ECE 385
Fall 2021
Experiment #2

A Logic Processor

Michael Stoens (mstoens2)
Ralph Balita (rbalita2)
Lab Section: AB3

1) Introduction: *Purpose of Circuit.*

The purpose of experiment 2.1 is to design and implement a 4-bit logic processor using integrated circuits. The logic processor can perform operations such as AND, OR, XOR, NAND, NOR, XNOR, logical 0 and logical 1. The purpose of experiment 2.2 is to extend the provided code for a 4-bit logic processor to implement a 8-bit logic processor. The experiments are similar since they both implement a Computation Unit with the same eight functions, along with a Routing Unit to select which data is sent back to the registers. Additionally, both experiments utilize finite state machines for their Control Unit. Since we are already familiar with the functionality of the 4-bit processor from experiment 2.1 and we are provided with code for a working 4-bit processor, experiment 2.2 provides an excellent platform to introduce SystemVerilog as the language that we use to program the FPGA.

2) Operations of the logic processor:

a) Data Loading Sequence

The sequence of switches the user must flip to load data into the A and B registers includes 7 steps.

(1) Ensure the circuit is in the reset state by checking that the execute switch is off
(2) Switch the #1 DIP switch to the on position to enter load mode of register A
(3) Enter data to register A but utilizing switches 3:0 on the DE-10 board
(4) Switch the #1 DIP switch to the off position to exit load mode of register A
(5) Switch the #2 DIP switch to the on position the enter load mode of register B
(6) Enter data to register B but utilizing switches 3:0 on the DE-10 board
(7) Switch the #2 DIR switch to the off position to exit load mode of register B

b) Initiating Computation

The sequence of switches the user must flip to initiate computation & routing operation has 3 steps.

(1) Select the function you would like to perform on the register values using the function select switches (Switches 10:8 on the DE-10 board).
(2) Select the routing of which signals you would like to be shifted into registers A and B using the routing select switches (Switches 8:7 on the DE-10 Board)
(3) Begin the calculation by setting the execute switch to a logical 1.

3) Diagrams

a) High Level Diagram

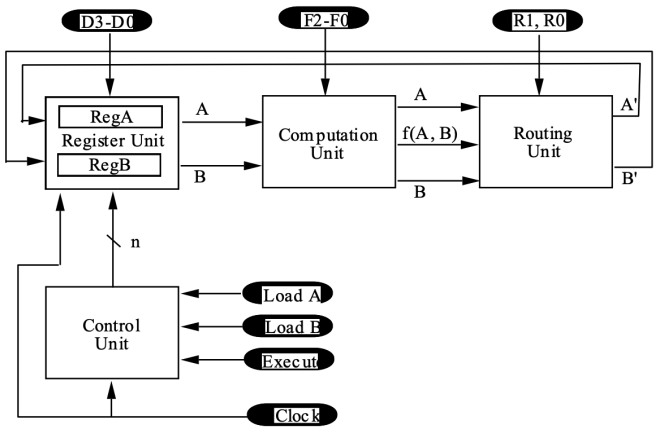
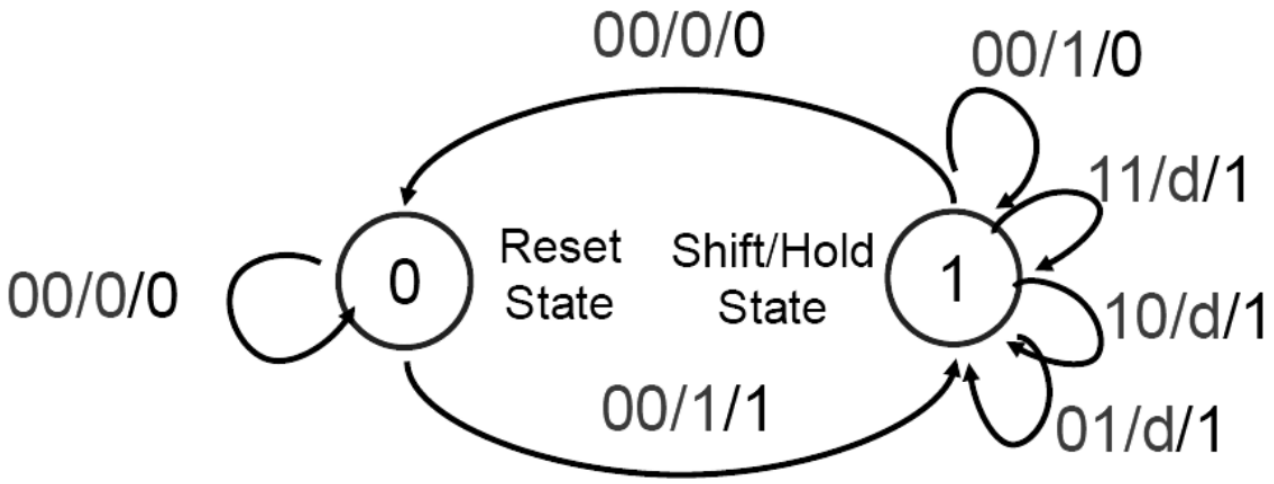


Figure 1: Block Diagram

High Level Diagram Descriptions	
Component Name	Component Description
Register Unit	The Register Unit holds the data that is utilized by the Computation Unit and routing unit. Registers can receive data by manually entering load mode from the reset state and loading all bits simultaneously, or right shifting in bits individually from the routing unit.
Computation Unit	The Computation Unit operates on the values with Registers A and B to implement eight logical functions. The logical function to compute is chosen using the three Function Select switches. The output of the Computation unit is then sent to the Routing Unit. The logical functions implemented by the Computation Unit are: • AND • OR • XOR • Logical 1 • NAND • NOR • XNOR • Logical 0
Routing Unit	The routing unit is used to select which values are shifted into the registers during the computation. This selection is completed using the two Select Switches. The Routing Unit implements four functions: • Both Registers A & B retain their original values • Register A retains its value and Register B receives the computation • Register B retains its value and Register A receives the computation • Register A and Register B swap their values
Control Unit	The Control Unit is used to control the state of the machine. The Control unit is built using a Mealy State Machine, meaning that the output of the machine is based on both the inputs and the current state. The Control unit enables the registers to shift in new values once the Execute switch is changed to a logical 1. After completing the selected computation and loading the data into the selected registers, the control unit disables register shifting and holds until the next computation is executed.

b) State Machine Diagram



For this experiment we built a Mealy Machine with two states. The machine utilizes its current state, along with three inputs to determine the output value. Inputs to the machine include the two least significant bits of the counter - notated using C1 and C0, along with the Execute switch - notated using E. The output of the machine is the Shift/Count Enable signal - notated using S.

