ECE 385

SP 2022

EXPERIMENT #2

A Logic Processor

Tianyu (Joe) Yu, Yulun(Ben) Wu

2/14/2022

Curtis Yu

Introduction

Our circuit can perform bitwise operations by taking bits from a register and shifting the result back to the designated register. The processor is capable of doing NAND, AND, NOR, OR, XOR, XNOR, all '0', and all '1' operations. Our lab2.1 is a hardwired design that can operate on 4 bits, and our lab 2.2 FPGA design can operate similar function in 8 bits system.

Operation of the logic processor

The sequence of switches the user must flip to load data into the A and B registers:

Step 1: Make all switches are open when initializing the system

Step 2: User will choose the binary value that needs to be stored in register A or B and then flip the corresponding four switches of D[3:0] (each represent one bit of 0(open) or 1 (close))

Step 3: close the Load A or Load B to load the value into register A or B

Step 4: Open the Load A and Load B switch to stop changing value in current loading register

Step 5: Repeat steps 2, 3, and 4 to load the data in to the other register

Describe the sequence of switches the user must flip to initiate a computation and routing operation:

Step 1: Make sure all switches are open, and A and B are loaded

Step 2: Open/close switches F2, F1, F0 to select the function the user wants to perform

Step 3: Open/close switches R1, R0 to select the register that the user want the results store in, or simply leave the value unchanged or switch the value in two registers

Step 4: Close the "execute" switch to execute the computation

Written description, block diagram and state machine diagram

a. Written description of each block in the high-level diagram

   i. Register unit

      The register Unit consists of two shift registers (74194) representing register A and B. The shift registers are connected to the CLK and get their input values directly from switches (D[3:0]). The shift registers will get their shift-in bit value from the routing unit, and their shift/load signal from the control unit's shift/load control unit.

ii. Computation unit

Computation unit consists of two NAND chips(7400), one NOR chip(7402), and one 8 to 1 MUX(74151). Computation unit has inputs from the Register unit and computes eight combinational logic at the same time and selects the desired output from input value F[2:0] from the switch. The output of the Computation unit are original inputs A and B, and the calculated result f(A,B).

iii. Routing Unit

The routing Unit is basically a 4 to 1 MUX (74153) with designed input order. It gets input logic values A, B and f(A,B) from the Computation unit, and R[1:0] directly from switches. The outputs of the Routing unit are connected to the shift-in bit on Register A and B, and this unit will send one of three inputs to each register's input via R[1:0].

iv. Control unit

The control unit is a Mealy State machine that gets input from the clock, load A & B and 'execute' logic from the switch. It has two major sub-units: shift/load control unit and counter/state control unit. The counter/state sub-unit is responsible for precisely running digit shift in 4 clock cycles, consisting of one Flip-flop (7474) representing the Q's state, one 3-input NAND (7410), one inverters (7404), one counter (74163), and one 2-input NAND (7400). It is has 2 states (Q=0 or Q=1) and 1 output (CLR') that clear the counter to zero after a complete computing cycle. The shift/load sub-unit will first compute an output S (shift). Then it will use two 4 to 1 MUX (74153) with designed input order and the value of Load A, Load B,and S, to compute and send each register S[1:0] value, making it load or shift value correctly.

