**ECE 385**

Spring 2021

Experiment 2

**A Logic Processor**

Mengyan Liu(mengyan4)

Angela Jiang (angelaj4)

Section ABD

TA: Andrew Gacek

**Introduction**

This lab serves as an introduction to Quartus and SystemVerilog. The purpose of the first part of lab 2 is to create a bit-serial logic operation processor in the real world using two 4-bit shift registers, several multiplexers, and a synchronous counter in our circuit. This circuit is a 4-bit logic processor that can perform eight different operations to the binary values loaded in register A and B. The values produced by the processor can be displayed depending on the router inputs.

The second part of lab 2 is to extend the 4-bit logic processor to 8-bits using SystemVerilog. The program uses several files that are intricately connected to replicate the real world circuit we built in the first week. Other than extending the processor, the program also uses the FPGA board’s switches and LED lights instead which is neater and more convenient to navigate than the breadboard circuit.

**Operation of the Logic Processor**

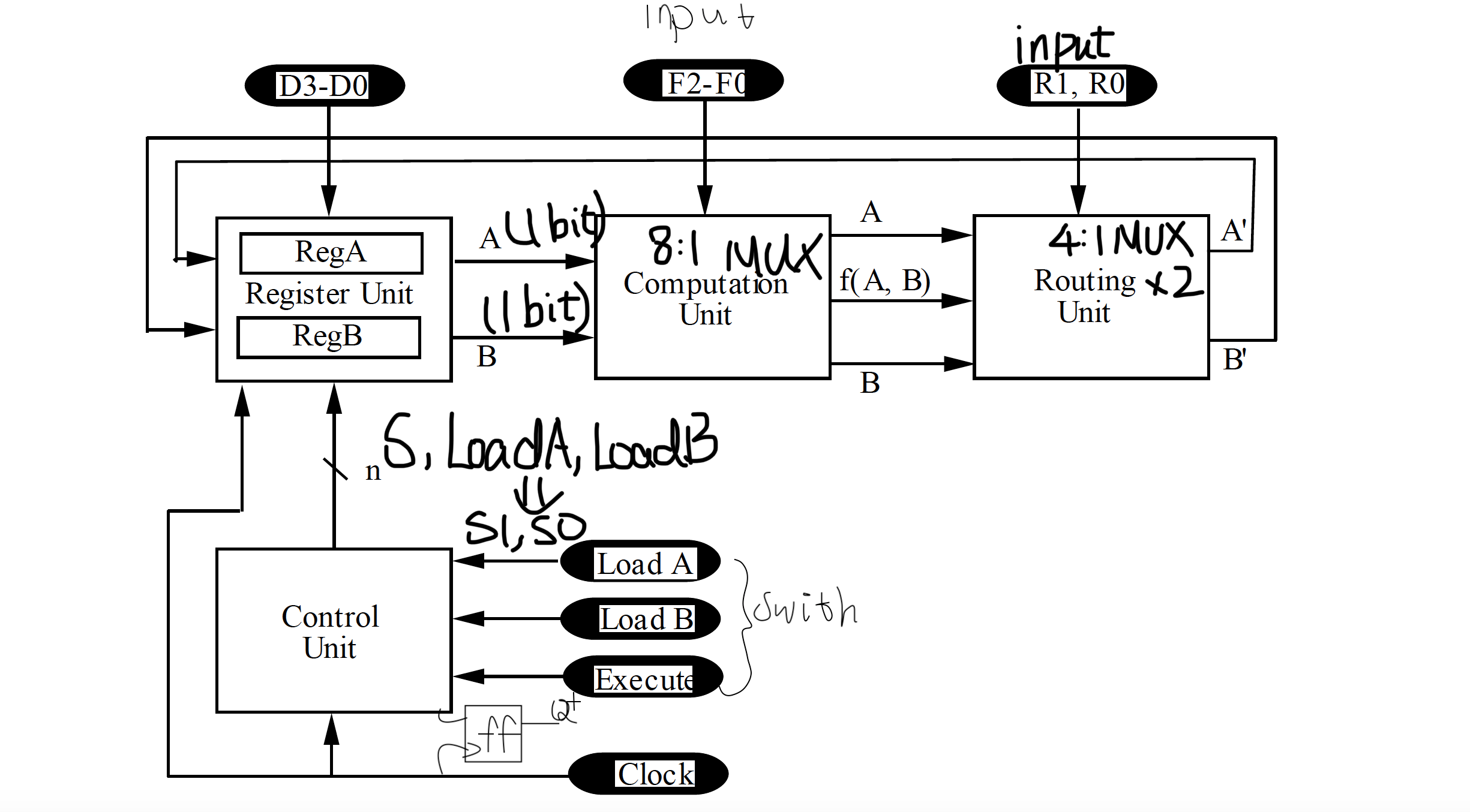
In the real world circuit, we first set the two different/same initial 4-bits input for register A and register B using eight switches. After that, set LoadA and LoadB from 0 to 1 in order for the shift register to parallel load. If the 4-bits outputs of both shift registers are connected to the LEDs, it is now easy to see that the LED output matches the designated 4-bits inputs. After setting up the shift registers to parallel load, the switches controlling LoadA and LoadB can be open in order to not affect the control unit. Next, we set the 3-bits computation and 2-bits router to their desired operation. Finally, we can flip the execute switch in order for the whole circuit to run through four cycles.

