ECE 385
Spring 2023
Lab 2

A Logic Processor

Gustavo Fonseca, Hunter Baisden
TA: Hongshuo Zhang

1. **Introduction**

   The purpose of the circuit that we developed is to act like a bit-serial operation processor. This processor can perform 8 operations: AND, OR, XOR, 1, NAND, NOR, XNOR, and 0. The processor we developed on our breadboard operates on 4 bits, while the one we developed on the FPGA operates on 8 bits. We also developed a routing element, so the processor is capable of storing the operation output in either register or swap values of the registers.

2. **Operation of the logic processor**

   a. **4-Bit Processor (Breadboard)**

      In order for the user to load data in the A and B registers on the breadboard design, they must change the 4 switches that are connected to input of both registers to desired values for register A. Then they must turn the "Load A" switch to high, this will load all the values set before into register A. After turning off "Load A", change the 4 input switches to the desired values for B. The user must then turn on the "Load B" switch to load all the values into register B. Before the user can execute the desired operation, they must set the correct computation and routing for the operation (details on which combination of switches does what is described in written description of circuit). After these switches are set, the user can flip the execute switch to high, making sure that "Load A" and "Load B" are both set to low beforehand. The user may switch the execute switch to low during the operation, as this will not affect the processor. To run the system again the user simply needs to switch the execute from low to high.

   b. **8-Bit Processor (FPGA)**

      Loading data in the A and B register is very similar to the 4-bit design. Using the switches on the FPGA we used switches SW0 to SW7 to control the input bits into both registers. The switches SW8 and SW9 act like "Load A" and "Load B" respectively. After setting SW0 to SW7 to desired values for register A turn SW8 to high, loading them in the first register, doing the same steps for SW9 and register B.

      Since the FPGA we used only has 10 switches and 2 buttons, we cannot control computation and routing options as 11 of these options are used already (a total of 5 more switches would be required for us to control these options). Because of this limitation, we hardcoded which computation and routing operation would be performed before programming the FPGA. Thus, if the user wanted to change these values, they would have to edit the System Verilog code to reflect the changes, and then reprogram the FPGA. Finally, for the user to execute the operation they must press the "Execute" button, KEY1. Again, make sure both the "Load A" and "Load B" switches are off as this would affect the operation execution.

3. Block Diagram

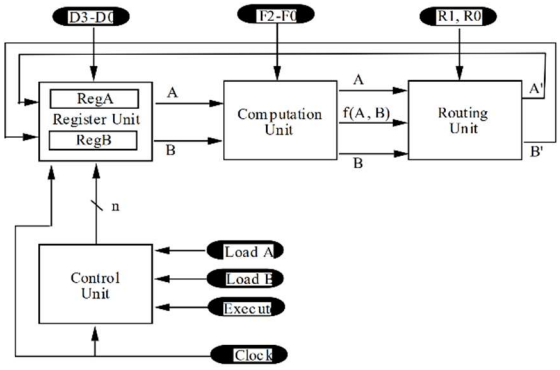


Figure 1: 4-Bit Block Diagram

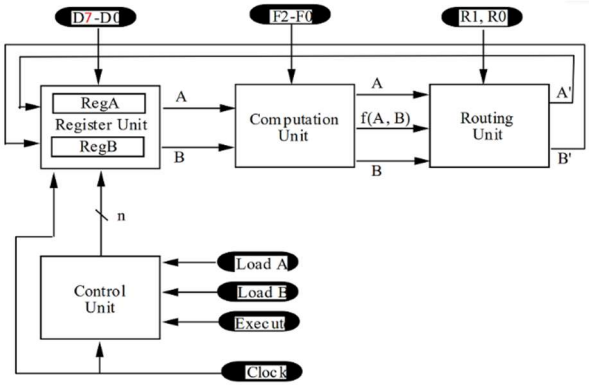


Figure 2: 8-Bit Block Diagram

4. Written Description

a. Register Unit

The purpose of the Register Unit is to store the values that go through the processor. The register unit is comprised of 2 registers, labeled as RegA and RegB. Both registers are controlled by the Control Unit. The serial input of both these registers are given by the output of the Routing Unit. The output for this unit is the most significant bit (MSB) of both registers.

b. Computation Unit

The purpose of the Computation Unit is to perform desired combinational logic for the processor. The inputs for Unit are the most significant bit from RegA and RegB, and function selection inputs, F2, F1, and F0. Figure 3 shows which combinational logic is performed based on the function selection inputs. The outputs for this unit are the most significant bit from RegA and RegB, and the

