University of Illinois at Urbana-Champaign

ECE385
Digital Systems Laboratory

Lab2: A Logic Processor

TA: Abigail Wezelis, Harris Mohamed

Yan Miao, Guangxun Zhai
yanmiao2, gzhai5

February 2021

Contents

1 Introduction

In this lab, we built a 4-it logic processor to perform loading and computing operations. The operations we accomplished included AND, NOR, XOR, all high and their corresponding inverse. To achieve such functions, our circuit design utilized two 4-bit shift registers, a counter, several multiplexers, a finite state machine with a flip-flop, and some logic gates.

2 Operation of the Logic Processor

In this section, we are going to describe both the sequence of switches the user must flip to load data into the A and B registers and the sequence of switches the user must flip to initiate a computation and routing operation.

2.1 Loading Operation

First, we set the 4 bits we want to load into the register by flipping the switch on our DE10 that represents those 4 bits($D_3D_2D_1D_0$). Then we pressed the Load button on DE10 for the corresponding register. Within one clock circle, we were able to see the 4 bits shown on LEDs that connected the 4 outputs of that register.

2.2 Computing and Routing Operation

To initiate a computation and routing operation, we first set the computation select bits and routing select bits according to our needs. Next, we set execute to high. After one cycle clock, we saw the LED which shows the outputs of register shift 4 times and halts with new sets of 4 bits after computation and routing operation.

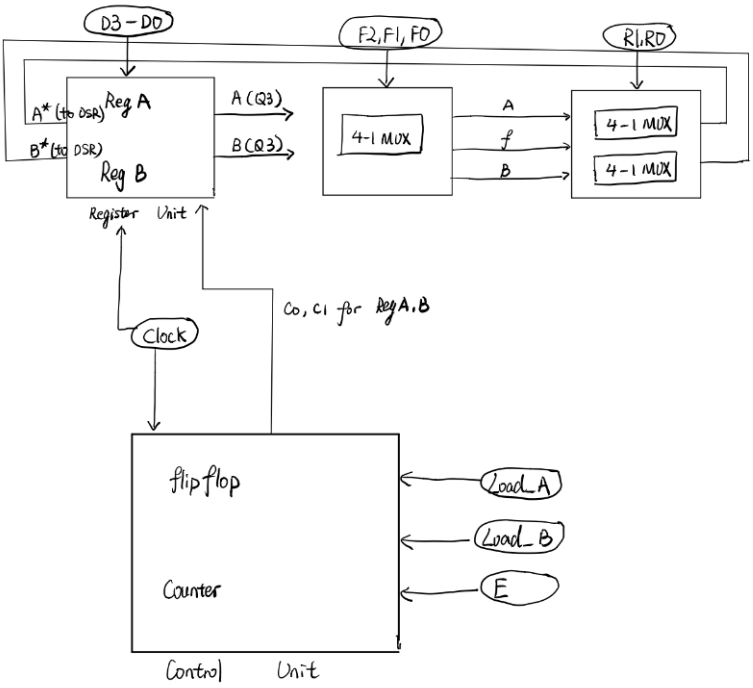
Function Selection Inputs			Computation Unit Output
F2	F1	F0	f(A, B)
0	0	0	A AND B
0	0	1	A OR B
0	1	0	A XOR B
0	1	1	1111
1	0	0	A NAND B
1	0	1	A NOR B
1	1	0	A XNOR B
1	1	1	0000

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

Detailed definition for Computation and Routing Units

3 Diagrams with Written Descriptions

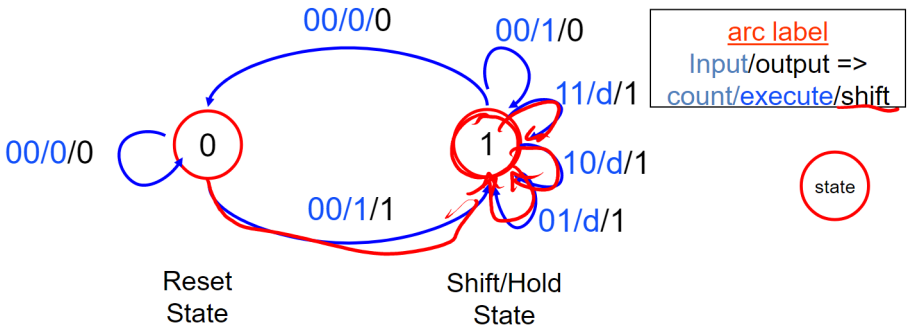
This is the high level block diagram of our design with all 4 units combined. Inputs controlled by us includes data($D_3D_2D_1D_0$), functional select($F_2F_1F_0$), and routing select(R_1R_0). The outputs($Q_3Q_2Q_1Q_0$) of registers are connected to LEDs for better visual inspection.



