ECE385

Lab2

Jiaming Xu

Rui Gong

Introduction

This lab is to build a circuit that is a four-bit processor that can calcul ate eight different functions based on the input and update the final value in 4 different ways. The circuit is composed of four different parts, contr ol unit, register unit, computation unit, and routing unit. Control unit is to decide when to load the value of register A and B, when to execute the o peration, and how many times the operation will take. Register unit is get ting the input values for A and B and shift one bit to the computation unit and do the operation. Computation unit decides which function should be use d in the operation including AND, OR, XOR, all ones, NAND, NOR, XNOR, and a ll zeros, and the routing unit decides where the new values should be store d at, either A or B.

Operation of the logic processor

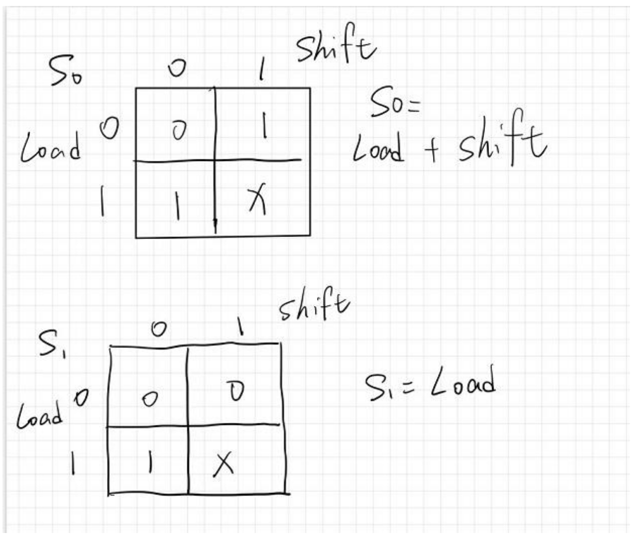
a. For the load data step, we first need to flip the load A and load B swit ches on in order to allow input values to be stored in A and B registers. T hen, we can flip the switches that connect with register A and B to load th e value we want as input. After loading the values to register A and B, we need to turn the switches of load A and load B off to avoid updating values of A and B during operation.

b. The user first needs to flip the computation switches to decide one operation, and then use the routing unit switches to decide where the output of computation should be stored at and update the value in register A and register B.

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

Register unit:

The register unit takes in four inputs, which are used to set the value of register A and register B. When the load A and load B are on, the switches related to register A can load the value of A and switches of register B can load the value of B. Then we use two 4-bit BIDIR Shift Regs to right shift both register A and register B one bit to the computation unit. For the 4-bit BIDIR Shift Reg, we set MR to always be high, and DSR takes in the output of the routing unit. D0 to D3 are the values we input by using switches. Q0 to Q3 is the output of the shift register. We connected D3 directly with the computation unit as input. CP we connected it with CLK input. For S1 and S0, we used boolean operations to get them. S1 = Load A/B, and S0 = Load A/B + Shift which can be directly derived from the control unit. For the OR operation in S0, we used one NOR gate and an inverter.



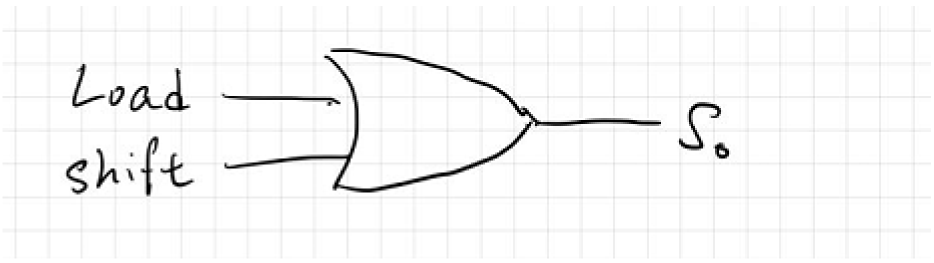
S_0	Shift 0	Shift 1
Load 0	0	1
Load 1	1	X

$S_0 =$ Load + shift

S_1	shift 0	shift 1
Load 0	0	0
Load 1	1	X

$S_1 =$ Load

Truth Table for Register Unit.



