ECE385

Fall 2021

Experiment #2

A Logic Processor

Haozhe Si

haozhes3

Section AB1

1. Introduction

In this lab, we designed a logic processor that can perform six types of bit-wise logic calculations including AND, OR, NAND, NOR, XOR and XNOR. We also designed the control unit, register unit and router unit for the processor. In the first week, we implemented the processor using TTL on bread boards that can operate on 4 bits. In the second week, we extend the processor to work on 8 bits by implementing it on the FPGA board.

2. Operation of the logic processor

a) Load Registers

We want to load data in parallel using the function provided by the SN74LS194 shift register chips, while sharing the same data input for the two registers. We achieve that by toggling the D3-D0 switches to the data we want to load, and then toggle the LoadA/B switch to select the target register we want to load the data into. Two sets of LEDs will indicate the data loaded into the register. Once the data for the two registers are load, set the both LoadA and LoadB into off, the data are saved in the register and can be accessed by the processor in future operations.

b) Setting Route and Initiate Computation

To select the calculation we want to perform, we can initiate them by entering the corresponding code for the function through the F2-F0 switches. We also need to specify the destination register for the computation results through the R1-R0 switches before we start the computation. After every thing is prepared, we need to toggle the Execute switch to start the computation. Once the computation is launched, the LEDs will display the value saved in the registers at each clock cycle, and during the computation, toggling Execute switch will no longer have any effect. The final result will be saved into the registers selected by the routing unit and being display by corresponding LEDs. After the computation is completed, the Execute switch will be functioning again for start another calculation, and one can restart the above procedures as needed.

3. Block Diagram and State Machine Diagram

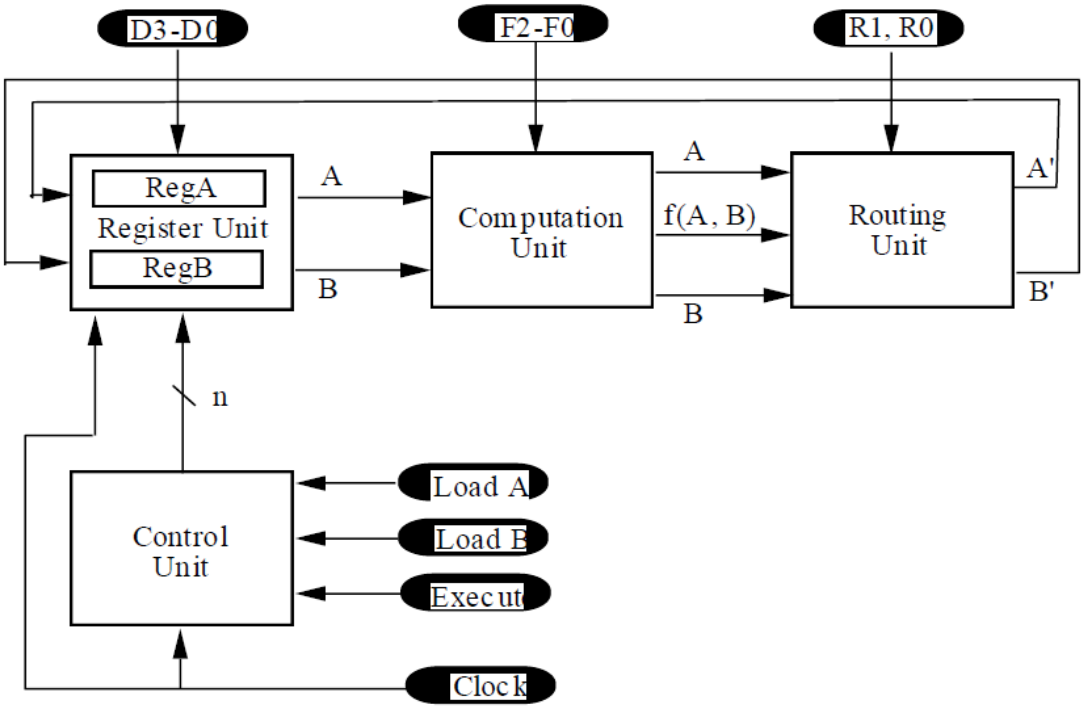
Register Unit: The register unit consists of two 4-bit shift registers, which stores the data we need in the calculation. The unit can parallelly load data from D3-D0 switches, or take serial input from the routing unit. The register unit will take shift signal and load signal from the control unit to indicate the way of loading data. It also has a clock input and serial output to the computation unit.

