ECE 385

Fall 2022

Experiment #2

A Logic Processor

Jialiang Xu/Jinrui Hu

NetId: jx17/jinruih2

Lab Section: NY / 09/01/2022

TA's Name: Neo Yuan

1 Introduction

In this Lab, our goal was to build a bit-serial logic operation processor. The processor is capable of :

1) loading and storing two manually set 4-bit binary values (see **register unit** in section 4.2);

2) calculating eight different functions: AND, OR, XOR, NAND, NOR, XNOR, CLR (output 0), SET (output 1) (see **computation unit** in section 4.3);

3) routing the results back to the register unit bit-by-bit in four ways: output to A only, output to B only, output to none, and swap A/B (see **routing unit** in section 4.4).

To achieve this goal, a **control unit** (see section 4.1) featuring a Finite State Machine was implemented overtop the other three units to control the general behavior of the circuit.

2 Operation of the Logic Processor

In this section we describe the details of performing two specific operations that the processor is capable of.

2.1 Loading Data into the Registers

To load data into the registers A and B, the user must first toggle on the switches to manually set up the intended 4 bit binary value (the D3-D0). In our design, these correspond to the switches 1-4 on the red mechanical switch box. Then, the user must flip on the switch for “Load A” or “Load B” depending on the intended loading target. In our design, these two switches are switches 7 and 6 on the FPGA switch box. After the values are loaded the switches should be restored off.

2.2 Initiating a Computation and Routing Operation

To initiate a computation and routing operation, the user must first select the type of calculation by flipping the switches for F2-F0. In our design, these switches correspond to the switches 6-8 on the red mechanical switch box. Then the user must select the routing pattern by flipping the the switches for R1-R0. In our design, these two switches are switches 9 and 8 on the FPGA switch box. Then the user must flip the switch according to the “execute” signal. In our design, this is switch 5 on the FPGA switch box. After execution is done, it should be restored off for the next run.

3 Written Description, Block Diagram and State Machine Diagram of Logic Processor

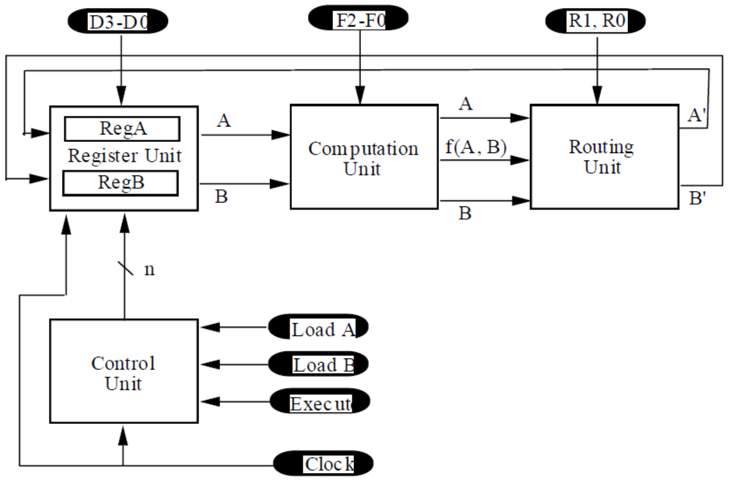
3.1 Written Description

The logic processor is consisted of the following 4 major components:

- **Control Unit**: The control unit consists of a Dual D-Type FF (SN74HC74N), a counter (SN74HC161N) and a 3-way NOR (SN74HC27N). It is responsible for sending out signals indicating the current status of the circuit. It controls the register unit in a way that when the calculation finishes, the calculation results get shifted bit-by-bit, and when the values are loading, the registers are configured to conduct loading in a parallel manner.
- **Register Unit**: two 4 bit BIDIR shift registers (CD74HC194E) are utilized to implement Register A and Register B. The registers can parallelly load data from D3-D0 switches, or take serial inputs from the routing unit. The registers hold the numbers to be calculated and the output results of the calculation. They receive signals from the control unit to decide whether to shift or load. The CLK pin is used to receive clock signals.
- **Computation Unit**: the computation unit in our design leverages five chips: Quad NAND (CD74AC00E), Quad NOR (CD74AC02E), Quad XOR (SN74HC86N), NOT (CD74AC04E), 8:1 MUX (SN74HC151N). Each time a calculation happens (Execute signal is sent), the computation unit parallely computes the results of all calculations, and the MUX selects (decided by the F2-F0 signals) the intended result as its output.
- **Routing Unit**: a single dual 4:1 MUX chip (SN74HC153N) is used to implement the routing unit, which determines the data flow path of the calculation results. Each 4:1 MUX determines the new value of one of the registers. With the R1-R0 switches, four ways of routing are addressed.

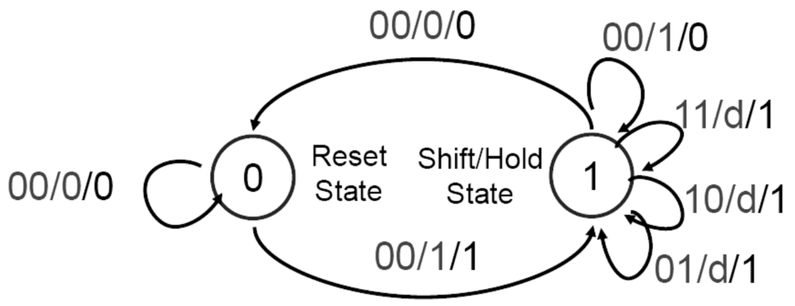
3.2 High-level Block Diagram.