The FPGA board works in a similar way. Using the suggested pin assignment, we first flip the LoadA switch to 1 in order to load the designated input bit. Then, we can set the values of Reg A using the eight switches counting from the right. For the output, we can see two Hex Drivers, also visible as LEDs, which separate the upper and lower four bits of the 8-bit input. After flipping the LoadA switch back to zero, the same process applies to loading the register B. First flip the LoadB switch to 1 and then change the 8 switches and see the other two LEDs to justify the input. Once the values are set we can hit execute to see the results.

**Written Description, Block Diagram, and State Machine Diagram**

In this lab, there are a total of four blocks for the processor to function as desired. They are the register unit, computational unit, routing unit, and control unit. The register unit uses two 4-bit shift registers to load two binary numbers, A and B, which are used to perform logically. After loading the two binary numbers in 4 bits, the computational unit is used to choose one of the eight logic operations and perform the logic operation on the two binary numbers A, B using 3 bits input. In this unit, since it requires eight operations and 3 bits can represent eight different choices, the input is chosen to be 3 bits. Next, the routing unit uses input 2-bit routing numbers to choose which to store back in register A and which to store back in register B. There are four combinations to choose from for this unit. Finally, the control unit is designed to store each of 4 bits back to the two registers one by one.

High-level Block Diagram:



For the four blocks diagram, some notes and explanations are added from debugging the circuit. First, 8:1 MUX is used to select one of the eight logic operations since we think it’s easier to debug if only one MUX is used. Next, two 4:1 MUXes are used separately for register A and register B. The ‘input’ sign is noted for F2-F0 and R1-R0 since it indicates that these five inputs are determined by switches. For the control unit, the output S and switches LoadA and LoadB are used to create a truth table, in order to determine the inputs, S1 and S0, for shift register 194. According to the datasheet for 194 shift register, S1 and S0 determine whether the register performs parallel load, shift right, or hold. Additionally, a flip-flop is added for the current state in the mealy machine.

State Machine Diagram:

For the control unit, I used a mealy machine that not only depends on the current state but also depends on two counter bits input.



The format of transition used for this mealy machine is C1C0/E, where C1, C0 stand for the second least significant bit and the least significant bit of the counter outputs, and E stands for ‘execute’ input. The execute input is determined by a switch and the two counter bits are generated by a counter, which is operated upon the ‘execute’ input.

Each state is followed by a binary number indicated by a binary number, Q. Q represents the current state latched by a flip-flop in a real circuit. In the ‘Shift’ state, the control unit supposes to send signals S to determine whether the shift register should be parallel loading for shifting. Since the minterm for S is C1 + C0 + EQ’, the S should be ‘0’ when the shift registers are parallel loading and the S should be ‘1’ when the shift registers are shifting. In the beginning, when the execute has to be ‘1’ to trigger the shifting, the S must turn to ‘1’ for the register to shift. Thus, the current state should stay at ‘0’ for a clock cycle and then turn to ‘1’ simultaneously when the counter counts to ‘01’ in order to let the state machine operate for four cycles. In this way, the shift register shifts for 4 cycles fluently using a flip flop to latch the current state Q.