Control Unit: The control unit takes in charge of generating shift signal and load signal to the register unit. It takes in the LoadA, LoadB, Execute and Clock

externally from switches and clock generators. The control unit implements a state machine that can decide when to output the shift signal according to the Execute switch and the current state given by the counter chip (SN74LS169). It also needs a flip-flop to store the internal next-state information.

Computation Unit: The computation unit is consisting of a NAND gate, a NOR gate and a XOR gate with three NOT gates to negate their outputs. The six outputs of the logical gate give the six calculation results, and such results, along with a logical high and a logical low signal, are fed into an 8-to-1 MUX (SN74LS151). The input to the MUX will be the Function Choose switched F2-F0. The computation unit will also take data input from the register unit.

Routing Unit: The Routing Unit takes in computed output and the two serial inputs and outputs the two selected data according to the Routing Choose R1-R0. This is implemented using two quad 2-to-1 MUX (SN74LS157). The inputs are taken from the computation unit and the register unit while the output goes into the register unit.

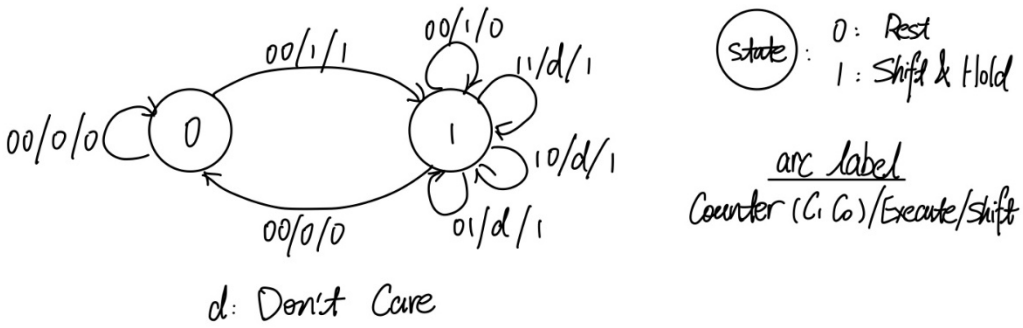


The Block Diagram

State Machine: In this design, we used a Mealy State Machine instead of the Moore State Machine. As we known, the output of the Mealy State Machine depends on the both the current state and input, while it requires less states; the output of the Moore State Machine depends solely on the current state thus requires more state. Noticing the state number and transition condition tradeoff between the Mealy State Machin and the Moore State Machine, we choose the Mealy State Machine because we can save one state by combining the Shift and

Hold state, which can make our implementation simpler and more reliable. **(POST-LAB Answer)**

Our resulting state machine has two states: Rest and Shift & Hold, where in the Rest state, the output of the state machine, Shift, is 0, and the counter will be set to 00 and never updates; meanwhile, in the Shift & Hold state, the counter will update for four clock cycles and frozen at 00, at this time, the Shift signal will be 0 again. The state transit depends on counter signal (C1C0), Execute and Shift signals, where counter and Execute signals are the inputs and Shift is the output.



The State Machine Diagram

4. Design Details

Control Unit: In order to design the logic for the control unit, we need to draw the K-maps from the state transition table.