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: Computation Unit Function

function output.

c. **Routing Unit**
The Routing Unit takes all the outputs from the Computation Unit (MSB of RegA and RegB, and function output, as inputs. The unit also takes the routing selection inputs R1, R0, as inputs. Based on R1 and R0, the unit's output is changed. Figure 4 shows the different ways that output is changed by R1 and R0 The outputs for this unit, "new A" and "new B" are connected to the least significant bit 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 4: Router Unit Function

d. Control Unit
The Control Unit's main purpose is handling the 3 inputs of "Load A", "Load B", and "Execute", and count to 4/8 clock cycles after "Execute" is pressed. When either "Load A" or "Load B" is pressed the Control Unit signals to registers to parallel load the inputs for switches D[3:0] for 4-bit design, and D[7:0] for 8-bit design, into either Register A or Register B depending on which Load is turned on. When "Execute" goes from low to high the Control Unit then signals to both registers to start shifting right, sending the MSB to the Computational Unit. The Control Unit will give this signal until the internal counter counts 4 clock cycles for the 4-bit design, and 8 for the 8-bit design. After this happens the Control Unit will signal Registers to hold their values until "Execute" goes from low to high again.

5. **State Machine Diagram**
   We used the Mealy State Machine.

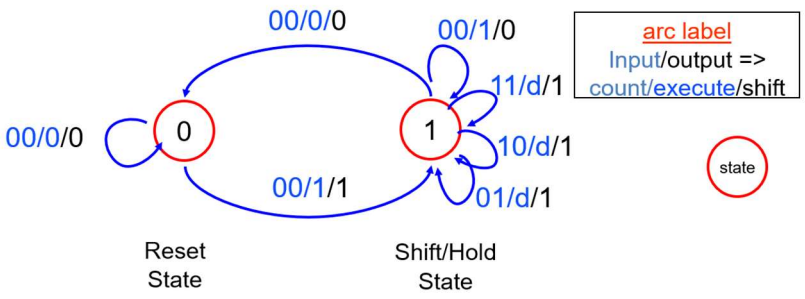


Figure 5: Mealy State Machine

6. **Design Procedure**
   a. **Register Unit**
      In our implementation of the Register Unit we used 2 74194 4-Bit Bi-Directional Shift Register ICs. We connected the control inputs (S[0:1]) for both registers to the output of the Control Unit that controls the Operating Mode of the registers. We then connected the DSR pin to the output of the routing unit, we did this as we chose to right shift the registers, thus the input in the LSB of the register is decided by the value of DSR. When considering other methods of implementation, we considered using the other register IC given to us, 74195 4-Bit Parallel Access Shift Register. However, this register is clearly inferior for this implementation. The main reason for this Register not being suitable is that there is no pin that dictates the new value of the LSB after the right shift: its value is controlled by the operating mode and not a specific pin. Thus, this would add unnecessary complexity to our system as we would need to add extra logic into our circuit to account for having to changing operating modes based on the incoming values.
   b. **Computational Unit**
      In our implementation of the Computational Unit we used 5 IC chips. These included: 7400 (NAND), 7402 (NOR), 7404 (NOT), 7486 (XOR), and 74151 (8:1 MUX). In our design we wired the F[2:0] inputs into the S[0:2] inputs of the MUX. We also wired the input's (A and B) to the inputs of a NAND gate, a NOR gate, and a XOR gate. This created 3 out of the 8 computational options for the processor. We then wired these outputs of these signals to the correct X inputs of the 8:1 Mux and the input of the NOT gate. This created the AND, OR, and XNOR logic that we needed. We then wired these outputs to the correct X inputs of 8:1 Mux. Finally, we wired the last 2 remaining X inputs to the low and high