States of the Mealy Machine		
State	Name	Description(binary flip flop values / meaningful outputs for each state)
Q = 0	Reset State	The Reset State is used to load values into the A and B registers. In this state, the circuit is prevented from beginning computations until the user changes the execute switch to a logical 1.
Q = 1	Shift/Hold State	The Shift/Hold State is used to enable shifting of the registers. Once in the Shift/Hold state, the registers will begin right shifting and a counter will be enabled that counts to four. When the counter reaches four, the shifting of the registers will be disabled. The circuit will then hold until Execute is switched to a logical 0. In the case that Execute is switched to a logical 0 before the counter reaches four, the circuit will immediately transition to the reset state when the counter reaches four

Arc Descriptions of the Mealy Machine			
State Q	Inputs C1C0/E	Output S	Description
0	00/0	0	Self Arc to Reset State. Waiting for next execution
1	00/1	1	Transition arc from reset to shift state
1	01/x	1	Self-loop to reset state. Count = 1
1	10/x	1	Self-loop to reset state. Count = 2
1	11/x	1	Self-loop to reset state. Count = 3
1	00/1	0	Self-loop to reset state. Count = 4. Counter Disabled.
1	00/0	0	Arc to reset state

4) Design steps taken and circuit schematics

a) Design Procedure

In order to design this machine, we began with analyzing the truth table of the computation unit. Once we had a design for our Computation Unit, we moved on to our Routing Unit and began analyzing it in the same manner. With a design for both our Computation and Routing Units complete, we started on the Control Unit. The Control Unit was the most difficult aspect of the machine to implement. Designing the control unit involved reading several data sheets for different components and figuring out how to use those components to implement the functionality of the Control Unit. We identified multiple designs for our Control Unit as we worked through the lab and ultimately chose the design that we found to be the easiest to implement.

Design Considerations and Revisions

When designing the circuit, the Register Units, Computation Unit, and Routing Unit were rather straightforward. Each unit only took one implementation to result in a simple and effective design. The Control Unit was more difficult. Our first Control Unit design did not utilize a counter from our lab kit as we planned to build our own counter using flip flops. After reading through the data sheets of the provided counters, we identified that implementing logic to pre-load and pause the provided counter would require far less logic than building our own. This made our design easier to implement and troubleshoot since it required less integrated circuits and wire connections.

Circuit Logic Diagrams (K-maps or Truth Tables)

Component Name	Truth Tables/K-maps

Register Unit

Register Load Control		
LD S	*S0*	*S1*
00	0	0
01	1	0
11	X	X
10	1	1

S0 = LD ^ S
S1 = LD

Computational Unit

Combinational Logic Component for Computation Unit [Truth Table]

Inputs		Output							
A	***B***	***AND***	***OR***	***XOR***	***1***	***NAND***	***NOR***	***XNOR***	***0***
0	0	0	0	0	1	1	1	1	0
0	1	0	1	1	1	1	0	0	0
1	0	0	1	1	1	1	1	0	0
1	1	1	1	0	1	0	0	1	0
		If F2 = 0				**If F2 = 1**			

4:1 Mux Component -> Computation Unit Output Logic [Truth Table]

For 4:1 Mux Output				
	Input	***F1***	***F0***	***Mux Output***
AND	~(~[A&B])	0	0	A & B
OR	~(~[A]&~[B])	0	1	A | B
XOR	A ^ B	1	0	A ^ B
1	1	1	1	1

Computation Unit Output	
F2	***Output***
0	~[F2] & Mux Output
1	[F2] & Mux Output

Routing Unit

For 4:1 Mux Routing Unit Output [Truth Table]

Cases	***R1***	***R0***	***A+***	***B+***
STAY	0	0	A	B
B <- f(A, B)	0	1	A	F(A, B)
A <- f(A, B)	1	0	F(A, B)	B
SWAP	1	1	B	A

Control Unit

Combinational Logic Component for Computation Unit [Truth Table]								
State	**Input**				**Output**			
Names	***E***	***Q***	***C1***	***C0***	***S***	***Q+***	***C1+***	***C0+***
Reset	0	0	0	0	0	0	0	0
	0	0	0	1	x	x	x	x
	0	0	1	0	x	x	x	x
	0	0	1	1	x	x	x	x
Shift 1	0	1	0	0	0	0	0	0
Shift 2	0	1	0	1	1	1	1	0
Shift 3	0	1	1	0	1	1	1	1
Shift 4	0	1	1	1	1	1	0	0
Counter Enable	1	0	0	0	1	1	0	1
	1	0	0	1	x	x	x	x
	1	0	1	0	x	x	x	x
	1	0	1	1	x	x	x	(x
Counter Enable	1	1	0	0	0	1	0	0
Shift 2	1	1	0	1	1	1	1	0
Shift 3	1	1	1	0	1	1	1	1
Shift 4	1	1	1	1	1	1	0	0

