ECE 385

Fall 2023

Experiment #2

Lab 2

Haoyu Huang, Eitan Tse

Lab section: ZY 16/Day & Time: 9.5

Your TA's Name: Yang Zhou

1. Introduction

The circuit performs bitwise logic operations, and it can operate on 4 bits. The circuit routes the output of a specified computation to a specified destination register. The operations that the circuit does are AND, OR, XOR, NAND, NOR, XNOR, CLR, SET, SWAP. We need to set the register values, choose computation method, where the result is being routed, and when to execute.

2. Operation of logic processor

Here are the instructions about how to use the logic processor. For the input part, there are load A, load B, a 3-bit function control, 2-bit router control, and execute. There are four switches which represent 4 bits. The user will first set 4 bits which can be loaded to register. When user press Load A and/or Load B, we can make 4 bits parallel load to register A and/or B. The function control decides which logic computation will be performed. The router control determines which register will store the result of the logic operation. It can also set the registers to swap or hold values. The register will get the shift-in value from the route part of the circuit.
Finally, the execution signal triggers the start of an operation. After logic computation, the circuit will halt when the execution is high. After turning the <Execute> off, we can reset A and B. Even if users turn off the <Execute> signal during shifting process, it will continue right shifting until all 4 bits have been shifted out and computation has completed.

3. Description of Block Diagram and State Machine

Overview:

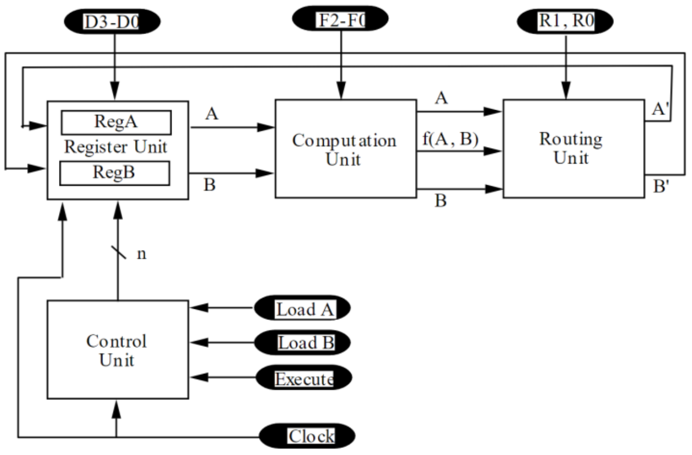

Figure 1: High-Level Block diagram

Control, register, computation, and routing units work together to implement the logic processor.

Register unit:

The register unit is composed of two registers (74194). The input of the unit is 4 bits which are set by user, the CLK signal, shift right signal, shift-in value, and the load signal. The shift-in value comes from the routing unit. The output of the register unit is shift-right value.

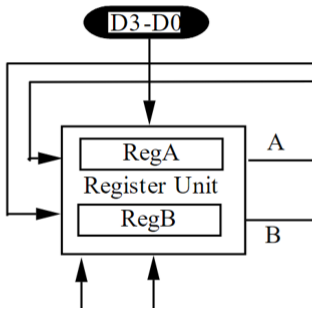

Figure 2: Register Unit

Computation unit:

Computation unit will accept the shift value from register and make logic computation like AND, OR, XOR, NAND, NOR, XNOR, CLR, SET, SWAP. Figure 3 shows the specific selection inputs of every logic computation. They are chosen by a 3-bit function control.

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

Figure 3: Function table

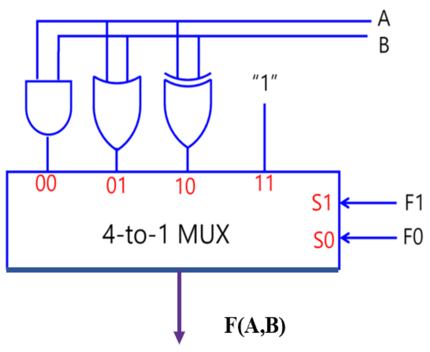

Figure 4: Computation Unit

Routing Unit

Routing unit will accept result from computation unit and 2-bit router control from the user. Figure 5 shows the router control function. The output of the routing unit is shifting the right value of register A and B. They will be connected to the Register unit.

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

Figure 5: Router control function

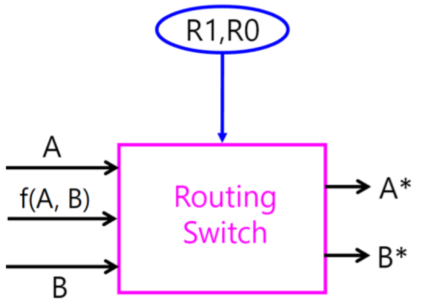

Figure 6: Routing Unit

Control Unit

Overview description:

The Control Unit is the most significant unit. It can decide when to load value, rest value, halt and execute. It can make register shift right every CLK cycle. It has input <CLK> and <Execute>. Its output is Shift signal. The unit is composed of a 4-bit counter, a flip-flop, and combinational logic.