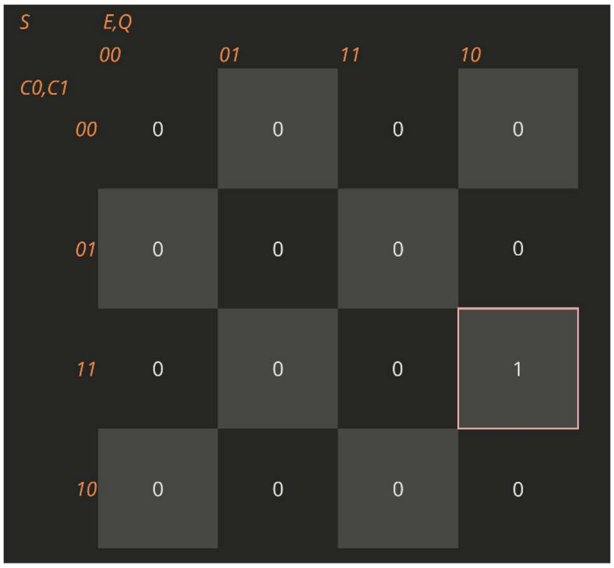
rails respectively. When examining other methods of implementation, we realized that we could have used a 4:1 MUX instead as X[4:7] are the inverse of X[0:3]. Thus, one could use a NAND gate with the output of the 4:1 MUX and F2 as the inputs. While this decreases the size of the MUX, this would not decrease the number of IC chips we would need for the implementation. Additionally, in this method we would need to make AND and OR, but we only have NAND and NOR gates. We would need to make AND with NAND and a NOT, and OR with NOR and a NOT. Taking this into account, this method adds unnecessary complexity as we already have the NAND and NOR outputs to make AND and OR. For this reason, we decided to stay with the 8:1 MUX design.

c. **Routing Unit**

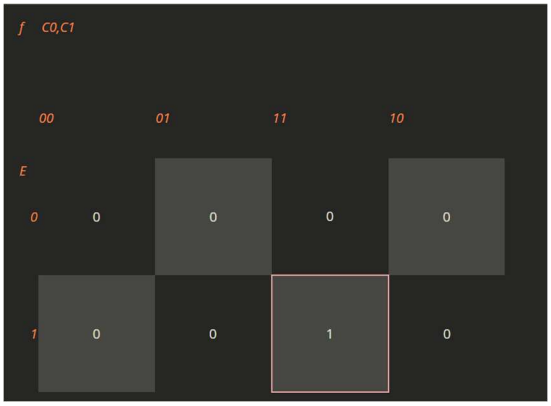
For the design of the Routing Unit, we only had to use one IC chip, the 74153 Dual 4:1 MUX. With this chip having 2 4:1 MUXes we could implement the output for both registers on this one chip. We first connected R[0:1] to the selection inputs S1 and S0. We then wired the outputs from the Computational Unit, A, B, and F into their appropriate inputs based on the desired routing output we wanted. We then connected the outputs to both registers' DSR pin to put the new bit into each register. When it came to other design ideas, there was not much to consider. This was a simple unit as there was no computation or storage needed. One issue that could have occurred with only using one chip is that you cannot control the output to Register A and Register B independently. However, this was not an issue as our design requirements did not require us to implement this feature.

d. **Control Unit**

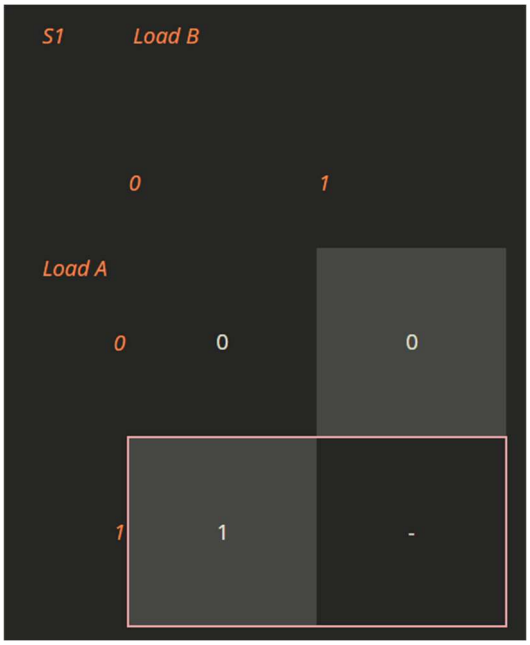
For the design of the Control Unit, we used 4 IC chips: 74161 4-Bit Counter, 7427 Triple Input NOT Gate, 7402 (NOR), and 7474 Positive Edge Triggered D Flip Flop. Our goal of the Control Unit was to have the counter begin counting to 4 after the execute switch goes to high. When this happens, the registers within the Register Unit would get into the Shift Right operating mode. Once the counter hits 4, the counting will stop and not reinitiate until the execute switch goes from low to high again. With this in mind, we decided to create a new value "Q" that would stop the counter from continuing.