Next, we will go through each unit's function and diagrams in details.

3.1 Control Unit(FSM)

In Lab2.1, we used a Mealy State Machine to implement our control unit. The machine consists of a SN74HC74N Flip-flop, a CD74HCT163E counter, and several logical gates.

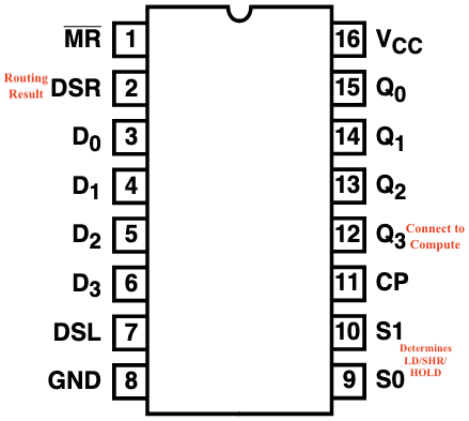


Our flip-flop would store value 0 if the FSM is currently in Reset State, and would have a value of 1 if it's in the Shift/Hold State. A feature of the Mealy Machine is that the output is decided by both the state and input. In our case, the output of the FSM(control unit), also known as S, will be determined by state and inputs(value of the counter and the Execute Signal).

We will discuss detailed expression as well as how we came up with this design in the next section(Section4.1).

3.2 Register Unit

In our register unit, we use CD74HC194E shift registers. Depending on the output of the control unit(S) and 2 inputs(LoadA and LoadB), the register unit will perform 3 different operations: hold value, parallel load and shift right.

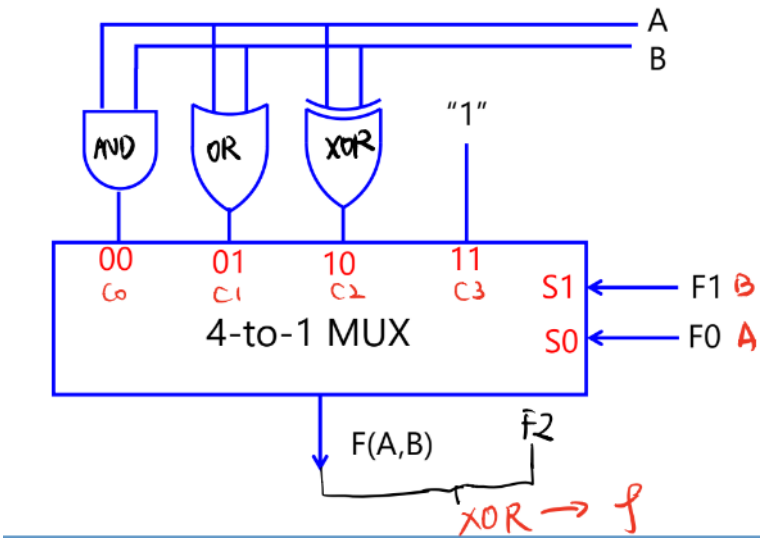


Please refer to Section4.1 to see how we derive the logical expression of S_1S_0 from LoadA, LoadB, and S(Shift) using truth table.

3.3 Computation Unit

In our computation unit, we used a SN74HC153N 4-to-1 MUX and several other logical gates implemented with NAND and XOR to accomplish the 8 required functions.

First, we implemented AND, and OR using NAND gate and connected the result with the input of the MUX while using F0 and F1 as the select bits of the 4 operations. Moreover, we discovered that the last operations are the negation of the first 4 operations; therefore, we decided to use an XOR gate and a select bit F2 to control whether we want to inverse the logic or not.

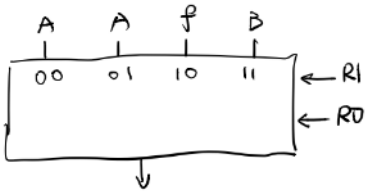


Please check the look-up table in Section 2.2 for detailed definitions.