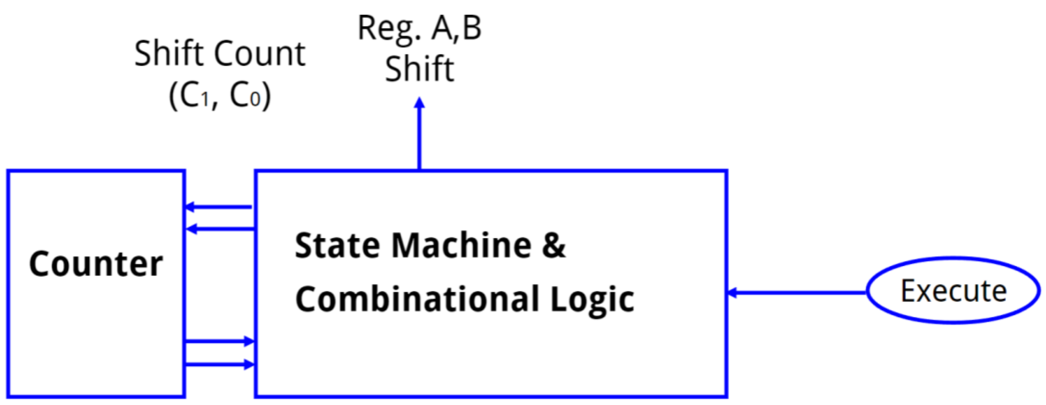

Figure 7: Control Unit (high-level and logic diagram)

State machine:

There are two ways to implement the state machine: Moore or Mealy machine. Moore has more state than Mealy. It is kind of complicated when building circuits, we choose Mealy machine.

Arc label
Input/ Output: Counter / Execute / Shift

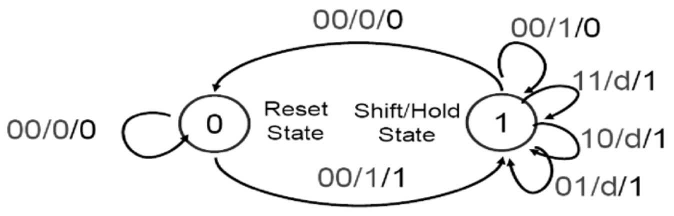

Figure 8: State machine

4. Design steps and circuit schematic diagram

Control unit is the most important part of the circuit, and it is easy to debug using FPGA. We decided to build it first. We start from the state machine (Figure 7). Since we must shift 4 times, we need a counter to count the cycle. We use Counter (74163 4-bit synchronous up-down counter): Since we must shift 4 times, we need two bits to count the cycle. We need to clear the counter when shift signal falls to 0. We also use Flip-Flop (7474) to store value Q. Since we use the Mealy machine, we need to store our current state. The flip flop with the CLK signal that took in Q+ (next state) as an input (output Q feedback to state machine to decide whether to shift). To implement the state machine, we need to derive K-MAP from the truth table to complete Boolean function.

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

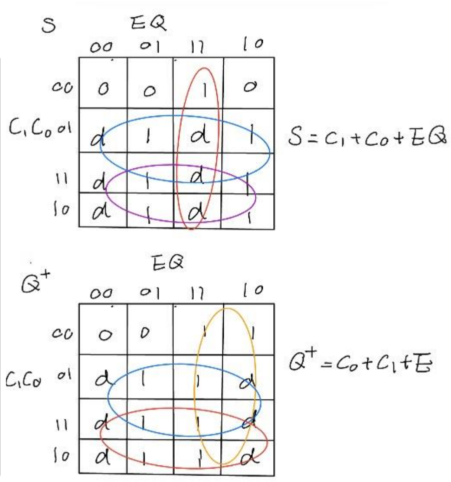

Figure 9 Truth table and K-map

Boolean function:

$$Shift = C1 + C0 + EQ$$

$$Q^{+} = C0 + C1 + E$$

According to the Boolean function, we complete the control unit with few logic gates. The inverted *Shift* signal is fed back to the counter which can reset counter when it has shifted 4 times. The rest of the gates are used to implement Boolean function.

We only have NAND gate and NOR gate. We combine invert gate with them to implement the circuit. We thought of using NOR2 gate instead of NOR3 gates. It may reduce the number of chips.