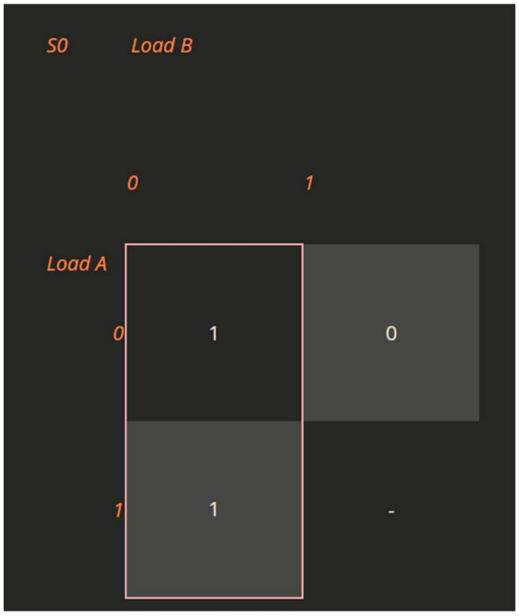
In this K-Map, C0 and C1 refer to the counter bits, E represents Execute, with the output, S, representing the shift state for the register. This K-Map results in the expression: S = C0 + C1 + E + Q'.
With the addition of this variable Q, we would need the next state logic for Q to go high once the counter went through 4 cycles and execute was pressed. Because we need the next state logic for the value of Q, we used a D Latch Flip Flop to store the value of Q.



With this K-map, we get that Q^+ = C0 + C1 + E. These two logic statements create the counter that we desire.
Finally, we had to implement the "Load A" and "Load B" commands. With these two commands, there were two things that were important: when both commands were at a logic low, then both registers would be at an operating state of Shift Right; when LoadA/ LoadB was pressed then Register A/B would be in the Parallel Load State while the other was in the Hold State. Since there were two bits, S0 and S1, controlling these states for both registers, we had to have two K-maps for S0 and S1. The K-maps for S0 and S1 for Register A are as follows.



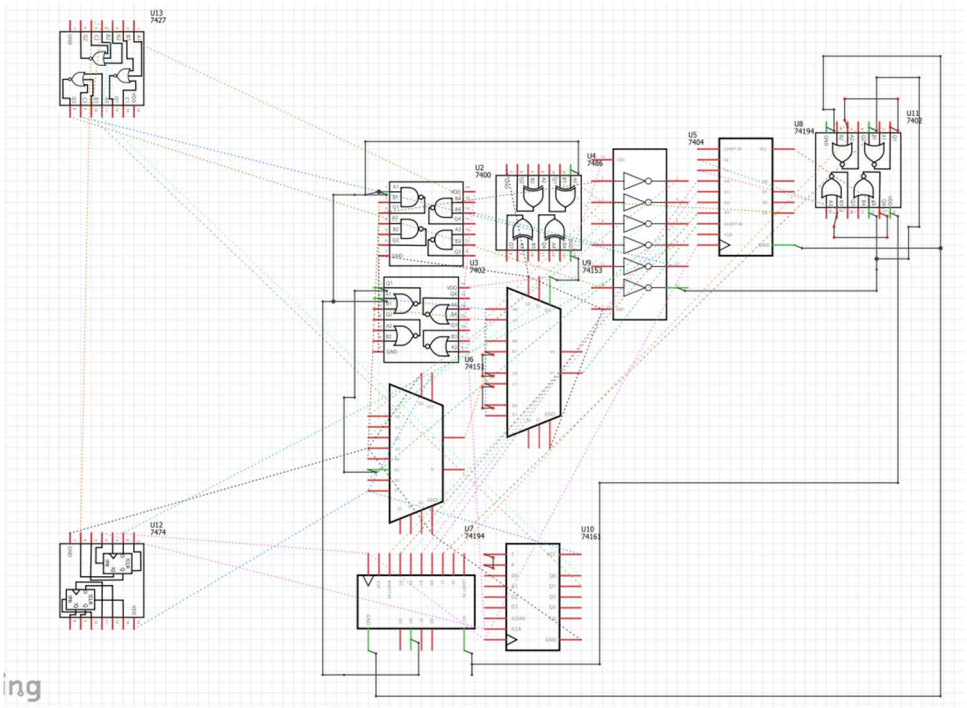
Thus, this would make S1 = Load A.