3.4 Routing Unit

In our Routing unit, we used two SN74HC153N 4-to-1 MUX again. Each of the MUX handles the routing operation for its corresponding register.

Since there are two outputs A*, B*, we use 2 MUXs to fulfill the design; each MUX will send the output to its corresponding register. Since two MUX are very identical, here we use the one with output A* as an example, A, A, f, B are 4 inputs to the mux, and R1, RO are the control switches to decide which input becomes our output A*.



Please check the look-up table in Section 2.2 for detailed definitions.

4 Design Procedure & Schematic View

In this section, we are going to share our truth table, K-map, our design thoughts as well as the schematic view.

4.1 Design Considertaion & Truth Table/K-Map

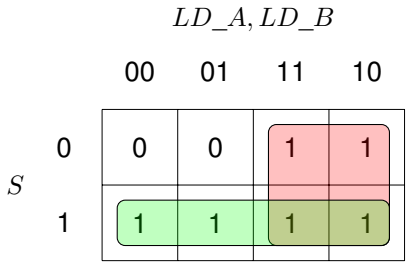
First, we wanted to determine when the register should shift, load and hold. We utilized the 2 pins S1, S0 on the shift register that controls whether the register should do parallel load or shifting. Since Professor Chen told us to assume that loading won't happen during shift, we decided to set the register to shift right whenever the output of the control unit(S) is true. Also we only do parallel load when S is false and the corresponding load button is pressed. Finally, the register should hold its value when all inputs are low. Below is a truth table for the timing of determining the values of S1, S0.

S	LD_A	LD_B	S0_A	S1_A	S0_B	S1_B
0	0	0	0	0	0	0
0	0	1	0	0	1	1
0	1	0	1	1	0	0
0	1	1	1	1	1	1
1	0	0	1	0	1	0
1	0	1	1	0	1	0
1	1	0	1	0	1	0
1	1	1	1	0	1	0

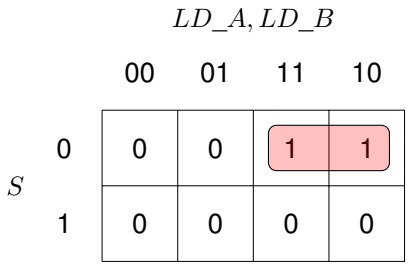
Truth Table for both the **naive** and the **optimized** circuit

LD_A means whether press the pushbutton to load data into register A. LD_A = 1 when the pushbutton for A I pressed and LD_A = 0 when not pressing. To make the register loading data, we have to make the register in parallel loading mode, which means that we need to set pin S1, S0 to be both at high. And the value of S1, SO is determined by the value of S and LD shown on the image (register table). When the register in parallel loading mode, S needs to be 0 and the pushing button for this register is pressed. For computing data, since here we are doing shifting right, so the rightmost bit Q3 will be sent into the computing system to process, and after running the routing unit, A* will be sent back to the pin DSR in register A.

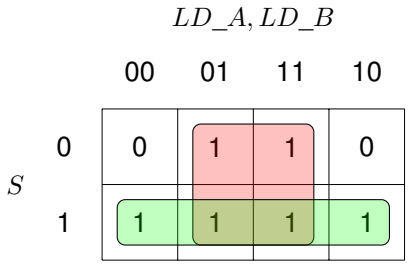
From the Truth table, we were able to draw up the K-map for S0_A, S1_A, S0_B, S1_B respectively.