In this experiment, we first physically implemented the circuit on the breadboard with n = 4. Then we switched to SystemVerilog and extended to n = 8 on the DE-10 Lite FPGA board.



3.3 State Machine Diagram

In this experiment, the Mealy machine is adopted for its simplicity in the number of states. There are two states in the Finite State Machine: **Reset** and **Shift/Hold**. The reason for the merge between Shift and Hold is that they share the same static parameters and the only difference lies in their outputs with different inputs. The arc labels are in the format **"Counter/Execute/Shift",** where **Counter** takes two bits and **d** means **"don't' care"**. Below is a diagram of the Finite State Machine.



4 Design Steps Taken and Detailed Circuit Schematic Diagrams

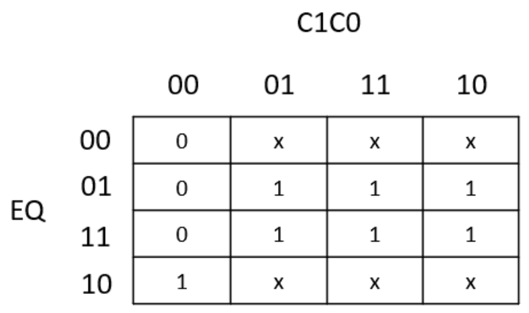
4.1 Control Unit

The control unit consists of a Dual D-Type FF (SN74HC74N), a counter (SN74HC161N) and a 3-way NOR (SN74HC27N). It is responsible for controlling the different chip behaviors in the circuit. Namely, a Shift signal is produced by the control unit to indicate whether the register should take in data from Din in a parallel manner or should they take in bit-by-bit function outputs while shifting right. When one calculation starts, the registers should start shifting for 4 clock cycles. When the calculation finishes and the execution switch is restored, the circuit should get ready for the next calculation. This behavior intuitively gives rise to a Finite State Machine with the following Truth Table. In the table, *Q* is the state variable, *C1* and *C0* are the counter bits indicating the passed clock cycle number. *S* is the output Shift signal, and the plus symbol indicates corresponding values for the next clock cycle.

Truth Table for the Mealy State Machine

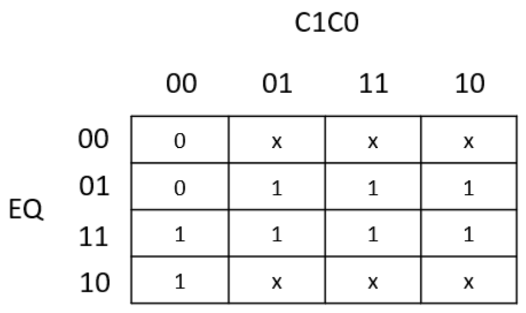
Exec. Switch ('E')	Q	C1	C0	Reg. Shift ('S')	Q^+	$C1^+$	$C0^+$
0	0	0	0	0	0	0	0
0	0	0	1	d	d	d	D
0	0	1	0	d	d	d	D
0	0	1	1	d	d	d	D
0	1	0	0	0	0	0	0
0	1	0	1	1	1	1	0
0	1	1	0	1	1	1	1
0	1	1	1	1	1	0	0
1	0	0	0	1	1	0	1
1	0	0	1	d	d	d	D
1	0	1	0	d	d	d	D
1	0	1	1	d	d	d	D
1	1	0	0	0	1	0	0
1	1	0	1	1	1	1	0
1	1	1	0	1	1	1	1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

The Truth Table can be converted to the following K-maps for the two important output signals:



EQ \ C1C0	00	01	11	10
00	0	x	x	x
01	0	1	1	1
11	0	1	1	1
10	1	x	x	x

1. K-maps for Output Shift (S)



EQ \ C1C0	00	01	11	10
00	0	x	x	x
01	0	1	1	1
11	1	1	1	1
10	1	x	x	x

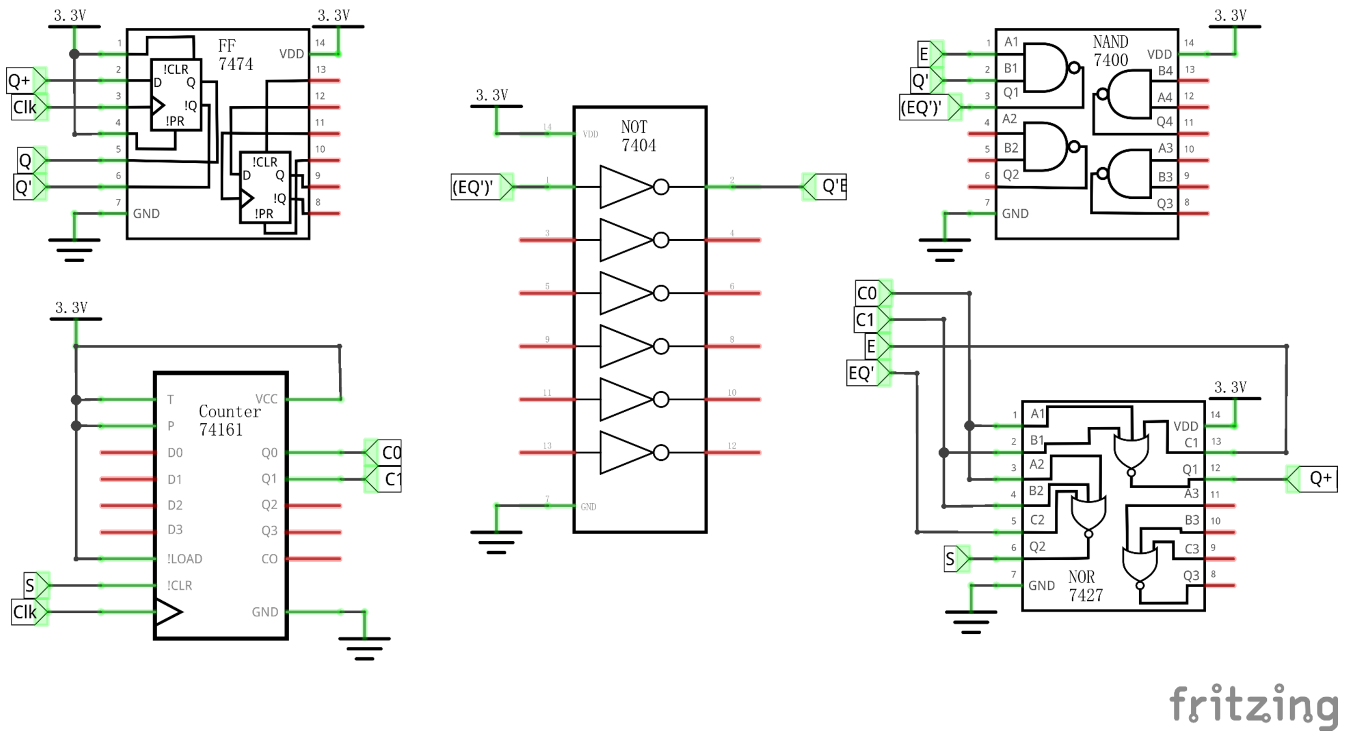
2. K-maps for Output Next State (Q+)

