ECE 385

Spring 22

Experiment #2

A Logic Processor

Siddarth Iyer, Geitanksha Tandon
TA: Ruihao Yao (RY)

1) Introduction:

In this experiment, we implement a bitwise logic operator. Our circuit takes in two 4-bit inputs stored in two shift registers, performs one of eight functions (preselected by the user) bit-by-bit using a computation unit, and stores values back into the registers via a routing unit. The operation of the circuit is controlled by a Mealy machine which uses a Double Flip Flop and a Counter. The logic operator executes the following functions: AND, OR, XOR, 1111, NAND, NOR, XNOR and 0000.

---

2) Operation of the logic processor

Our logic processor works using two shift-registers in our register unit. We first use the data input switches D_3 - D_0 to set the 4 bit value to be loaded in either of the registers. Then, we set the Load button for whichever register we want to set. For example, if we set our D_3 - D_0 switches and press the Load A button, the four bit value gets parallel-loaded into register A.

To initiate a computation, the user must first decide which operation to execute, from a table as shown below (Taken from the G.G.) and switches $F_2F_1F_0$ must be toggled to match the corresponding function. Next, switches R_1R_0 must be toggled based on the table to decide the new value to be loaded into Register A and B. For instance, if the user wanted to execute the XOR function and store the result in register A while retaining the value of register B, the switch selection would be as follows:
$F_2F_1F_0$ = 010, R_1R_0 = 10. Once values are loaded into register A and B, and the function and route is decided, the logic processor can be executed by pressing the Execute button.

TABLE 1: Functions

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: The functions to be executed, the routing unit to be implemented; taken from Experiment PDF.

---

3) Written description, block diagram and state machine diagram of logic processor

Register Unit: The register unit consists of two 4 bit shift registers RegA and RegB. When switches Load A / Load B are high, the register operating mode is set to Parallel Load and a 4 bit value set by data input switches D_3 - D_0 are loaded into the register. When the Load Buttons and the Execute button are low, the register holds the value in the register. When Execute is pressed, the shift register starts shifting right for 4 clock cycles. The least significant bit is shifted into the Computation unit and the final value coming from the routing unit is inputted serially back into the register.

Computation Unit: The computation unit receives two 1-bit inputs A and B from the register unit on which the bitwise logical operation is to be executed. Four logic operations (AND, OR, XOR, 1111) are implemented and sent into a 4-to-1 MUX. Switches F_2-F_0 are used to select which function to implement. Switches F_1 and F_0 are used as select bits for the MUX, and the output of the MUX is then put into an XOR gate, with the second input of the XOR being F2. If F_2 = 0, then the above functions are executed, which form the top half of the required truth table. For F_2 = 1, the four functions implemented above are simply inverted, which form the lower part of the truth table. Once the function is executed, the F output is sent to the routing unit to be routed back into the registers.

Routing Unit: The routing unit controls the inputs of the registers. It decides what input must be put into them after execution. It consists of two 4-to-1 MUXes with inputs AAFB and BFBA for A and B respectively. The input F comes from the Computation Unit, while A and B come from the Register Unit. The final value to be serially inputted is determined by switches R_1 and R_0 which serve as control bits for the MUX.

Control Unit: The control unit consists of a D-flip flop, 4 bit counter (where only the lowest 2 bits are used), switches for Load A, Load B, Execute, a clock, and additional combinational logic. The control unit uses two switches Load A and Load B to parallel-load data into registers A and B when they are set to high. It also consists of a Mealy FSM comprising a D-flip flop and counter. When the Execute switch is triggered, the counter starts transitioning for 4 clock cycles (one cycle per bitwise operation).

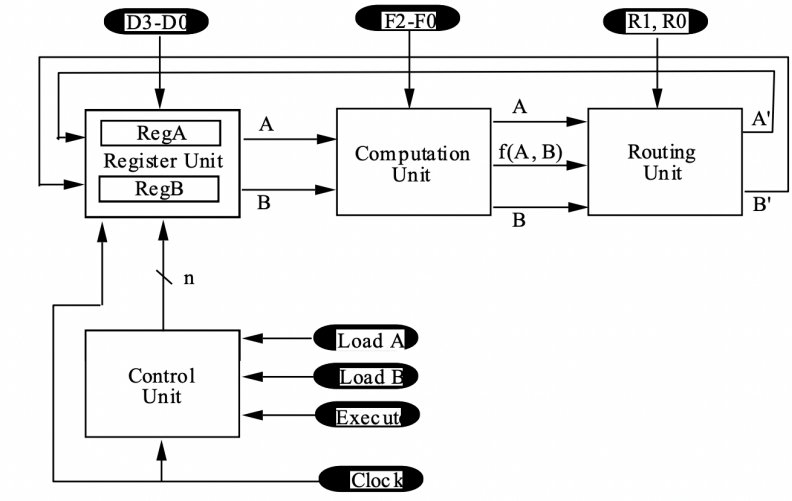


Figure 2: High Level Block DIagram of our Circuit, taken from Experiment 3 PDF.

Our control unit makes use of a Mealy Finite State Machine, for which the output depends on the current state as well as the inputs. Our FSM consists of only 2 states - the Reset state and the Shift/Hold State. The inputs of our FSM are: Execute, C_1C_0 (Counter bits), Q (Current state bit).
The outputs of our FSM are: Shift, Q+ (Next state bit).

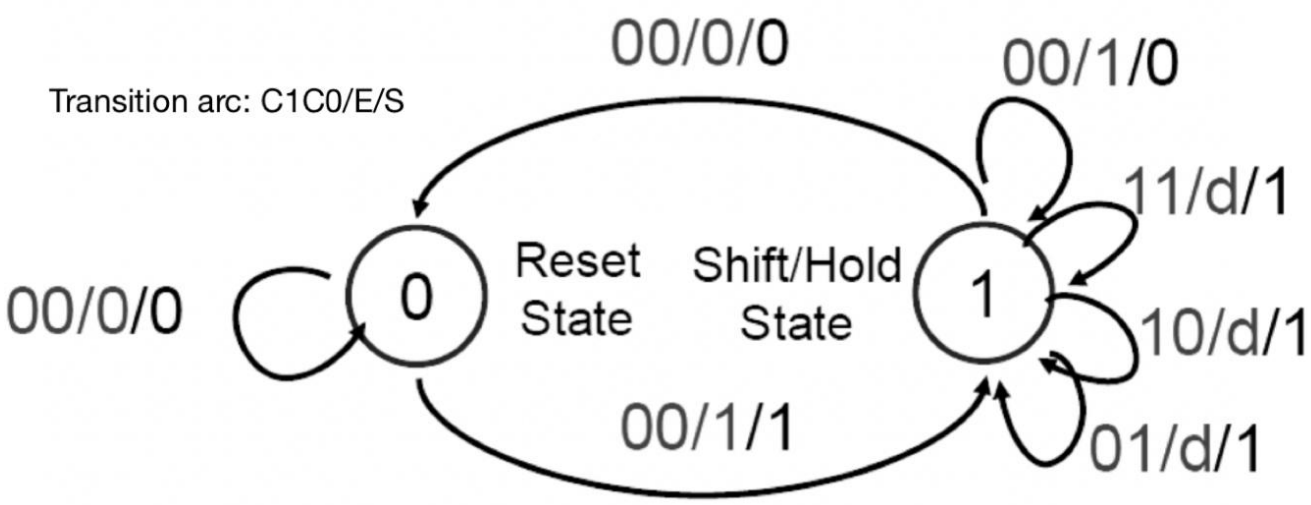


Figure 3: Mealy State Machine for the Control Unit, taken from the Experiment 3 PDF.

---

4. Design steps taken and detailed circuit schematic diagram.

We began by creating the Next state Table for the Mealy state machine, drew out Kmaps for S and Q+, and derived boolean expressions for the same. S and Q+ depends on E, Q, C1, C0.