b. high-level block diagram

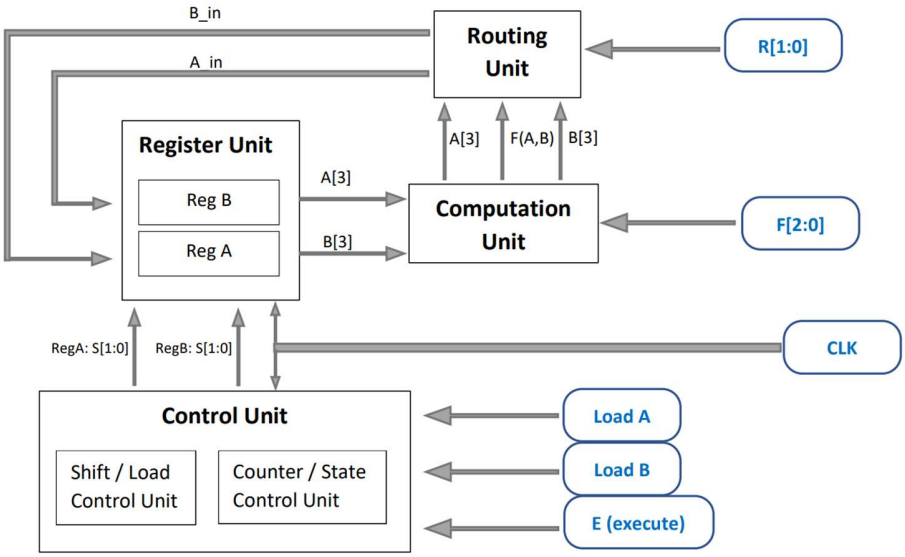

Figure 1, high-level block diagram

c. State Machine Diagram

We used a Mealy machine to represent the control unit's design. There are two states of Q which stores in a flip-flop., each representing Rest (Q=0) or Shift/Hold (Q=1). With input Count (C1 and C0 from the counter) and the Execute button, the machine can generate two results, Shift and CLR'.

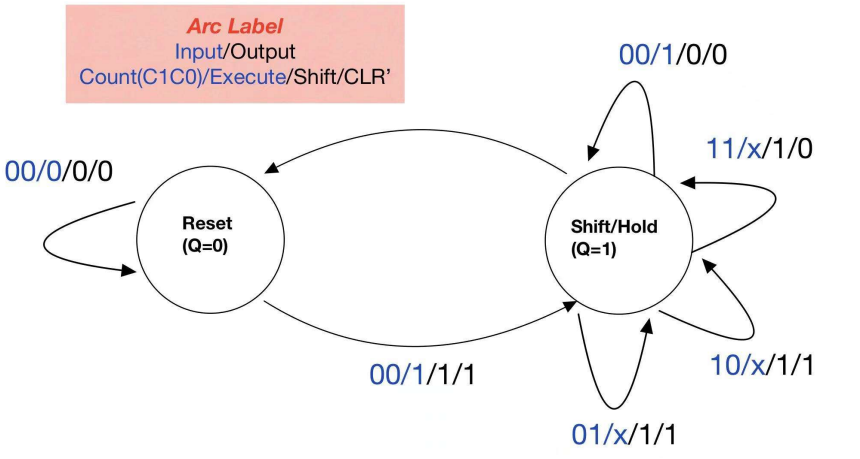

Figure 2, State Machine Diagram

Design steps taken and detailed circuit schematic diagram

a. Written procedure of the design steps taken

E	Q	C[1]	C[0]	S (shift)
0	0	0	0	0
0	0	0	1	x
0	0	1	0	X
0	0	1	1	X
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	X
1	0	1	0	X
1	0	1	1	X
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

C[1]C[0] \ EQ	00	01	11	10
00	0	0	0	1
01	x	1	1	x
11	x	1	1	x
10	x	1	1	x

E	Q	C[1]	C[0]	Q+
0	0	0	0	0
0	0	0	1	x
0	0	1	0	X
0	0	1	1	X
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	X
1	0	1	0	X
1	0	1	1	X
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

C[1]C[0] \ EQ	00	01	11	10
00	0	0	1	1
01	x	1	1	x
11	x	1	1	x
10	x	1	1	x

E	Q	C[1]	C[0]	MR' (Clear counter)
0	0	0	0	0
0	0	0	1	X
0	0	1	0	X
0	0	1	1	X
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	1
1	0	0	1	X
1	0	1	0	X
1	0	1	1	X
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

C[1]C[0] \ EQ	00	01	11	10
00	0	0	0	1
01	x	1	1	x
11	x	0	0	x
10	x	1	1	x

Figure 3, K-map and truth table for shift signal (S), Q state, and CLR' for the counter

i. At the beginning of design, we didn't apply the counter in the control unit. Instead, we tried to use pure combinational logic to replace the counter and directly compute the Shift value and Q state. However, after carefully making the truth table and K-map, we discovered that this design will take too much logic gate to complete, and therefore it will be extremely difficult to debug. So we abolished that design and implemented a counter in the circuit to help us count the 4 clock cycle.