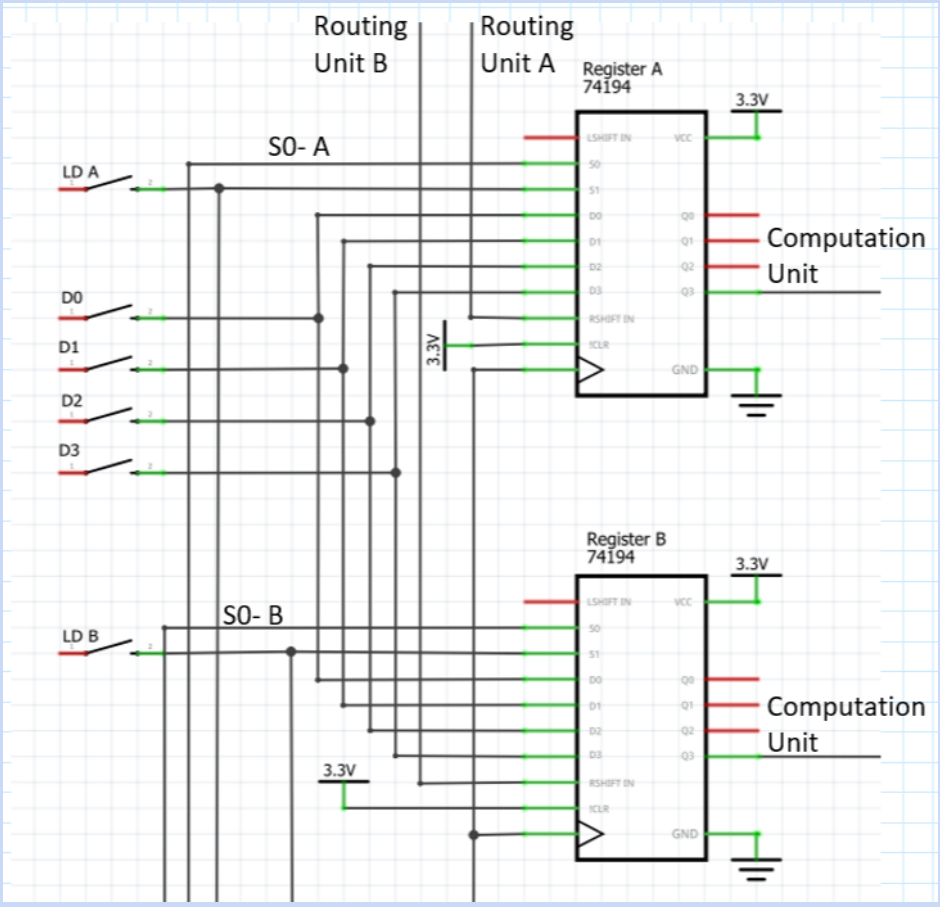
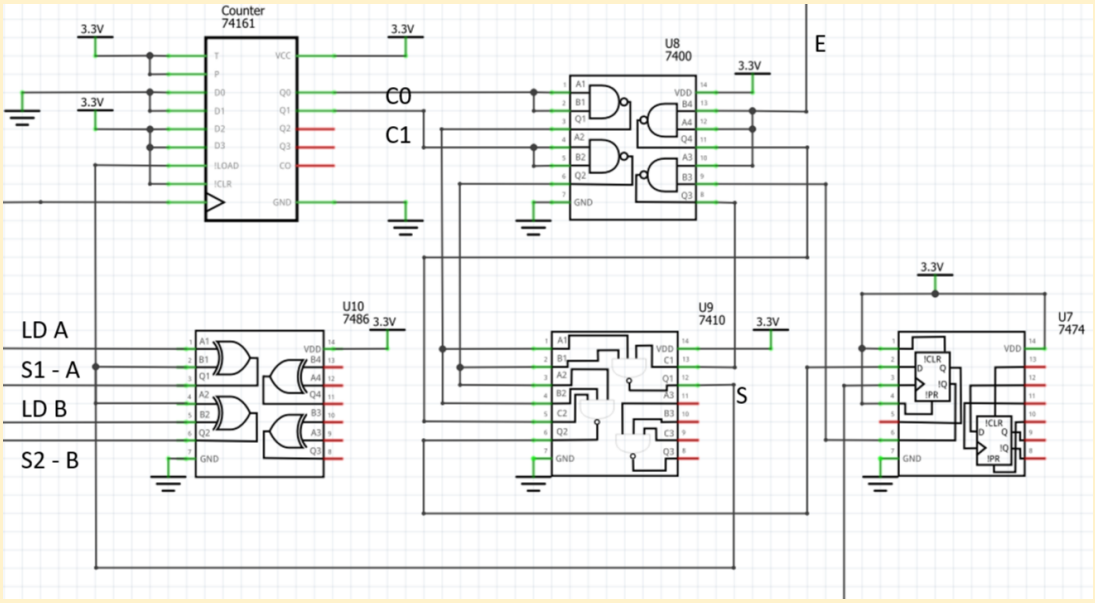
Computation Unit K-map				
S	***C1C0***	***C1C0***	***C1C0***	***C1C0***
EQ	***00***	***01***	***11***	***10***
00	0	X	X	X
01	0	1	1	1
11	0	1	1	1
10	1	X	X	X

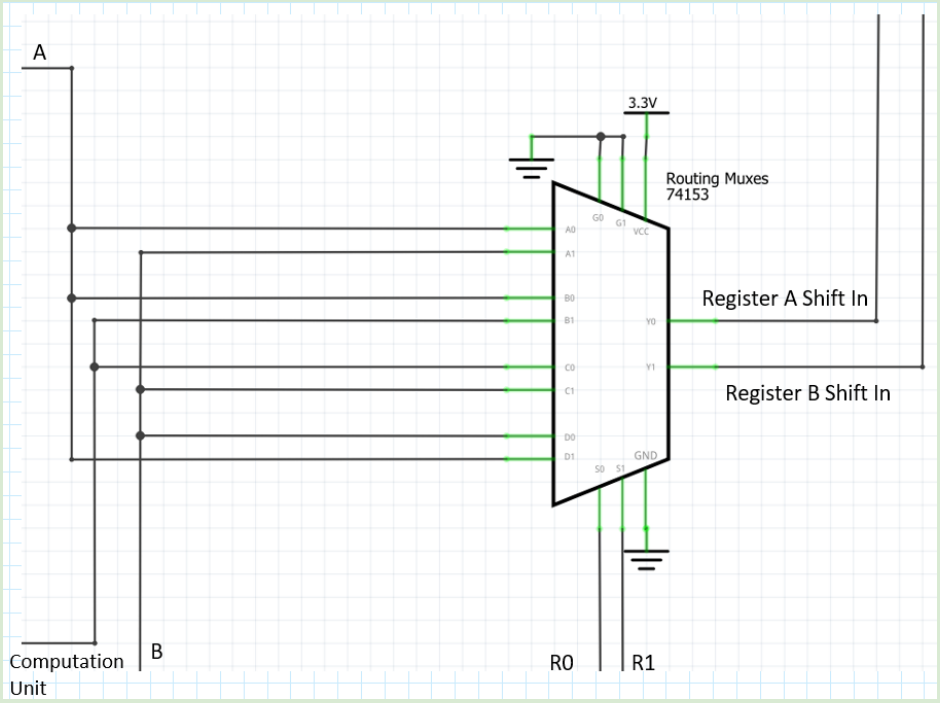
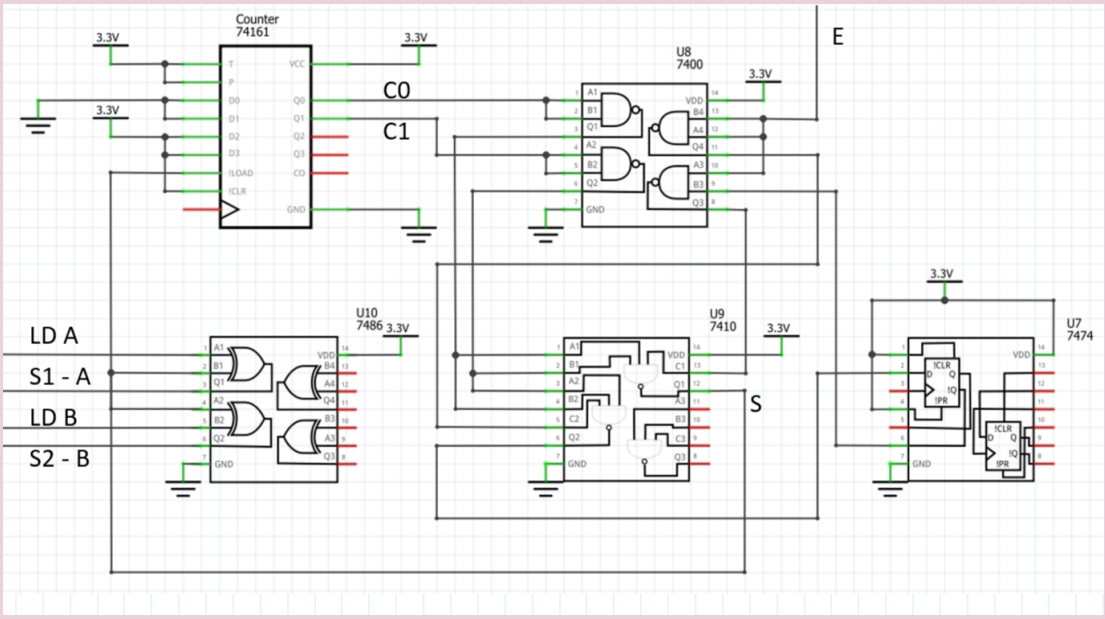
Computation Unit K-map				
Q+	***C1C0***	***C1C0***	***C1C0***	***C1C0***
EQ	***00***	***01***	***11***	***10***
00	0	X	X	X
01	0	1	1	1
11	1	1	1	1
10	1	X	X	X