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

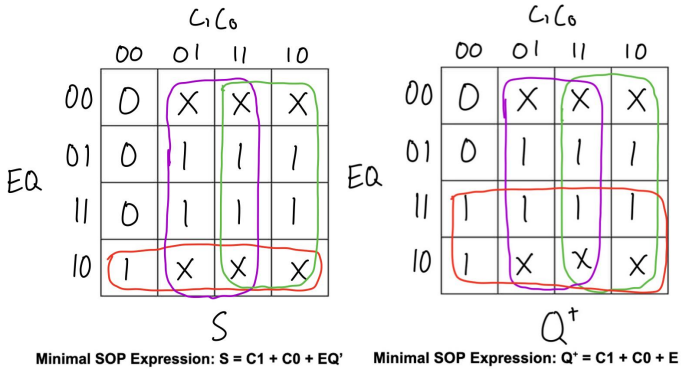
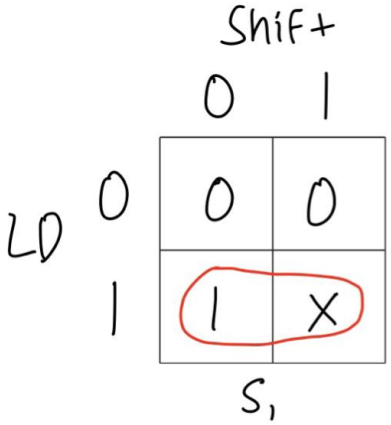


Figure 4: Mealy Machine's implementation taken from Experiment PDF, KMAPs created for S, Q+.

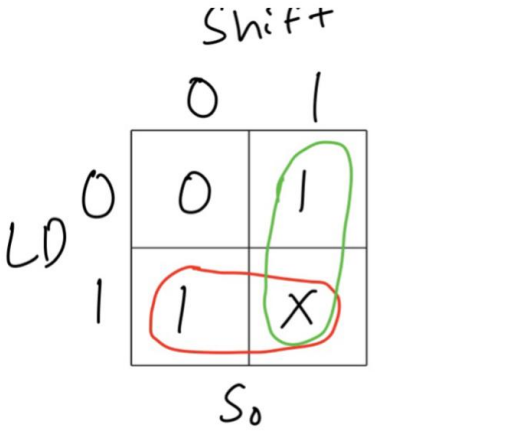
The shift registers contains various modes controlled by pins S_1 and S_0 on the register. The values of Load A / Load B and Shift determine the values to be inputted to S_1 and S_0. Using the datasheet to learn about the operation modes, and drawing out K-Maps we derive expressions for S_1 and S_0.

MODE	S1	S0
Hold	L	L
Shift right	L	H
Parallel load	H	H

LD A/B	Shift	S1	S0
0	0	0	0
0	1	0	1
1	0	1	1
1	1	d	d



Minimal SOP Expression: S1 = LD A/B



Minimal SOP Expression: S0 = LD A/B + S

Figure 5: Truth tables and derived KMAPs for the creation of the SOP Expression for State Bits S1, S0.

Our main objective while designing the circuit was to make the design as modular as possible. We divided the circuit into 4 main units (r*egister unit, computation unit, routing unit, and control unit*). Though this slightly increases the area of the circuit, it made the designing, debugging, and testing process much easier. We designed units as optimally as possible to reduce the number of chips used. For the control unit, we chose the Mealy implementation for the FSM in order to reduce the number of flip-flops used. Instead of creating a separate counter FSM we used a 4 bit counter and used the lowest 2 bits as inputs to the FSM to count through 4 clock cycles. In the register unit we decided to use CD74HC194E Bi-Directional shift registers instead of the CD74HC195E, since the 194 chip offers holding functionality, where it can hold the value that is required, which would be necessary when we are not shifting the values of A and B. In addition, it offers both parallel load in and out and shift in and out capabilities. In the Computation unit, we decided to use the F_2 signal as the second input to our XOR so we could create the bottom half of our truth table from the top half implementation. This made it possible for us to use a 4-to-1 MUX instead of an 8-to-1 MUX and prevented the need to implement 4 additional functions. We also used unused NAND, XOR, NOR gates as inverters.

Quartus Diagrams for the Circuit:

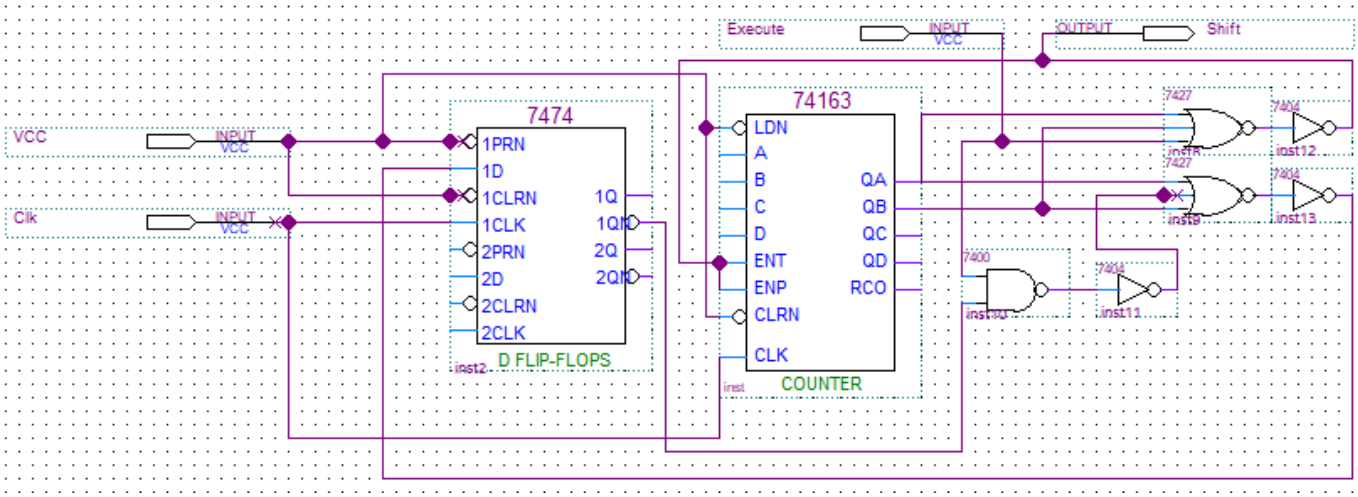


Figure 6: Control unit of our circuit. Note: Output Shift serves as Input in register unit.

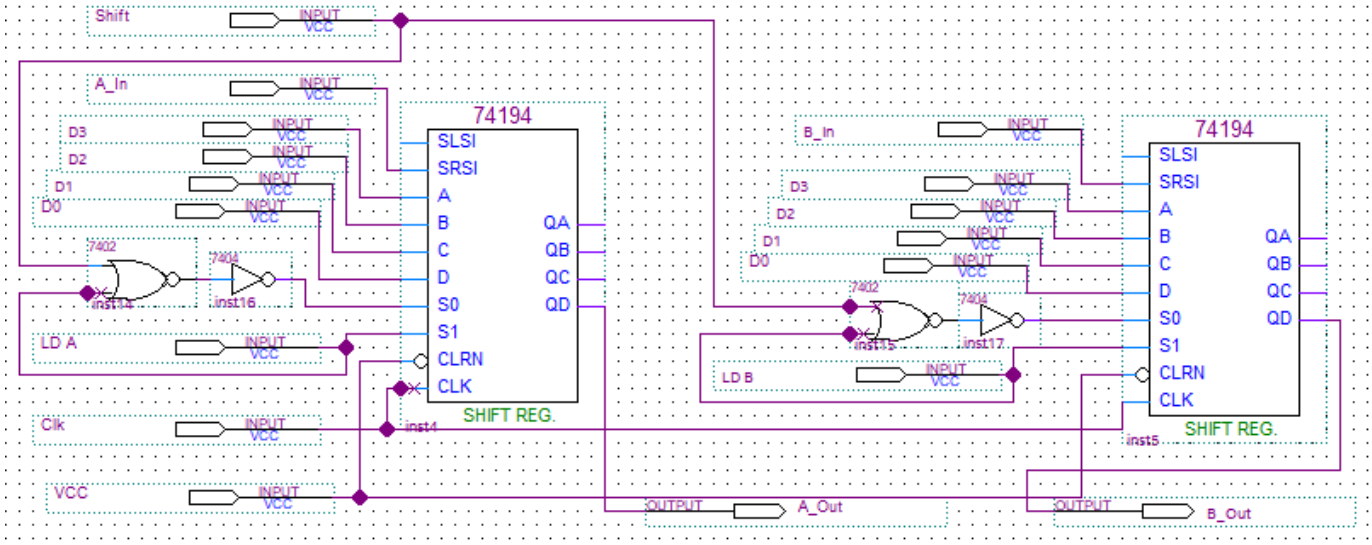


Figure 7: Input Shift from Control Unit. Output A_Out, B_Out serve as inputs for Computation and routing unit.