ii. This circuit build was fairly straightforward, we did take some shortcuts (Using the values generated mid-way as input for other logic functions) in the Computation unit to minimize the number of chips. This slightly increases the work for debugging but greatly reduces the space and power taken by the computation unit. This is a great trade off since we have limited space on the breadboard.

b. Detailed Circuit Schematic

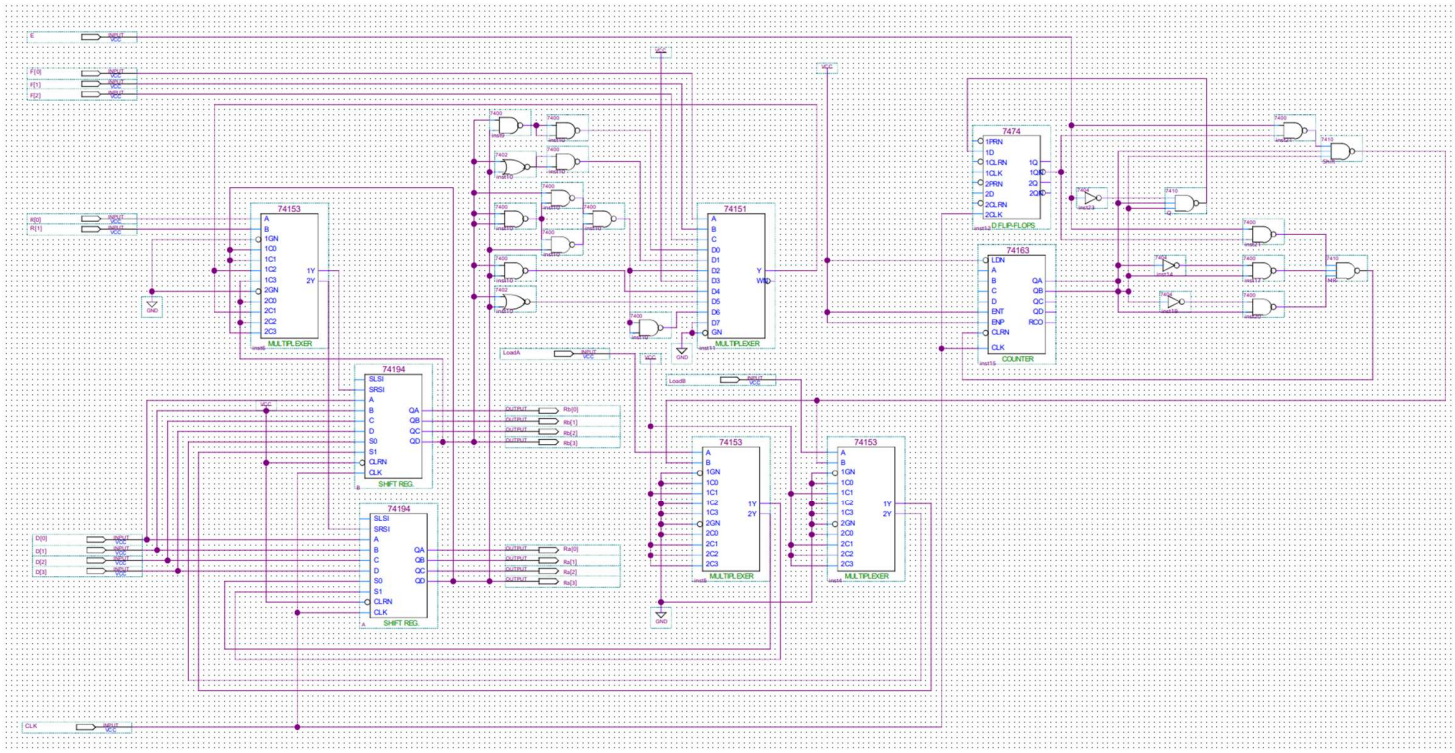

Figure 4, Detailed Circuit Schematic

Breadboard view / Layout sheet

a. Layout sheet

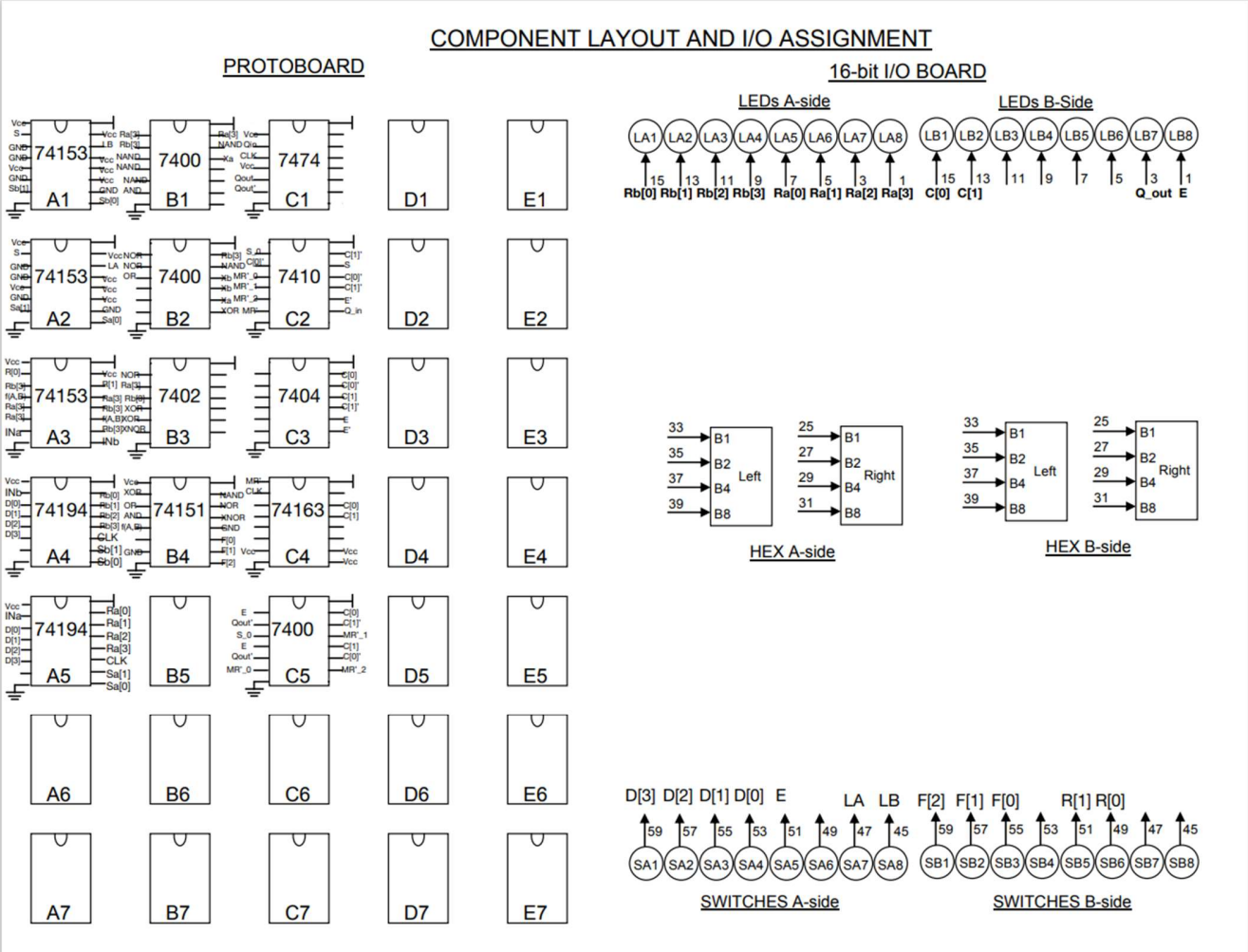

Figure 5, Layout sheet

8-bit logic processor on FPGA

a. Summary of all .SV modules and the changes you made to extend processor to 8-bits.