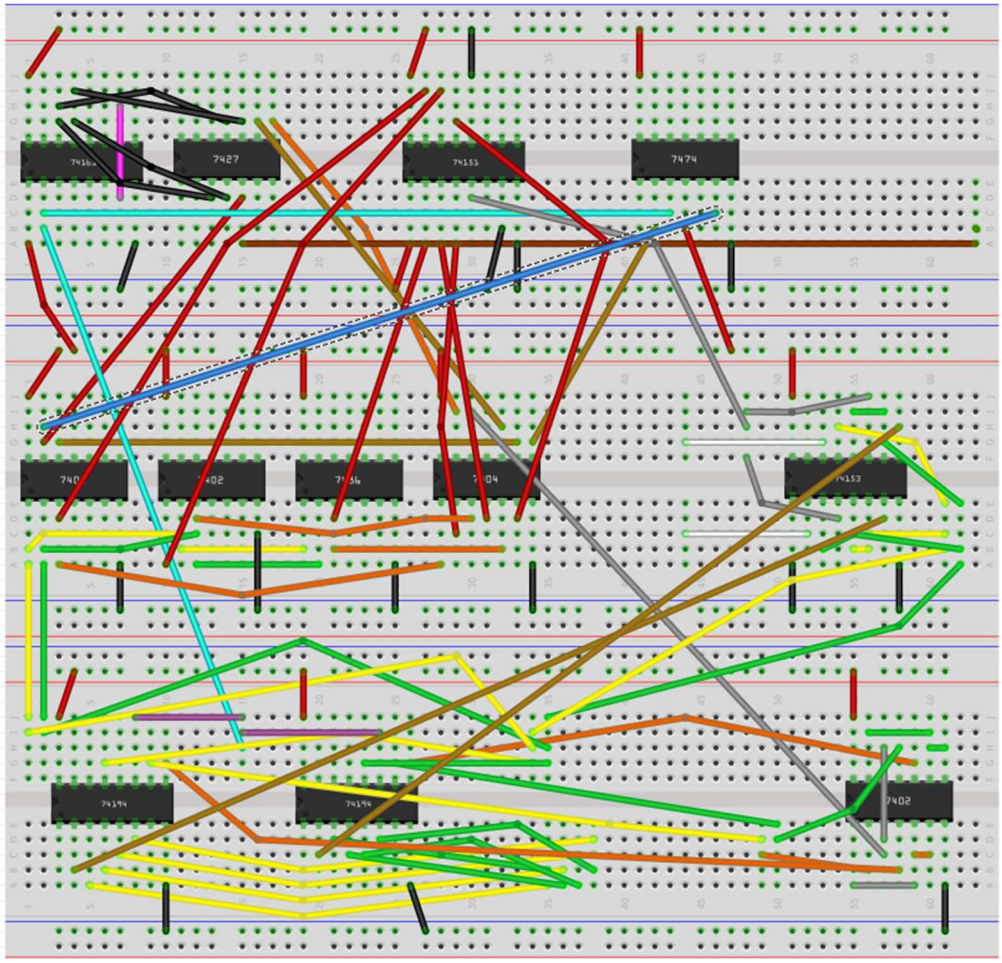
This K-Map makes S0 = (Load B)'.
Since each register needs to act in reverse logic from each other, Register B's S0 and S1 are as follows: S0 = Load B and S1 = (Load A)'.
In terms of other design considerations, the idea of using the 74109 J K Flip Flop came to mind. However, we found that the a D Latch would be better suited for the simple next state logic that we had to implement for the counter. We also considered having separate switches for the inputs of both registers; however, this obviously would almost double the amount of switches on the board and would

make us fail our design requirements. For these reasons, we decided to only use one set of input switches for the registers and implement the simple combinatorial logic required.