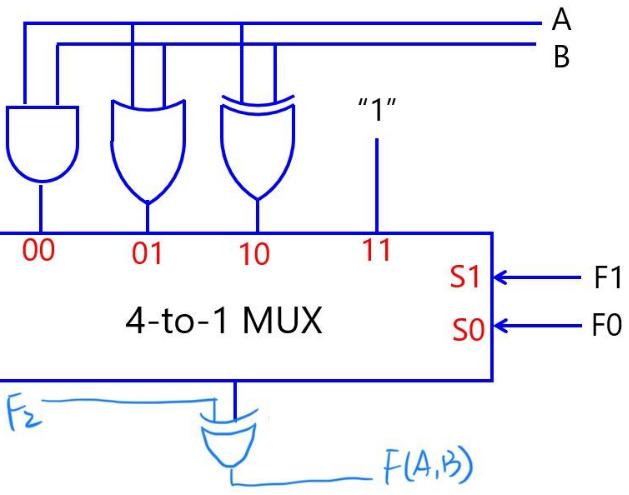
Block Diagram for Register Unit.

Computation unit:

The Computation Unit has five inputs: RegisterA, RegisterB, F2, F1, F0 and three outputs: RegisterA, ResgisterB, f(A, B). Inputs RegisterA and RegisterB are containing the value of A and B, coming from the Register Unit. Inputs F2, F1, F0 serve as a select of what functions we plan to implement.

To implement the Computation Unit, we utilize a 4-1 MUX, where F1 and F0 serve as the two select inputs, so that we could implement 4 functions: AB, A+B, AXORB, ‘1’. Then we connect the output of the MUX to a XOR gate with F2, so that we could implement the other 4 functions.

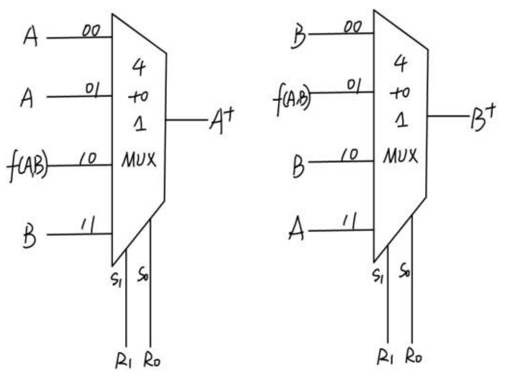
Outputs RegisterA and RegisterB will contain the original values of A and B, same as the input, while the output f(A, B) will contain the value of the result after we implement the function on A and B. These three outputs will feed into the Routing Unit.



Block Diagram for Computation Unit.

Routing unit:

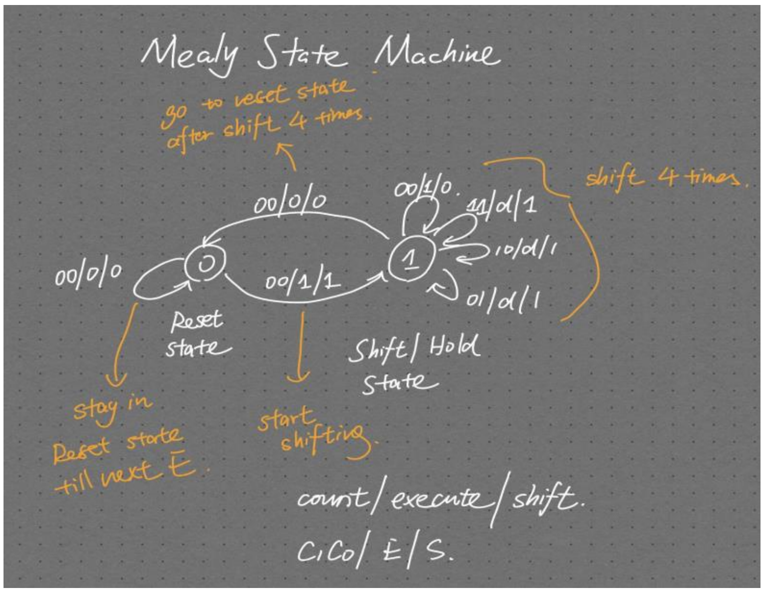
The Routing Unit has five inputs: A, B, f(A,B), R1, R0 and two outputs: A*, B*. The purpose of the Routing Unit is to select two from A, B, f(A,B) and feed into the A, B at the Register Unit, serving as the new value for RegisterA and RegisterB. To implement the Routing Unit, we utilize two 4-1 MUXs. The inputs R1, R0 will serve as two selects of the two 4-1 MUXs, while A, B, f(A,B) will serve as the inputs of the two MUXs.