**Design Steps Taken and detailed Circuit Schematic Diagram**

The first unit that’s being considered is the register unit since, in this unit, it loads the very first 4-bits to the registers. Thus, eight switches are used to decide the 2 4-bits for 2 registers, A and B. Then, using the notes from the lecture, we used the synchronous register 194 with all three states used: parallel load, shift right/left and hold. The next unit for the processor that was easy for us to understand is the computational unit. Initially, we intended to implement the six logic operations using NAND, NOR, XOR gate, and an 8-1 MUX to select the correct logic operation based on F2-F0. For each operation, a diagram including gates and wires are drawn in order to correspond to the truth table. For example, for AND gate, we draw the gate-level diagram as follows:

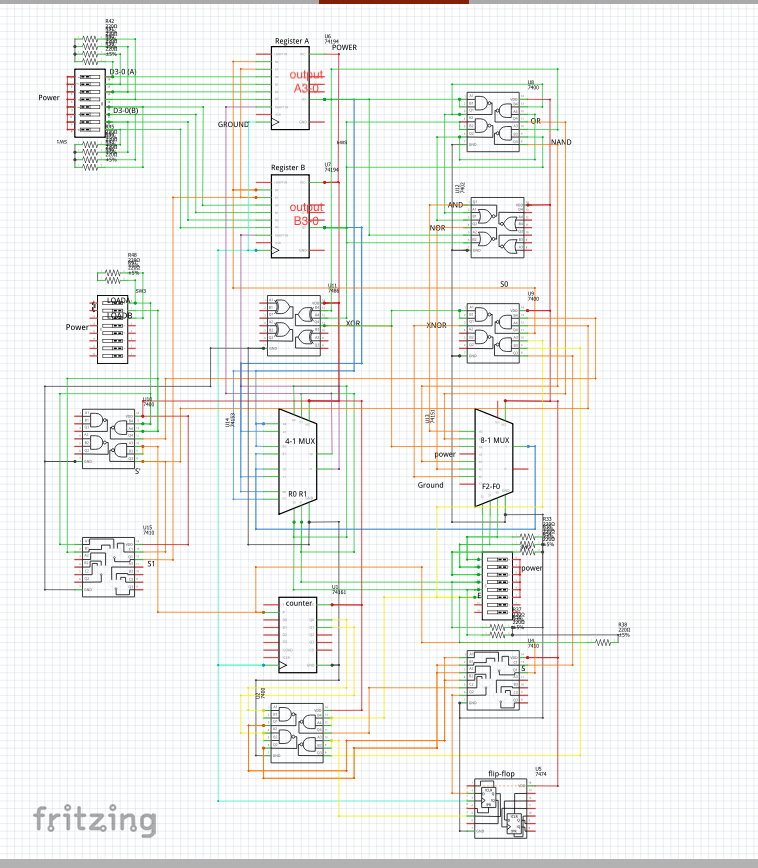
Then, after designing the six logic operations for the computational unit, it is discovered that the first four operations are exactly the inverse of the last four operations correspondingly. From this, we realized that there’s another way to build the computational unit. Using this method, we can use a 4-1 MUX and a 2-1 MUX to replace the 8-1 MUX for the computation unit. The F1 and F0 can be the selection bits for 4-1 MUX to select one of the four operations neglecting if it’s inverted. The F2 selection bit can connect the 2-1 MUX to select the final operation since it decides whether the operation is among the first four or the last four operations. This method requires less logic gate and may take less time to debug all the six complicated logic operations. However, we are not comfortable using MUX in the real world circuit and it’s easier to debug the logic gate than a MUX. Finally, the first implementation of the computational unit is taken. For the routing unit, a dual 4-1 MUX is selected to choose the correct shifting bit for register A and B. Then, for the control unit, two k-maps are created for S(shift) and Q(current state). K-map for S is:

| | S | EQ | | | | | | --- | --- | --- | --- | --- | --- | | C1C0 |  | 00 | 01 | 11 | 10 | | 00 | 0 | 0 | 0 | 1 | | 01 | d | 1 | 1 | d | | 11 | d | 1 | 1 | d | | 10 | d | 1 | 1 | d | | | Q | EQ | | | | | | --- | --- | --- | --- | --- | --- | | C1C0 |  | 00 | 01 | 11 | 10 | | 00 | 0 | 0 | 1 | 1 | | 01 | d | 1 | 1 | d | | 11 | d | 1 | 1 | d | | 10 | d | 1 | 1 | d | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

The minterm for S is C1+C0+EQ’ and the minterm for Q is E+C0+C1. Using 2-input NAND and 3-input NAND inverters and a flip-flop for the current state Q and next state Q+, the routing unit can generate the output correctly. The last step is to design the output S, input LoadA and LoadB for the register to shift. The truth table is:

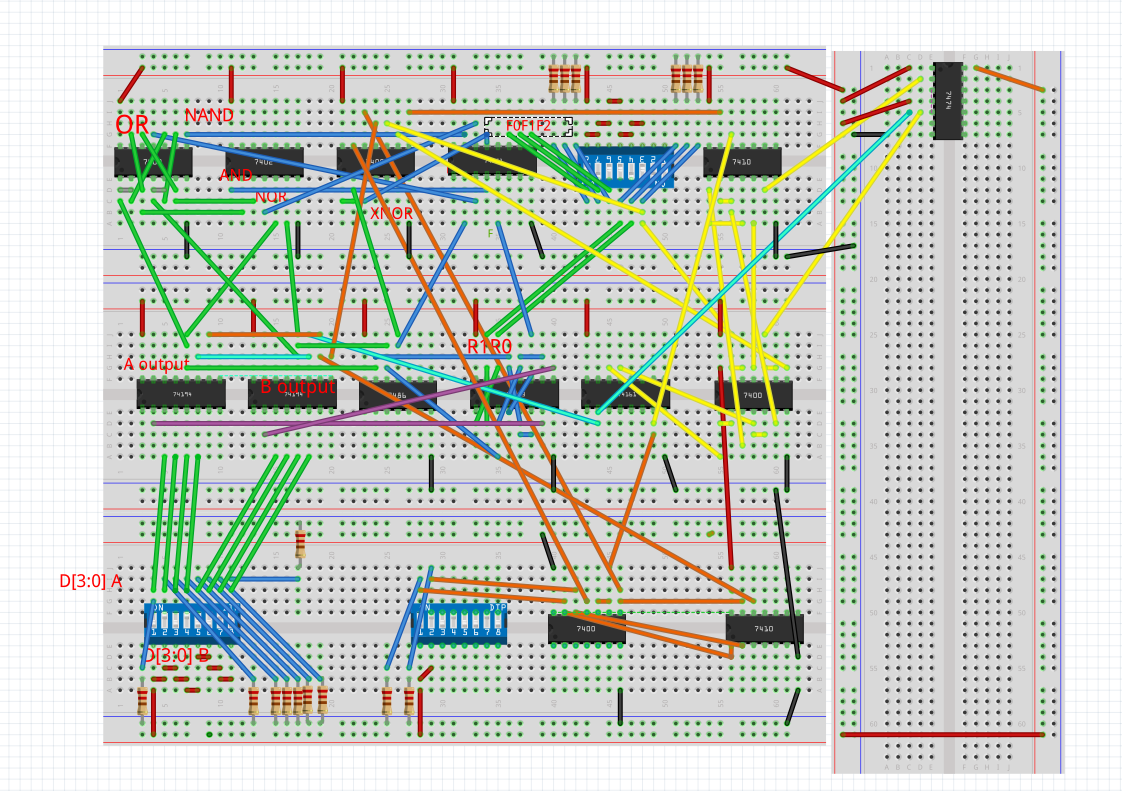
| S | LoadA | LoadB | S1 | S0 |
| --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | d |
| 1 | 0 | 1 | 0 | d |
| 1 | 1 | 0 | 0 | d |
| 1 | 1 | 1 | 0 | 1 |

Thus, the minterm for S0 is S+LoadALoadB, S1 = S’LoadALoadB. Here, the S0 and S1 is the input for 194 shift registers to decide whether it’s parallel load, shift right or hold. Also, S should connect to the LSR input in the register to control the right shifting.