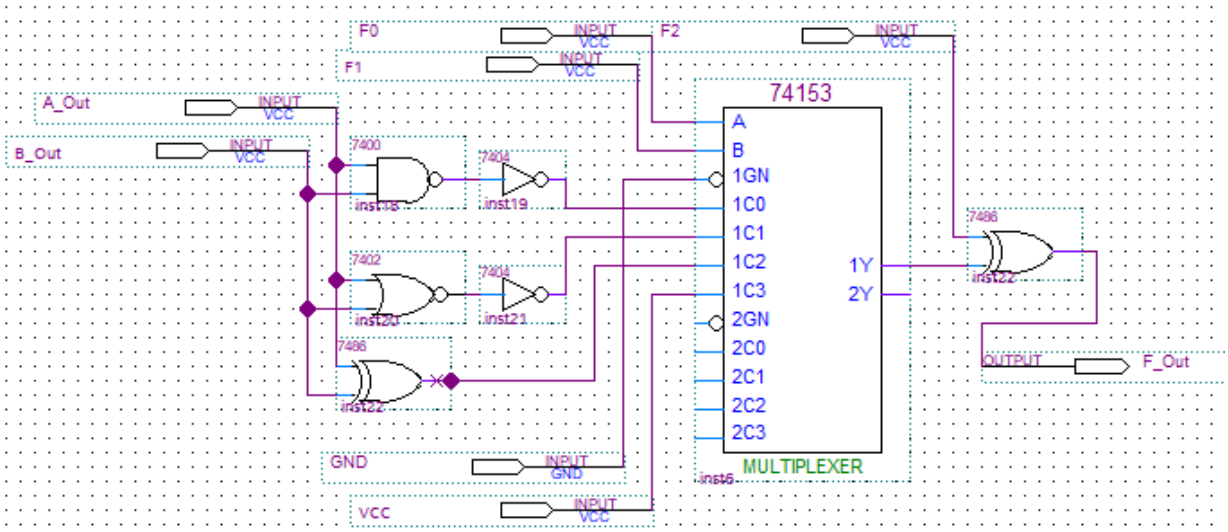


Figure 8: Computation Unit(Input A_Out, B_Out from Register Unit. Output F_Out is input for Routing Unit).

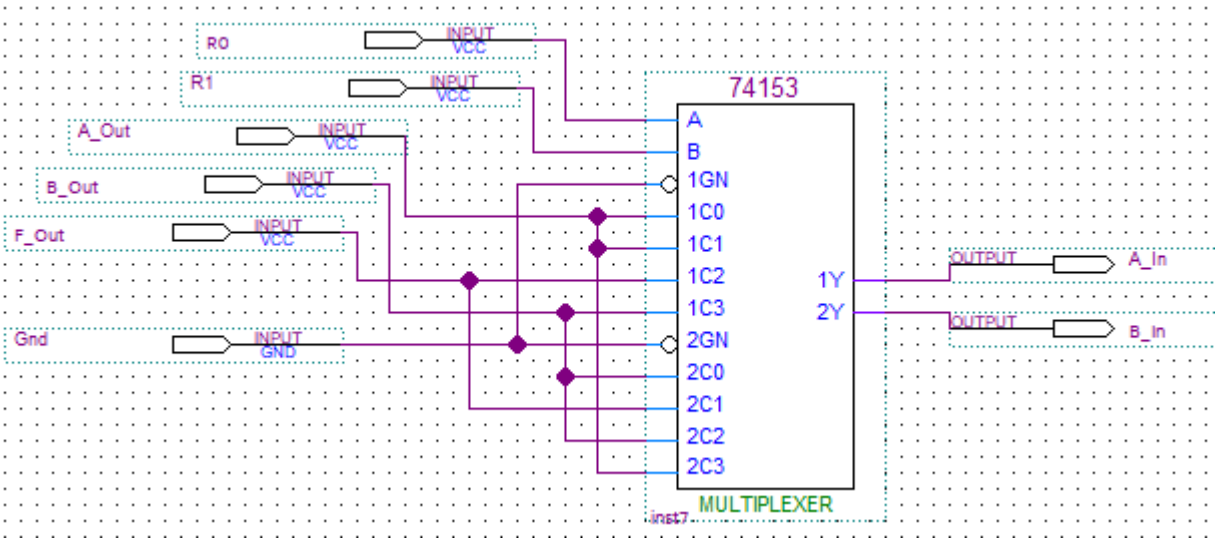
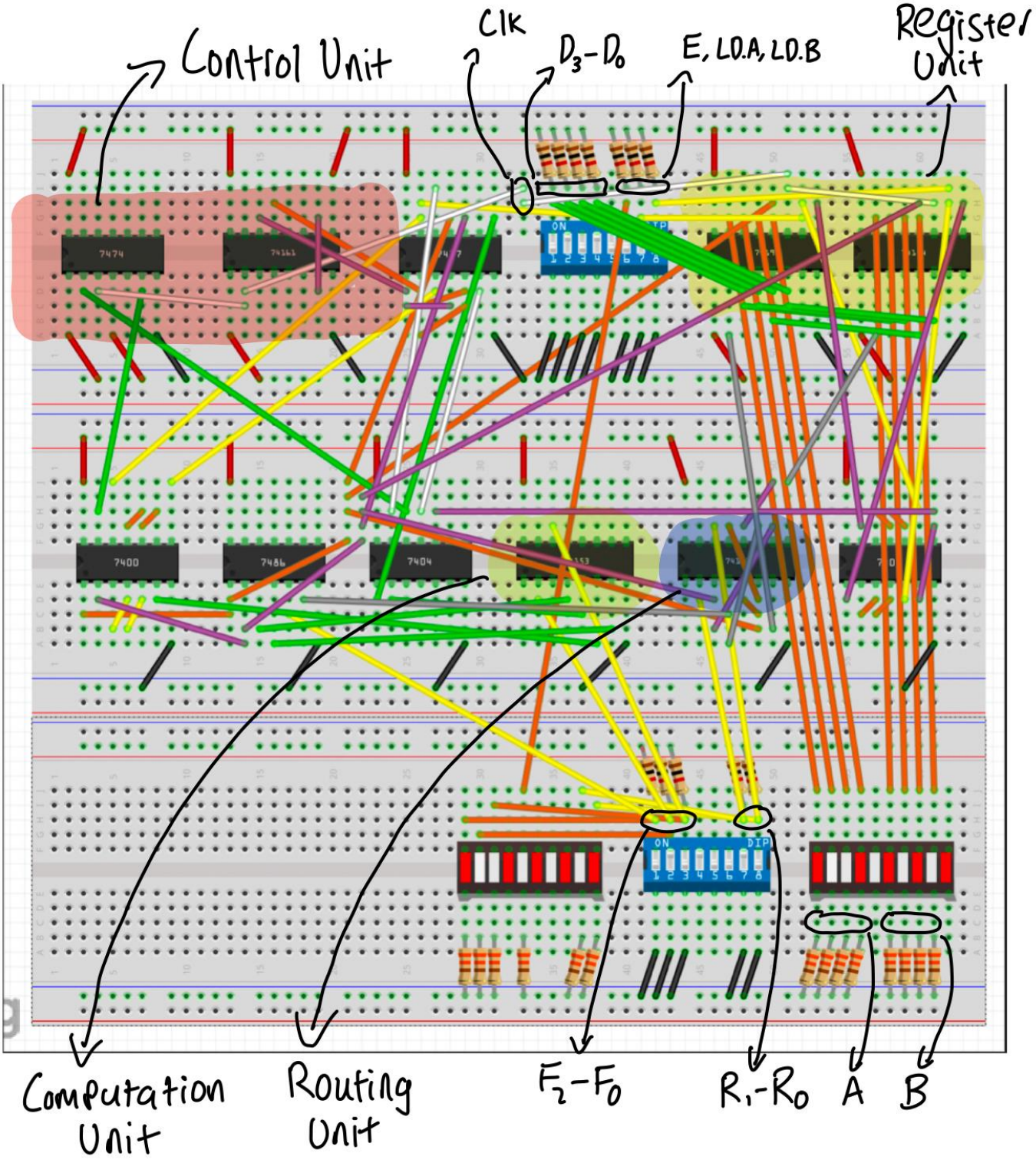


Figure 9: Routing Unit (Input A_Out, B_Out from register unit, F_Out from computation. Output A_In, B_in serves as input for Register Unit).