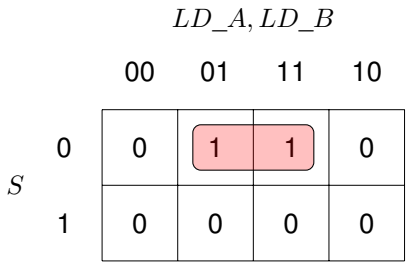
SOP: S0_A = S + LD_A



SOP: S1_A = S' • LD_A



SOP: S0_B = S + LD_B

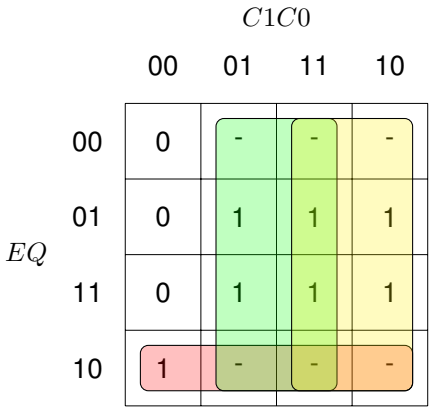


SOP: S1_B = S' • LD_B

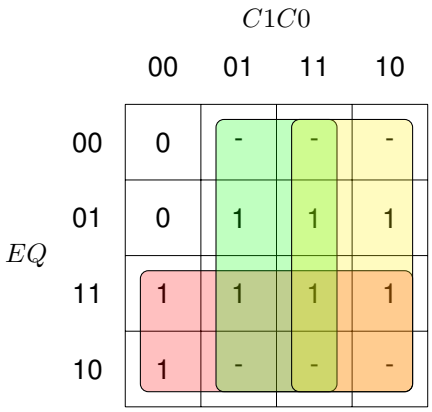
From the truth table of state machine below, we then concluded the k-map and the Boolean expression for output S and next state Q+.

TABLE 1: Control unit state transition table using 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



SOP: S = E·Q'+ C0 + C1



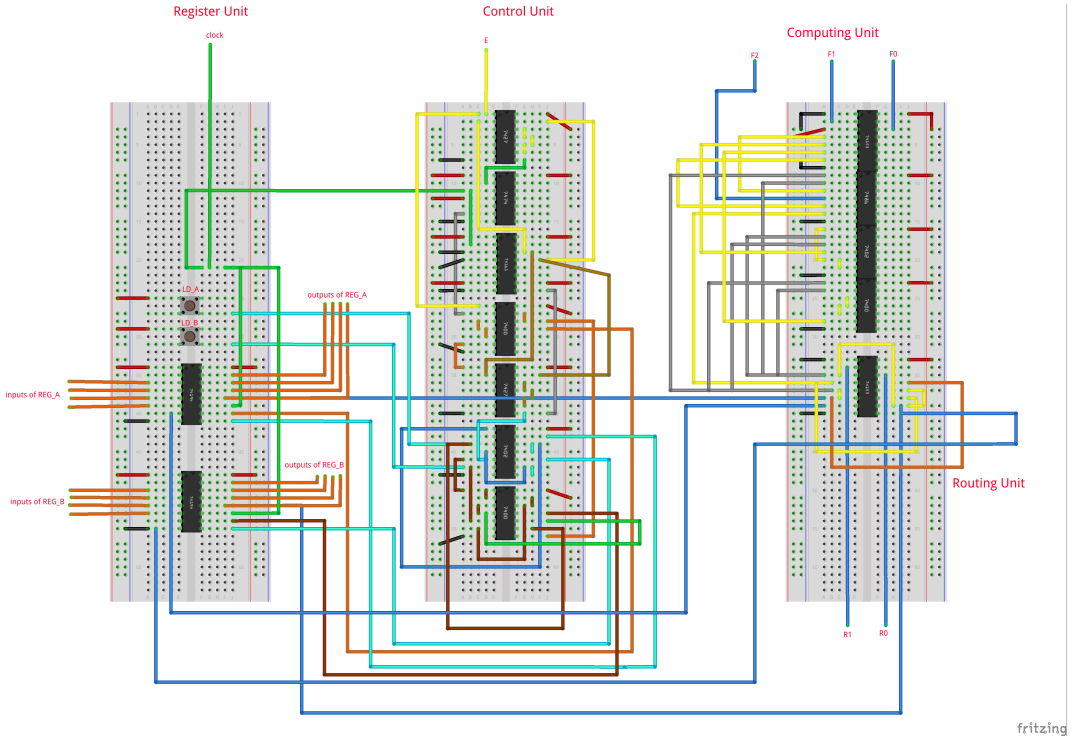
SOP: Q+ = E + C0 + C1

Once we get all the expressions, we used two 3-input NOR gates included in the SN74HC27N chip to implement the control unit and the register unit. According to the logical expression: when S is high, the register should be doing shifting and the counter should go into a counting mode and count exactly 4 times. As a result, we set pin PE, pin TE to high. When S is low, the register should hold, and the counter should show 00 when halting. In order to fulfill that, we utilize the parallel load mode in the counter. We loaded 00 from pin P0 and P1 TO achieve that. In addition, we also connect S to the counter, so that the counter will only count when S is high and reset to 0 when S became low.