From the K-maps above, the following expressions can be derived:

$$S = EQ' + C0 + C1$$

$$Q+ = E + C0 + C1$$

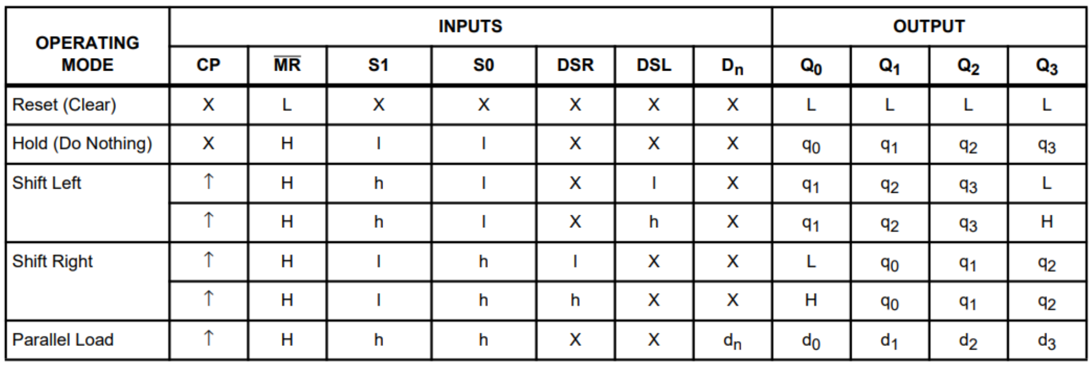
Additionally, we use the shift signal to change the registers' operating modes, as when the shifting happens the parallel loading should be shut down and the serial loading should begin. Below is a detailed schematic of the control unit as a whole:



4.2 Register Unit

The register unit is responsible for storing the values to be calculated and the output of the calculation. Two 4 bit BIDIR shift registers (CD74HC194E) are utilized to implement Register A and Register B. The loading mode is specified by the S1S0 pins (see the Truth Table below from the datasheet).

TRUTH TABLE



OPERATING MODE	INPUTS							OUTPUT			
	CP	$\overline{MR}$	S1	S0	DSR	DSL	D_n	Q_0	Q_1	Q_2	Q_3
Reset (Clear)	X	L	X	X	X	X	X	L	L	L	L
Hold (Do Nothing)	X	H	l	l	X	X	X	q_0	q_1	q_2	q_3
Shift Left	↑	H	h	l	X	l	X	q_1	q_2	q_3	L
	↑	H	h	l	X	h	X	q_1	q_2	q_3	H
Shift Right	↑	H	l	h	l	X	X	L	q_0	q_1	q_2
	↑	H	l	h	h	X	X	H	q_0	q_1	q_2
Parallel Load	↑	H	h	h	X	X	d_n	d_0	d_1	d_2	d_3

Here, a combinational logic is used to determine the loading modes of the registers. The registers should only load in parallel if the Load A/B signals are set to logic ones, and the registers should do Shift Right and Serial Loading if the circuit is in calculation status and the results are being produced. This observation gives the following K-maps:

Shift \ Load	0	1
0	0	1
1	0	1

1. K-maps for Output S1

Shift \ Load	0	1
0	0	1
1	1	1

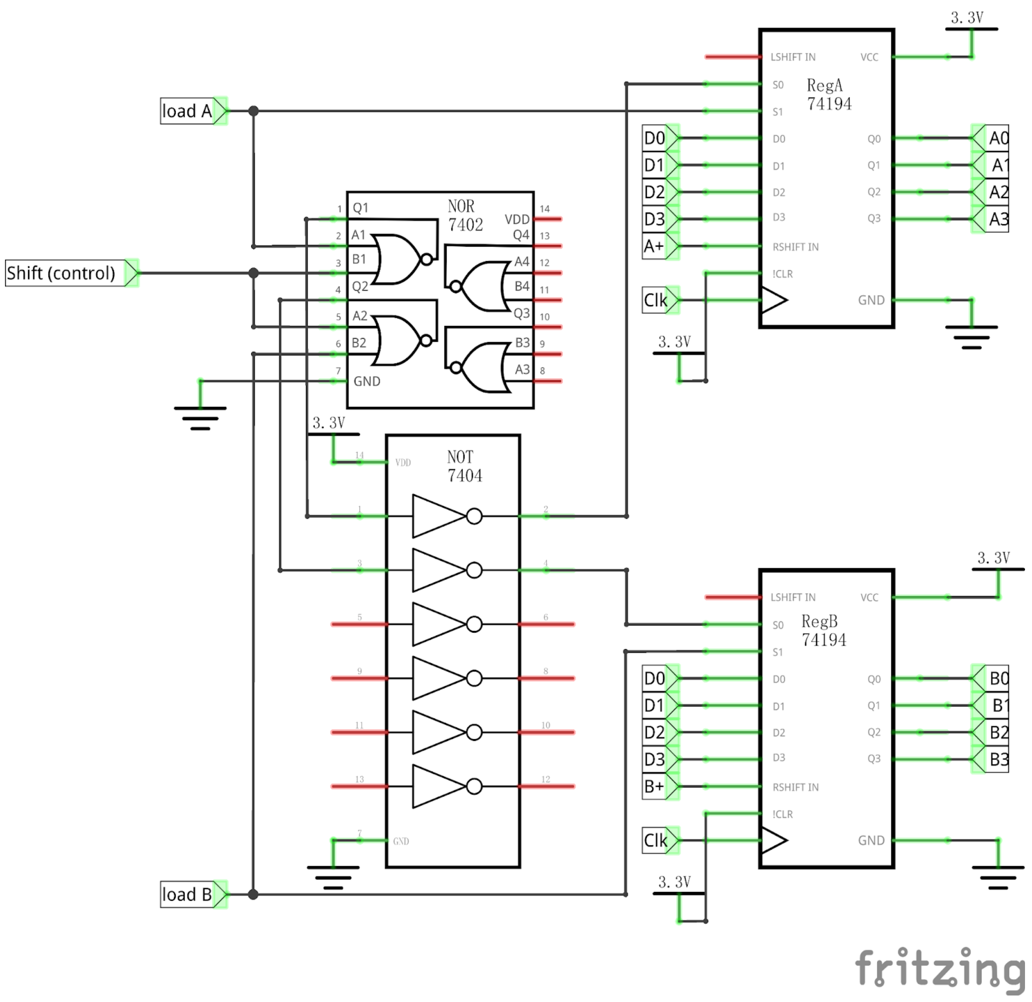
2. K-maps for Output S0

Thus, the following expressions that can be derived:

$$S1 = Load$$

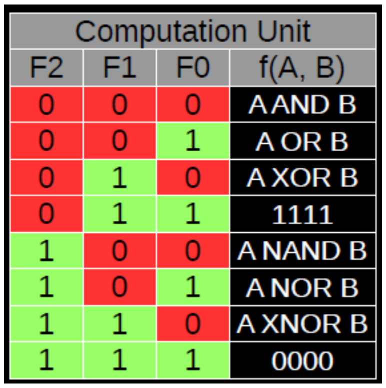
$$S0 = Load + Shift$$

Below is a detailed schematic of the control unit as a whole:



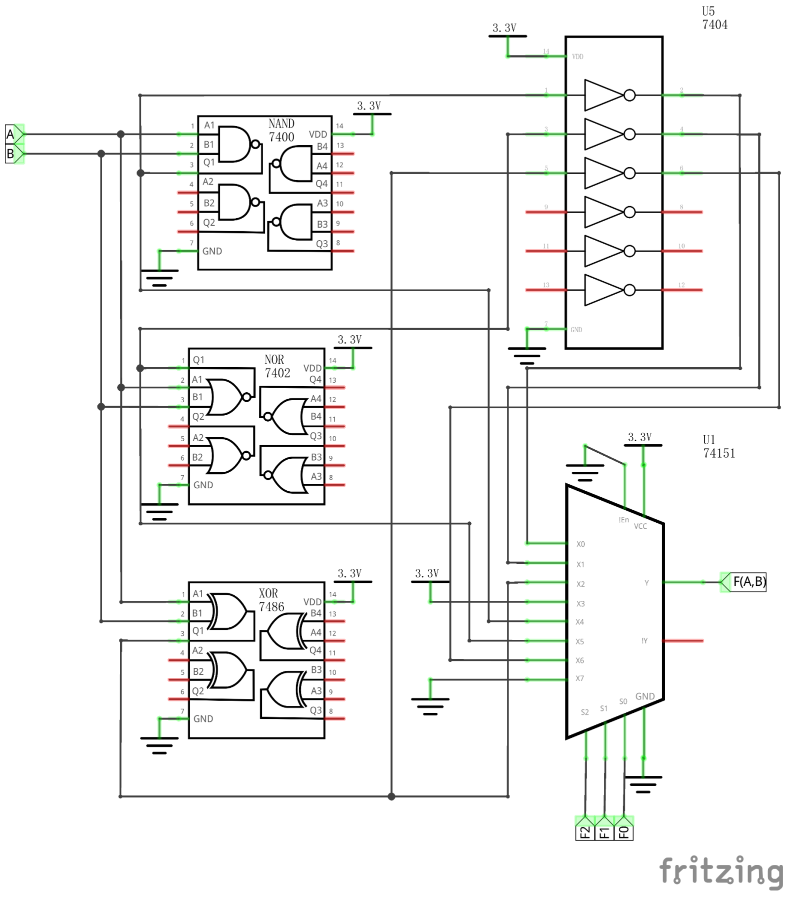
4.3 Computation Unit

The computation unit is responsible for producing the final calculation results of 8 operations specified below with logic gates.



