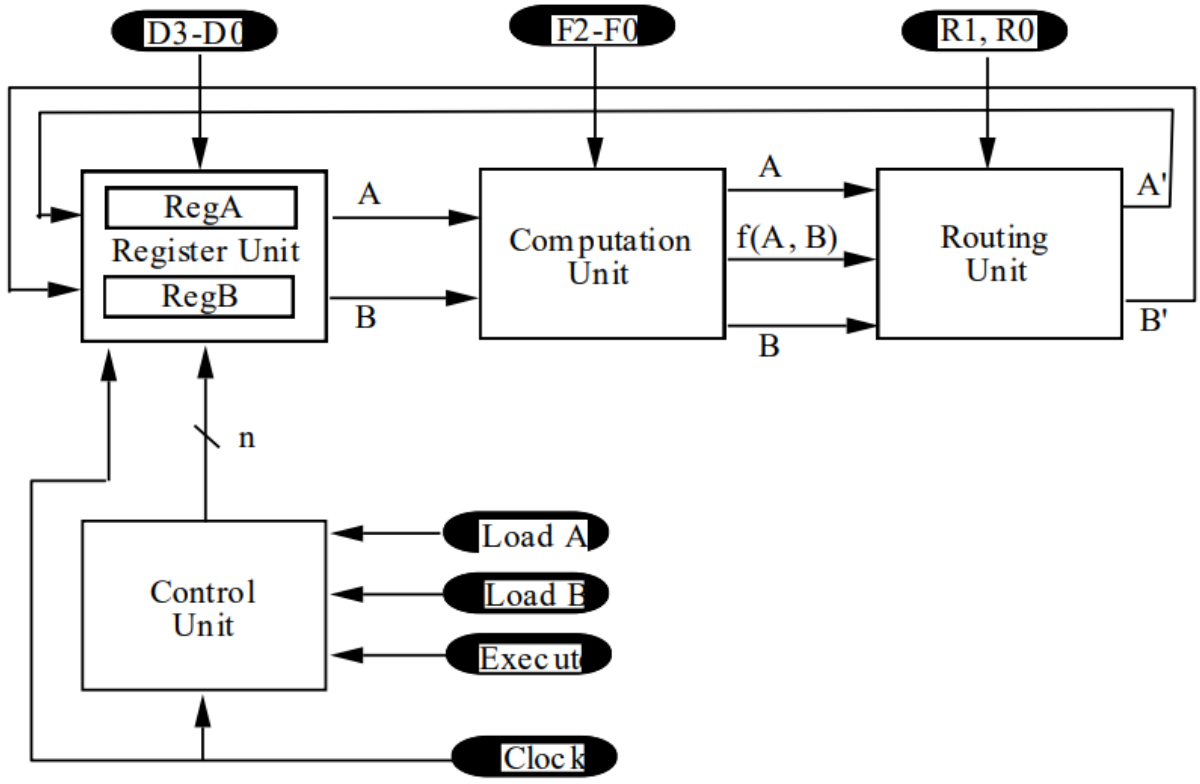


Figure 5: High Level Block Diagram of Breadboard Implementation

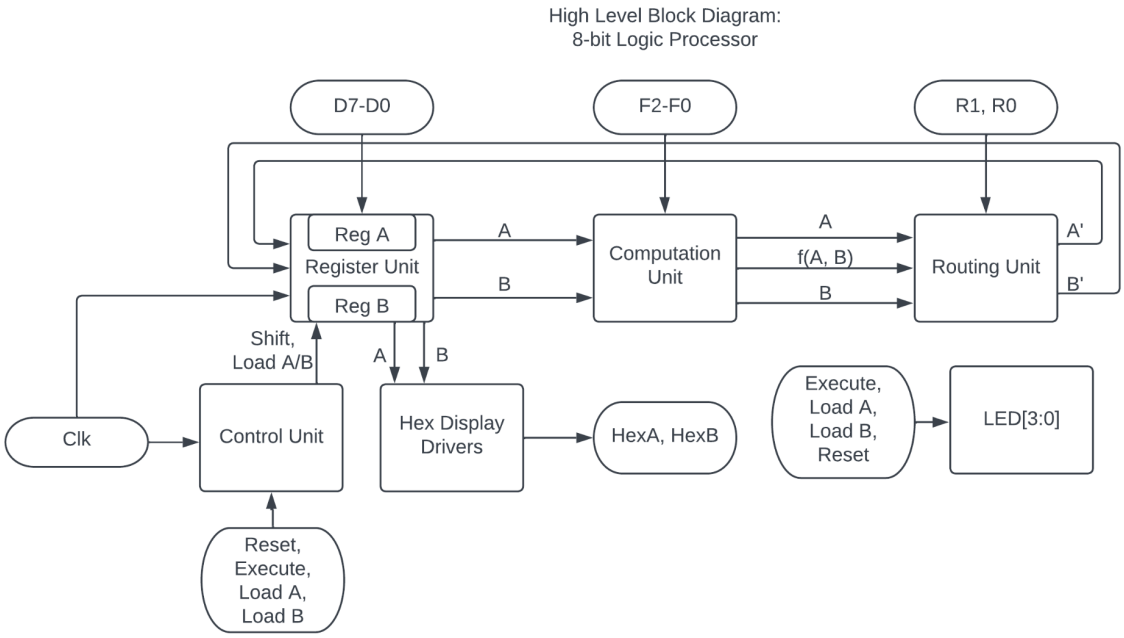


Figure 6: High Level Block Diagram of FPGA Implementation

Design Decisions:
Lab Part 1)

1. **Flip-Flop Initialisation**

We spent some time debating whether we needed additional logic to ensure the FSM flip-flops were initialized in the correct state (0000), and not in a don't care state which could cause ambiguous behavior. However, carefully inspecting the actual logic circuits from our Kmaps showed us that all don't care states eventually led back to 0000 within a few clock cycles, meaning we did not have to waste hardware on initializing our flip-flops.