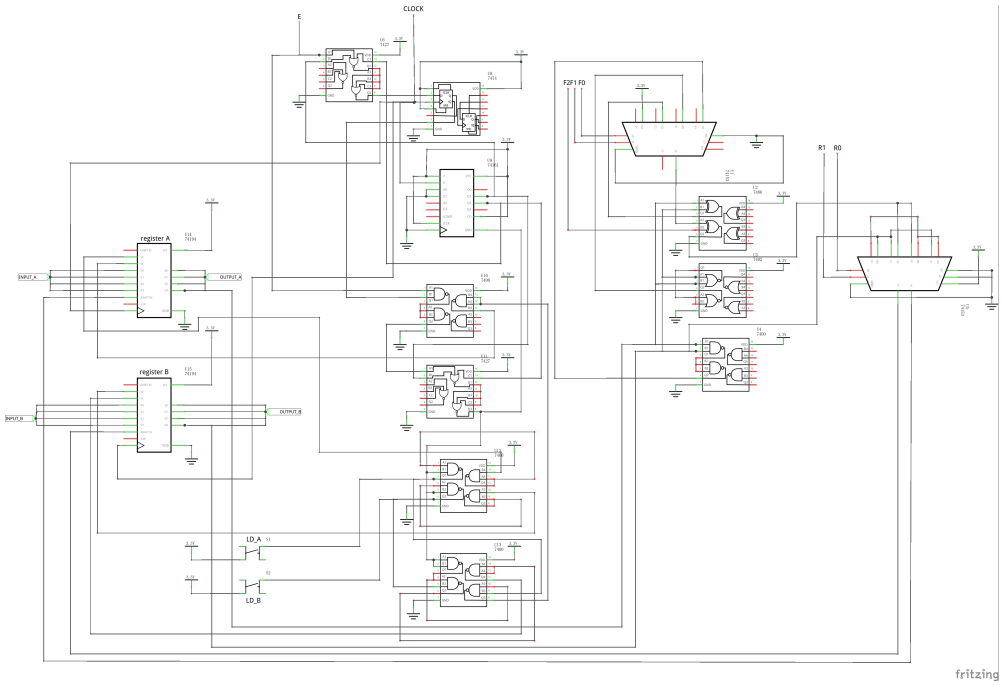
Overall, we spent most of time googling different components' datasheets and trying to figure out each pin's target. The control unit takes us the most time in both writing the expressions of S and Q+ from the Mealy State Machine's truth table and trying to make sense of them. Once we got the logical part correct, it's just wiring things up.

During debugging the hardware circuit, we used the old school LED method introduced in the Lab1 homework. We connected one end of an led to GND, and left the other end as the debugging pin. If the debugging pin is high, then we should see the LED light up. We used this debugging pin to check whether certain values meets our expectations. If it is opposite to our theoretical value, then we know the wiring up to this node might be wrong and can then backtrace the problem source.

4.2 Detailed Breadboard View



5 Circuit Schematic View



6 Extended 8-bit Logic Processor on FPGA

6.1 Summary of All Modules

(a) **Processor.sv**

This is the top-level module of our 8-bit logic processor. In this module, we extended values of A and B from 4-bit to 8-bit; moreover, we enabled 2 new hex drivers, because, we need 4 hex drivers in total to represent 2 8-bit values.

(b) **Compute.sv**

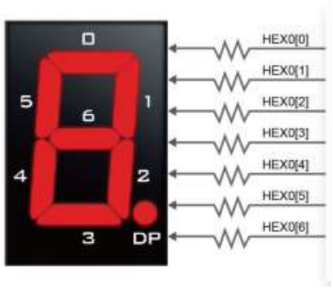
This module is in charge of calculating the bit-wise operation result of the 2 bits fetched from the registers. Since we still used the same functions as the 4-bit processor, we didn't make any changes to this module.

(c) **Control.sv**

This module implemented our control unit using a Moore Machine. It defined our state transition and reset/next-state conditions. We added 4 mores states to the Moore FSM to incorporate 4 more shifts in the 8-bit processor than the 4-bit one.

(d) **HexDriver.sv**

Hex module is used to output the hex value on LED using the 7-segment method. Each hex value has its unique corresponding 7-bit binary value. The LED will light up its segment if the corresponding bit is low. We stilled used the same encoding method so we didn't make changes to this file.