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

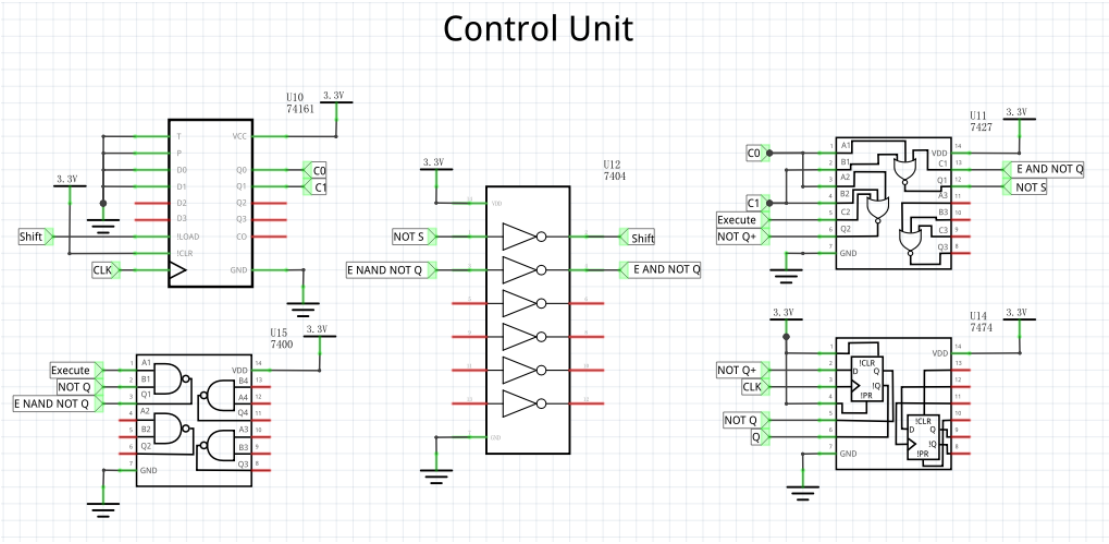
Since $C1^+$ and $C0^+$ are counter's outputs, we can use the counter chip to implement the count can hold logics. To be specific, since holds in our implementation means constantly outputting 0, we can make use of the load function on the counter chip, therefore, whenever we need to clear the counter or hold it at 0, we constantly load 0 to the counter. Thus, we initiate a new signal called Enable to enable counter to count and Enable' means loading 0. We need to draw three K-maps from the transition table: Shift, Q^+ and Enable.

S		C1C0			
		00	01	11	10
EQ	00	0	d	d	d
	01	0	1	1	1
	11	0	1	1	1
	10	1	d	d	d

Q^+		C1C0			
		00	01	11	10
EQ	00	0	d	d	d
	01	0	1	1	1
	11	1	1	1	1
	10	1	d	d	d

En		C1C0			
		00	01	11	10
EQ	00	0	1	1	1
	01	0	1	1	1
	11	0	1	1	1
	10	1	1	1	1

From the K-maps, we can see that the logical expression for S is S = C0+C1+EQ', the expression for Q^+ is Q^+ = C1+C0+E, and for Enable is En = C0+C1+EQ'. Surprisingly we find that the expression for Shift is the same as the one for Enable. Thus, we can simply use the Shift signal to control the counter.



The Control Unit Schematic

Register Unit: According to the datasheet for the SN74LS194 4-bit shift register, the load mode is specified by the signal S1S0. In our processor, we want to parallel load data when setting the Load signal true and load data in serial, which is inequivalent to right shift the data, when the Shift signal is true. Therefore, we will have four K-maps for S1S0 for the two registers. Since the two registers are parallel, they will have the same logic, thus, only two K-maps are needed.

FUNCTION TABLE

INPUTS									OUTPUTS			
CLEAR	MODE		CLOCK	SERIAL		PARALLEL						
	S1	S0		LEFT	RIGHT	A	B	C	D	Q_A	Q_B	Q_C	Q_D
L	X	X	X	X	X	X	X	X	X	L	L	L	L
H	X	X	L	X	X	X	X	X	X	Q_{A0}	Q_{B0}	Q_{C0}	Q_{D0}
H	H	H	↑	X	X	a	b	c	d	a	b	c	d
H	L	H	↑	X	H	X	X	X	X	H	Q_{An}	Q_{Bn}	Q_{Cn}
H	L	H	↑	X	L	X	X	X	X	L	Q_{An}	Q_{Bn}	Q_{Cn}
H	H	L	↑	H	X	X	X	X	X	Q_{Bn}	Q_{Cn}	Q_{Dn}	H
H	H	L	↑	L	X	X	X	X	X	Q_{Bn}	Q_{Cn}	Q_{Dn}	L
H	L	L	X	X	X	X	X	X	X	Q_{A0}	Q_{B0}	Q_{C0}	Q_{D0}