5. Breadboard view / Layout sheet & Circuit Schematic:



6. 8-bit logic processor on FPGA: Appendix:

Modules:

- **Module: compute.sv**

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

Outputs: A_Out, B_Out, F_A_B

Description: This is a one bit ALU which implements 8 functions on A_in and B_in. It uses F[2:0] to determine which function to Execute, and sets the value of F_A_B to the output of the resulting function. A_Out and B_out are simply set to A_In and B_In respectively.

Purpose: This module is used to create a 8-to-1 mux, which uses F[2:0] as select bits, and outputs the function to be Executed.

- **Module: Control.sv**

Inputs: Clk, Reset, LoadA, LoadB, Execute

Outputs: Shift_En, Ld_A, Ld_B

Description: This is a Moore FSM with a reset state, 8 transition states for shifting, and a hold state. The FSM starts in the Reset state, and transitions to the next state when Execute is pressed. Once Execute is pressed, the FSM transitions for 8 clock cycles, and then stays in the hold state till Execute is 0. If reset is pressed, the FSM transitions back to the reset state regardless of current state.

Purpose: This module describes a Moore FSM and transition conditions for the logic processor

---

- **Module: HexDriver.sv**

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: This makes use of a unique case switch to set the 7 bits of Out0 to the corresponding 4 bit In0, to display the 4 bit hex number on the 7-segment LED.

Purpose: This module maps a hex number to its corresponding value so it can be displayed on a 7-segment LED.

- **Module: Processor.sv**

Inputs: Clk, Reset, LoadA, LoadB, Execute, [7:0] Din, [2:0] F, [1:0] R

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

Description: This is the final Logic Processor. The register_unit, compute, router, control, Hex_driver are instantiated and the necessary I/O connections made to attach all the modules together

Purpose: Top-level entity that describes the overall circuit. Contains all the inputs, outputs, and modules for the subunits of the circuit. Describes the overall functioning of the circuit and ties the 4 units together

---

- **Module: Reg_8.sv**

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

Outputs: Shift_Out, [7:0] Data_Out

Description: This is a positive-edged 8-bit right shift register. When Reset is high, the register synchronously resets and sets Data_Out to 0. When Load is high, D loads into Data_Out. When Shift_En is high, the register right shifts, Data_Out[7] takes the value of Shift_In, Data_Out[7:1] shifts to Data_Out[6:0], and Shift_Out takes the value of Data_Out[0].

Purpose: This module is used to create an 8-bit shift register with holding, shifting, parallel loading capabilities

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

- **Module: Register_unit**

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 creates 2 instances of reg_8 module and accounts for the i/o connections, which will be used to store 8 bit values of A and B

Purpose: This module is used to create the register unit comprising of register A and register B

- **Module: Router.sv**

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

Outputs: A_Out, B_Out

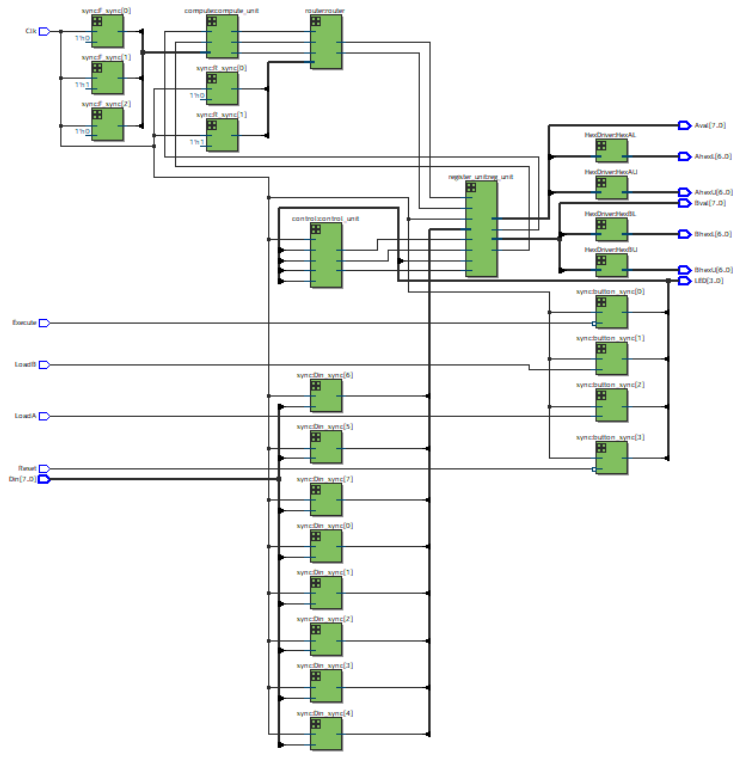
Description: This is a routing unit that uses two 4-to-1 muxes to decide the serial inputs to the register. R[1:0] is used as the select bits while the inputs of the muxes are {B_In, F_A_B, A_In, A_in} and {A_In, B_In, F_A_B, B_In} for A and B respectively. The output of the mux is then set to A_Out and B_Out

Purpose: This module is used to create the routing unit to select the final values to be serially inputted into the registers.

- **Changes made:**

Since the Computation unit and Routing unit only deal with one bit at a time, the only changes that had to be made to convert the 4 bit logic processor to 8 bits were in the control unit and register unit. In the control unit, the FSM had to be changed from 6 states to 10 states to account for the 4 additional shifts to be made. To fix this, in the Control.sv we change the state declaration from enum logic [2:0] to enum logic [3:0], add the 4 additional states, and modify the state transitions to incorporate the 4 states. In Processor.sv, input Din[3:0], output Aval[3:0], Bval[3:0], and additional variables A, B, Din_s must all be changed to [7:0] since the register size has changed to 8 bits. Additionally, since the contents of register A and B are now 8 bits, they require two Hex LEDs each, and two additional instances of hex drivers are to be created to display the upper 4 bits of each register. In reg_4.sv, all the 4 bit inputs and outputs are changed to 8 and the expression for Dout changes from Data_Out <= { Shift_In, Data_Out[3:1] } to Data_Out <= { Shift_In, Data_Out[7:1] }. Similarly, in Register_unit.sv, inputs and outputs of size 4 are changed to 8. Lastly, while running simulations, we use the 8 bit testbench to test instead of the 4 bit version.