(e) **Register_unit.sv & Reg_4.sv**

The register modules handled all operations in our register unit. Depending on various input values, it would either reset, hold, shift, or parallel load. We simply adopted the change to 8-bit by extending the variable length.

(f) **Router.sv**

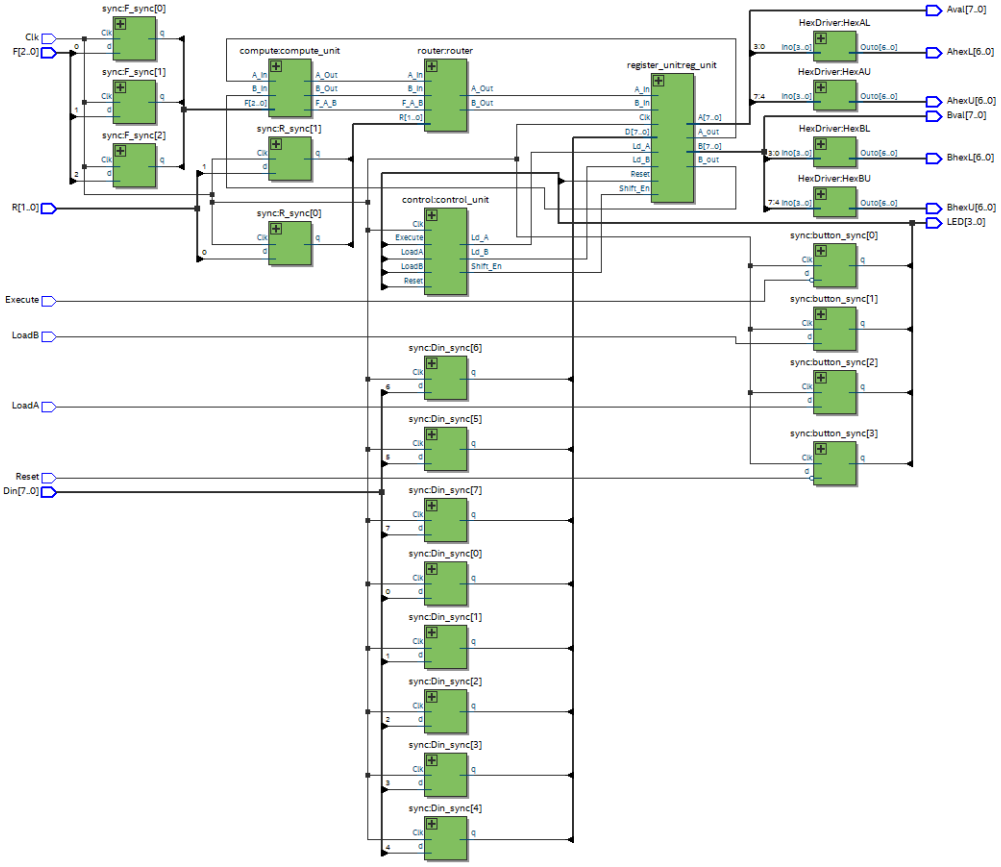
The Router module takes care of sending the computed result from Compute.sv back to the registers. Since we still used the same routing options(check Section 2.2 for details), we didn't make any changes to this module.

(g) **testbench_8.sv**

This module is a test case provided to us to verify our design through simulation. It simulated 2 XOR operations and then swapped the values in Register A and Register B. Next, it compared the computed value with the theoretical value and checked for errors. We only changed the module name to avoid conflict with the other 4-bit testbench.

6.2 RTL Block Diagram

With the help of ModelSim, we generated the RTL Block Diagram using "RTL Viewer"



6.3 Simulation with Annotation

With the help of ModelSim, we generated the RTL Block Diagram using "RTL Simulation"

6.4 SignalTap ILA Trace Procedure

Due to the limited switches we have on DE10, we decided to hardcode the function select bits(XOR) and our routing select bits. After that, to enable SignalTap Trace, we first opened the SignalTap window and compiled our code. Once everything has been compiled successfully, we upload the program to our FPGA. Next we set the signals we want to trace(in this lab, it's A, B and execute); we set the trigger event to the either edge of the execute signal and applied the same clock to all signals. Once we started the acquisition in signalTap, we loaded the desired value(8'h33 and 8'h55) into register A and register B respectively and executed. Finally, we were able to see how the value of register A and register B changed in different cycles/timestamps. And the final output became 8'h66.