2. **Binary Counter**

We had initially planned to use 4 flip-flops in the control unit, one for each of S, Q, C1 and C0. However, visiting office hours helped us realize that this would leave us with a lot of extra logic and chips to implement on an already crowded board. Our chip kits already came with binary counters, which we could use to implement C1 and C0, thanks to the way the FSM truth table was laid out. What we learnt from this was to carefully evaluate all the resources at hand before deciding on an implementation.

3. **SOP vs POS**

For every Kmap we had, we compared the SOP and POS implementations to see if one was considerably better than the other. In most cases, we saw no clear advantage. However, for the next state logic of S, we found the POS method superior because we could implement all of it on a triple 3 input NOR chip.

4. **Do Nothing State**

For the register control logic, we chose to implement a unique safety state if a register load button was pressed while shifting. Instead of letting the module continue shifting, we chose to simply hold the data, since this represents a user trying to do two conflicting operations at the same time. In a more robust design, we would probably extend this design to include a warning that tells the user an error has occurred. This design may even help identify button failures.

Lab Part 2)

1. **Control Unit**

To extend our design to 8 bits, we had to change the FSM to have more shift states, so that each bit in the registers could pass through the computation unit.

Logic Diagrams:
The logic diagrams for controlling the registers from the control unit as shown below. Additionally, the circuit diagrams from the compute and routing unit are shown in Figure 10 and Figure 11 respectively.