i. Summary of all .SV modules:

The system in Systemverilog basically consists of the same function that we realized in breadboard. The register_unit module contains 2 sets of reg_4 modules which has the same functionality as the shift registers (74194). The compute unit work as an ALU, and it select the correct output logic by implementing “unique case”. The router

module contains two always_comb parallel works as MUXs to send out the correct value to register_unit with “unique case”. The control module in “Control.sv” accomplished its mission with Moore machine written inside, and each state is directly encoded with its S (shift) value and the possibility to load A or B with Ld_A or Ld_B. The “Processor.sv” is the top-level file and it act like a “main” file to combine all modules we wrote together and run them. Else .sv files are responsible for testing (“testbench_8.sv”) or drive the Hex display in our FPGA correctly in this lab (“HexDriver.sv”).

ii. The changes we made:

1. We added four extra states, changing from A-F to A-J, so that now the register can shift 8 times for every execution.
2. We change the length of the registers from 4 bits to 8 bits by changing both input and output logic. We also changed the several logic within the Processor.sv file, like Din, Din_S, Din_sync, Aval, A, B, to match the number of bits with registers. In the same file, we adjusted the HexDriver module to allow 2 Hex digits displayed on the FPGA board.

b. RTL block diagram - please only include the top-level design if using the RTL viewer.