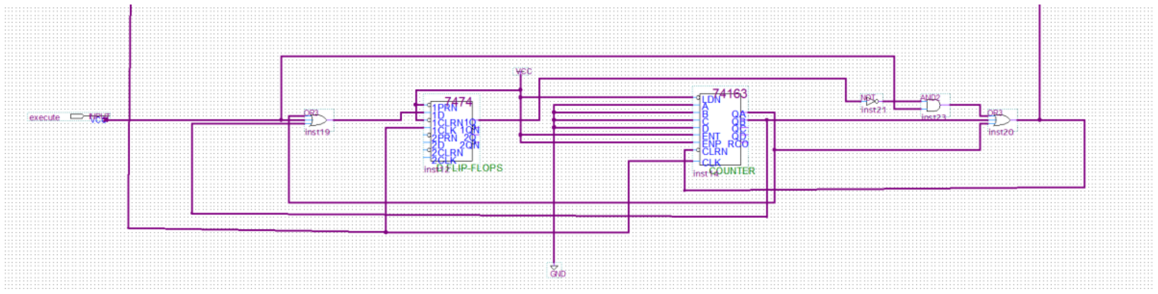

Figure 10 Control Unit schematic diagram

Secondly, we implemented the register unit. To implement our register unit, we used two 4-bit shift registers. These registers have 9 inputs. A, B, C, D are parallel loads. CLK is connected to clock signal. CLRN is connected to VCC. SRSI is connected to routing unit. To make register shift, we fed the load signal to S1, and OR the shift signal with the load signal to S0. S0 and S1 can decide when to load value and when to shift right. If <*Execute*> signal is high after 4 shifts, the control unit will remain halt. If <*Execute*> signal is turned off when shifting bit, the circuit will complete shift and change to reset state.
We tested the circuit immediately after we built the register and control unit. We test whether the register can do parallel load, and we first connect the right shift value to logic 0 and test whether the register can shift with control unit.
We only have NAND gate and NOR gate. We combine invert gate with them to implement the circuit.

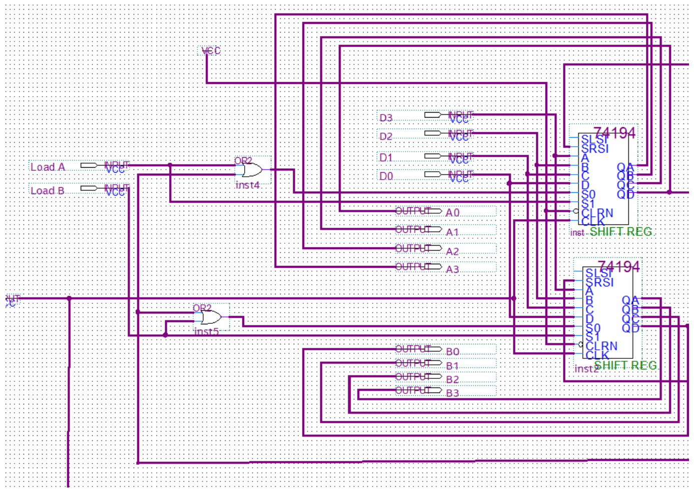

Figure 11 Register Unit schematic diagram

Thirdly, we built computation unit. I connect the right output of register to OR, AND XOR gate. (Since we don't have OR AND gate. I use NOR, NAND gate and inverter instead.) The results (OR | AND | Ground) are connected to a multiplexer (MUX 74153). Since NAND, NOR XNOR, 0000 are inverses of the previous 4 function, we can control them through the first bit of function control. When F2 is equal to 1 we need to implement the invert. So F2 should be connected to XOR gate with the output of MUX.