Block Diagram for Routing Unit.

Control unit:

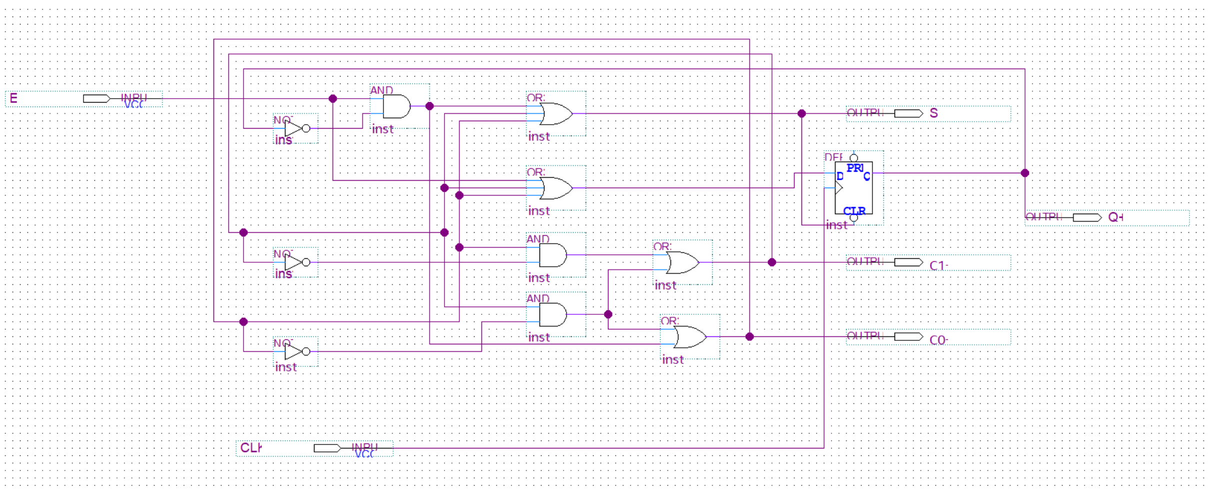
The Control Unit has four inputs: Load A, Load B, Execute, and the Clock signal. The Execute will determine when the registeres are ready to begin the computation. The Control Unit will output S, Shift, which control whether to shift the registers for exact four times and stops until another Execute. To implement the Control Unit, we construct a Mealy State Machine and a corresponding truth table based on the State Machine Diagram below.



Mealy State Machine for the Control Unit.