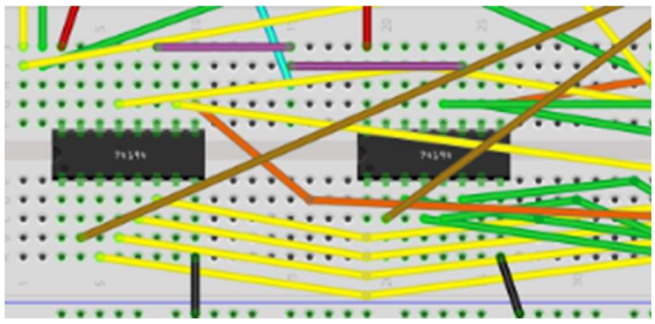
7. **Circuit Schematic**



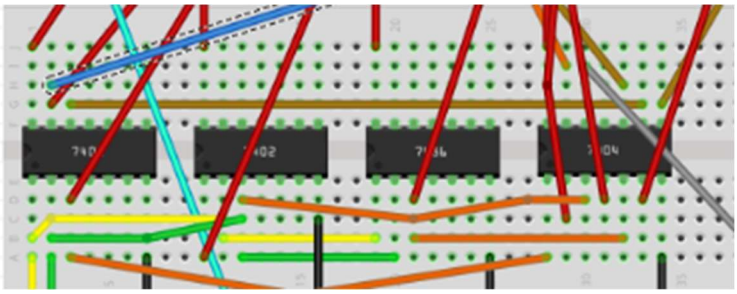
8. **Breadboard View**



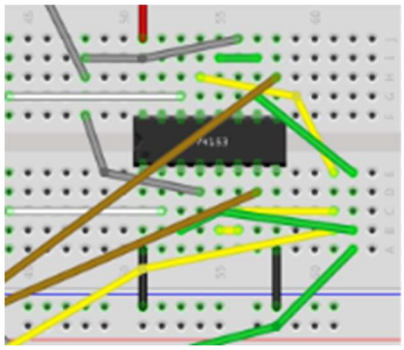
Breaking this down further we have the register unit:



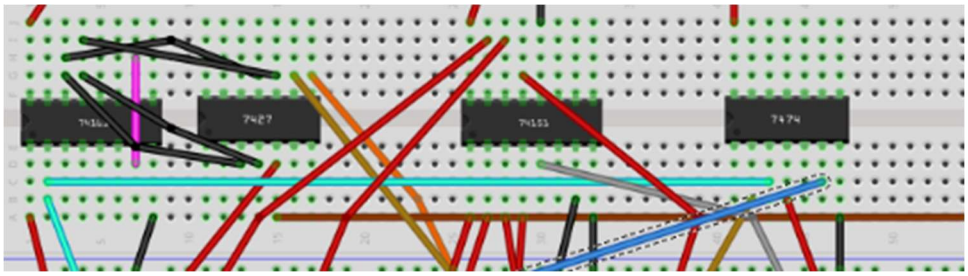
The computation unit:



The routing unit:



And the control unit:



9. **.SV Modules**
    a. The first module that needed to change was the top-level module, Processor.sv. Within this module we extended the input logic and output logic to 8 bits instead of 4 bits. This gave a top-level module as:

```
module Processor (input logic   Clk,     // Internal
                                Reset,   // Push button 0
                                LoadA,   // Push button 1
                                LoadB,   // Push button 2
                                Execute, // Push button 3
                  input  logic [7:0]  Din,     // input data
                  //Hint for SignalTap, you want to comment out the following 2 lines to hardwire
values for F and R
                  //input  logic [2:0]  F,       // Function select
                  //input  logic [1:0]  R,       // Routing select
                  output logic [3:0]  LED,     // DEBUG
                  output logic [7:0]  Aval,    // DEBUG
                                Bval,    // DEBUG
                  output logic [6:0]  AhexL,
                                AhexU,
                                BhexL,
                                BhexU);
```

    b. We also extended the Din_sync to 8 bits within the top-level module
    c. The compute module stayed the same, as everything was done bitwise

d. The register unit inputs and outputs were extended to 8 bits:

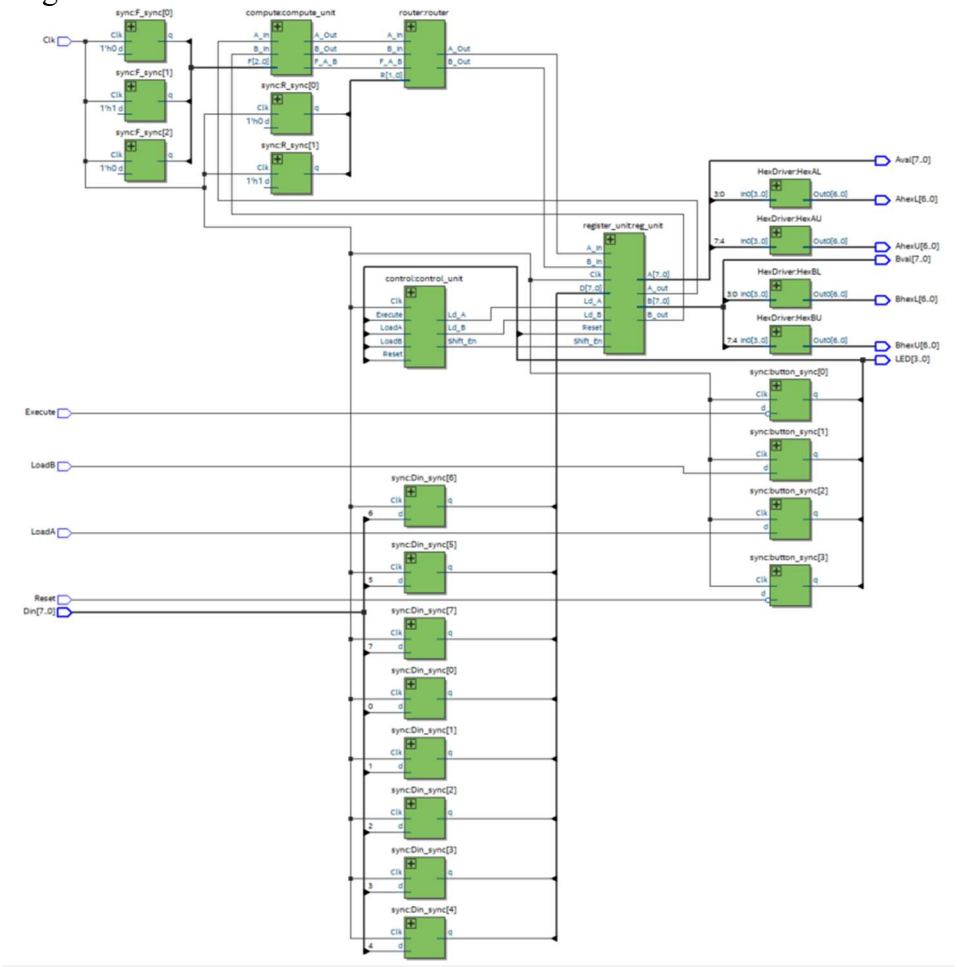
```
module register_unit (input  logic Clk, Reset, A_In, B_In, Ld_A, Ld_B,
                                   Shift_En,
                      input  logic [7:0]  D,
                      output logic A_out, B_out,
                      output logic [7:0]  A,
                      output logic [7:0]  B);
```

e. The control unit required the addition of the G, H, I, and J states for the processor to run through. These were added like below:

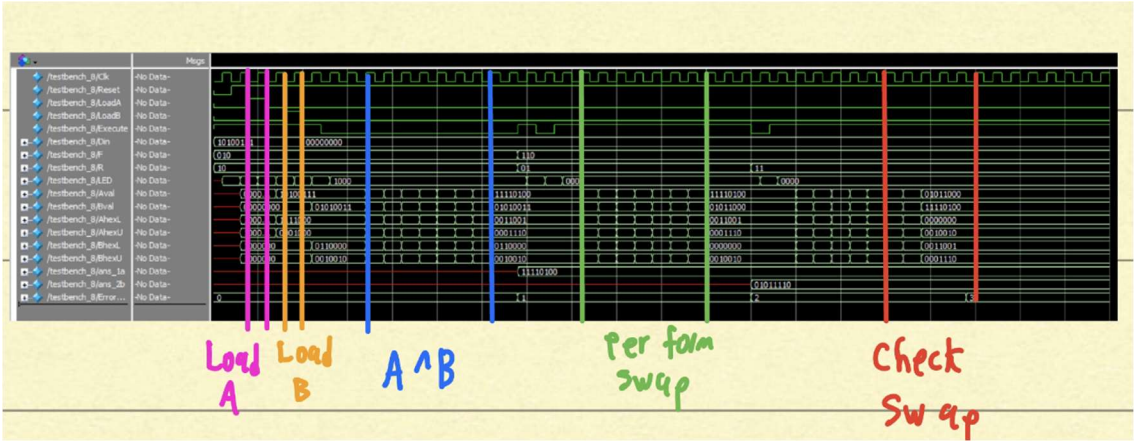
```
enum logic [3:0] {A, B, C, D, E, F, G, H, I, J}   curr_state, next_state;

    //updates flip flop, current state is the only one
    always_ff @ (posedge Clk)
    begin
        if (Reset)
            curr_state <= A;
        else
            curr_state <= next_state;
    end

    // Assign outputs based on state
    always_comb
    begin

          next_state  = curr_state;    //required because I haven't enumerated all possibilities
below
        unique case (curr_state)

            A :    if (Execute)
                       next_state = B;
            B :    next_state = C;
            C :    next_state = D;
            D :    next_state = E;
            E :    next_state = F;
                F :    next_state = G;
                G :    next_state = H;
                H :    next_state = I;
                I :    next_state = J;
            J :    if (~Execute)
                       next_state = A;

        endcase
```

f. Finally, to test everything we edited the testbench file. All the inputs and outputs were extended to 8 bits.