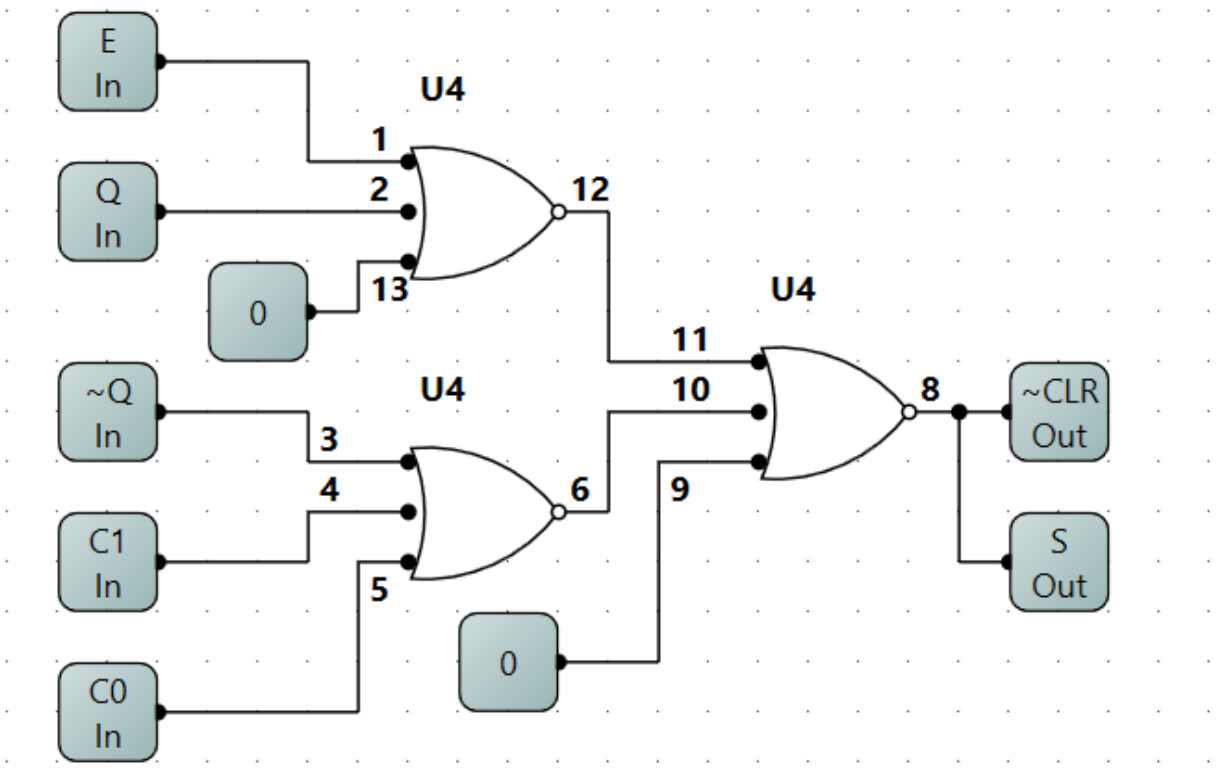


Figure 7: ~CLR and Shift Signals for 4-bit Shift Registers

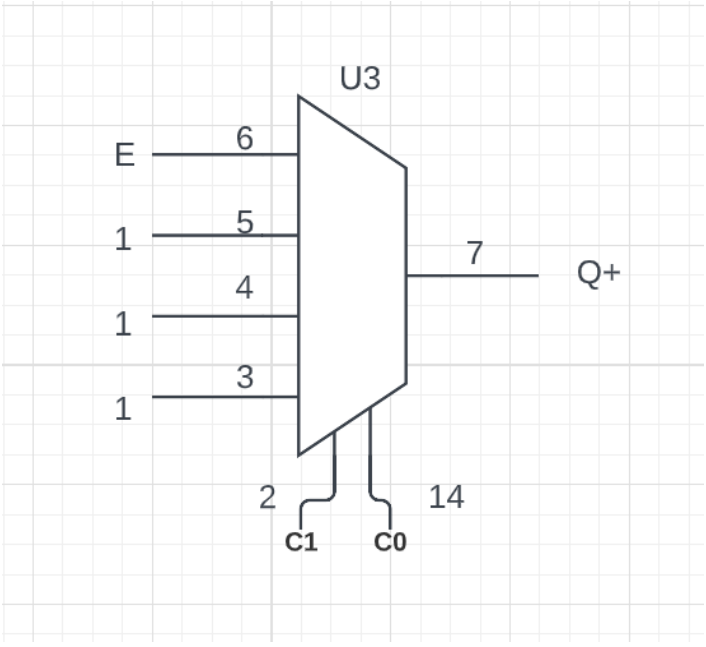


Figure 8: Q+ Logic Implemented Using Spare 4:1 Mux

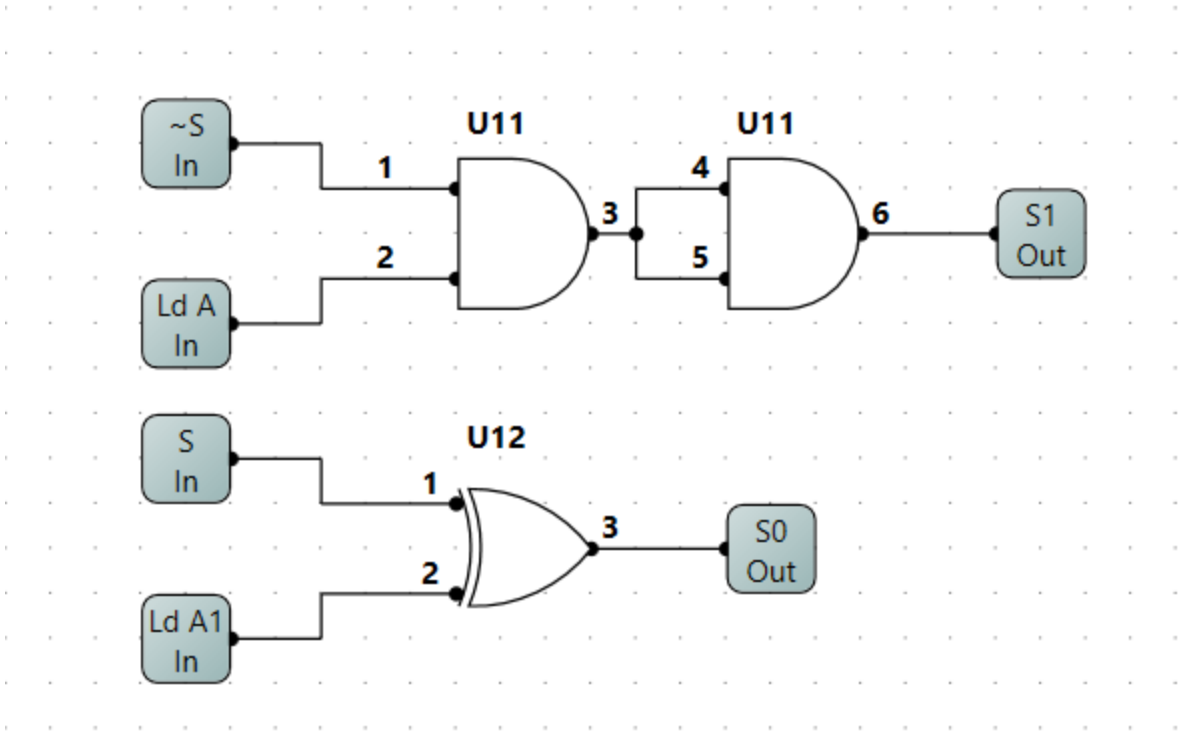


Figure 9: Register Load A / B for 4-bit Register

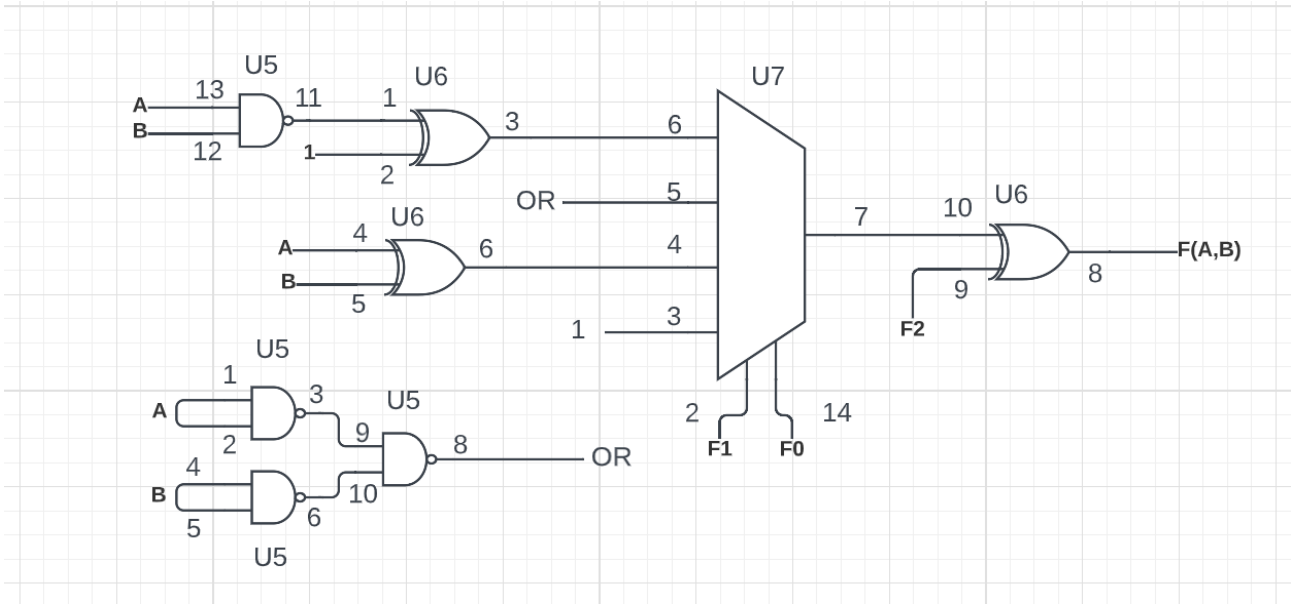


Figure 10: Compute Unit Logic Circuit Design

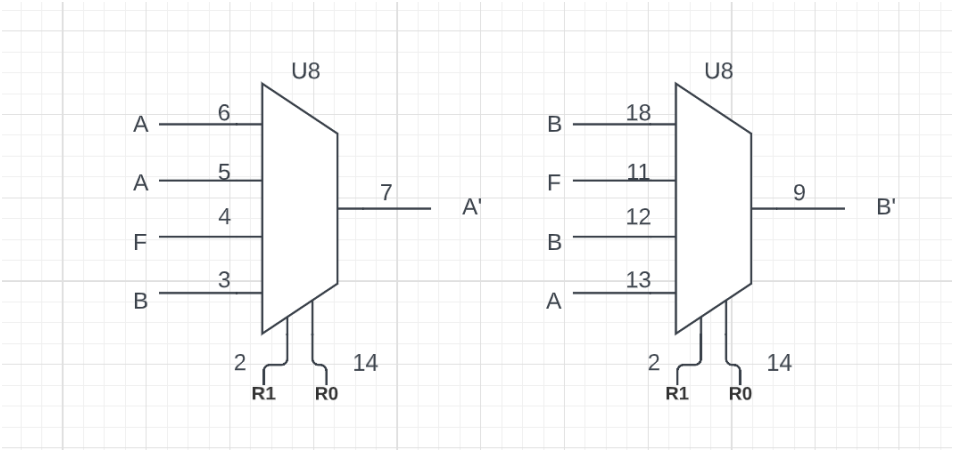


Figure 11: Routing Unit Logic Circuit Diagram

State Diagrams and Tables:

The state transition diagram and table for the 4-bit control unit are shown below in Figure 12 and Figure 13 respectively. This was the implementation of the control unit for the breadboard 4-bit Logic Processor. The FPGA implementation of the state transition diagram is shown in Figure 14.

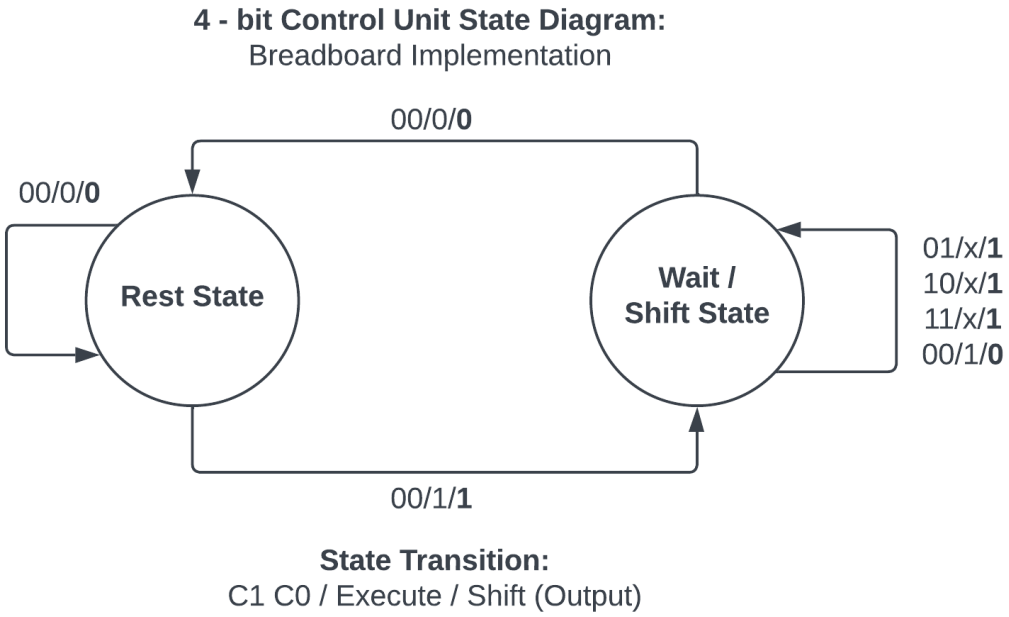


Figure 12: 4-bit State Diagram for Breadboard Implementation

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 13 : 4-bit State Transition Table for Breadboard Implementation

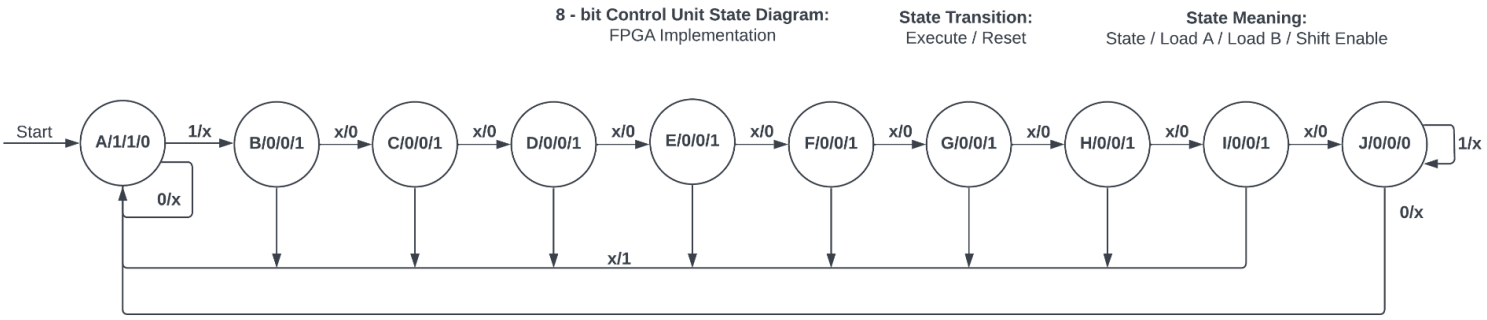


Figure 14: 8-bit State Diagram for FPGA Implementation

Component Layout:

The Fritzing layout shown below in Figure 15 is the complete breadboard layout of the 4-bit logic processor.