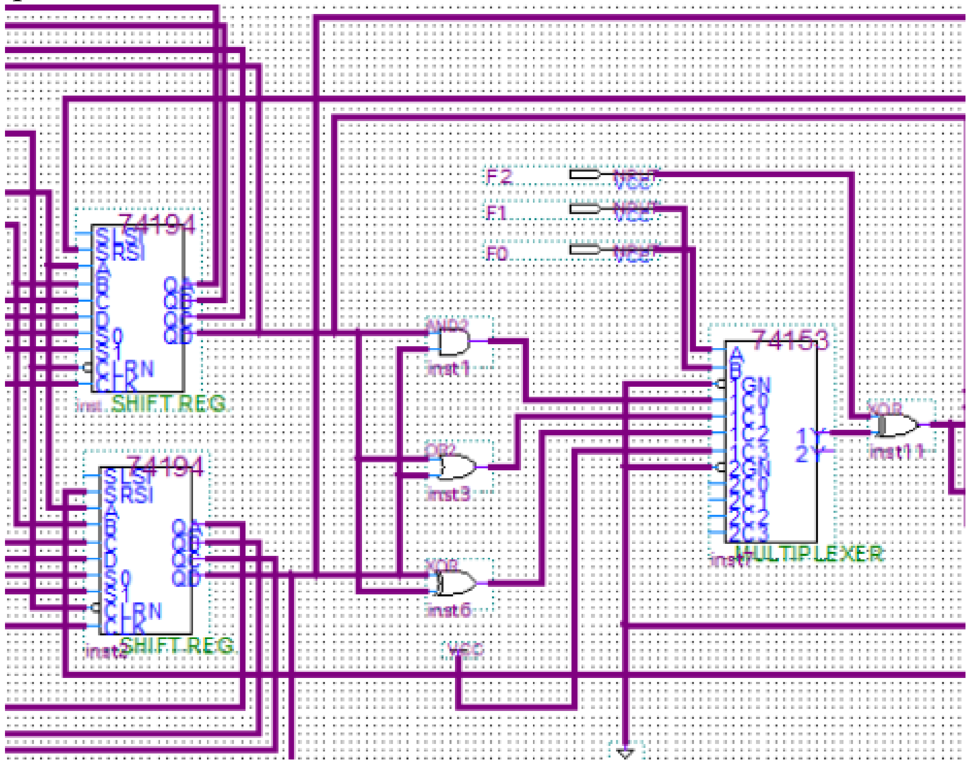

Figure 12 Computation Unit schematic diagram

Lastly, we build routing unit. Its inputs are RegA, RegB, and
F (A, B) from the computation unit. They will be determined by R1, R0. The outputs will be fed back into the register unit. To implement the routing unit, we used two MUXs (74153) and connected outputs to SRSL.

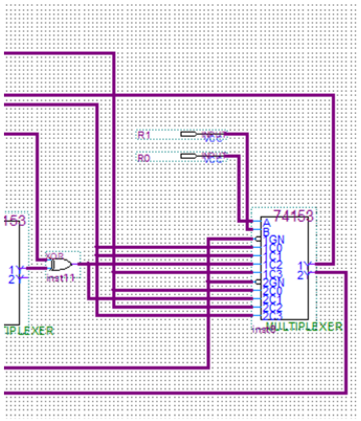
Figure 13 Routing Unit schematic diagram

Design Considerations/Trade-offs

For computation unit, one way is to use 8 inputs. We can use another 4 inverters to inverse the first 4 input. The other way is to use another XOR gate and F2 to inverse the output signal. This is an easier way to implement the circuit.