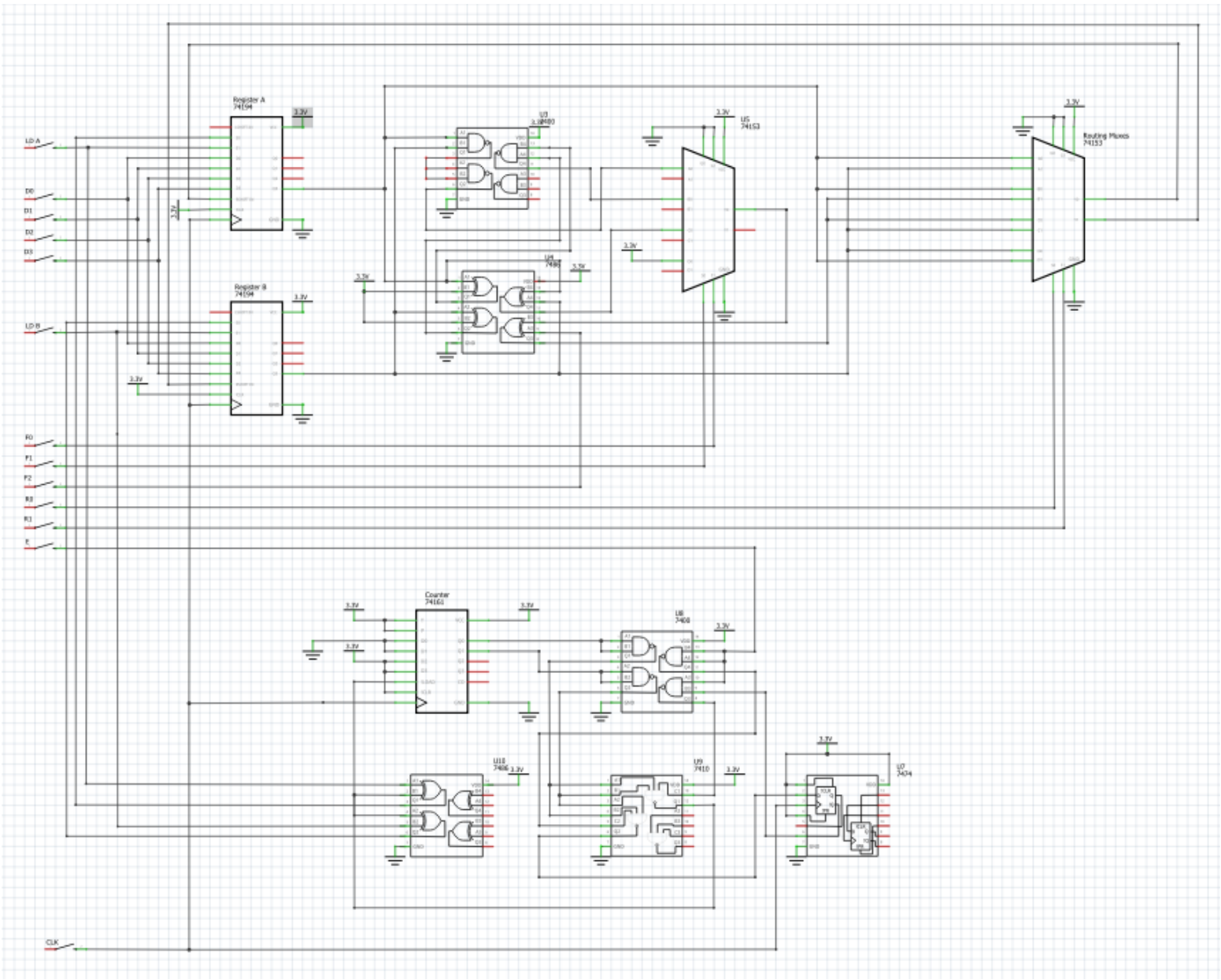
$$S = C1 + C0 + E(\sim Q)$$
$$S = \sim[\sim(C1)\sim(C0)\sim(E\sim(Q))]$$