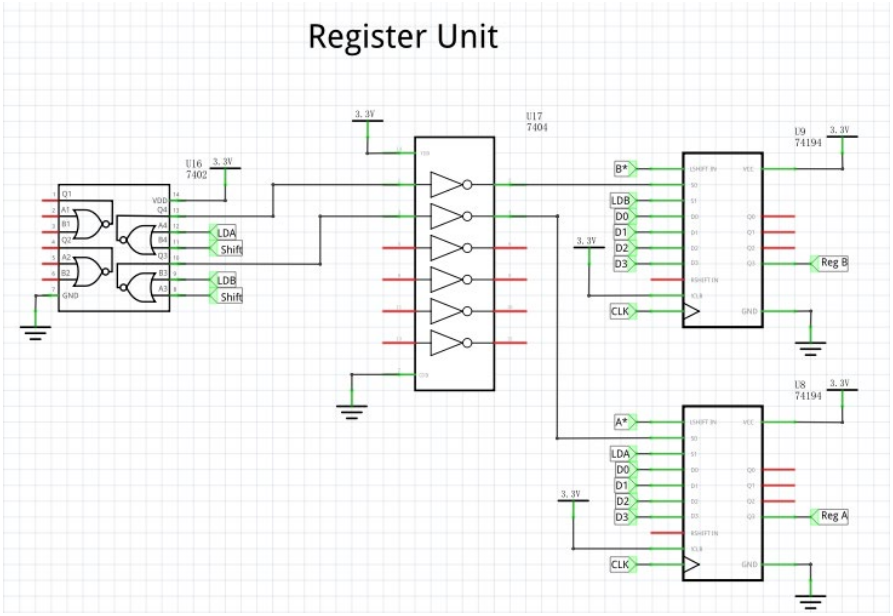
H = high level (steady state)
L = low level (steady state)
X = irrelevant (any input, including transitions)
↑ = transition from low to high level
a, b, c, d = the level of steady-state input at inputs A, B, C, or D, respectively.
Q_{A0}, Q_{B0}, Q_{C0}, Q_{D0} = the level of Q_A, Q_B, Q_C, or Q_D, respectively, before the indicated steady-state input conditions were established.
Q_{An}, Q_{Bn}, Q_{Cn}, Q_{Dn} = the level of Q_A, Q_B, Q_C, respectively, before the most-recent ↑ transition of the clock.