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

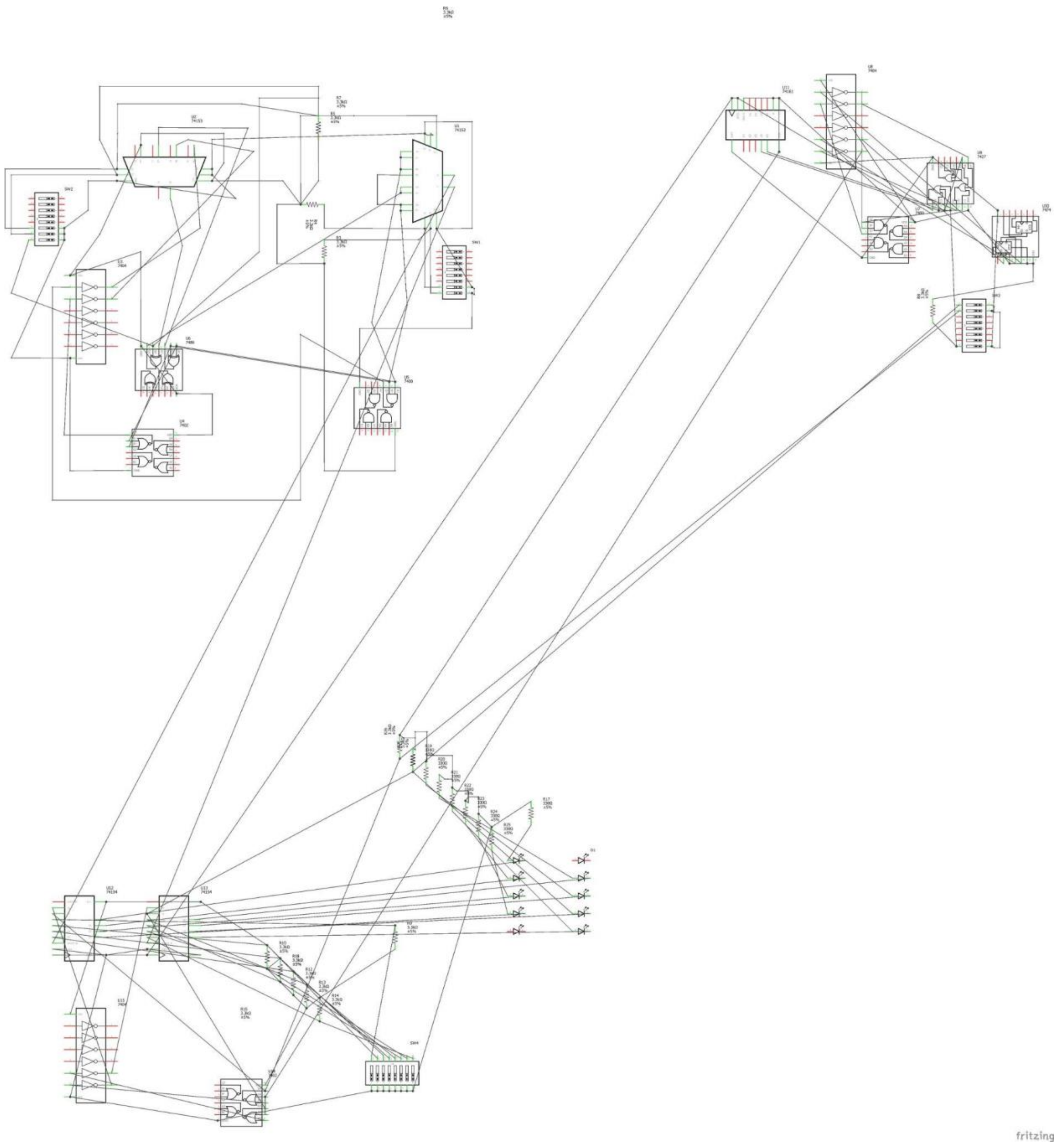
Truth Table for the Control Unit.



Block Diagram for Control Unit.