$$Q+ = C1 + C0 + E$$
$$Q+ = \sim[\sim(C1)\sim(C0)\sim(E)]$$

b) Detailed Circuit Schematic

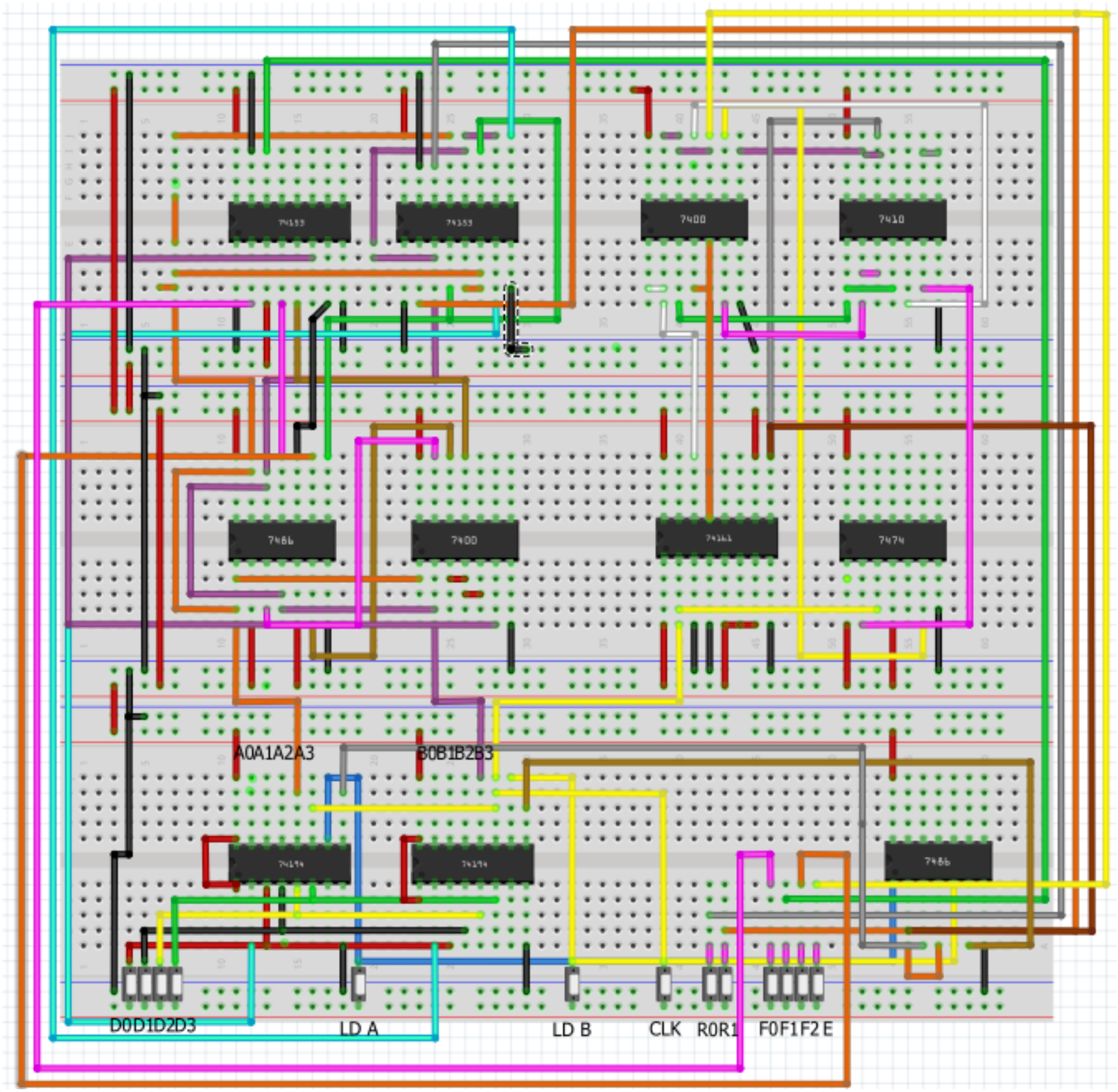
Gate Level Schematic	
Component Name	Fritzing Gate Level Schematic
Register Unit	
ALU: Computational Unit	

Routing Unit	
Control Unit	


Register A
74194
Register B
74194
U3
U5
74153
Routing Muxes
74153
U8
7408
Counter
74161
U6
7400
U18
7406
U9
7410
U7
7404
LD A
LD B
D0
D1
D2
D3
F0
F1
F2
R0
R1
E
CLK
3.3V