10. **RTL Block Diagram**
    a. To generate a RTL block diagram of the circuit, we used Quartus and going through Tools > Net List Viewers > RTL Viewer we were able to generate the diagram

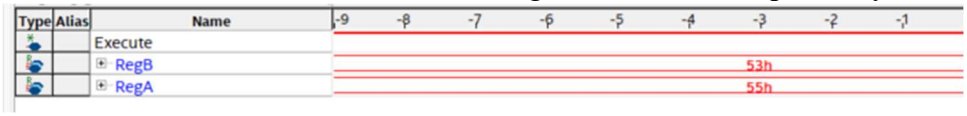


11. **Simulations**
    a. To run simulations on the 8-bit processor we ran the testbench file in ModelSim-Altera. This generated a waveform which can be seen below



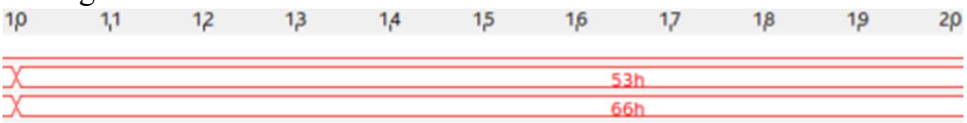
b. As labeled above, the processor first loads A and B according to the test bench, then performs A XOR B in a bitwise fashion, and then swaps the destination registers according to the routing unit

12. **SignalTap**
    a. In order to perform a Signal tap on the processor, a few values had to be hard coded into the FPGA. For more information, about this see the operation of the 8-bit processor at the top of this report. After hard coding F to 3'b010 (the 3 function selection bits) and R 2'b10 (the 2 routing bits) we were able to run a signal tap on the FPGA.
    b. First, we loaded 8'h53 and 8'h55 into Registers B and A respectively



    c. Then once execute fell (the FPGA is active low) the FPGA performed the XOR operation

    d. And finally, the FPGA stored the values back in the registers according to the routing bits set earlier where 8'h66 is 8'h53 XOR 8'h55



    e.

13. **Bugs**
    a. In the physical circuit we found once the circuit was fully built that we were experiencing unpredictable behavior in the control unit. In order to solve this, we systematically reviewed the control unit until we found a floating input on the flip flop, and after fixing this we had a working control unit
    b. During the demo of the 8-bit processor, in attempting to load 8'h53 and 8'h33 into Registers B and A respectively, the loading of register A was off, as whenever we entered 8'b00110011 into the onboard switches, it would load 8'h55 into register A. We found out that this bug was related to us mixing up the pin assignments in the assignment editor for the switch pins. After fixing the assignments the loading worked as intended.

14. **Conclusion**
    a. **Summary**
        i. This lab first implemented 4-bit processor on a breadboard using a register unit, computation unit, routing unit, and control unit. This processor was then implemented in the FPGA and extended to 8 bits on the FPGA.
    b. **Post-Lab Questions**
        i. Optional inverter
        We can either use a NAND gate or NOR gate to invert the desired input.

For a NAND gate if we connect the signal to one input and a logic high on the other, then the system acts like an inverter to the signal. For a NOR gate, by connecting one input to the signal and the other to logic low we can have the gate act like an inverter. This is useful for this lab because we can conserve the amount of Inverter ICs and just use a NAND or NOR gate on a chip we are already using.

ii. Modular Design

A modular design improves testability because we can isolate each module and test them separately, simplifying the debugging process. This simplification cuts down on development time as we spent less time debugging the entire circuit. Additionally, breaking the system into modules makes it easier to conceptualize the entire complicated circuit, allowing us to connect each module together easier.

iii. Designing the State Machine

When designing our state machine, we had to consider the three states that could occur: Reset, Shift, and Halt. When we developed the Moore machine, we could not consider the value of the output, in this case Q, thus we had to have 3 different states in the state machine. When we used the Mealy machine, we can use the value of the output to determine the next state, this allows us to consolidate the Shift and Halt states together. This consolidation allows us to decrease the complexity of the truth tables; however, we need to add a flip flop IC chip to store the value Q to be able to use it in computational logic. After taking these factors into account we decided to use the Mealy machine for decreased complexity.

iv. Differences Between ModelSim and SignalTap

The difference between ModelSim and SignalTap is that ModelSim uses the given TestBench file to generate the waveform, while the SignalTap takes the values generated by the FPGA. In situations where we have more than the 12 inputs that FPGA has, ModelSim would be preferred as we can control a nearly unlimited number of inputs that can be controlled with the TestBench. SignalTap is useful when we want to test the functionally of the FPGA, this is important because there are cases where our SystemVerilog code passed our TestBench tests but has an issue on the FPGA itself.