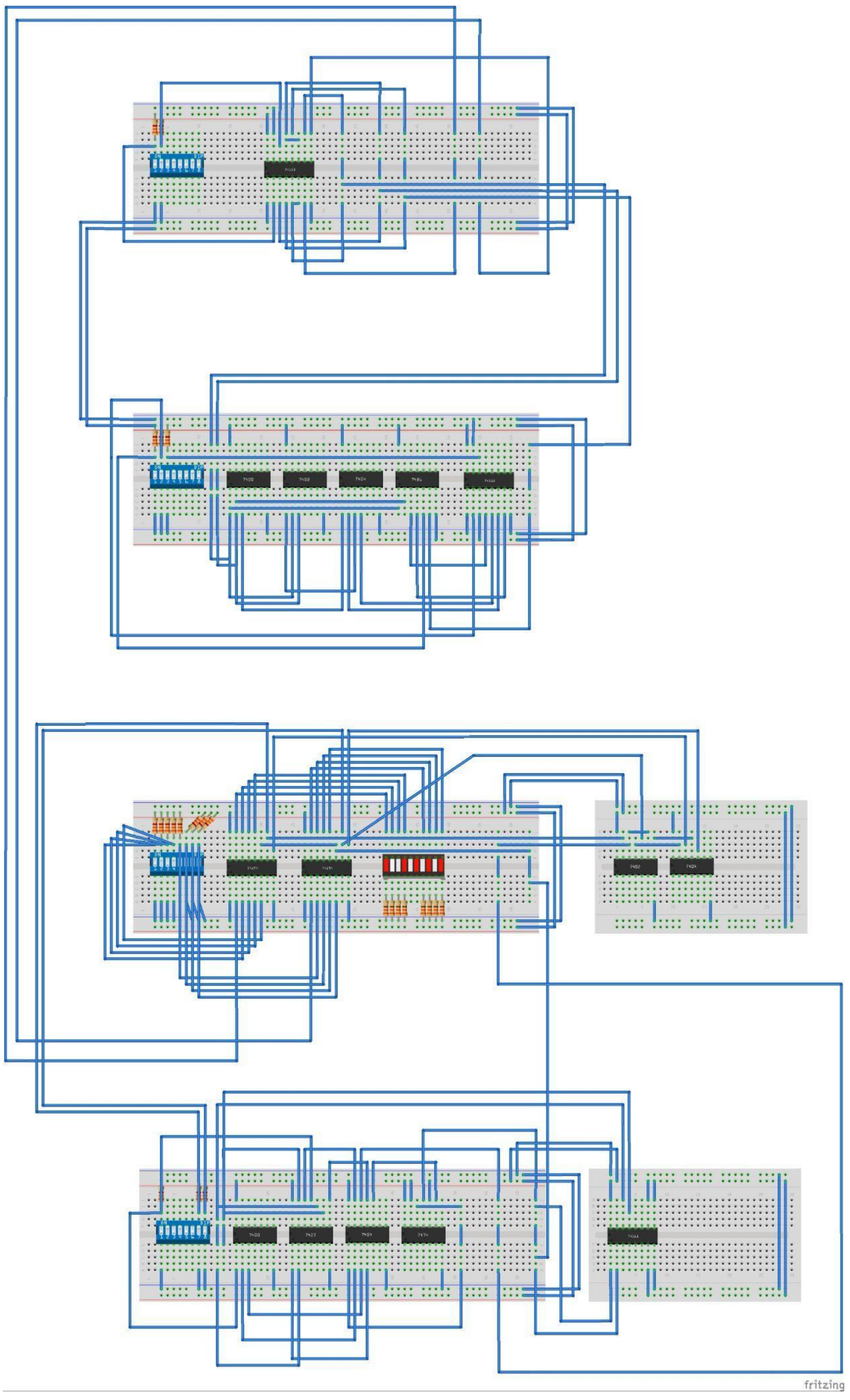
With the State Machine and the truth table, we come to the logic expression for S and Q+. Also, to hold the value of Q, we need to connect the output Q to a Flip-Flop, and the output of the flip-flop should be Q+.

Detail Circuit Schematic



The top left block is the Computation Unit and the Rounting Unit. The top right block is the Control Unit. The b ottom block is Register Unit.