RTL Block Diagram:



Simulation of the Processor:

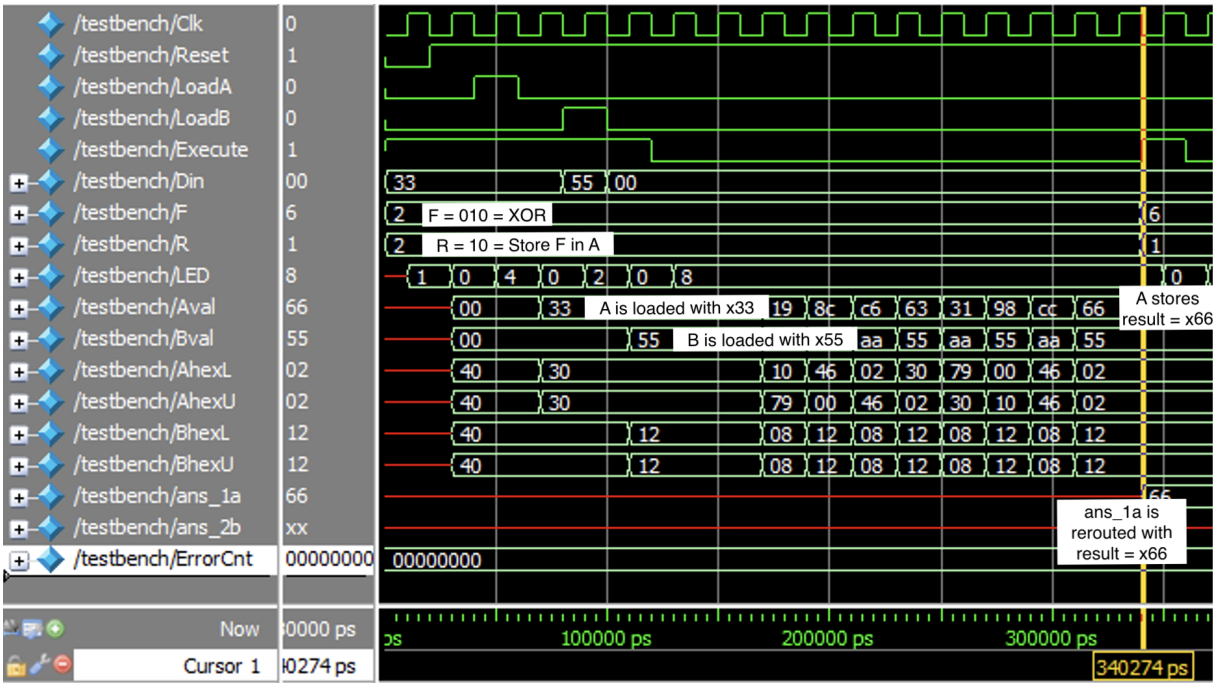


Figure 10: Testbench's ModelSim Simulation.

The testbench executed two functions:

1. Screenshot 1: The testbench set Input A = x33 (at *t ~ 68885 ps*), and Input B = x55 (at *t ~ 109552 ps*). It declared $F_2F_1F_0$ = 010 which means that it decided to perform an XOR operation. The routing unit was set to R_1R_0 = 10 which means that the result was to be stored in A, leaving B unchanged. The simulation executed this and the output was displayed in ans_1a = 66 (which was acquired at *t = 340274 ps*) which is the correct response that we expected from the testbench.

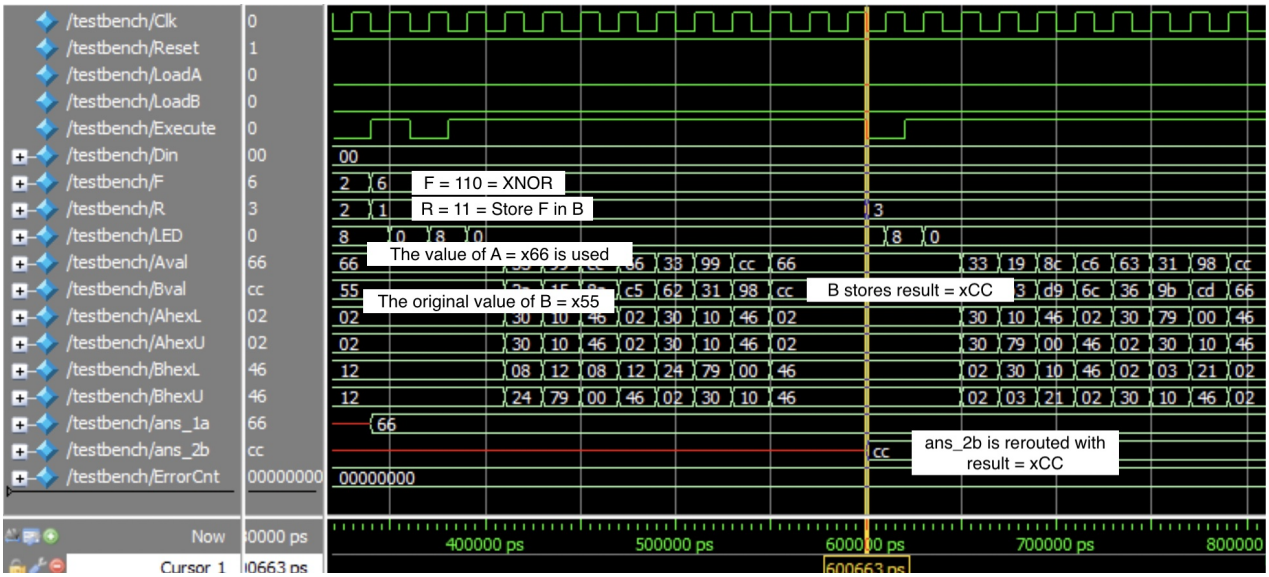
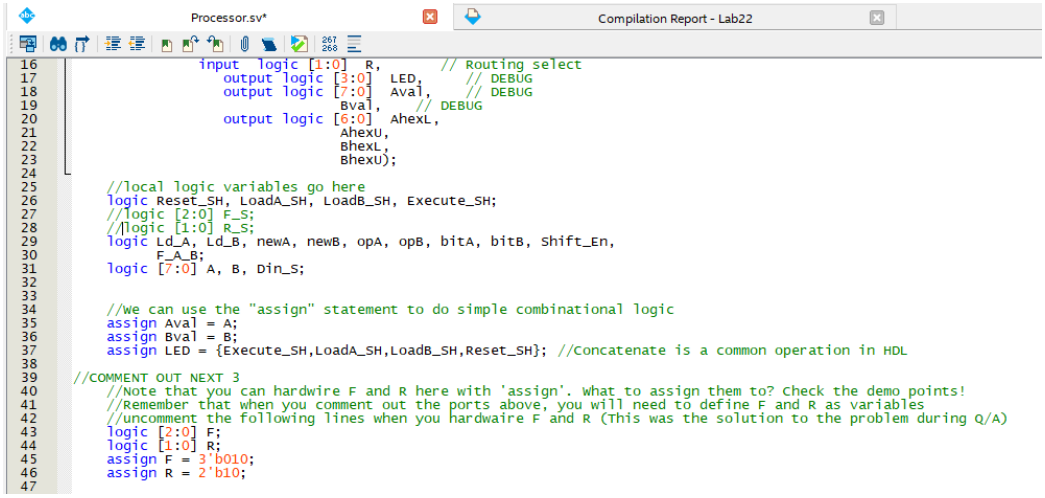


Figure 10: Testbench's ModelSim Simulation.

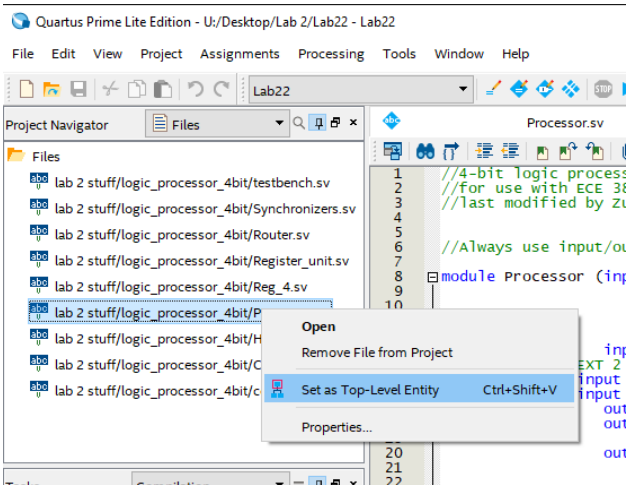
2. Screenshot 2: The testbench then declared $F_2F_1F_0$ = 110 which means that it decided to perform an XNOR operation. The routing unit was set to R_1R_0 = 01 which means that the result was to be stored in B, leaving A unchanged. It used the ans_1a = x66 which was stored in A, and used the original value of B = x55 as inputs, and stored the answer in the output ans_2b = xCC (which was acquired at *t = 600663 ps*) which is the correct response that we expected from the testbench.