****

The schematic diagram above is plotted using Fritzing. The red wires indicate that the ports are connected to Vcc. On the contrary, the black wire means that the port should be connected to the ground. Also, the cyan wires are connecting all the clock signals. To differentiate different parts of the circuit, orange wires are used to connect the output from six logic operations and the blue wires indicate the input for 4-1 MUX in the routing unit.

**Breadboard View**

****

The breadboard view is drawn on Fritzing and uses label to label the input for switches and six logical operations for easy implementation. This breadboard view matches highly with the real world circuit built for week 1 demo.

**8-bit Logic Processor on FPGA**

.sv Modules

Module: Synchronizers.sv

Inputs: clk, d

Outputs: q

Description: This is a synchronizer that takes in the input of the clock and data, d, to output logic q. Using these, the program will be able to sync the inputs and output at the same time

Purpose: The synchronizers are used to bring asynchronous signals into the FPGA. The synchronizer module is used to present the data output at the same time.

Module: Router.sv

Inputs: [1:0] R, A\_In, B\_In, F\_A\_B

Outputs: A\_Out, B\_Out

Description: The routing unit will take in the three inputs: A\_In, B\_In, F\_A\_B and will shift in the newA and newB depending on what R is set to. This program uses two unique cases in order to assign the new 8-bit data to its respective register.

Purpose: This module is used to control what new values will be shifted into each register after the one of the eight operations is performed. Depending on what the router input, R, is set to, the appropriate output will be shifted in and displayed on the FPGA’s LED for each register.

Module: Register\_unit.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: The register unit uses data gathered from other modules and lists out the functions that can be done to the registers. The register can shift in, load in the hardwired input, shift out, and display the new data created by the operations/router modules.

Purpose: This module is used to let the registers know which functions/operations can be done to it on a base/internal level. For example, it can take in parallel loads or shifts.

Module: Reg\_4.sv

Inputs: clk, Reset, Shift\_In, Load, [7:0] D

Outputs: Shift\_Out, [7:0] Data\_Out

Description: This is a positive-edge triggered 8-bit register with a synchronous reset. When the clock reaches the rising edge, it can reset the data, parallel load in the input [7:0] D, or it can start shifting values in from the routing unit while shifting out the initial values for a computation cycle.

Purpose: This module is used to parallel load in the values of the data hardwired into the FPGA board to the registers. It can also take in the bits from the routing unit and start shifting out the bits one at a time as the resulting bits from the routing unit is shifted in for each computation cycle. In addition, it can also synchronously rest the register units.

Module: Processor.sv

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

Outputs: [3:0] LED, [7:0] Aval, [7:0] Bval, [6:0] AhexL, [6:0] AhexU, [6:0] BhexL, [6:0] BhexU

Description: The processor is an 8-bit top level module that finalizes the inputs and outputs of the FPGA board on a surface level. It brings together all the inputs and outputs in the files to create the block diagram. The processor includes the inputs that will be used by the switches and buttons, and the outputs shown on the LEDs and HexDrivers on the FPGA board. The inputs for the computation and router units, F and R, are hardwired in as there are not enough switches on the board while the other inputs are set to switches. The logic processor instantianizes the other modules using these inputs/values. The processor also helps take the asynchronous inputs and make them synchronous along with the synchronous file. This allows the outputs to be present at the same time.

Purpose: This module combines all the other files to simulate a bit-serial logic operation processor and takes care of everything on the surface level of the high-level block diagram. This module takes in the hardwired inputs and is able to convey them to the internal process of each unit. It also helps synchronize the data, so it can appear at the same time when the clock is running.