SN74LS194 4-bit Shift Register Data Sheet

S1		Load	
		0	1
Shift	0	0	1
	1	0	1

S0		Load	
		0	1
Shift	0	0	1
	1	1	1

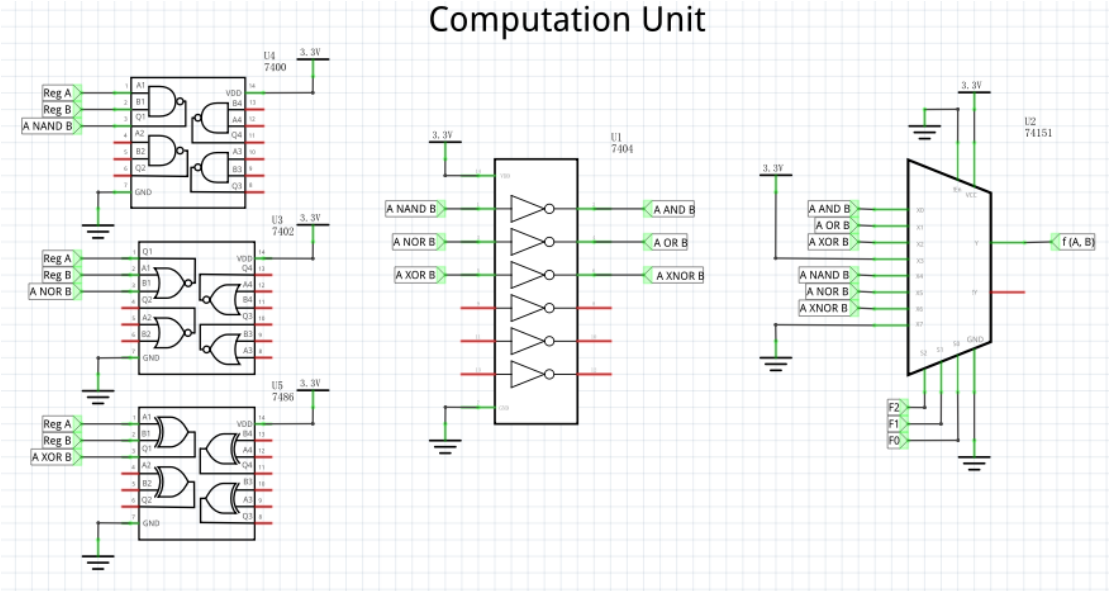
Therefore, the logic expression for S1 is S1=Load and the one for S0 is S0 = Load + Shift. The following is the schematic for the register unit.



The Register Unit Schematic

Computation Unit: The computation unit is simply consisting of NAND, NOR and XOR gates and three NOT gates that can negate the output. We made use of a 8-to-1 MUX to combine the signals. The Function Choose switches F2-F0 will feed directly into the MUX to select the output.

One may also consider to use two bits of the function choose to select the function and use the last bit to select whether to output the original result or the inverted result. This can be easily implemented by using a 4-to-1 MUX for function choose and a 2-to-1 MUX for inverter choose. For the 2-to-1 MUX, the two input will be the output of the 4-to-1 MUX and the inverted output resulting from the former signal passing through an inverter. This is the simplest circuit that can optionally invert a signal. However, we did not choose such implementation because although it saves two inverters, we are actually wasting the other five inverters on the hex inverter chip. Also, we need an extra MUX than our current implementation needs. Therefore, for the sake of saving board space, we think our design is better. **(POST-LAB Answer)**