5. Circuit Schematic

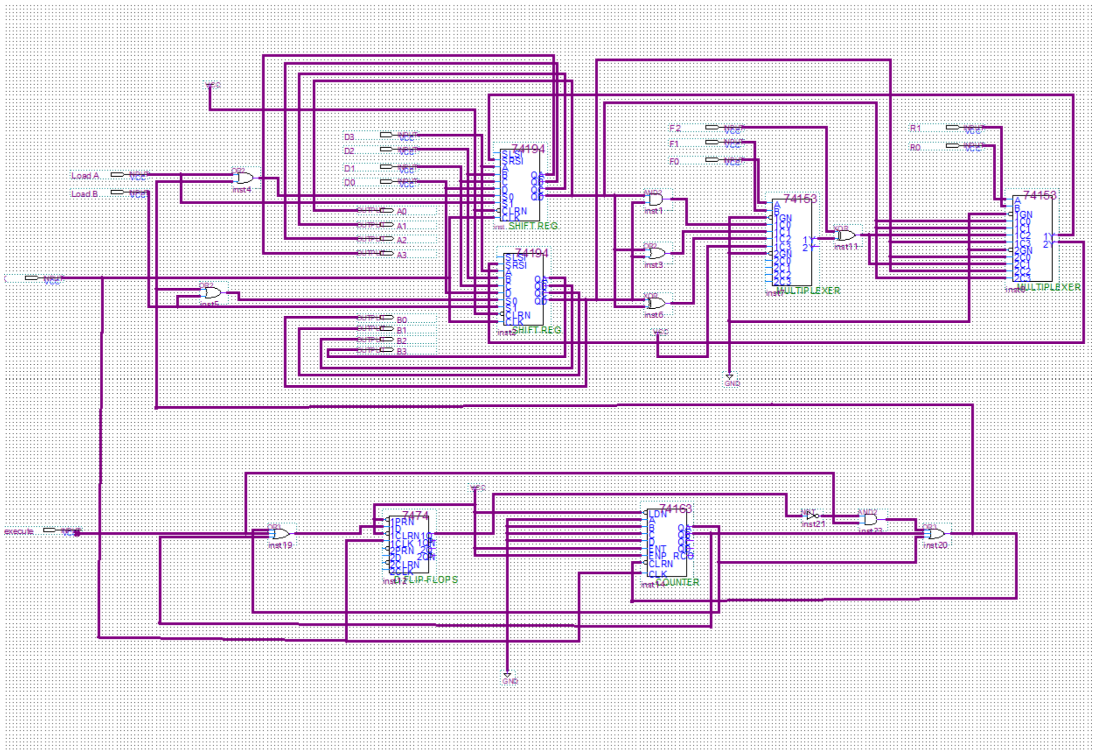
Figure 14 schematic diagram

4. Component Layout

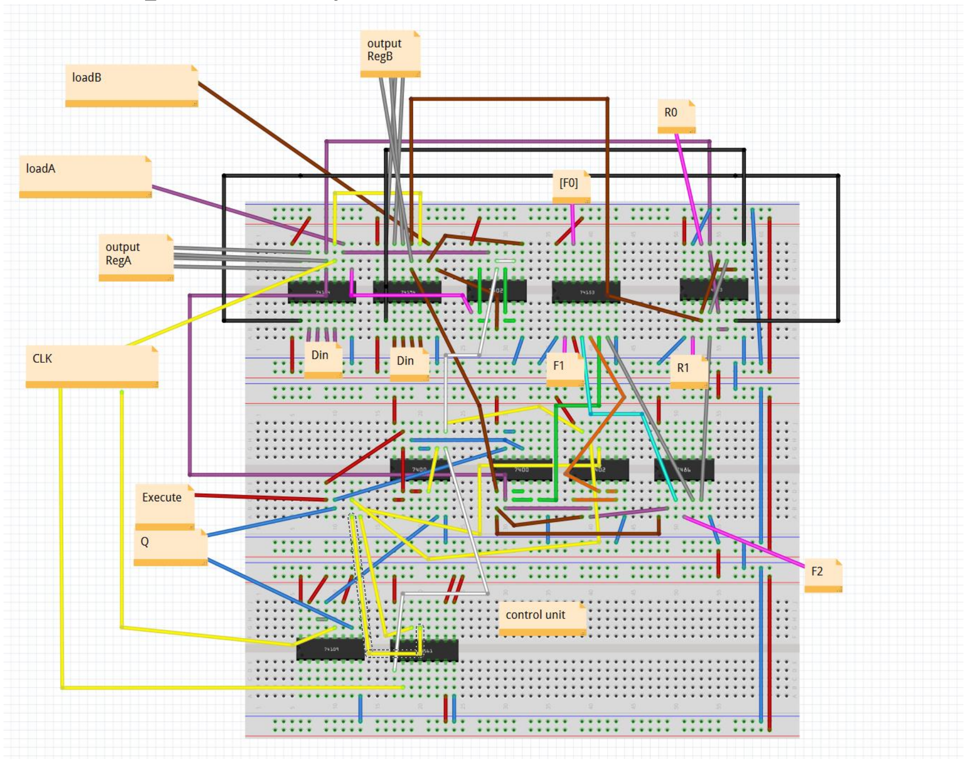

Figure 15 Breadboard Schematic from Fritzing

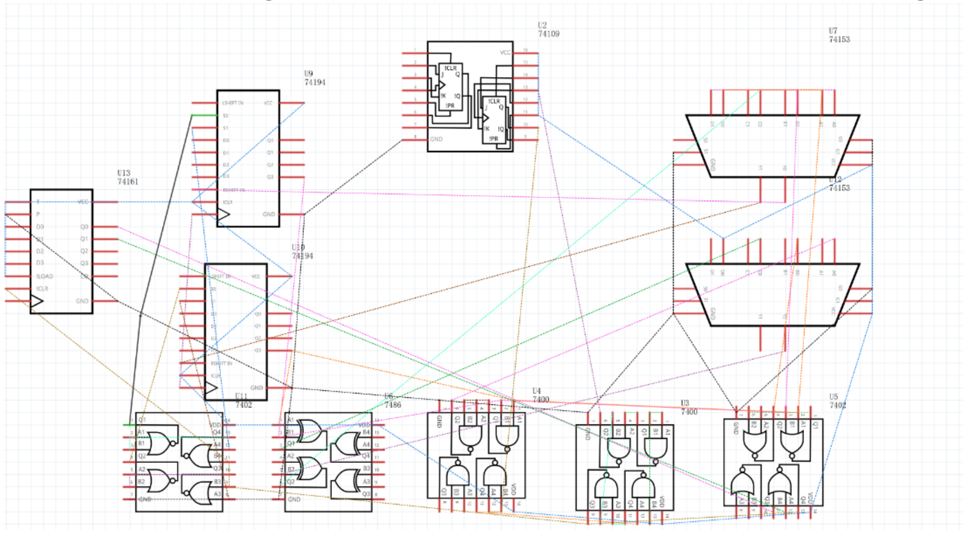

Figure 16 Processor Top-level Schematic

I use (74163) counter. But I didn't find it in the provided file. I use 74161 instead. I optimize the circuit by replacing NOR3 with NOR2.