Module: HexDriver.sv

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: Each HexDriver is a 7 segment LED light on the FPGA board and only accepts one 4-bit hexadecimal digit. Each 4-bit hexadecimal digit is converted into its corresponding 7 segment LED code. In this program, four HexDrivers are used as the program will display 16 bits of data.

Purpose: This module is used to transform the binary forms of the upper and lower bits of Registers A and B into a hexadecimal digit that will show on the LED.

Module: Control.sv

Inputs: clk, Reset, LoadA, LoadB, Execute

Outputs: Shift\_En, Ld\_A, Ld\_B

Description: The control unit keeps track of the current state after the execute switch is flipped and runs a cycle for each bit that will be shifted in. When execution starts, the computation cycle begins, and the control unit will shift the register unit the appropriate amount of times and halts until the next execution begins. This is produced by the state value A through J which keeps track which state the computation is at, and will go back to the first state A once the cycle is finished.

Purpose: The control module serves as a state machine and keeps track of where the current state is. The control unit starts the program when the execution switch is flipped and will begin by loading in LoadA and LoadB. It will then begin the computation cycle and go through the whole process before halting everything until the next execution.

Module: compute.sv

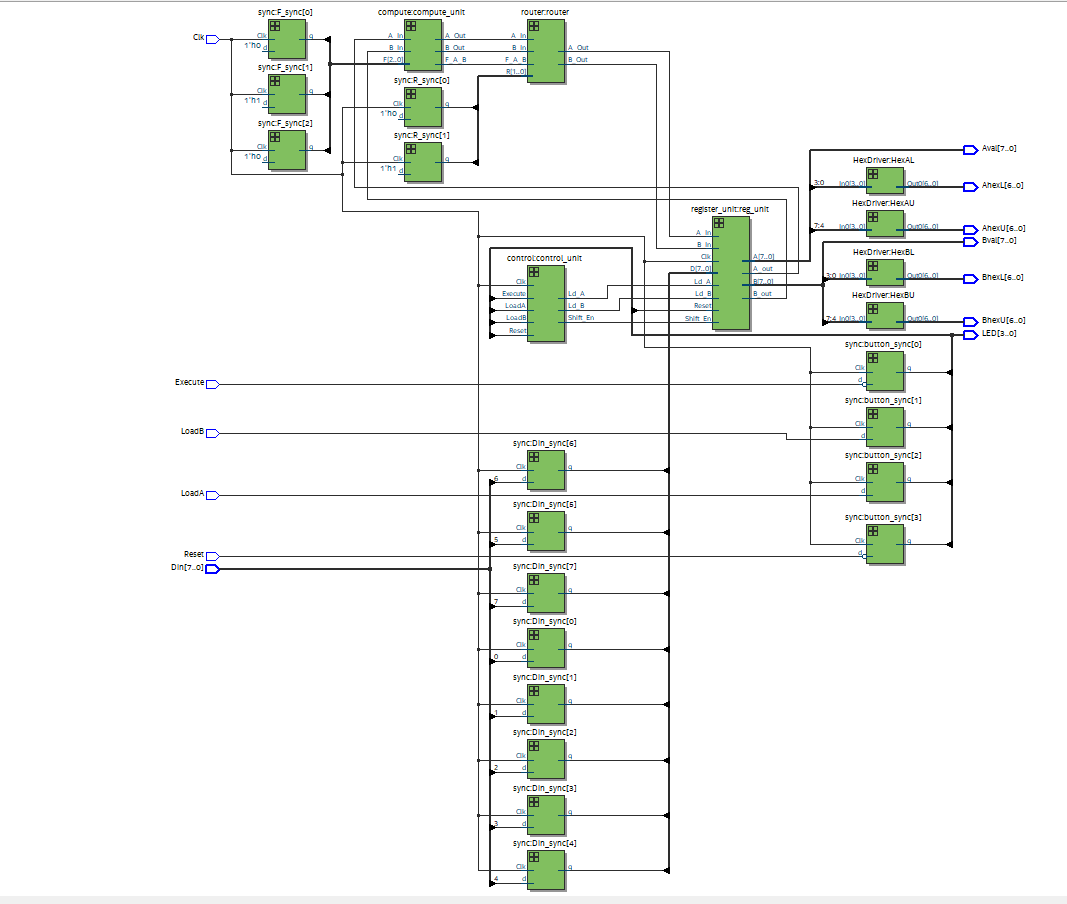
Inputs: [2:0] F, A\_In, B\_In

Outputs: A\_Out, B\_Out, F\_A\_B

Description: The computation will perform one out of eight operations and will be controlled by the function select inputs. The program assigns an operation for each [2:0] F input which is based on the table given and it will produce the appropriate outcome for the associated F input. These values will then be sent to the routing unit, which will choose which values to display to the register unit.