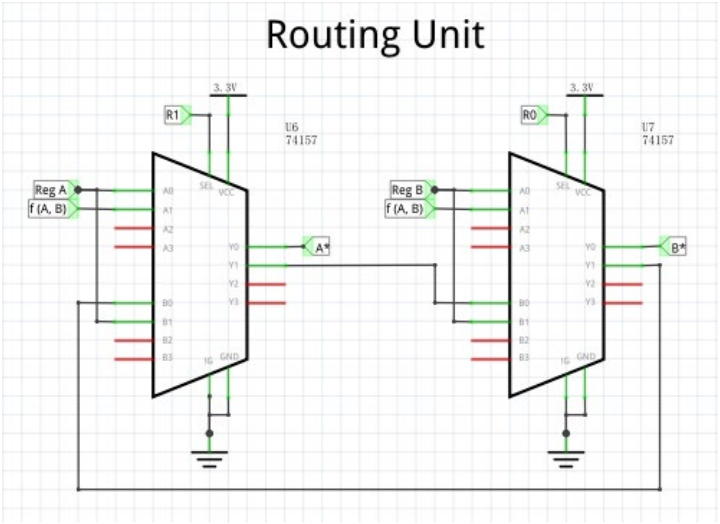


The Computation Unit Schematic

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

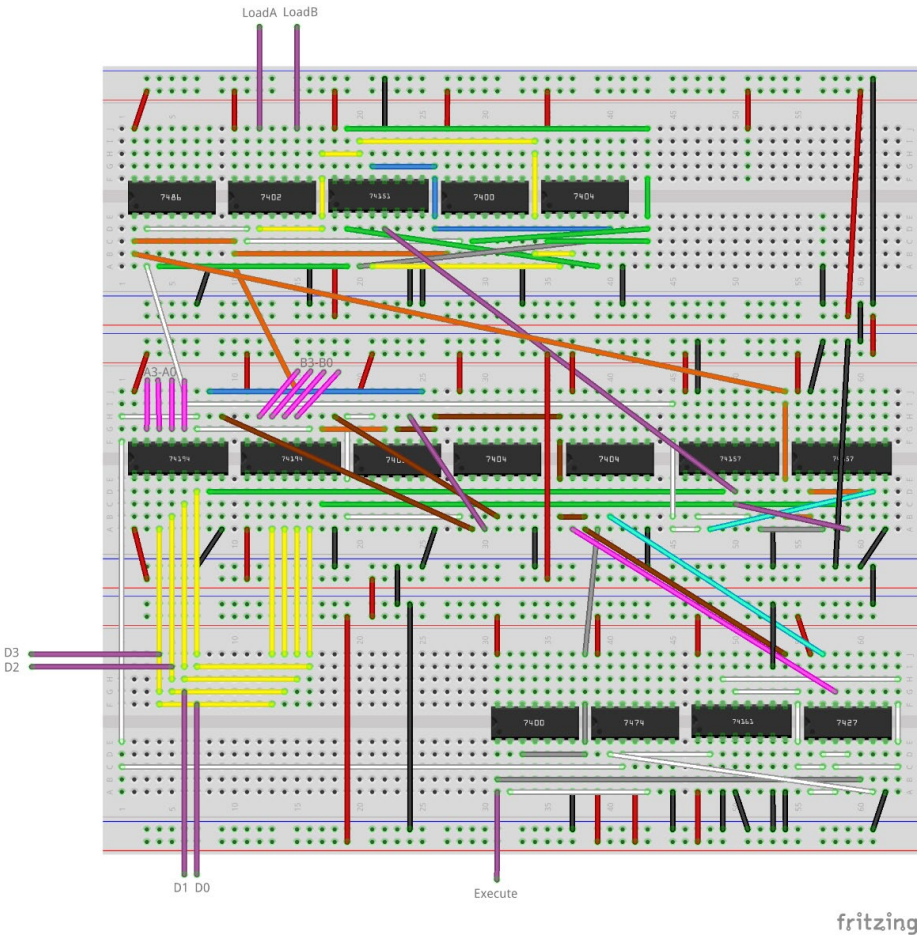
The Function Table for Routing Unit

Routing Unit: From the function table for the Routing Choose switches, we can see that the routing unit will route data from register A back to register A whenever R1 is logical high, and when R1 is logical low, the value goes into register A depends on R0: the resulting data from computation unit will go to register A if R0 is logical low, the data from register B will go to register B otherwise. Similar logic holds for data routed to register B. Since this is a two-step choice making depend on the value of R1 and R0, we use two quad 2-to-1 MUX to implement the logic.