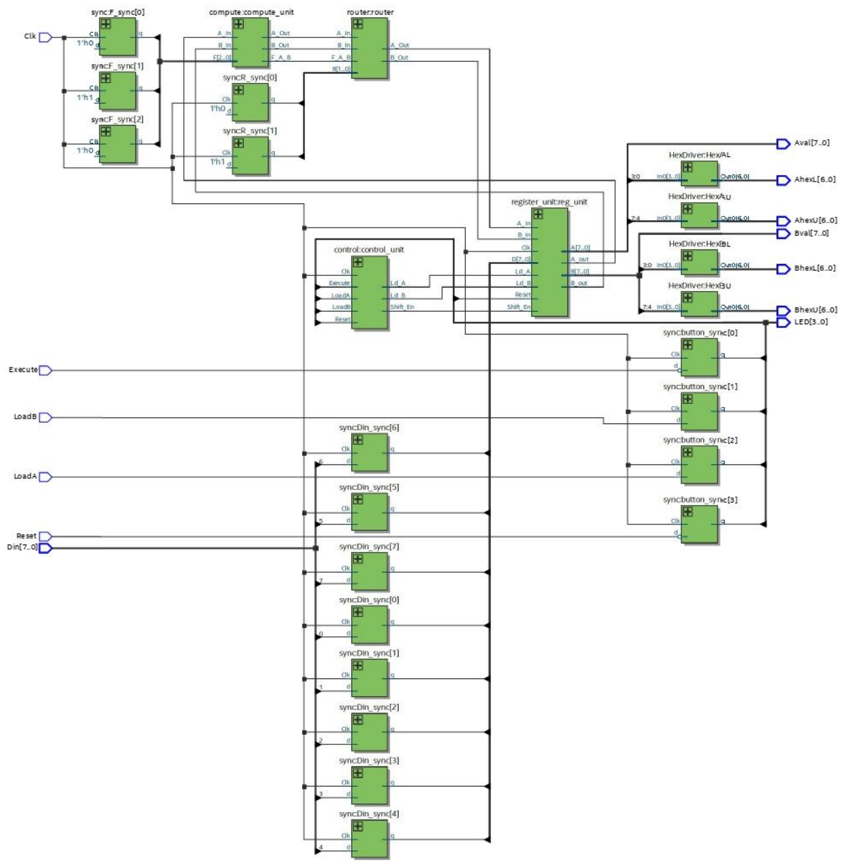
Breadboard View



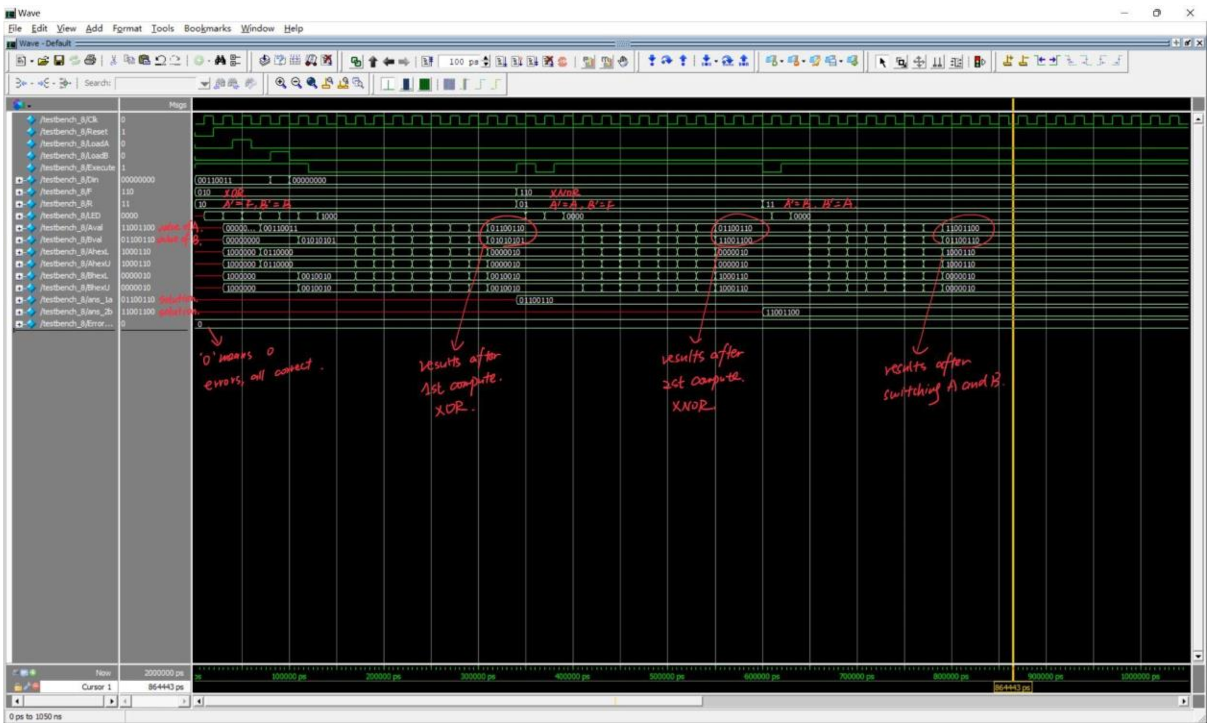
From top to bottom: Routing Unit, Computation Unit, Register Unit, Control Unit.

8-bit Logic Processor on FPGA

To convert a 4-bit Logic Processor to 8-bit, first of all, we changed the size of all the 4-bit register to 8-bit register in all the files. In the Control.sv the previous 4-bit processor design only uses 6 states, which is not enough for the 8-bit processor. To accomodate the 8-bit design, we add 4 more states to the final state machines, and revise the corresponding coding. In the Processor.sv, we also add two more Hexdriver so that we could view the upper part of the 8-bit registers. The previous design only has two Hexdrivers, which can only accomodate two 4-bit registers, while by adding two more Hexdrivers, we are able to accomodate two 8-bit registers.



8-bit Processor RTL Block Diagram.