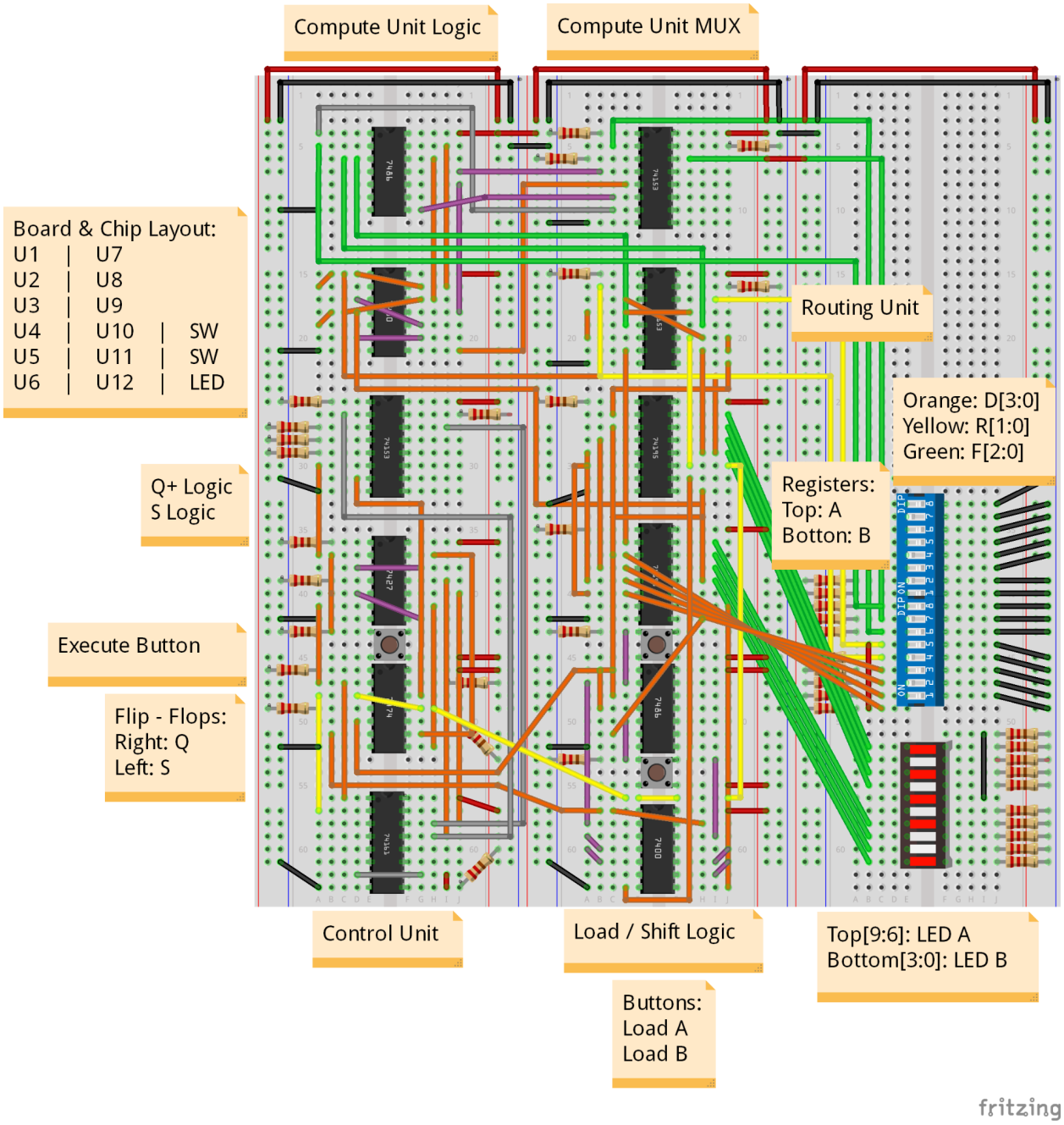


Figure 15: Fritzing Board and Component Layout

Answers to Post-Lab:

Q1.) We encountered three difficulties when trying to debug our circuit. One, our wiring was messy making it difficult to see signals between modules. We solve this problem by taking a lot of time to trim our wires to the appropriate length. Second, we have signals from other modules interfering with our testing. Accidentally, we would have multiple drivers from our board and testing switch box to the same pin. We solved this problem by

disconnecting the wires from other interfering signals. Third, we had some confusion with our LEDs and Switches corresponding to which register and inputs. This occurred due to the complexity and number of components in the design. We solved this issue by having Register A components always above Register B. The modular approach definitely helped us isolate specific faults in our design. For example, we unit tested our control unit and found that it was working properly. When we encountered problems with our register loading and shifting operations, we did not have to consider the control unit as a source of error since that was unit tested already. Instead we isolated our load A and load B logic; in the end, we found an error with the S1 and S0 control signals to the registers having the wrong truth table.

Q2.) The simplest circuit would be XOR Gate as shown in Figure 16. For example, if A is pulled up to logic level 1, the gate acts as an inverter. Meanwhile, if A is pulled down to logic level 0, the gate acts as a buffer that passes a signal to the output. This functionality acts as an optional inverter. This is useful in the construction of the lab because it can simplify the logic of the compute module. This is shown in the logic diagram of the compute module shown in Figure 16. Additionally, extra XOR gates can act as inverters without having to use an extra hex inverter chip.

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

Figure 16: XOR Truth Table

Q3.) A modular design allows the designers to isolate and test each module. Testing by module increases the reachability of the designers testing because every aspect of the module should be easily accessible. If the whole design was tested at the same time, it would be difficult to examine small implementation details. Testing by module also reduces development time because it will help the designer to isolate problems when debugging instead of trying to take away variables from the whole design.