5. The 8 bits logic processor on FPGA

6. RTL Block Diagram

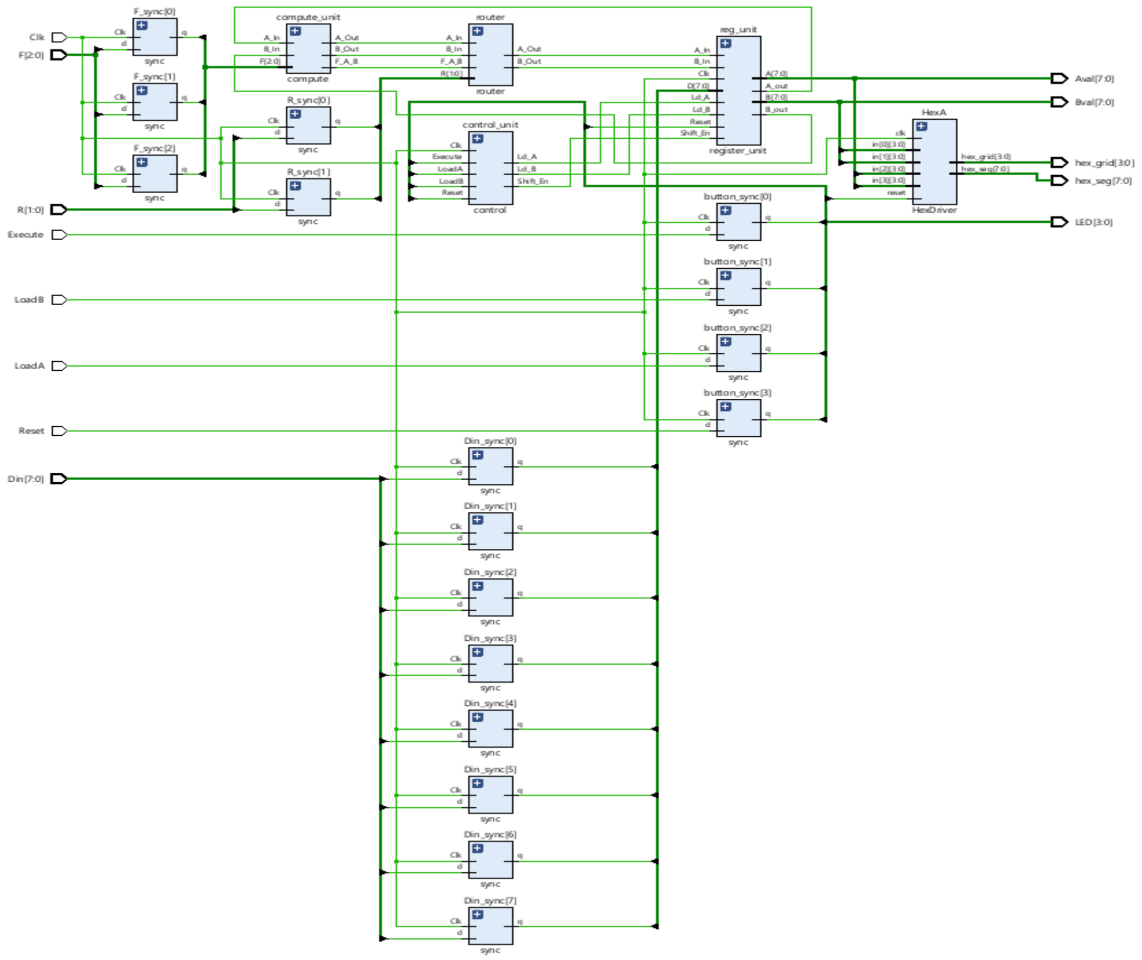

Figure 17 RTL diagram

7. Annotated Waveform

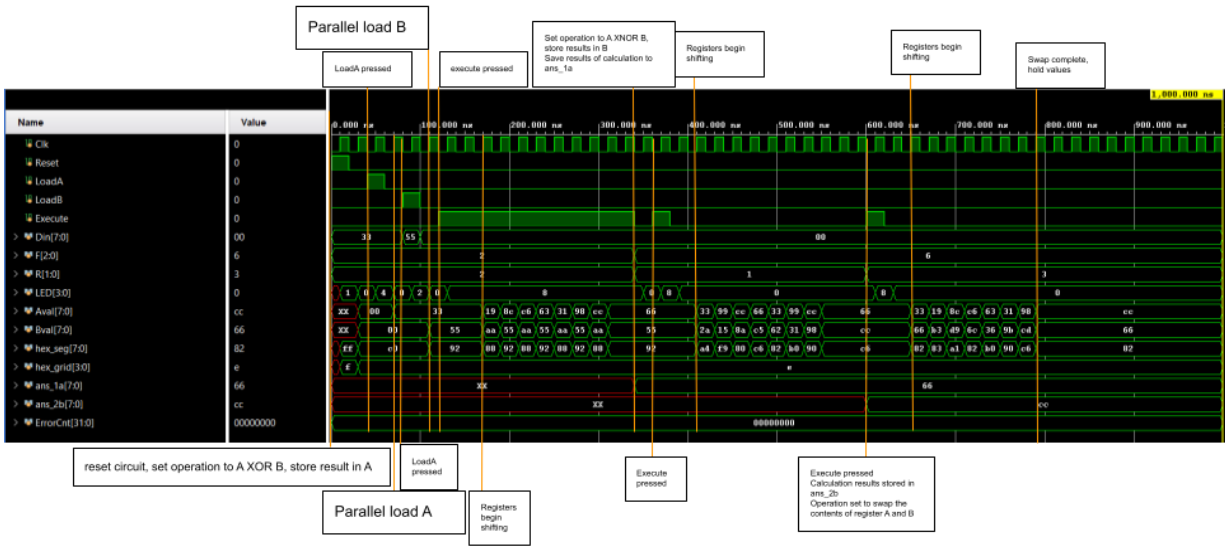

Figure 18 Simulation Waveform

8. Modules

Module: **RegisterUnit.sv**

Inputs: Clk, Reset, A_In, B_In, Ld_A, Ld_B, Shift_En, [7:0] D

Outputs: A_Out, B_Out, [7:0] A, [7:0] B

Description: Contains two register instances. It can control when to parallel load or shift. Ld_A, Ld_B are used to load. Shift_En is shift signal. A_Out and B_Out are the right shift data.