5) Breadboard view / Layout Sheet

a) Fritzing Breadboard Layout

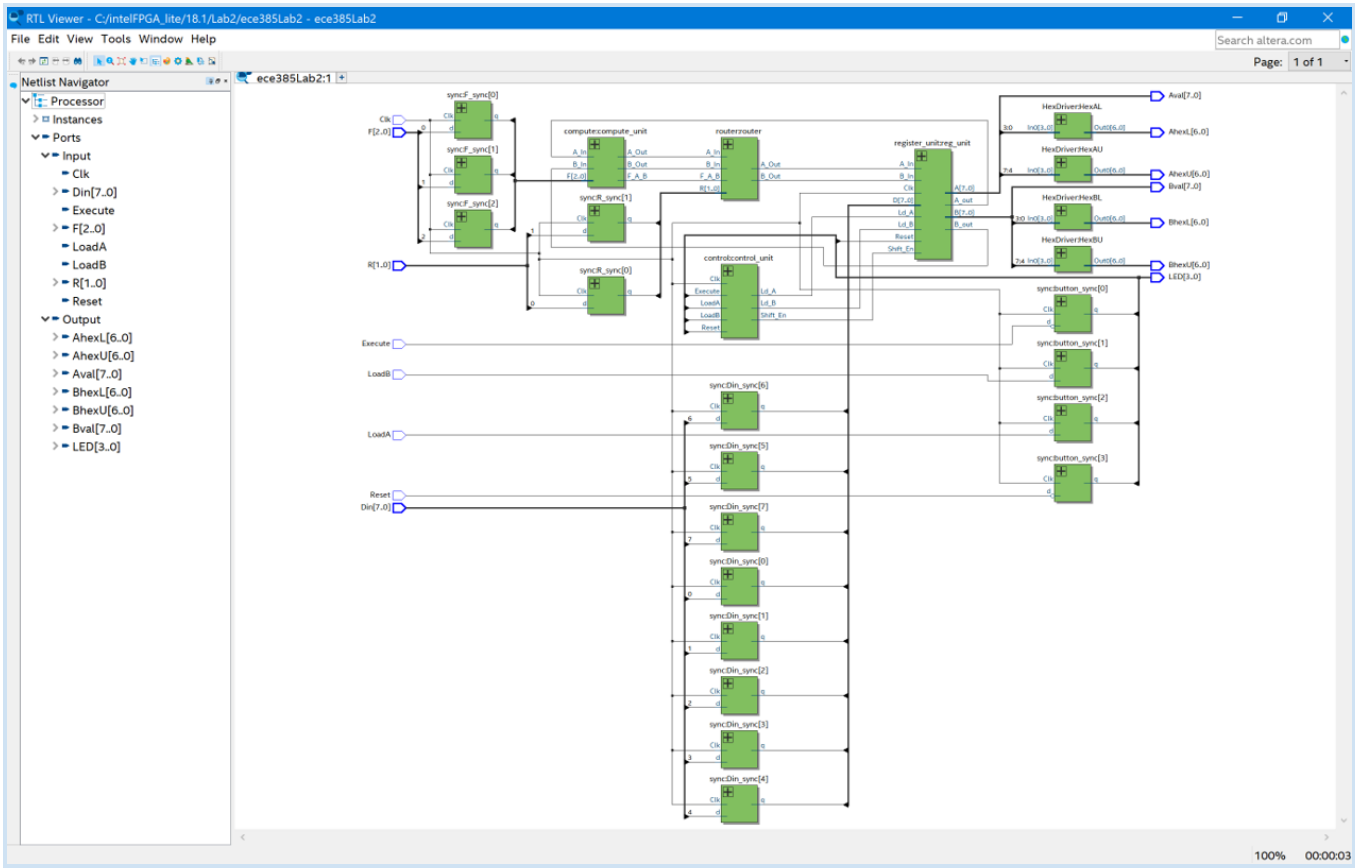


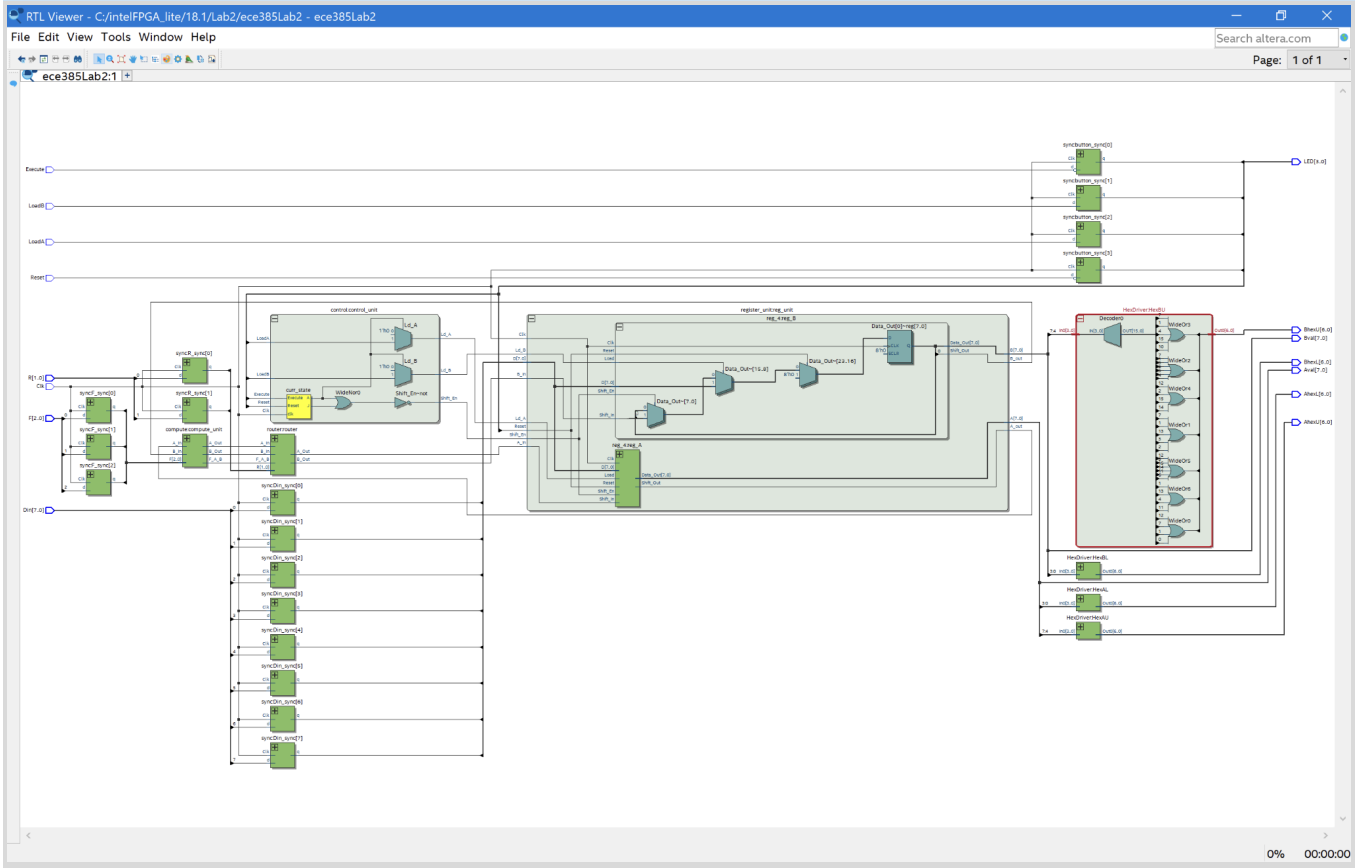
6) 8-Bit logic processor on FPGA

a) Summary of .sv Modules and Changes Made to Extend the Processor to 8-bits

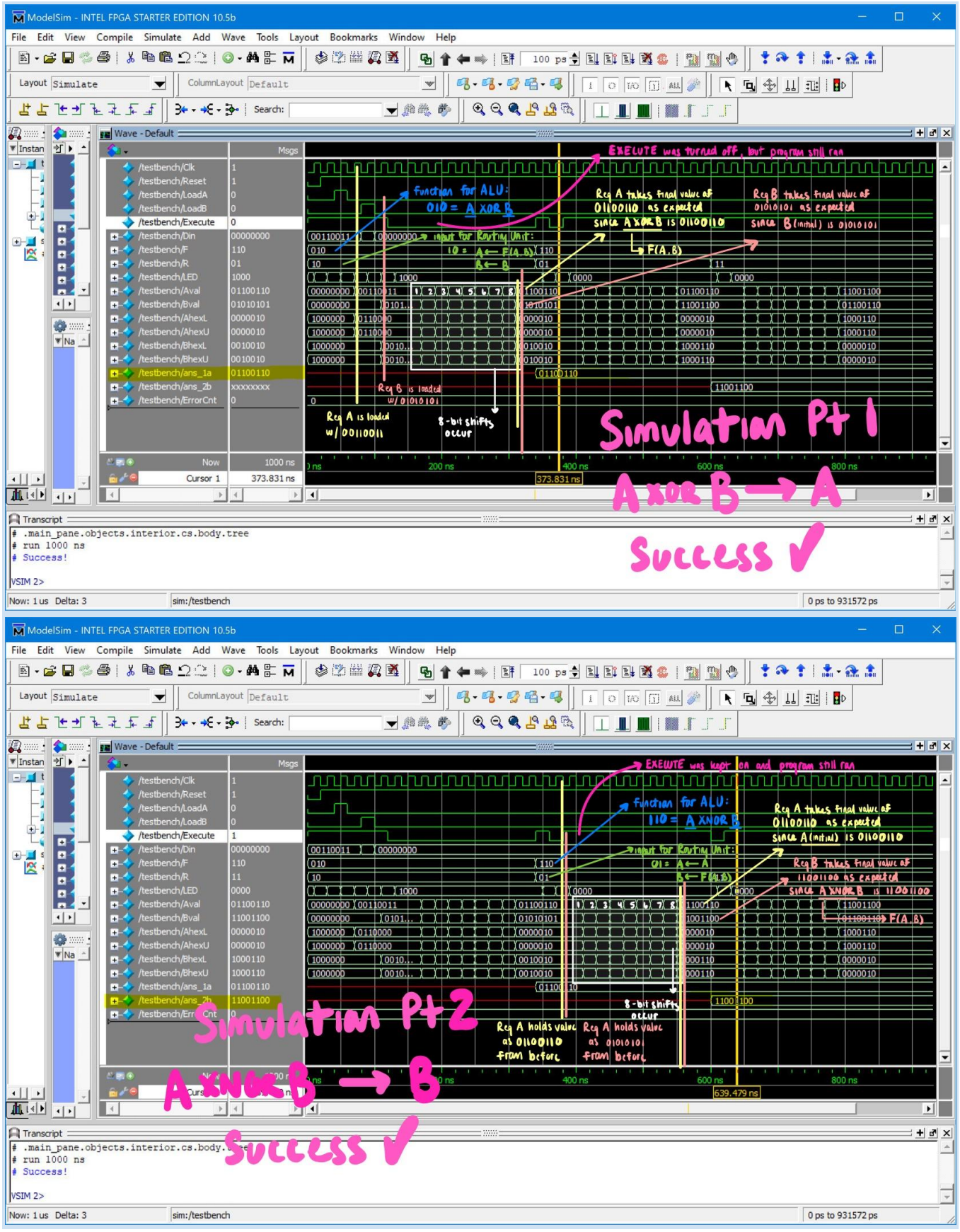
compute.sv	**Summary**	Computation Unit: responsible for holding the logic functions
	Changes	No changes were made, since the unit operates one bit at a time
Control.sv	**Summary**	Control Unit: responsible for implementing the Finite State Machine
	Changes	8 states (A - J) are necessary since we will need to do 8 shifts. Added new cases (A - J) for curr_states and new_states
HexDriver.sv	**Summary**	Hex Display Unit: responsible for holding the 6 bit HEX Display values
	Changes	No changes were made. Already done
Processor.sv	**Summary**	8 Bit Processor Unit: responsible for communicating between modules
	Changes	[3:0] Din, Aval, Bval, A, B, Din were changed to hold 8 bits [7:0] Added bits for the upper ends of the A and B Hex Driver [7:4] Din_sync[3:0] -> Din_sync[7:0]
Reg_4.sv	**Summary**	Register Unit: responsible for holding the shift Register Values
	Changes	Module parameters, D, Dataout were changed to 8 bit [3,0] -> [7,0] Hex and Right shift Dataout values were re-assigned with 8 bit values
Register_unit.sv	**Summary**	Register Unit: Responsible for holding the Shift Register Values
	Changes	Module parameters D, A, B were changed to 8 bit [3,0] -> [7,0]
Router.sv	**Summary**	Routing Unit: responsible for routing 1 bit to the Shift registers (A or B)
	Changes	No changes were made. Router only works one bit at a time
Synchronizers.sv	**Summary**	Synchronized Clock Unit: responsible for running a synced clock (1Hz)
	Changes	No changes were made. clock is independent of 8-bit processing
testbench_8.sv	**Summary**	8 Bit Testbench Module: testbench for 8 bit processor
	Changes	We changed the values of the 4-bit testbench to an 8 bit testbench
testbench.sv	**Summary**	4 Bit Testbench Module: testbench for 4 bit processor
	Changes	No changes were made, since the Computation operates one bit at a time