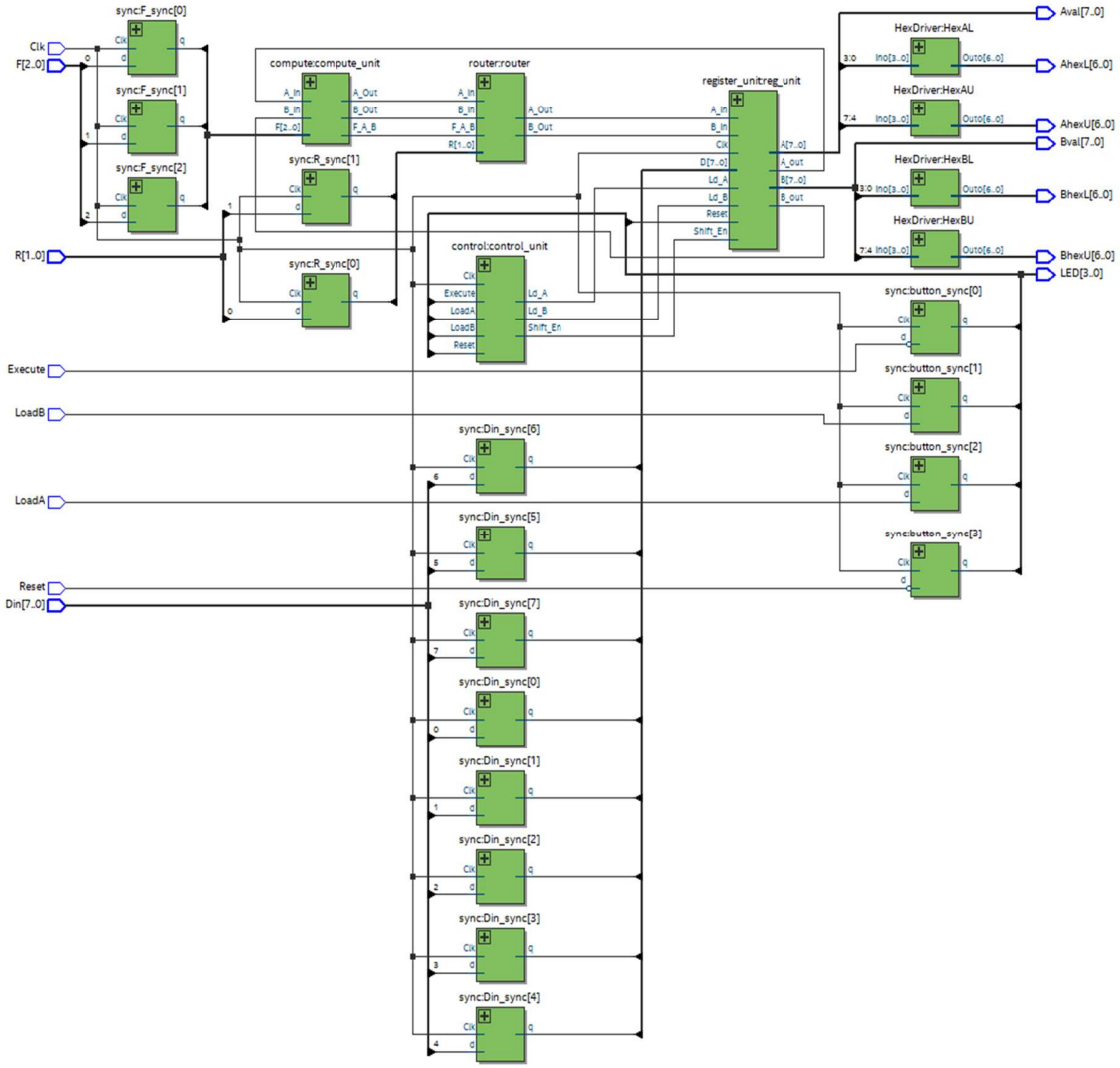

Figure 6, RTL block diagram

c. Include a simulation of the processor that has notes (annotations) that give information such as what operation is being performed, where the result was stored, etc.