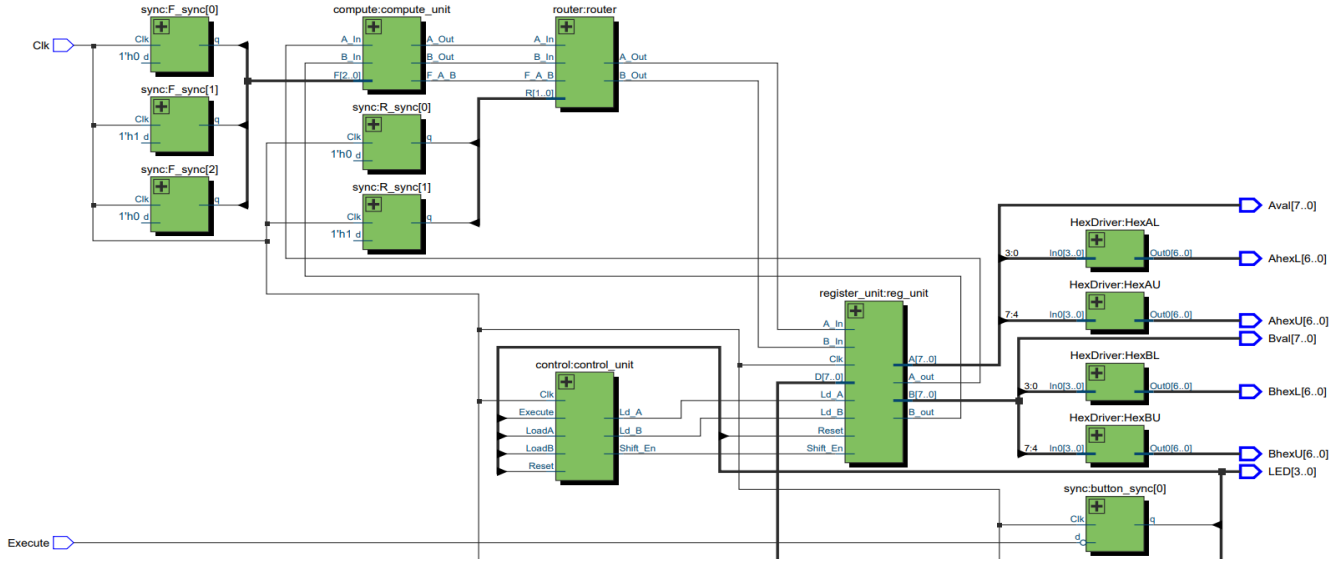
Q4.) As shown in Figure 12 and Figure 14, the state diagrams of a Mealy and Moore machine can be very different. The Mealy machine has 4 inputs while the Moore machine has only 2 inputs. Meanwhile, the Mealy machine has 2 states, while the Moore

machine has 10 states. In terms of design tradeoffs, the Mealy machine has more complexity, which is its number of inputs; however, it needs less states as a result.

Q5.) ModelSim ***generates*** simulation waveforms using SystemVerilog code. Although it is non-synthesizable, it is mainly used for testing existing SystemVerilog code by instantiating the Unit Under Test (UUT) module and probing the input, output, and internal logic signals. On the other hand, SignalTap is an integrated logic analyzer that is compiled and placed in the FPGA fabric along with the design. It is designed to ***capture*** waveforms from the actual hardware implementation on the FPGA. The designer must configure the SignalTAP ILA to trigger and observe certain traces in their design, which means no SystemVerilog code is written from the designer. In terms of use cases, ModelSim is great for testing designs quickly. ModelSim should typically be used before Signal Tap ILA in the design process to verify modules written in SystemVerilog. On the other hand, Signal Tap can be used to test if the SystemVerilog implementation is working correctly in hardware. Additionally, can be used to test for glitches and hazards that will not be simulated in ModelSim since ModelSim does not account for physical hardware with propagation delays.

FPGA Labs: Block Diagram from RTL:

The top-level RTL block diagram from Quartus's RTL Viewer is included below in Figure 17.



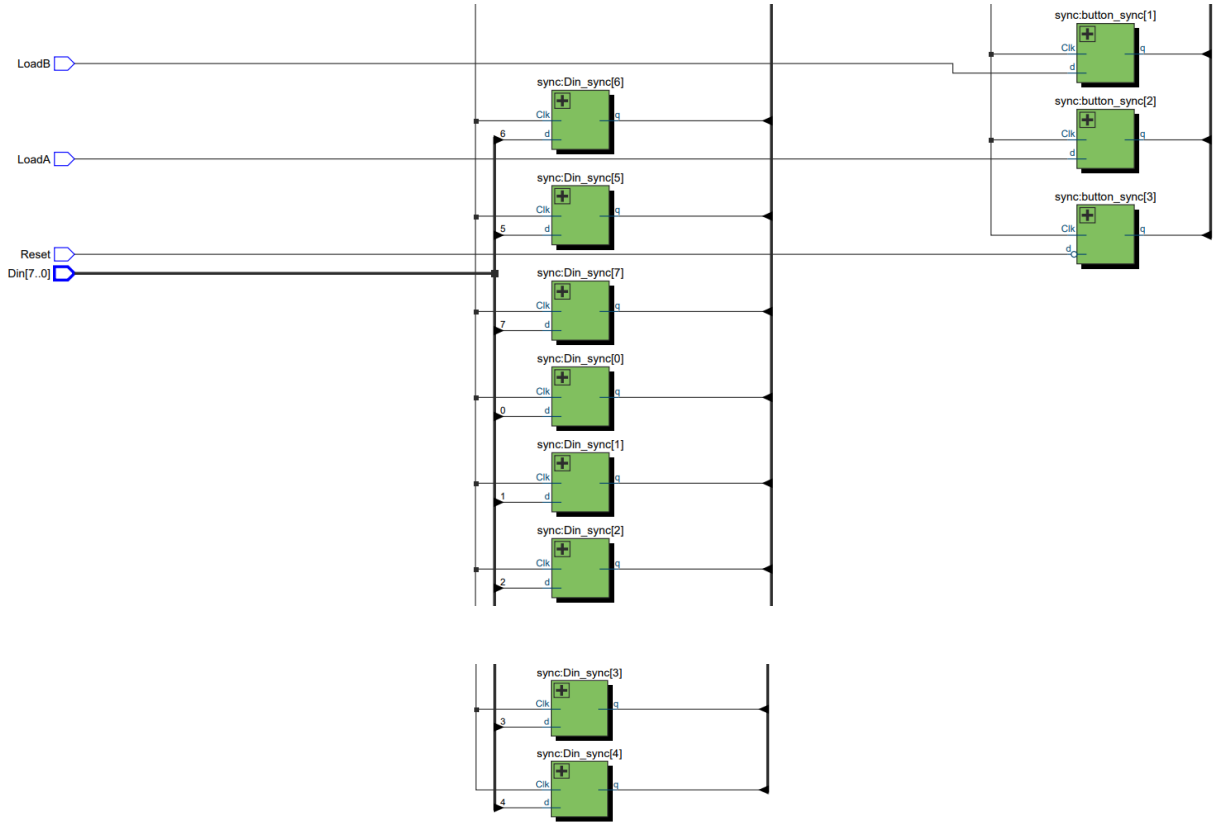


Figure 17: RTL Viewer Top Level Block Diagram

FPGA Labs: Design Module Information:
Changes to SystemVerilog Modules:

1. Extended registers to 8 bits by changing connections from [3:0] to [7:0].
2. Increased the number of counting states from 4 to 8 in the control unit.
3. Included the new upper 4 bits of A and B in the hex display.