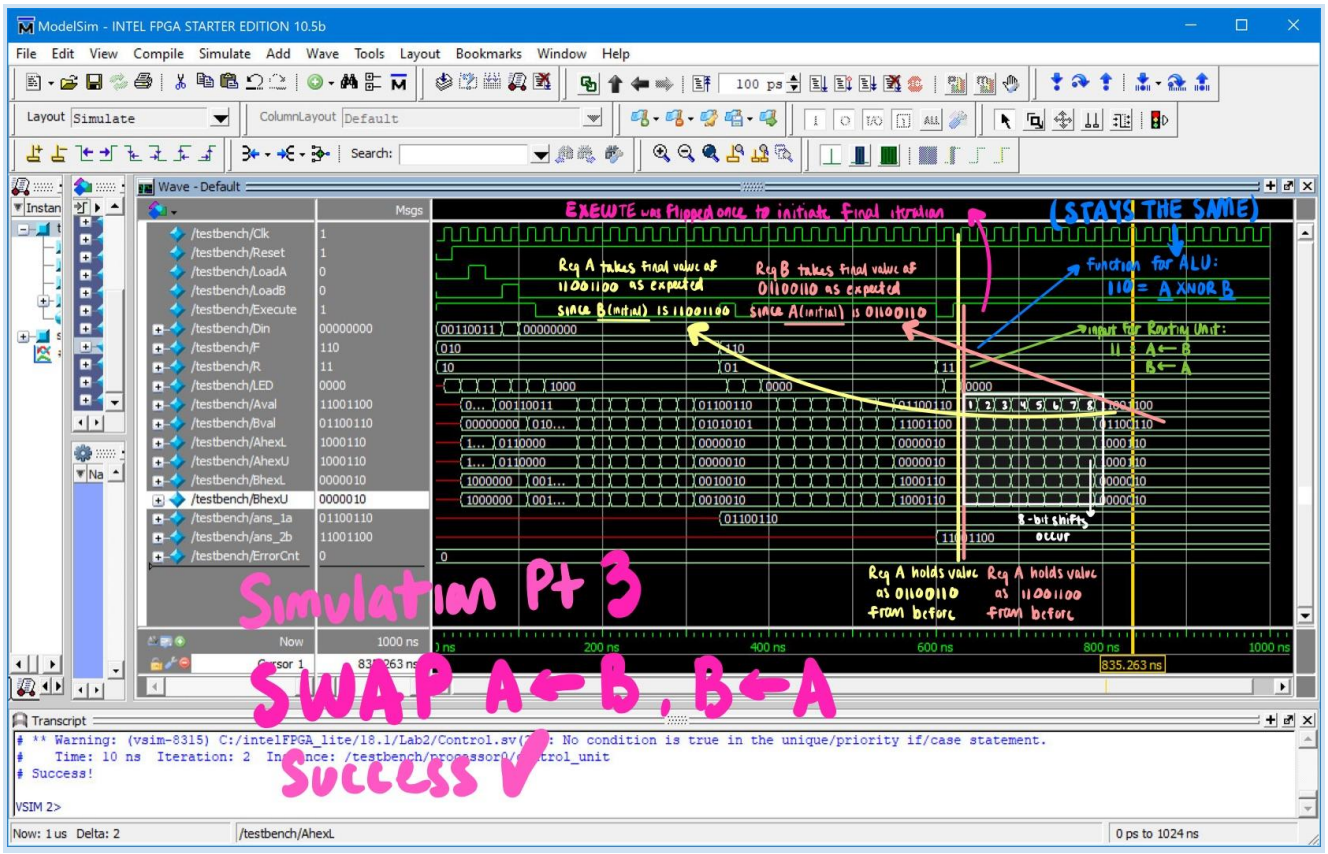
b) RTL Top Level Block Diagram





c) Annotated Simulation of the Processor





d) <u>Procedure to Generate SignalTap ILA trace & Result of Trace Executing 8'h33 XOR 8'h55</u>

(1) Comment out the input parameters (F and R) of the Processor.sv file, and instead "hardwire" the variables F and R in the Processor module since we do not have enough switches on the DE-10

(2) "Assignments" -> "Pin Planner". Set up the pin planner using the recommended assignments in Appendix 2

(3) Open SignalTap Logic Analyzer, set up the clock and add nodes (Register A DataOuts, and Register B DataOuts)

(4) Add node for Execuate, trigger on "either edge" in order to view the important moments where the EXEC becomes active, and we would like to see the changes of Reg A and Reg B

(5) Now we program the device in SignalTap by going to the JTAG Chain Configuration and add the .sof file of the project, and then hit program device

(6) Now we are ready to Acquire. We click on the auto_signaltap_0 and then hit "Acquire"

(7) We should get a reading saying "Waiting for trigger", at this point, we will need to load the 8 bits for A (which are 8'h33) and load in the bits for B (which are 8'h55) on the FPGA switches

(8) Now we can hit the Execute button, and we will get a clear waveform of the Register A and B after we hit the Execute

(9) In this waveform, we should be able to see the 8 cycles of bit shifts/changes of Reg A and Reg B being processed through the 8-bit logic processor

(10) The result for the 8'h33 XOR 8'h55 should be an 8'h66

7) Description of all the bugs encounters, corrective measures taken

For Experiment 2.1, we had bugs with both our computation unit and our control unit. To debug the circuit we used a multimeter combined with known inputs to probe the circuit and identify where the issues were coming from. For our control unit, the issue came from accidentally using the Q signal as the counter enable instead of the S signal. After correcting the misplaced wire the Control Unit worked as intended. The bug we encountered with the Computation unit revolved around not the data appearing to not be shifting properly. After reading through the data sheets several times, we identified that we were accidentally sending the most significant bit of each register to the Computation Unit instead of the least significant bit. After fixing that issue, the computation unit worked perfectly. For Experiment 2.2, one of the learning points happened after compile time. We had already completed the compile, Analysis and Synthesis, and were moving towards creating a waveform from ModelSim. Everytime we had tried to compile and run the simulation through ModelSim, we had been using the alias name of “testbench_8”. We had not noticed that the ModelSim testbench name must be the same as the testbench Modules instead of the name of the .sv file. Only after looking at the code for both the 4-bit and 8-bit testbenches and seeing both files had the module name “testbench” did we recognize our error. It took us hours to realize this, but we now make sure that we are utilizing module names instead of file names.

8) Conclusion

a) Summary

Lab 2 consisted of 2 parts, both of which introduce the idea that hardware implementations (2.1) can be instead implemented using SystemVerilog software implementations (2.2) on the FPGA. As said in 385 lecture, before FPGAs were created, there were PALs and GALs (permanent combinational logic) and CPLDs (combinational logic w/ sequential logic). In the same way that those developments have inspired the new FPGA technology, we were able to appreciate learning the replacement of hardware (hard components) with software (SystemVerilog programming) on the FPGA. This experiment gave us the opportunity to implement the same design in both hardware and software components so that we could learn how to translate between the two.

b) 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.

Input	Invert	Out
0	0	0
0	1	1
1	0	1
1	1	0

XOR Truth Table

The simplest circuit that can optionally invert a signal is a single XOR gate. We can see from the provided truth table that when the Invert input is a logical 0, the output signal is equal to the input signal. When the invert signal is a logical 1, the output signal is the equal to inverted input signal. This is useful for the construction of our circuit because four of the eight functions of our Computation Unit are simply an inverted version of the other four functions (ex. AND vs NAND). By using a XOR gate to optionally invert the output from the Computation Unit, we can simplify our Computation Unit to only implement four functions.

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

In Experiment 2.1, we implemented the Computation Unit, Routing Unit, Register Unit, Control Unit (in that order) all separately. Although we had some problems with actually finding the way they would communicate and work simultaneously (wiring them together), we were able to learn and appreciate the idea of debugging components one at a time. This allowed us to feel the security that our solution would work correctly before actually using them together. We knew that if each component worked properly in an individual sense, it would be easier for us to test and debug our problems when connecting them together.

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

To design our state machine, we first began with identifying how many states and transitions we would need for one computation cycle. From our previous courses such as ECE 120, we were most familiar with Moore state machines so we started to design a state machine with outputs that are solely dependent on its current state. However, we quickly realized that the logic required to implement a Moore machine was quite complex. In lecture we were introduced to Mealy machines in which the output is dependent on both the current state and the inputs. This allowed us to simplify our state machine down to only two states. Although the Mealy machine was harder to understand than a Moore machine, we ultimately chose to design a Mealy machine due to ease of implementation and troubleshooting.

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?

ModelSim is solely used to simulate the SystemVerilog code as though it was on the FPGA. SignalTap on the other hand loads the program onto the FPGA and records the signals being processed in real time. In this case, we looked at the Reg A DataOuts, Reg B Dataouts, and looked at them Triggered by the Execute switch. Considering that both of them generate waveforms, ModelSim's waveforms come from preset inputs from the code of the testbench while SignalTap waveforms come directly from the signals on the FPGA board. ModelSim is used in cases such as testbenching SystemVerilog code on a "step" basis (no need for an FPGA). SignalTap is used to monitor the signals and verify the functionality of the program after implementing the code onto an FPGA.