SignalTap ILA Trace:

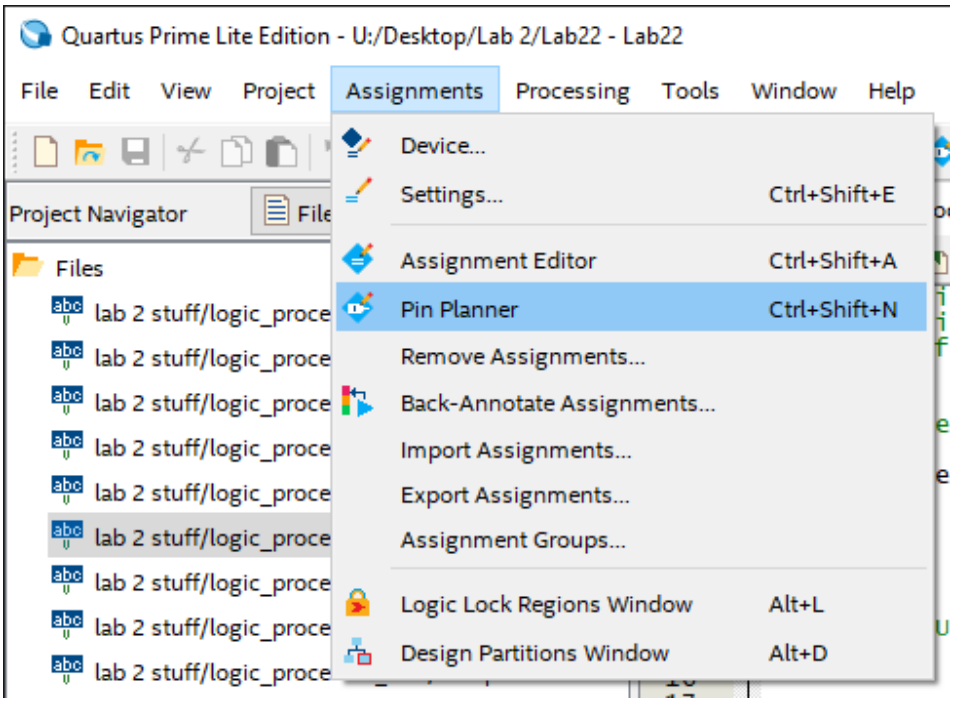
1. After executing the ModelSim version of the circuit, we need to first comment out the version of the F, R that was implemented in logic and hardwire them. Do this by opening 'Files' on the Project Navigation tab, and double click the Processor.sv file. Then, comment out the lines of code that state input logic for F, R; and uncomment out the lines that hardwire a specific value of F, R. In our example, we set F, R to 010 and 10 respectively.



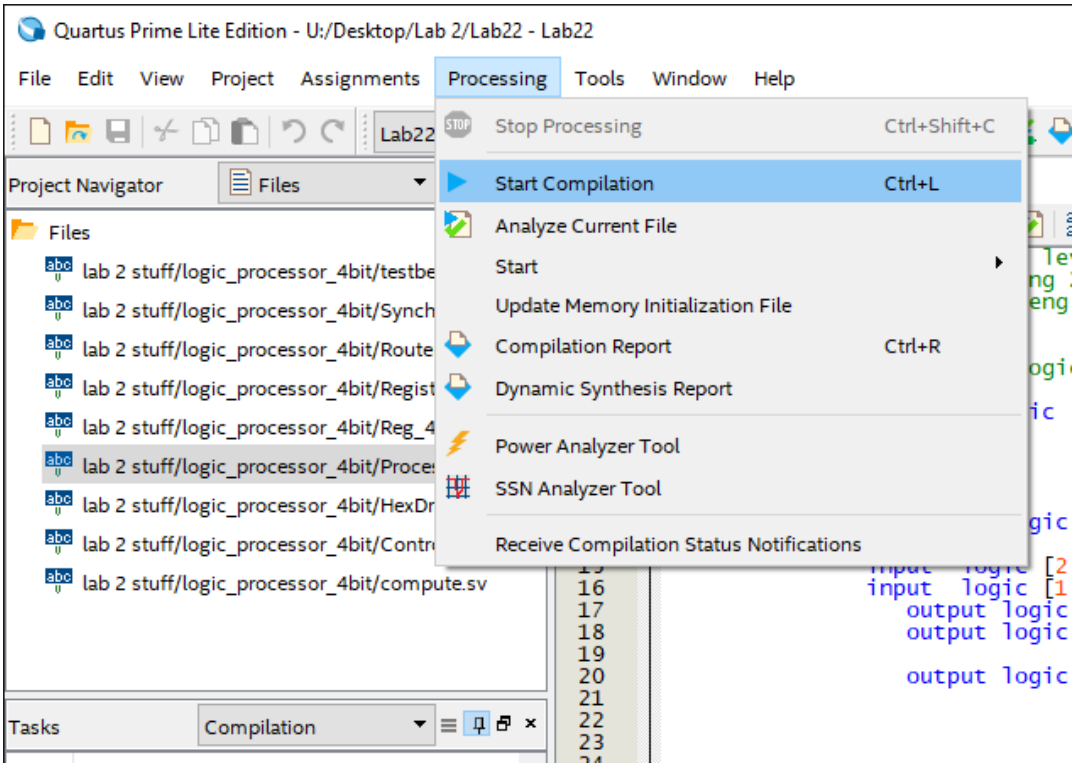
2. We then need to set the top-level Entity. Open ‘Files’ on the Project Navigation tab, and right click the Processor.sv file. Then, select ‘Set as Top-Level Entity’.



3. Set the pin assignments, in order to ensure that the pins mapped can execute the necessary functions via the switches and LEDs assigned. This can be done by selecting ‘Assignments’ on the top tab, selecting ‘Pin Planner’ and assigning the Pins to the necessary values.

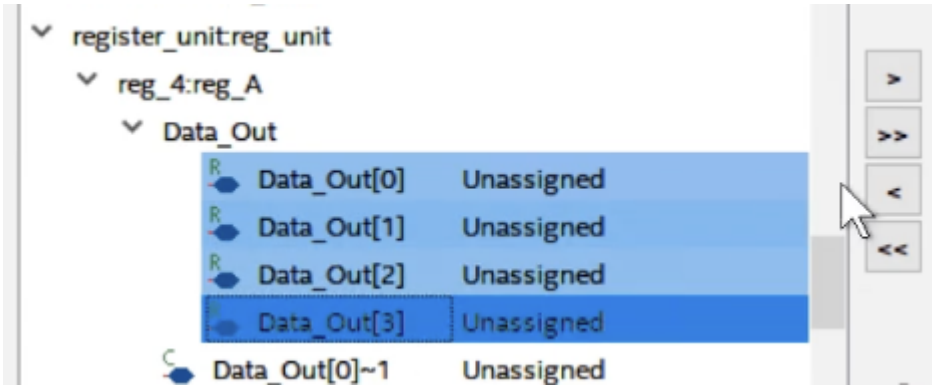


4. Save this file and compile the code. Do this by selecting ‘Processing’ in the top tab, and choosing ‘Start Compilation’. Correct the code until the code compiles successfully.