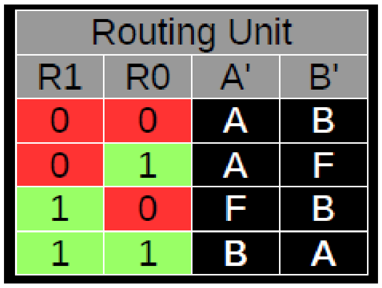
Computation Unit			
F2	F1	F0	f(A, B)
0	0	0	A AND B
0	0	1	A OR B
0	1	0	A XOR B
0	1	1	1111
1	0	0	A NAND B
1	0	1	A NOR B
1	1	0	A XNOR B
1	1	1	0000

In our design, five chips are leveraged: Quad NAND (CD74AC00E), Quad NOR (CD74AC02E), Quad XOR (SN74HC86N), NOT (CD74AC04E), 8:1 MUX (SN74HC151N). Each time a calculation happens (Execute signal is set), the computation unit parallely computes the results of all calculations, and the MUX selects (decided by the F2-F0 signals) the intended result as its output. The output is then routed back to the register unit for display.
Below is a detailed schematic diagram of the computation unit.



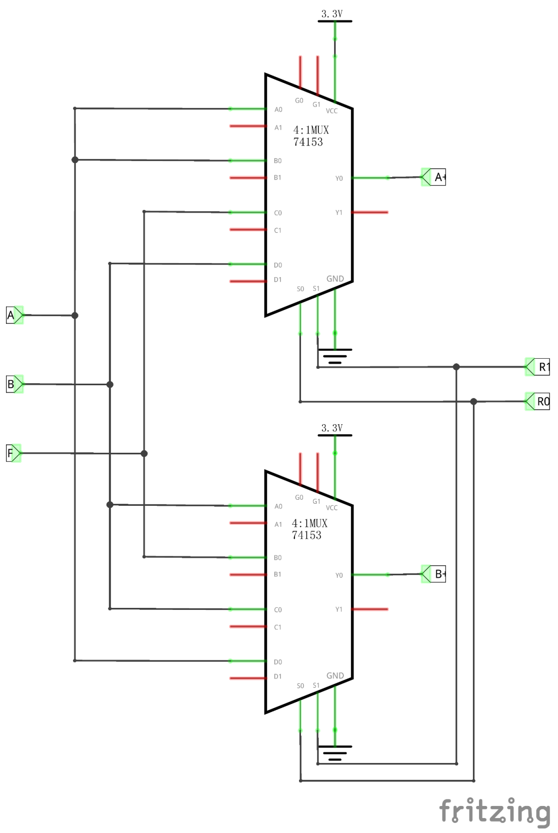
4.4 Routing Unit

The routing unit is responsible for correctly routing the calculation results to the intended register, give a 2-bit command code as specified below.