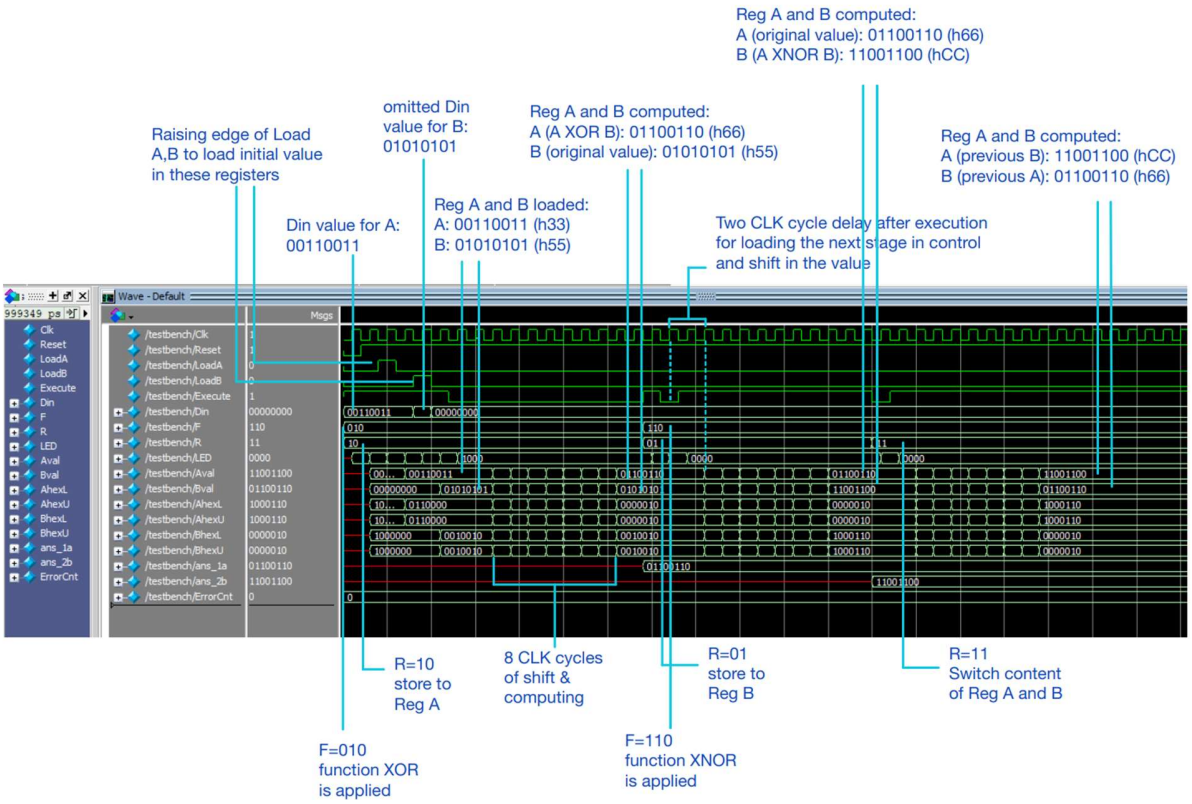

Figure 7, simulation of the processor with annotations

d. Include procedure used to generate SignalTap ILA trace, as well as the result of such trace executing the 8'h33 XOR 8'h55 operation.

i. Step 1: Set up Clock in "Signal configuration"

ii. Step 2: Add the nodes that we want to exam

iii. Step 3: Set trigger condition on the "Execute (E)" to either edge on execute signal

iv. Step 4: Start Compilation and implement the program on FPGA

v. Step 5: Load values into registers and execute, observe the triggered SignalTap trace.

Figure 8, result of SignalTap executing the 8'h33 XOR 8'h55 operation

Description of all bugs encountered, and corrective measures taken

We spend most of our time debugging the issue of floating gates. On the 8 to 1 mux (SN74HC151), according to the data sheet, the STROBE pin should be L in order to have designated output, but we didn't notice it and left it floating. We first tested the signals right before entering the 8 to 1 mux to make sure that the circuit prior to the 8 to 1 mux is functional. Then we switched two new chips to eliminate the bug within the chip. Finally, we re-read the datasheet carefully and found the solution.

Conclusions

In the first section of this lab, we accomplished a 4-bit bit-serial logic operation system with RTL design. The system is designed based on the Mealy machine with different units that guide the data calculated and routed correctly. We input values through switches and get the output response through connecting LEDs to the registers. We adjusted and implemented an 8-bit system with Systemverilog code in FPGA in the second section. Each unit is encoded with modules in Systemverilog with similar functionality. We learned to observe the RTL simulations and the SignalTap result to examine our work.

Answers to Post-Lab Questions

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.

- Modular design was helpful because we can test each module individually before connecting them together, making our debugging process much faster. For example, we had an unknown bug in the computation unit during our building process. If we didn't follow the modular design, we had to debug the whole circuit, but now, since we have modular design, we simply test the inputs of the 8 to 1 MUX and quickly find that we have a wrong connection for the MUX.

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 A and B, output = ((AB')'(A'B)')', or we can have output = A XOR B
- This is useful for out computation unit since we can express XOR using only NAND gates, which mean that we don't need to place down an extra chip just for the XOR function

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

- Modular design allows us to unit test each section which greatly improves testability. It also simplifies a complicated design into small segments which is easier to understand and cuts down development time. It also allows people in a group to focus on different modules which further decrease the development time.

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

- We designed a Mealy machine because we first designed a state machine with only reset and shift states, then we added a hold state inside the shift state and made the output depend on both input and the state.
- Mealy machines have fewer states, since we only have two states, only one flip-flop is required. However, the trade off is that we need to add some extra logic to make the output depend on both the state and the input.

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 software based. It initializes values of registers and applies different input cases through the testbench file. Then it will simulate the running process and return an oscilloscope diagram.
- SignalTap is hardware based. The testbench is no longer needed and we will receive real time feedback from the SignalTap with our manual input on the switches on the FPGA board. We generally applied a trigger on the Execute (E) to better observe the detailed change within a computation cycle.
- ModelSim is preferred in an earlier stage of the project, where we can test various cases through changing values in the testbench file without actually applying the FPGA board. On the other hand, SignalTap is preferred when we want to test the performance of a program that is actually running on an FPGA board. Thus, we manually adjust the SignalTap setting and operate on the FPGA board and receive feedback on the software.