SystemVerilog Modules Description

1. **Compute:**
    a. **Input:** F [2:0], A_In, B_In
    b. **Output:** A_out, B_out, F_A_B
    c. **Description:** This module computes the result of the logical operation between A and B. The available logic operations are AND, OR, XOR, 1s, NAND, NOR, XNOR, 0s. The executed logic result is selected by the three F bits using a 8:1 Mux. The function output is passed to the routing module along with A and B.

2. **Control:**
    a. **Input:** Clk, Reset, LoadA, LoadB, Execute

b. **Output:** Shift_En, Ld_A, Ld_B
c. **Description:** The control module determines whether the shift registers, A and B, should be loaded or shifted using a Moore Finite State Machine with 10 states. It transitions to the next state on the positive edge of its input clock and returns back to A when the reset button is pressed. State A is the resting state in which the registers can be loaded. States B-I are the 8 shifting states for the registers. J is the reset state in which "execute" must be low to transition back to A, the resting state. The Shift_En, Ld_A, Ld_B are sent to the register unit.

3. **HexDriver:**
    a. **Input:** In0[3:0]
    b. **Output:** Out0[6:0]
    c. **Description:** This module acts as a translator between binary to hexadecimal. A set of four binary bits are taken as input and linked to their corresponding pattern on the 7-Segment Hex Display using a case statement.

4. **Processor (Top Level):**
    a. **Input:** Clk, Reset, Load A, Load B, Execute, Din [7:0], F[2:0], R[1:0]
    b. **Output:** LED [3:0], Aval | Bval [7:0], AhexL | BhexL | AhexU | BhexU [6:0],
    c. **Description:** This is the top level module in the project, which is in charge of instantiating all the modules. It creates internal logic variables to connect each module together. The modules that it instantiates are the register unit, compute unit, routing unit, control unit, hex drivers, and synchronizers.

5. **Reg_4:**
    a. **Input:** Clk, Reset, Shift_In, Load, Shift_en, D [7:0]
    b. **Output:** Shift_Out, Data_Out [7:0]
    c. **Description:** This module creates an 8-bit shift register. The serial output bit is assigned to the LSB of the current 8-bits in the register. On the rising edge of each clock cycle, the serial input is shifted into the MSB when shifting is enabled. If reset is pushed, the shift register's data is set back to all 0s. If load is enabled, the shift register will parallel load the data from the switches.

6. **Register_unit:**
    a. **Input:** Clk, Reset, A_In, B_In, Ld_A, Ld_B, Shift_En, D [7:0]
    b. **Output:** A_out, B_out, A[7:0], B[7:0]
    c. **Description:** The register unit module is the top level module for the shift registers. It instantiates two 8-bit shift registers providing the load and shift signals along with the input data.

7. **Router:**
    a. **Input:** R[1:0], A_In, B_In, F_A_B
    b. **Output:** A_out, B_out
    c. **Description:** The routing unit decides the serial input to the A and B shift registers. It uses two case statements acting as 4:1 Muxes with R [1:0] as select bits. The select bits are used to indicate which register, A or B, should take the function output from the compute module. Additionally, registers A and B could be stationary or swapped.

8. **Schronizers:**
    a. **Input:** Clk, d
    b. **Output:** q
    c. **Description:** The synchronizer module is used to bring asynchronous signals into the FPGA and synchronize them to the common clock.

FPGA Labs: Design Statistics:

	Resource	Usage
1	Total logic elements	842 / 49,760 (2 %)
1	-- Combinational with no register	198
2	-- Register only	279
3	-- Combinational with a register	365
2		
3	Logic element usage ...number of LUT inputs	
1	-- 4 input functions	250
2	-- 3 input functions	146
3	-- <=2 input functions	167
4	-- Register only	279
4		
5	Logic elements by mode	
1	-- normal mode	480
2	-- arithmetic mode	83
6		
7	Total registers*	644 / 51,509 (1 %)
1	-- Dedicated logic registers	644 / 49,760 (1 %)
2	-- I/O registers	0 / 1,749 (0 %)
8		
9	Total LABs: partially or completely used	75 / 3,110 (2 %)
10	Virtual pins	0

11	⌄ I/O pins	61 / 360 (17 %)
1	-- Clock pins	1 / 8 (13 %)
2	-- Dedicated input pins	1 / 1 (100 %)
12		
13	M9Ks	1 / 182 (< 1 %)
14	UFM blocks	0 / 1 (0 %)
15	ADC blocks	0 / 2 (0 %)
16	Total block memory bits	8,704 / 1,677,312 (< 1 %)
17	Total block memory implementation bits	9,216 / 1,677,312 (< 1 %)
18	Embedded Multiplier 9-bit elements	0 / 288 (0 %)
19	PLLs	0 / 4 (0 %)
20	⌄ Global signals	3
1	-- Global clocks	3 / 20 (15 %)
21	JTAGs	1 / 1 (100 %)
22	CRC blocks	0 / 1 (0 %)
23	Remote update blocks	0 / 1 (0 %)
24	Oscillator blocks	0 / 1 (0 %)
25	Impedance control blocks	0 / 1 (0 %)
26	Average interconnect usage (total/H/V)	0.3% / 0.3% / 0.3%
27	Peak interconnect usage (total/H/V)	5.3% / 5.0% / 5.7%
28	Maximum fan-out	349
29	Highest non-global fan-out	74
30	Total fan-out	4155
31	Average fan-out	2.67

Figure 18: Compilation Report Statistics and Summary

FPGA Labs: Annotated Simulation Waveform:

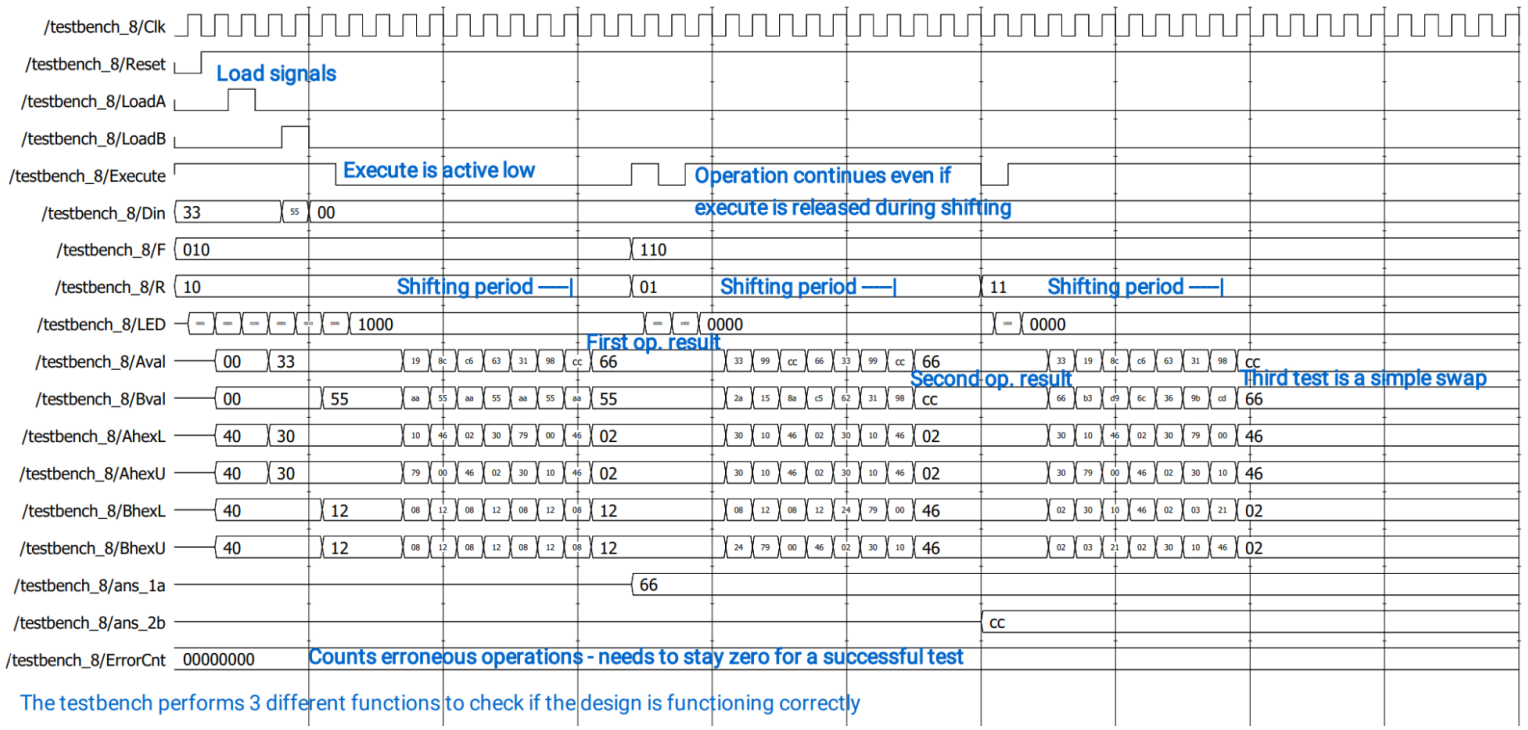


Figure 19: Annotated Simulation Waveform

FPGA Labs: Signal TAP ILA Instructions

Step 1: Head to Tools > Signal Tap Logic Analyzer

Step 2: Enter the "Setup Tab" and double left click on the window to open Node Finder

Step 3: Find Execute, Aval[7:0], and Bval[7:0] in the Node Finder

Step 3.1: Enter the names of the signals and filter for "Pins:all"

Step 4: Right click on the Trigger Conditions for Execute and select "Either Edge"

Step 5: Start compilation and program the board using the generated .sof file

Step 6: Once programmed, click on "Run Analysis" or Press F5 to set up the trigger

Step 7: Load the test data using the switches and LEDs, then press execute

An example capture is shown below in Figure 20.

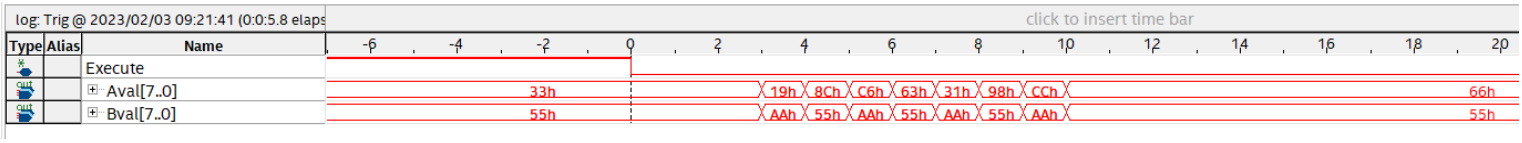


Figure 20: Example SignalTap ILA trace

Bugs:

1. **Bad/Missing Wire Connections**

Despite our best organizational efforts, we did encounter some issues caused by bad or missing wires. This was just due to the tedious nature of the physical lab, and was the motivation behind teaching us to use FPGAs in the first place.

2. **Floating Voltages**

While building our physical circuit, we encountered some unpredictable behavior, where certain functions would act correct at some times but not others. We forgot that even chip signals that are meant to stay low need to be connected to ground. If not, they are “floating voltages”, whose value is unknown and could cause the chip to act in ways we did not intend. Once identified, it was a simple fix.

3. **Multiple drivers while testing**

While unit testing sections on the breadboard, we encountered an issue where the test inputs we were applying were being interfered with by other parts of the circuit we had already built. This was because we had eagerly connected all the units together before unit testing all of them, and then forgot to disconnect those channels while debugging. This is not best practice, so we learnt the importance of isolated testing here.

Conclusion:

We enjoyed the initial planning stage of the physical lab where we could debate implementation choices. However, after going through the physical build process and then the FPGA lab, the power of VHDLs cannot be denied. While we lose some control over exact implementation, the speed of development is far greater. Additionally, a lot of designs can be simplified by the software better than us, as long as we do a good job specifying the operation of the modules in code.