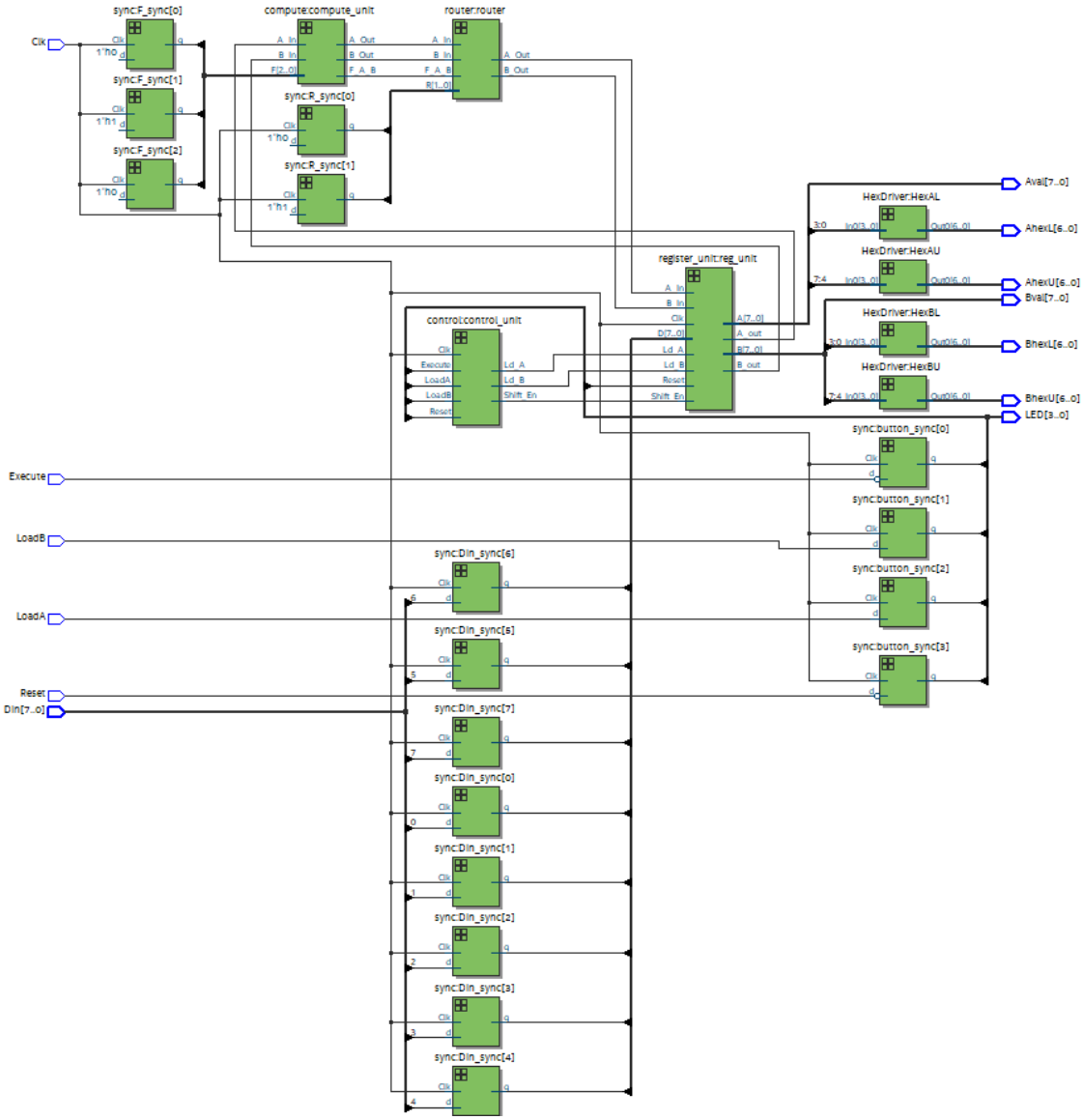
The Routing Unit Schematic

5. Breadboard View



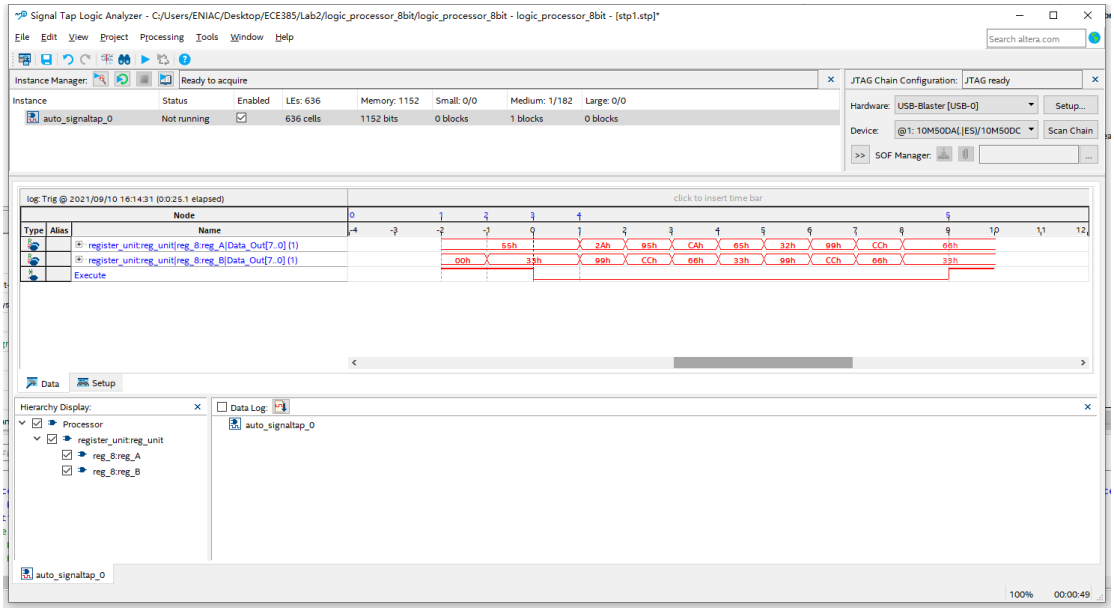
6. 8-bit Logic Processor on FPGA

In order to extend the 4-bit logic processor into 8 bits, we need to modify the modules for registers and control units. For the register unit, we need to extend the size of each variable from 4 bits to 8 bits. For the control unit, we need to add four more states so that the logic processor can finish the 8-bit calculation before it enters the hold state. We also need to modify the top-level file Processor correspondingly to be compatible with the modified register unit and control unit. We do not to modify the routing unit and the computation unit since they are performing bit-wise operations originally and do not care about how many bits we have.



The RTL Block Diagram

Simulation of Processor



SignalTap ILA Trace of Executing 8'h33 XOR 8'h55

SignalTap Trace Generating:

1. Select the nodes we want to keep track of. Here we want to keep track of the output signal for the two registers and the Execute signal. For the sake of interpretability, we can group the eight outputs for each register. Also select Clk signal for the clock signal of SignalTap.
2. Select the Trigger condition for Execute signal to "Either Edge".
3. Recompile the project and reprogram the FPGA Board.
4. Press "Run Analysis" and we can the SignalTap start to acquire the data.
5. Load data into the registers and press Execute. After the calculation ends, we can stop the data acquisition and the waveform of the signal we tracked will be displayed.

ModelSim v.s. SignalTap: Both ModelSim and SignalTap can generate the waveform. However, the waveform of ModelSim is generated by the software simulating the function implemented by the System Verilog code according to the testbench while the waveform generated by SignalTap is generated by the FPGA board being programmed by the code we wrote. Therefore, when we want to test the functionality of the code, checking