Purpose: It acts as a register unit and instance two register.

Modifications: I change the input D and the outputs A and B to 8-bit values ([8:0])

Module: **Synchronizers.sv**

Input: CLK, d

Output: q

Description: The module will assign d to q. The Module is synchronizers required for bringing asynchronous signals into the FPGA. They can output a clock-synchronized signal. The module includes 3 synchronizer modes which can have no reset, reset to 0 or reset to 1. It is instantiated in the Processor.sv. I use array to instantiate it.
Purpose: It is used to synchronize input signals.
Modifications: None**.**

Module: **Router.sv**

Inputs: [1:0] R, A_In, B_In, F_A_B

Outputs: A_Out, B_Out

Description: A_In is shift result of register A. B_In is shift result of register B. F_A_B is the computation result. With the control of two-bit R, the router assigns outputs A and B to either register A, register B or F (A, B).
Purpose: To assign shift output A, B, and computation result to desired destination. The module uses two always comb units to decide the output.
Modifications: None.

Module: **Compute.sv**

Inputs: [2:0] F, A_In, B_In

Outputs: A_Out, B_Out, F_A_B

Description: The inputs are 3-bit F to control the computation method. The module takes shifted bits from registers. The module uses an always_comb unit to decide which logic computation will use. A_Out and B_Out are the shift result of register A and B.
Purpose: It acts as computation unit
Modifications: None.

Module: **Control.sv**

Inputs: Clk, Reset, LoadA, LoadB, Execute

Outputs: Shift_En, Ld_A, Ld_B

Description: The module is the body of a state machine which have two always units. For an 8-bit processor, we have reset state, halt state and we need to shift 8 times. In that case, we have 10 states which need 5 bits to represent these states. For the Reset part, we need to go back to state A (first state) when Reset is high. Otherwise, it will go to the next state

automatically. It is like flip-flop in circuit which can update state. For state transition, since we have not listed all the possibilities, we need to set default to the next state. When <Execute> signal is high, it begins to update state and goes back to state A when <Execute> is low. Apart from that, the procedure will not stop, even if the <Execute> signal becomes low. For state A, register will load data and Shift is low. In the Shifting state, Shift is high. In the final state, Shift is low.
Purpose: It acts as a control unit. It implements the state machine.
Modifications: Changed the state number to 10 from A to J, to allow for 8 shift states instead of the 4 it had prior.

Module: **Hexdriver.sv**

Inputs: Clk, Reset, [3:0] in [4]

Outputs: [7:0] hex_seg, [3:0] hex_grid

Description: The module takes output bits of registers in the register unit. It will change 3-bits into Hex number and display on the FPGA. The input is array type (each 4-bits) which can take in 16 bits in total. It can convert upper 4bits and lower 4bits of register to each hex display respectively. Output is 8-bits hex_seg and 4-bits hex_grid. Hex_seg is used to display numbers in LED. Every LED has 7 lights. Number can be expressed in 7 bits in LED. For example, 0 can be expressed as 11000000. Hex_seg [7] is constant 1 in this case. Since FPGA has multiple HEX displays, hex_grid will instruct the board to use specific displays.
Purpose: The module is used to display Hex numbers on FPGA boards, allowing the user to see the contents of registers A and B.
Modifications: None

Module: **Processor.sv**

Inputs: Clk, Reset, LoadA, LoadB, Execute, [7:0] Din, [2:0]F, [1:0]R,

Outputs: [3:0] LED, [7:0] Aval, [7:0] Bval, [7:0] hex_seg, [3:0] hex_grid

Description: The module instantiates all of the necessary hardware to perform the actions specified by the switch bank, and display the results on the hex displays. It also connects the instantiated modules together, allowing

them to communicate and pass data in between each other to perform the function of an 8-bit logic unit.
Purpose: The module links all of the switch and clock inputs, instantiates internal logic and sub-modules, and links all of them together to read user input and display the results.
Modifications: We change output and input to 8bits. We changed hex driver inputs to A [7:4] A [3:0] B [7:4] B [3:0].

Module: **Reg_8.sv**
Inputs: Clk, Reset, Shift_In, Load, Shift_En, [7:0] D
Outputs: Shift_out, Data_out
Description: The module is a single register which can reset to 0 when the reset input is high. Besides, it can do parallel load and right shift. When Load is high, Data_out was assigned D. When Shift_In is high, Data_out is assigned to Shift_in and D [7:1]. Shift_Out is the right most bit.
Purpose: It acts as a single register. It will be instantiated in register unit.
Modifications: We changed the input D and Data_Out to 8 bits. Shift procedure changed to concatenate the 7 leftmost bits with the shift_in value.