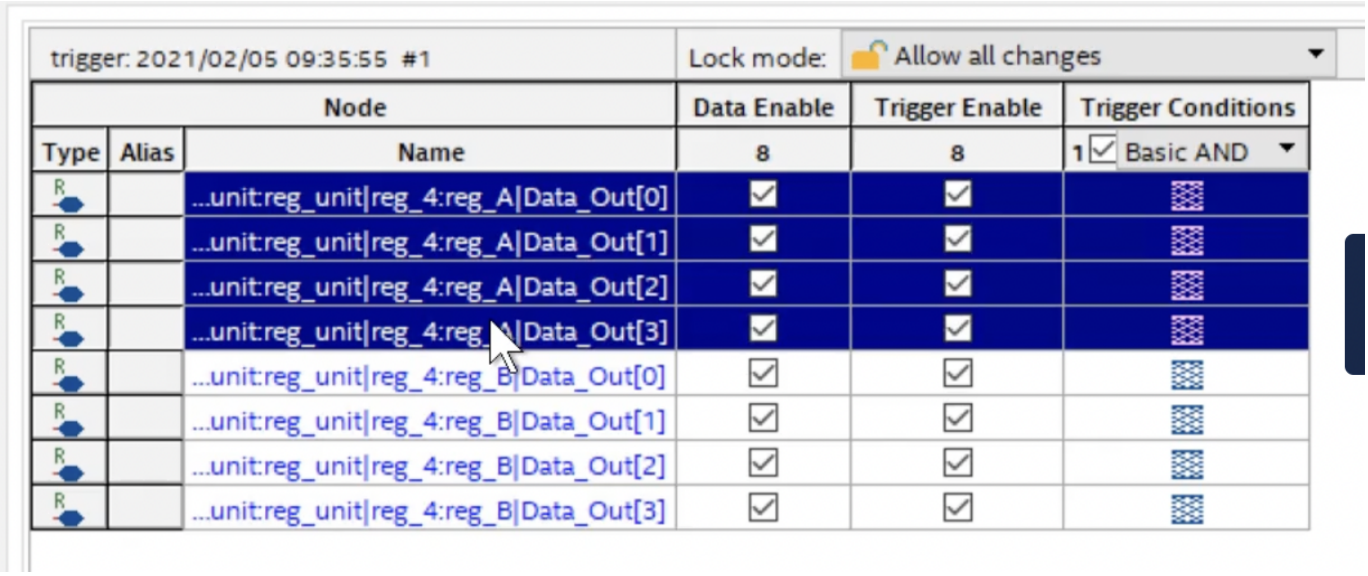


5. Finally, go to the ‘Tools’ Tab, and select ‘Signal Tap Logic Analyzer’. Since it is a logic analyzer, it works with synchronous designs, and so we need to set our own clock for it. We then select the … option near the Clock and type in the *clk* node, selecting the one that has a Pin Assignment that corresponds to our clock. Once the clock is configured, change the sample depth to 64.

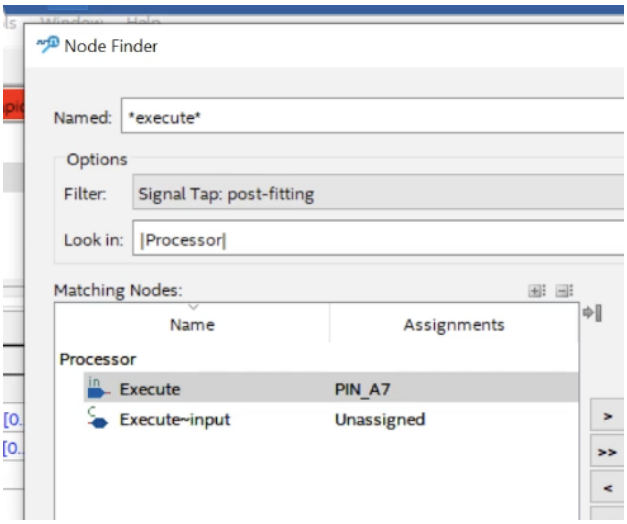
6. Now, start selecting the nodes. Double click in the nodes area as displayed, select *Reg* and shift-select the reg_a's Data_Out and select all outputs from 0-3, then click the '>' arrow. Repeat for reg_b.



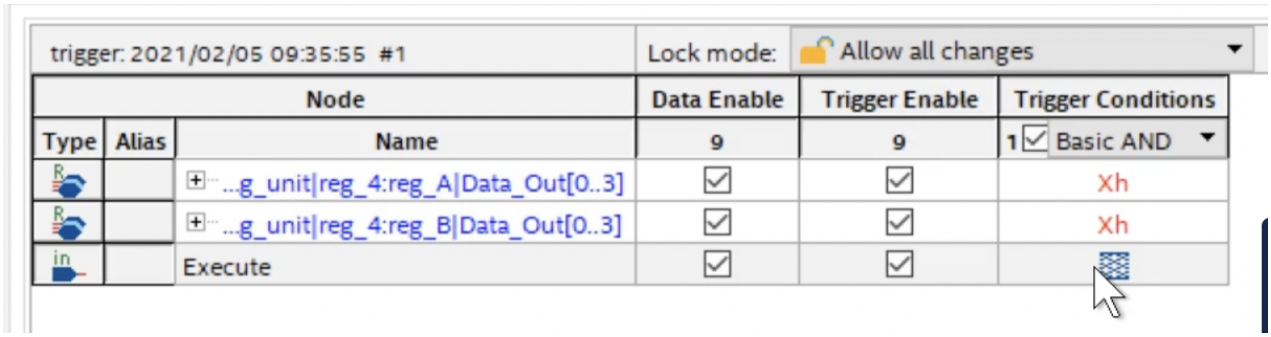
7. Then, select the four for A and right click, select 'Group'. Repeat for B.



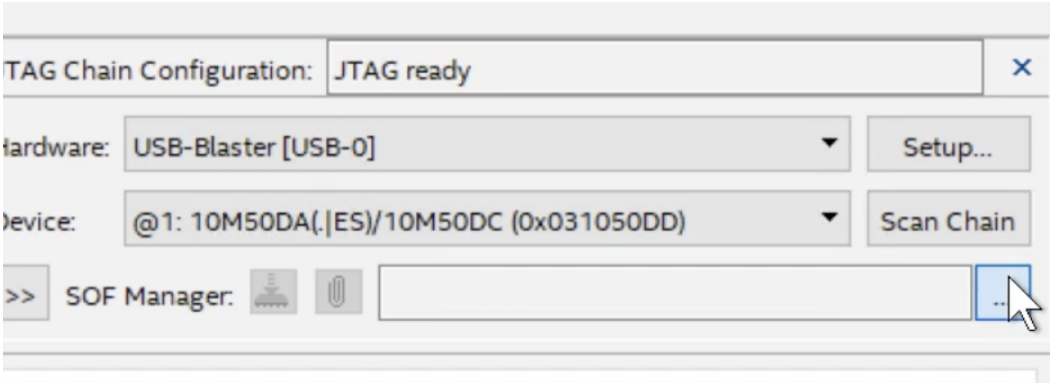
8. Add the execute pin as shown:

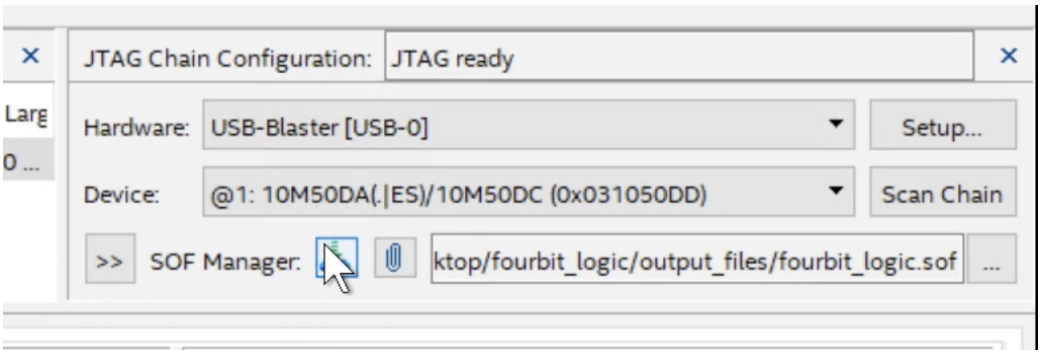


9. Right click on the Trigger condition for the Execute and select the ‘Either Edge’ option.



10. Then, re-compile as shown in step 4. After this, we can select ‘USB Blaster’ from the SOF Manager. Click the ‘>>’ Icon to load it in, and then the download button to load it into the FPGA.





11. Once we can see the Acquisition in Progress' we can then hit the execute button, and on SignalTap observe the waveform and confirm whether it behaves as we expect it to.