8-bit Processor RTL Simulation Diagram

SignalTap of 8’ h33 XOR 8’ h55 Operation:

1. Since the FPGA only has 10 input switches, whic could not accommodate all the inputs, we need to first hardcode the value for F and R. To operate XOR, we need to set F to 3’ b101, and we save the value of F as in the new A by setting B to 2’b10.
2. After saving files, compiling, finishing the pin planning, plugging in the FPGA, we open the Signal Tap Logic Analyzer, and add device.
3. Then, we add clk to Clock, and add Reg_unitA[0] and Reg_unitB[0] all the all to Reg_unitA[7] and Reg_unitB[7] and add Execute, so that we could monitor the value of A and B.
4. Then, we hit the Run Analysis, and we are ready to load on FPGA.

5. We first input 8’ h33 into A by inputing ‘00110011’ on the switches and hit ‘LoadA’ .
6. Then, we input 8’ h55 into B by inputing ‘01010101’ on the switches and hit ‘LoadB’ .
7. Then, we hit ‘Execute’ , the result should pop out on the Singal Tap Analysis, and the value of f(A, B) as well as AXORB will be stored in to A.

Description of all bugs encountered, and corrective measures taken

In lab 2.1 we met many bugs and it took us much time to solve them. The fir st problem we met is we have trouble connecting the shift register. For S0 and S1 we thought they are simply connected with shift, the output of the c ontrol unit. After testing, we found they should be connected differently. By re-looking the truth table of the datasheet and having some hints from T as, we realize we should use load value and shift value to get S0 and S1. A nother bug we had is when we load the values of A and B, after turning off the load switches, but didn’ t turn on the execute switches, the register A and B values update automatically, and eventually disappear. The measure we took is adding a flip flop which can store the value and hold it until we t urn on the execute switch.

Conclusion

This lab is kind of difficult. In lab 2.1, we spent a lot of time figuring out how this circuit works, how to build the circuit, and debugging. For lab 2.1, we think the most challenging part is to build the control unit. We need to use switches to decide when to load value and when to execute. In lab 2.2, we extend the 4-bit processor in lab 2.1 to 8-bit processor by using quartus. It is very similar to the 4-bit processor except it extends the input and output value to 8 bits.

Post-lab Question

1. Document changes to your design and correct your Pre-Lab write-up, explaining any difficulties you had in debugging your circuit. Outline how the modular approach proposed in the pre-lab helps you isolate design and wiring faults, be specific and give examples from your actual lab experience.

For debugging, it is very difficult to find which parts we got wrong, especially with so many wires. When a test fails, we need to check all the related parts to make sure we connected it correctly. The methods proposed in the prelab about studying each of the chips is very helpful for us to isolate design and wiring faults. In the experiment, we used counter chips. There is a function table in the 4-bit Sync Counter datasheet. We first don’t know how to deal with each pin. After studying the chips function table and getting familiar with how each pin works, we understand how to connect the input and output with the chips. For example, MR, the first pin of a counter chip. When we read the function table, we found that MR is high when counti ng and low when resetting, so we decided to connect it with shift, derived from the previous step. When shift is high, this means the register shifts the least significant bit to the computation unit, and the counter starts c ounting. When the shift is low this means the register stops shifting, and the counter should reset. By using this method, we set up our counter corre ctly.

2. Describe the simplest (two-input one-output) circuit that can optionall y invert a signal (i.e., one input determines if the output is equal to th e other input or equal to the other input inverted). Explain why this is u seful for the construction of this lab.

This circuit can be built by XOR gate. By using XOR gate, if the select inp ut is 0 then the other input stays the same. If the select input is 1, then the other input should be inverted. This circuit can help us to determine w hether to invert the input or not manually. Also, it only uses one gate, wh ich can make the circuit simple and make debug easier.

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

A modular design can help us divide the circuit into many different parts. When we test the circuit, and there are bugs, we can easily find the parts that are related to the problem. Also, modular design makes each unit that serves the same function together, helping us to build the circuit in a clear way.

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

We used the Mealy machine in this experiment. In the control unit we need the combination of the current state and current input to decide the next state. Mealy machines compute the output based on both current input and current state. Moore machines only depend on the current input.

5. 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 perform the simulation of the design rather than the actual value of the device. When we use software testing, ModelSim is preferred. When we use hardware testing or for demo, SignalTap is more appropriate.