if everything is implemented correctly before we actually program it to the board, ModelSim works well to take the role of sanity check. When we want to see how the data are transferred and processed on the FPGA board, we can use the SignalTap to acquire the data and visualize them as the waveform. **(POST-LAB Answer)**

7. Bug Log

The major issue I encountered in doing this experiment is generating the SignalTap ILA Trace. This is because I did not set the trigger condition correctly to "Either Edge", but leaving it at the "Don't Care", which leads to the SignalTap cannot successfully acquiring the data from the board. When I realized the issue and corrected the setting, the trace successfully generated.

Thanks to the modular design, I did not encounter any bug during assembling the TTL circuits. During the modular design, I can test every module once after I finish build up them. Since the task for each module is relatively simple comparing to the whole logical processor, we can easily tell if it is functioning correctly. Even if I encountered any bug, I would be sure that the bug is inside the module, which would limit the range for me to check the circuit and saves time for debugging. Since each module becomes an individual, we can leave them unconnected until each module is tested function properly. In this way we can protect the finished module from being destroyed by malfunctioning like short. **(POST-LAB Answer)**

8. Conclusion

In this lab, we built the logical processor from the sketch. We decomposed the logical processor into four blocks and analyzed their functions. From the functions, we come up with the logical implementations to realizing them. Given the designs, we built and tested a 4-bit logical processor using the TTL circuits. We also use System Verilog to extended a 4-bit logical processor to an 8-bit one and implemented it on the FPGA board. This lab practice our ability of designing a hardware project from the logical level and build up every component step by step.