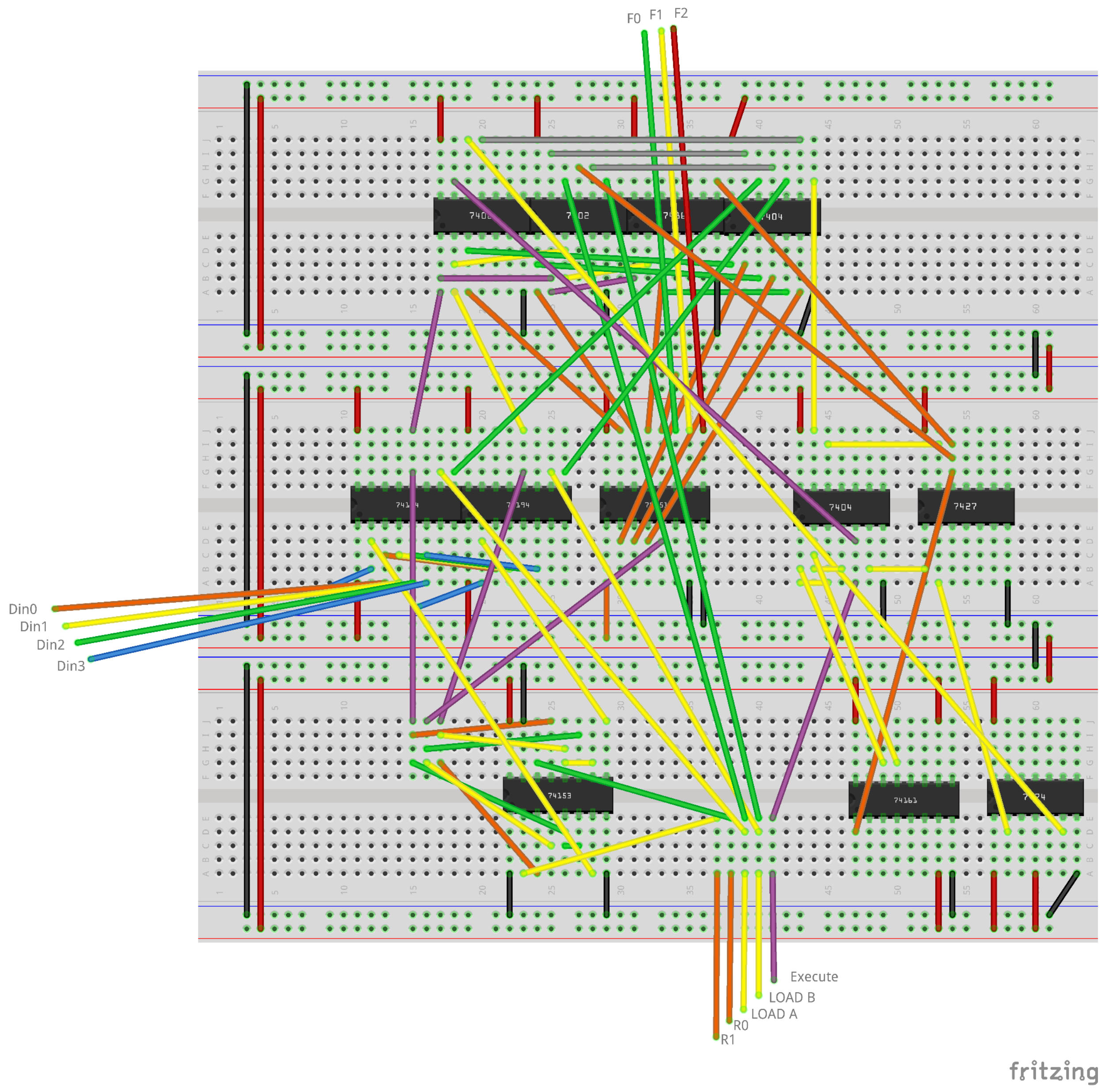


Routing Unit			
R1	R0	A'	B'
0	0	A	B
0	1	A	F
1	0	F	B
1	1	B	A

In our design, a single dual 4:1 MUX chip (SN74HC153N) is used to implement the routing unit, which determines the data flow path of the calculation results. Each 4:1 MUX determines the new value of one of the registers. With the R1-R0 switches, four ways of routing is addressed.
Below is a detailed schematic diagram of the routing unit.



5 Breadboard view / Layout sheet



6 8-bit logic processor on FPGA

6.1 Changes Made to Extend the Processor to 8-bits

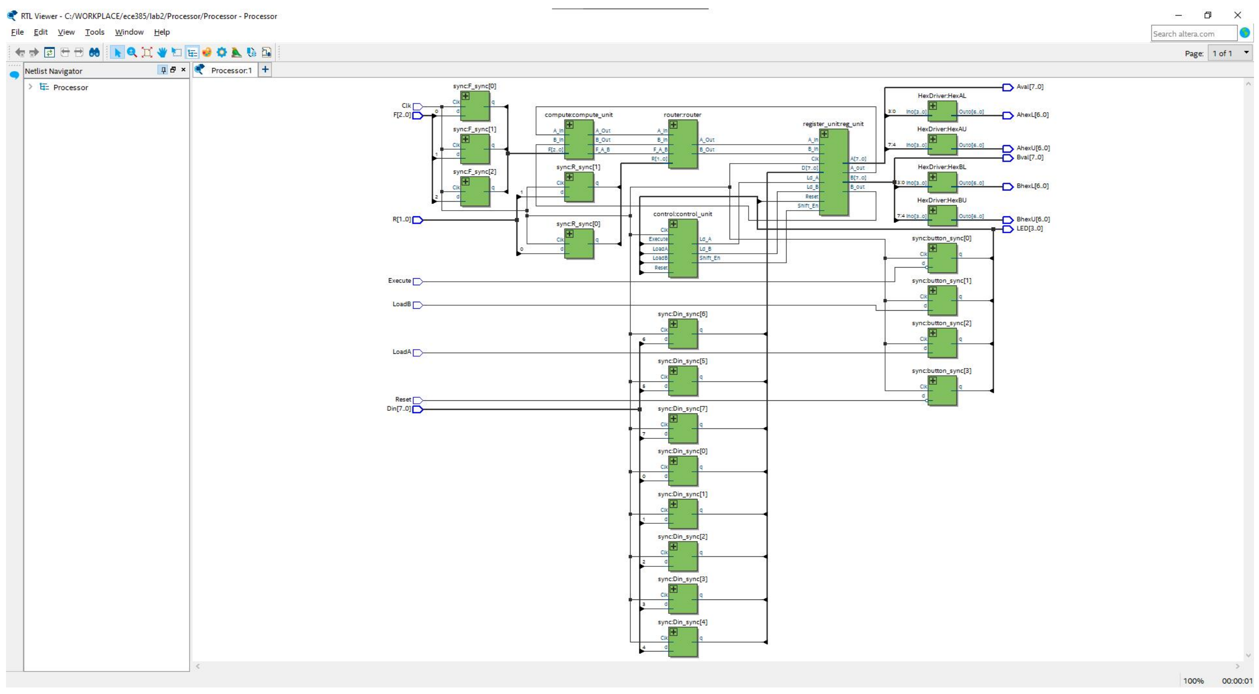
To extend the original 4-bit logic processor into a 8-bit version, these changes are made to the following aspects of the circuit:

1. The register units are extended from 4 bits to 8 bits.
2. The Din width is extended from 4 bits to 8 bits, and corresponding synchronizers are added.