7 Bugs & Fixes

(a) **Bug: Our counter continued to count after 4 cycles.**

We checked every pin on the counter chip and found that we connect pin (SPE)' to VCC. Theoretically, we need the counter to stop when S = 0, so we should connect S to (SPE)' to decide whether stop or not. We corrected this and the counter works.

(b) **Bug: Pushbutton doesn't always work**

In the beginning, our pushbutton for control loading A, B was not connected to the ground, so sometimes the value of loading flip between 1 and 0 and made our design not working. We originally it is the fault of the pushbutton, so we changed it into 8 - DIP Switches, and all the problems disappeared. When comparing the connections of those two kinds of switched, we found that we forgot to connect to the ground when doing with Mini Pushbutton Switch.

(c) **Bug: Flip-flop value fluctuates**

After further examining using the LED debugging method mentioned in Section 4.1, we discovered that each pin on the flip-flop chip should be connected and couldn't be left floating. This will cause an undesired consequence. The pins we missed were "PRE" and "CLR"; after we set the correct value, Flip-flop behaves as normal.

(d) **Bug: Circuit connected the clock to the wrong pin.on error**

Since it's a complicated design, we sometimes misread the datasheet and connect the wire to the wrong pins even though our logic is correct. As a result, we connected the clock to the wrong pin. We fixed the bug using the LED Debugging method described in Section 4.1.

8 Post-Lab Questions

(a) **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 help you isolate design and wiring faults, be specific and give examples from your actual lab experience.**

The main difficulty we had during debugging and unstable behavior(E.g. when we shook the table a little bit, the value changed from 0011 to 1111). With the modular approach proposed in the pre-lab, we isolated the circuit part by part, where one of us debugging the control unit and the other working on the compute unit. We set up 2 LEDs with one end connected to GND, and have the other end as the testing pin to check up the signal in our circuit. With this debugging method, we discovered that the reason for the unstable behavior is we didn't ground the switches and we also missed a clock signal.

(b) **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.**

In our compute unit, we utilized an XOR gate, a select bit, and the original signal to implement the above circuit. Due to the nature of XOR, if the selecting bit is 0, the output will be exactly the original signal; while when the select bit is 1, the output will be the inverted original signal. This design is extremely useful because it simplified our circuit. Without this XOR design, we would have to use an 8-to-1 MUX, which means more logical gates, more wiring, and potentially more debugging work to do. However, if we apply the XOR design, we only have to use a 4-to-1 MUX and an XOR gate, they saved us more work and costs.

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

From our own experience, employing modular design saved us much time because we can apply the divide-and-conquer technique. While one of us is working on the compute unit, the other can already start debugging the control unit. It's more efficient for working together. It also cuts down both debugging and development time since working on a module took fewer steps than checking up the whole circuit.

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

The difference between Mealy Machine and Moore Machine is the output of the former depends on both state and input while the one of the latter only depends on the state. We choose to implement a Mealy machine in that it only involves 2 states, and thus can be represented with only 1 Flip-flop. The trade-off is even though we used fewer states and flip-flops, our logical expression would be more complicated because it also involves an extra term(input); as a result, we used more logical gates. But after all, it's just a design choice: more flip-flops versus more logical gates.

(e) **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?**

The main difference between ModelSim and SignalTap is that ModelSim runs the simulation with testbench we provide to it, while SignalTap captures and displays the real signals on real hardware(in our case, switches/button on DE10 FPGA). In the early stage of debugging, SignalTap might be better, because it can provide real-time feedback on small sets of test cases; however, if we want to run a large portion of verification(E.g. a nested for loop with 10000 iterations), ModelSim might be preferred in that it's more efficient and easy to check for errors.

9 conclusion

This is a long but interesting lab, it took us nearly 20 hours to finish both parts and the lab report. Through Lab2.1, we learned many based techniques when building a TTL circuit, like reading from datasheet and how to efficiently wire up the entire circuit. More importantly, we learned the debugging method using LED to fix the bug when our circuit didn't behave normally. During Lab2.2, we were exposed to some intro-level SystemVerilog; we understand what modules are for and how should we wire up the modules using implicit/explicit port connections. Moreover, we also learned how to generate simulation with our testbench and using SignalTap for real-time debugging. This lab lays a solid foundation for us to understand basic TTL circuits and HDL/HVL.

Appendices

.1 Our Circuit