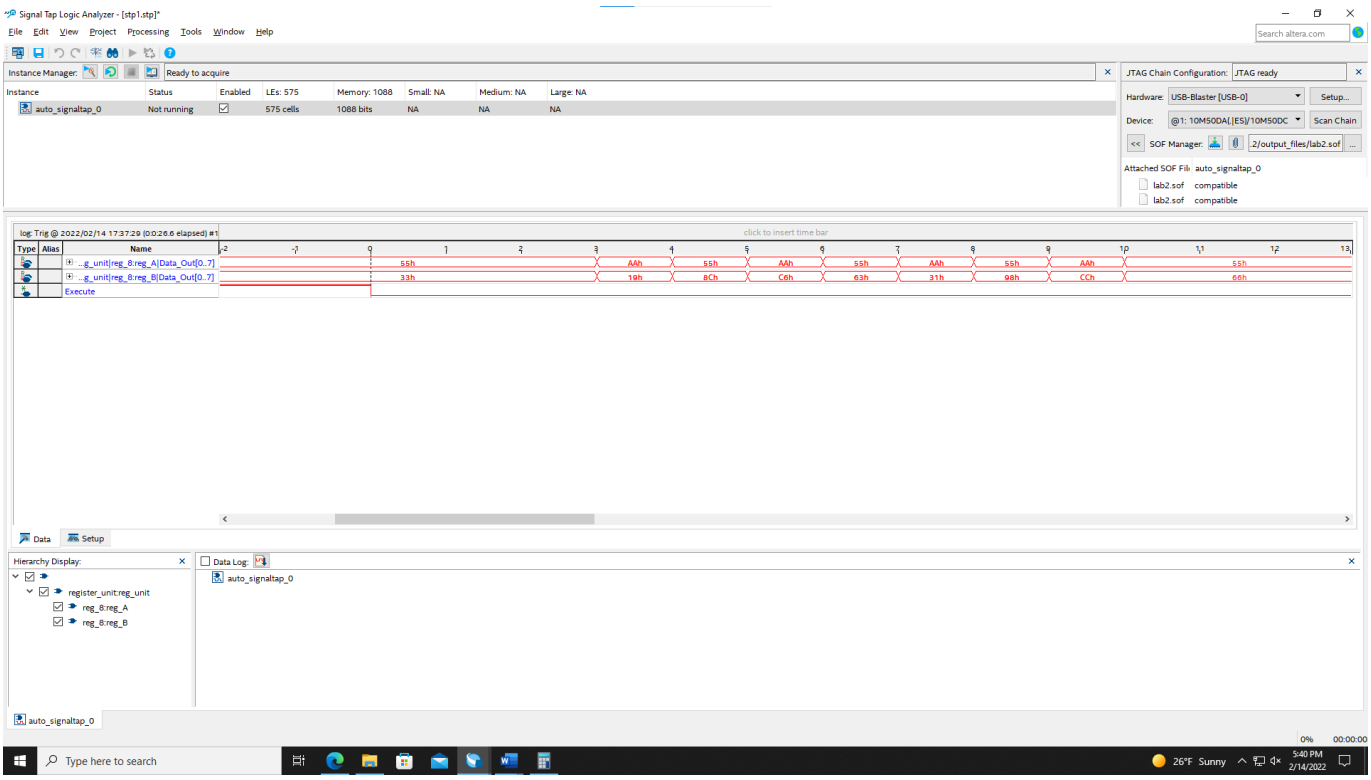


Figure 11: Testbench's SignalTap Simulation.

In this SignalTap result, we have executed A XOR B where A = x55, B = x33. The result is x66 which is clearly visible from our diagram result.

---

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

Initially our counter kept counting, instead of halting after 4 clock cycles, because of which the registers kept shifting. We were able to identify the bug by wiring the control signals such as C1C0, Execute, FSM current state, Shift to LEDs. To fix this, we referred to the data sheet and realized Pins PE and TE of the counter had to be wired to the Shift signal such that the counter would only trigger when the shift signal went high. Once we got the control unit to work as per normal, we noticed that there was an issue with the routing unit. Upon reviewing our circuit design, and referring to the datasheet, we realized pin 1G', 2G' had to be connected to ground in order to enable the mux.

---

8. Conclusion:

In this lab, we first learned how to build a functional hardware bitwise logic operator on a breadboard, by implementing our Mealy FSM. Our circuit was 4-bit based, and we learned how to optimize our circuit for space and design efficiency so it could be extended to a larger set of inputs. At the end of the first week, we were familiar with the various components of our kit and also utilizing the resources available to us to make appropriate design choices. During the second week, we learned how to utilize Systemverilog to implement the circuit we had created in the first week so we could extend our design from 4-bits to 8-bits. We became familiar with the syntax, the usage of ModelSim, the virtual simulator, to monitor our design and any bugs that may be associated with the implementation. Once we created this virtual simulation, we translated the design onto our FPGA, so we could test the functionality on hardware using SignalTap, which examines the signals on the board. For us, our circuit worked but we had to ensure that we were aware of the limitations of our circuit (like Load A, Load B being hindrances to the execution of our circuit). We could enhance the design by attempting extend our 8-bit design to 16 bits via parallel computation, and doubling the creation of the Register, Computation, and Routing units to execute this with more time efficiency. We could enhance our circuit by trying to have additional combinational logic so that the enabling of our load buttons would not override the execution of the control unit.

Post-Lab Questions:

Q1) Describe the simplest (two-input one-output) circuit that can optionally invert a signal. Explain why this is useful for the construction of this lab.

Ans: The simplest circuit that can optionally invert a signal is a 2 input XOR gate. With one input fixed as the input signal, when a ‘0’ is put in as the input to the other terminal, the signal passes through, while when a ‘1’ is put in as the input, the signal is inverted. This is particularly useful while designing the Computation unit. In our function table, the bottom 4 rows ($F_2 = 1$) of the Function table are just inverted versions of the first 4 rows ($F_2 = 0$), and so the final function to be executed can optionally inverted using F_2. Hence, in our design, we only implemented 4 out of 8 functions. This further optimized our circuit because we used a 4-to-1 MUX instead of an 8-to-1 MUX, reducing the area of the circuit. We also use an extra 2-input XOR gate as an inverter by wiring one of the terminals to VCC, to make the design simpler.

Q2) Explain how a modular design such as that presented above improves testability and cuts down development time.

Our circuit design consists of 4 different standalone units (register unit, computation unit, routing unit, and control unit). Each unit can be unit tested by simulating input values and monitoring the output and functionality. This improves the development and debugging process, since each unit

can be individually tested, and bugs can be narrowed down to an individual unit making it easier to identify. For example, the control signals, counter value, FSM state, register contents can all be easily monitored to check if any part malfunctions.

Q3) Discuss the design process of your state machine, what are the tradeoffs of a Mealy machine vs a Moore machine?

Ans: Our circuit requires an FSM that can stay in a reset state, which switches states when Execute is pressed, shifts for 4 clock cycles, holds the state till Execute is disabled, and goes back to the reset state. The FSM takes inputs Execute, C_1C_0 (from the Counter) and produces Shift as the output. The next step was to decide between the Mealy and Moore implementation. The output for Moore Machines depends only on the current state, while for Mealy it depends on the current state and input. Hence, the Mealy machines typically require less number of states. For our implementation, the Moore machine would require 3 states (Rest, Shift, Hold), while Mealy would require only two (Rest, Shift/Hold) since the shift and hold state can be combined. This is why we decided on using a Mealy Machine. The trade off was that by using the Mealy Machine, we had slightly more complex circuitry to compute the output and next state. In the Moore machine, however, we would have more states and so our design would occupy more space. We thus decided on the Mealy implementation, we then drew out the next state table, derived expressions for the next state and output Shift, and designed the circuit using a DFF, 2 bit counter, and few NAND/NOR gates.

Q4) 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?

Ans: ModelSim is a virtual simulator that creates a simulation of the design, and is limited to testing the circuit in theory. It does not check the signals present on the device on which the circuit is being implemented. SignalTap is a logic analyzer that monitors the actual signals on the board. ModelSim requires extra code in SystemVerilog in the form of initializing of the values of the circuit, as well as a test bench to test the circuit in the virtual environment.
ModelSim is more appropriate for cases where one might want to test the individual components of the circuit virtually to debug sections with more minute errors, like forgotten pieces of code, or smaller mistakes that would otherwise be virtually impossible to catch while debugging on hardware. If we attempt to use SignalTap for this, trying to catch the component where the error is being made is infinitely more difficult.
One would prefer using SignalTap, however, to test the discrepancy between the execution of the program in virtual time, versus how it would actually execute in hardware, and SIgnalTap is a more appropriate representation of how the circuit would react in real-time, taking into account the differences caused by hardware limitations, like delays and glitches.