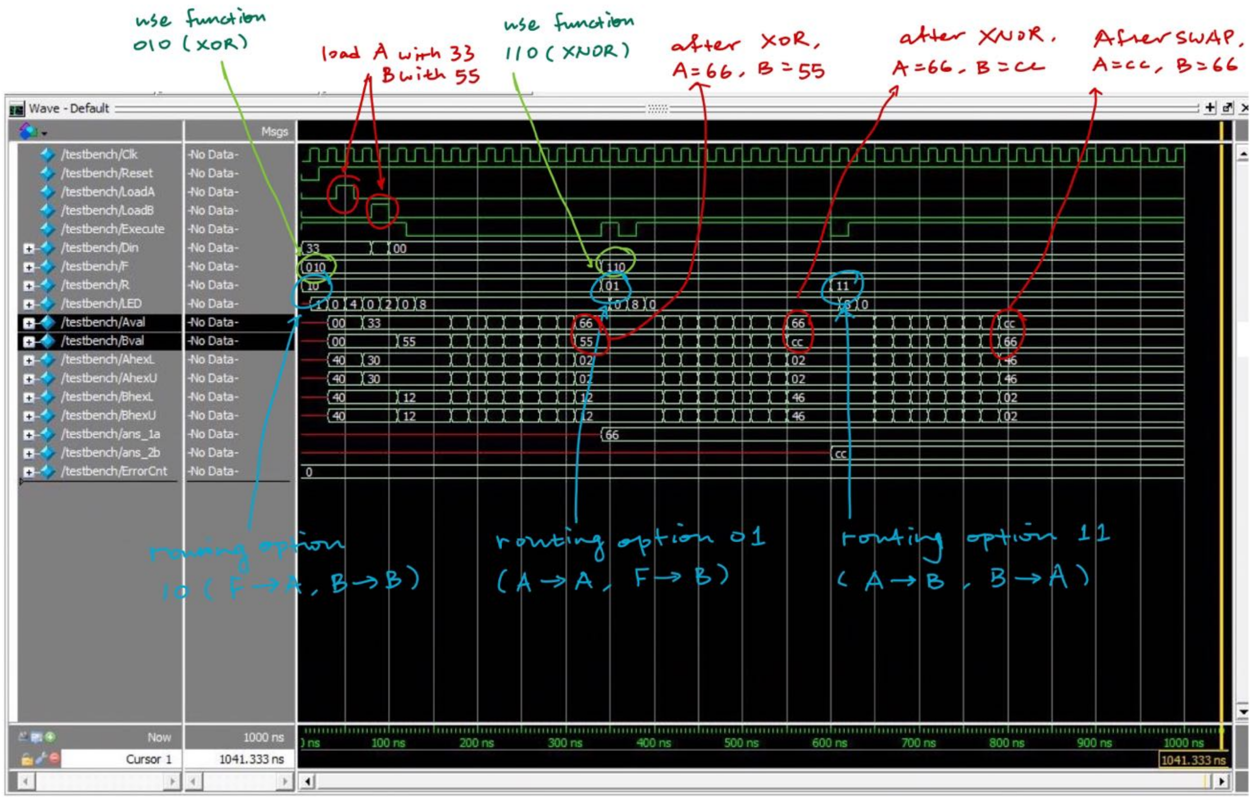
3. Two more HexDriver’s are added to display higher 4 bits of the extended register values.
4. Four more states are added to the Finite State Machine such that after exactly 8 cycles will the FSM enter the hold/shift state.

6.2 RTL Block Diagram

The following diagram shows the changes mentioned in 6.1 are indeed made and reflected on the RTL Block Diagram.



6.3 Annotated Simulation of the Processor

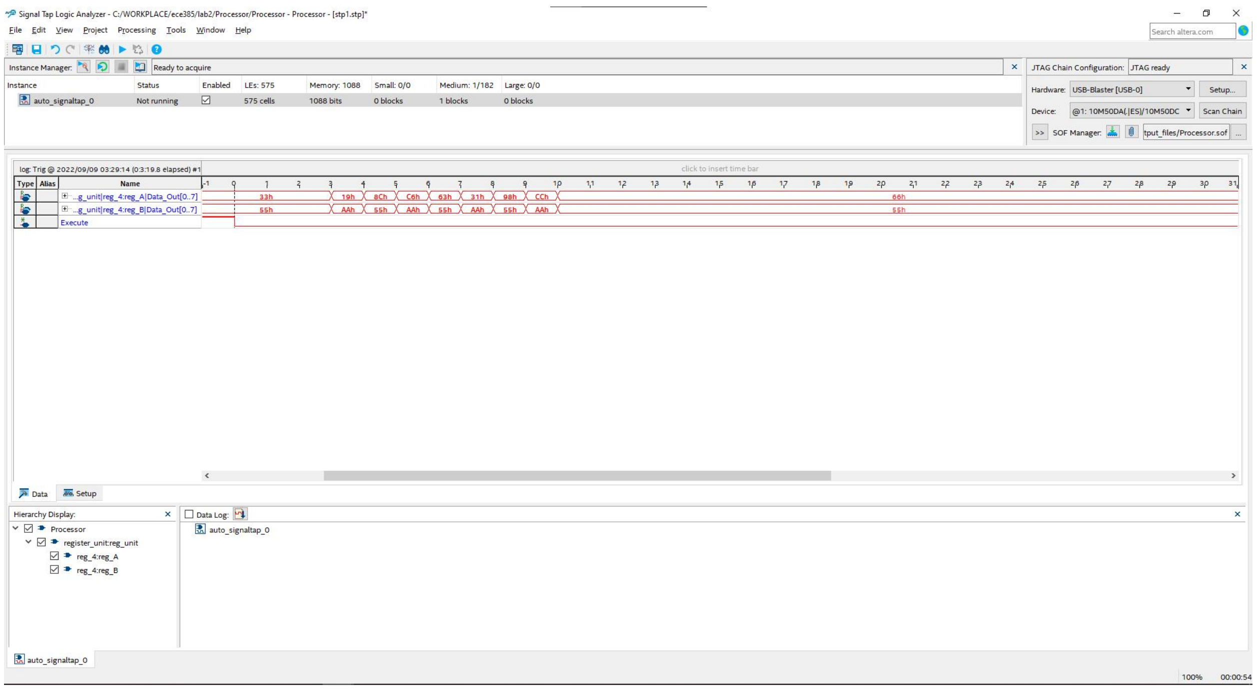


6.4 Procedure Used to Generate Signal Tap ILA Trace and the Results executing the 8'h33 XOR 8'h55 operation

The procedure of generating SignalTap trace:

1. Select, add, and group the output signals of the registers A and B, trigger on the signal Exectute with condition "either edge" so the behavior right before and after the execution is recorded.
2. Compile the project, flash the program into the FPGA chip, after the flashing is finished use "Run Analysis" and start the acquisition in the SignalTap window.
3. Manually load data 8'h33 and 8'h55 into A and B respectively, set up the function switches F2-F0 to be 010 (XOR), and toggle the execute switch, observe in the SignalTap window, the following result should come up.

The results can be found in the following image:


Signal Tap Logic Analyzer - C:/WORKPLACE/ece385/lab2/Processor/Processor - Processor - [stp1.stp]*

Instance Manager: Ready to acquire

Instance	Status	Enabled	LEs: 575	Memory: 1088	Small: 0/0	Medium: 1/182	Large: 0/0
auto_signaltap_0	Not running	✓	575 cells	1088 bits	0 blocks	1 blocks	0 blocks

JTAG Chain Configuration: JTAG ready
Hardware: USB-Blaster [USB-0]
Device: @1: 10M50DA(.|ES)/10M50DC
SOF Manager: tput_files/Processor.sof

log: Trig @ 2022/09/09 03:29:14 (0:3:19.8 elapsed) #1

Type	Alias	Name	Values
		...g_unit|reg_4:reg_A|Data_Out[0..7]	33h, 19h, 8Ch, C6h, 63h, 31h, 98h, CCh, 66h
		...g_unit|reg_4:reg_B|Data_Out[0..7]	55h, AAh, 55h, AAh, 55h, AAh, 55h, AAh, 55h
		Execute	

Hierarchy Display: Processor, register_unit:reg_unit, reg_4:reg_A, reg_4:reg_B
Data Log: auto_signaltap_0

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

The biggest problem we encountered during the construction was the selection of the register chip. At first, we chose the 4-bit Shift register with Parallel (CD74HC195E) because we thought it would be easier to load inputs in parallel, but when it came to shifting bits to the right, it became hard to distinguish between Shift and Load without disrupting the data in the register. Therefore, we changed the register to a 4-bit BIDIR Shift register (CD74HC195E), which we could use its S1 and S0 to control Shift and Load signals.

8 Conclusion

In this lab, we built a 4-bit serial logic processor on a breadboard and extended it to 8-bit with SystemVerilog on FPGA. The main four components of our circuit are the register unit, the computational unit, the routing unit, and the control unit. After finishing the construction on the breadboard, we extended the 4-bit logic processor to 8-bit logic processor on the FPGA board by writing SystemVerilog code and test it with ModelSim and SignalTap. We learn how to build a complicated circuit from the sketch and debug separate modules. It also offers us a great opportunity to be familiar with Quartus so that we can be prepared for the future labs.

9 Answers to Post-Lab Questions

9.1 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 or equal to the other input inverted). Explain why this is useful for the construction of this lab.

The simplest circuit that can optionally invert a signal is the XOR gate. Let's assume input A is the one to determine whether the output is equal to the other input B or not.

- If A = 1, A XOR B = B' (1 XOR 1 = 0; 1 XOR 0 = 1)
- If A = 0, A XOR B = B (0 XOR 1 = 1; 0 XOR 0 =0)

This is useful for the construction of this lab because for the computational unit, the results of NAND, NOR, XNOR, and SET are the inverted results of AND, OR, XOR, and CLR. We can easily implement another XOR chip to invert the output signal to achieve our goal, therefore, less gates and wires are needed.

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

A modular design such as that presented above improves testability and cuts down development time because each individual module has its own functionality and can be easily tested inside the module without connecting with other modules. In this way, we can test every module without messing up the whole circuit design, and if all modules work as we expected, our whole circuit would also function well.

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

When designing our state machine, we implement the knowledge we gained in ECE 120 and build a Finite State Machine with states reset, shift, and halt, and our input is count and execute signal. But then we realized that we could combine our outputs with inputs (the four-times shifting process) to reduce the number of states.

A Mealy machine depends on its current state and current input whereas a Moore machine only depends on its current state, which means a Moore state is a subset of a Mealy machine. A Moore machine might be more intuitive and straightforward, however, a Mealy Machine requires less states (only two states, reset state and shift/halt state), which means less wires and gates are needed, making debugging easier and faster. We only need one bit to represent these two states and therefore only one flip-flop needs to be implemented. Besides, a Mealy Machine responds faster as it depends on the input.

9.4 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 simulates the behavior of the circuit virtually inside the computer, not involving hardware at all, whereas SignalTap uses the hardware on the FPGA and it captures the waveforms of the FPGA. If we want to test the functionality of the System Verilog code, before we program it onto the FPGA board, we can write our own testbenches and run the ModelSim to check. When we want to check how data is actually processed on the FPGA board, we have to use the SignalTap to generate waveforms.