9. Vivado Debug Core

Step1: set up debug core. We open synthesized design and find "set up debug" option. We add <Aval>, <Bval>, and <Execute>. (<Execute> has probe type data and trigger. Aval and Bval have probe type data). We set the sample of data depth to 1024.
Step2: Rerun synthesis, implementation, regenerate bitstream, then program the FPGA device.
Step3: Trigger <Execute> to both transitions.
Step4: Click on "Run trigger for this ILA core". The debug core is waiting for trigger.
Step5: I set Reg A to 33. Reg B to AA. I apply XOR computation and route output to Reg A.
Step6: Push the execute button and display the output.

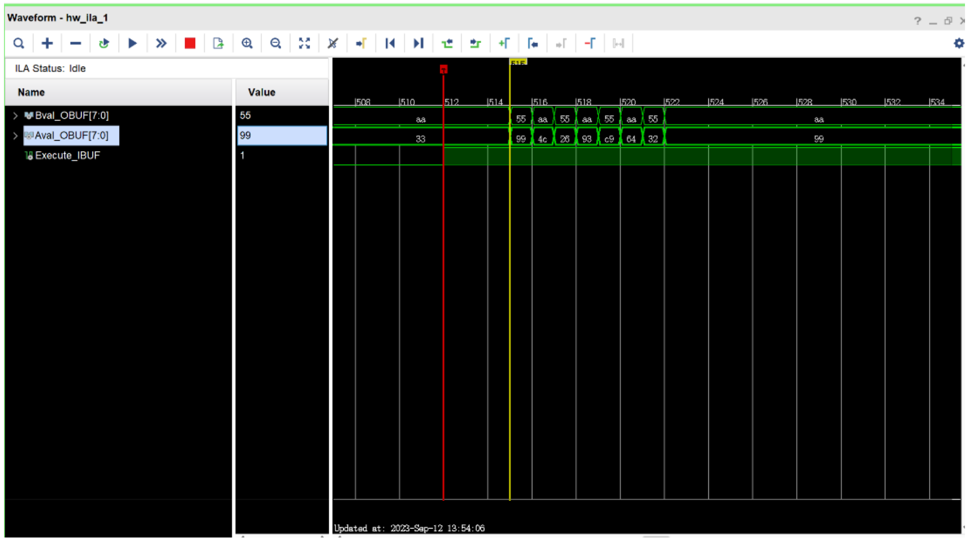

Figure 19 Vivado Debug Core

10. Answers to Post-Lab Questions

The simplest circuit that can optionally invert a signal consists of the two signals fed into an XOR gate. This is very useful in the lab because it allows us to eliminate the need of an 8:1 multiplexer and instead use a 4:1 multiplexer, since half of the logic operations are just inverted versions of their originals.

The modular approach to construction as suggested by the pre-lab helped greatly in debugging. Testing each module's behavior individually made it much easier to check whether it worked as intended and if it needed debugging, before incorporating it into the overall processor. Without doing that, determining which module was problematic would prove quite the exercise in frustration. This also extends to testing individual chips- if the module does not work, testing and ensuring all chips are functional and making sure they are connected properly saves a lot of time. By cutting out much debugging time, that speeds along development massively.

Eitan settled on the Mealy machine, due to requiring less states and thus being simpler to implement. The Moore machine requires three states, which would necessitate the addition of a second register to store the state. Thus, the Mealy machine was chosen to cut down on register counts. A Mealy machine typically has a lower state count than the Moore machine, though a

Mealy machine has to account for more- outputs depending on state and input- versus the Moore machine, which just depends on current state.

vSim cores are very, very useful for checking functionality of the hardware before synthesis and implementation. Because it makes so many assumptions about the behavior of the hardware, it is not a good check for the actual behavior of the hardware on the FPGA. It is a fantastic check for making sure your hardware's logic is correct and outputting the correct values, however. Debug cores are physical cores on the chip tapped into the nets specified, which makes them great for monitoring the actual behavior of the hardware, including timing.

11. Bugs

Eitan had a problem with his register loading, where it would not load correctly. Unfortunately, since it was built into the entire processor already, the issue was thought to be part of the shift control logic due to the registers constantly shifting right. With module unit testing, it was determined that the load logic was overcomplicated and causing problems with constant shifting, and after rebuilding worked perfectly. This is an example of learning the hard way that modular testing would have saved much time and headache during debugging.

Haoyu had a problem with register loading. At first, I use a mechanical switch which will bounce. I use the bug by using debounce switch in FPGA. I also design debounce switch by connecting ground with switch through a resistance.

12. Conclusion

On the breadboard, we designed and implemented a 4-bit logic unit with an emphasis on a modular design and unit testing to ensure functionality. This clearly demonstrated the advantages of such a process. Then, we extended a SystemVerilog implementation of the 4-bit logic unit to 8 bits. This served both as an introduction to the main hardware design language of the class, and as a demonstration of the power of HDLs.