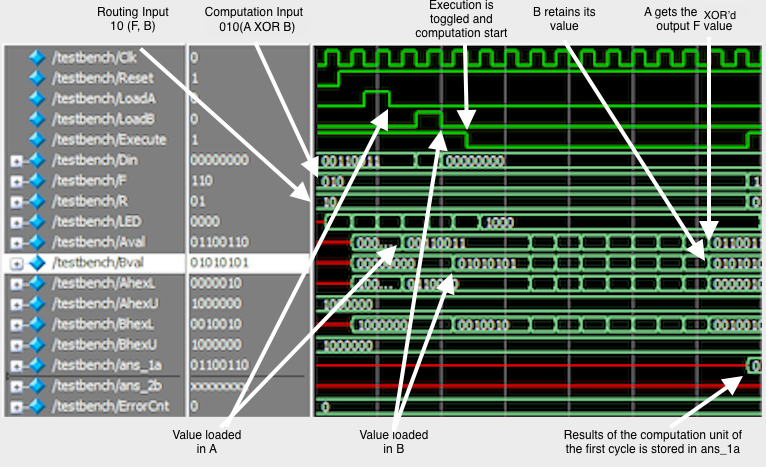
Purpose: This module takes in the values loaded in Registers A and B to perform eight different operations according to the table provided for lab 2. Based on the function selection inputs, it will perform one of the operations and store it into F\_A\_B.

**RTL Block Diagram View**

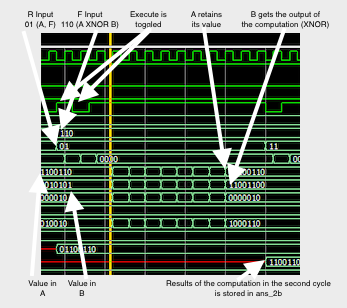
****

**Simulation of Processor**

In this lab, we are provided with a testbench\_8.sv file that allows us to see the process the program is running through for each cycle. In the photo below, there includes annotations of the first cycle of the process.



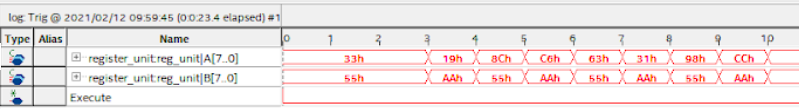
The picture above shows the first cycle of the process and the computation is successful with 0 error count. A XOR B is successfully computed and the new A gets the output while B retains its output as directed by the router unit. The computation result is stored successfully in Aval. The seven triangles between means that the values for Aval and Bval had shifted seven times. After shifting the eighth time, the final result was stored.

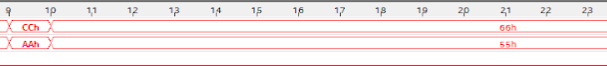


The picture above shows the second cycle of the process. As can be seen, the process is still successful as the operations and outputs are being shown appropriately. The second cycle’s output is stored successfully in Aval and the expected value for register B stores in and\_2b. It continues to have 0 errors, so the process is running smoothly.

**SignalTap Procedure**

To generate the SignalTap ILA trace, we first must hardwire computation input F and router input R as there are not enough switches. Then we run the SignalTap Generator and set it up by adding the three nodes: register\_unit:reg\_unit|A[7:0], register\_unit:reg\_unit|B[7:0], Execute. In this procedure we will be using the 8’h33 XOR 8’h55 operation. After the setup, we run the SignalTap program. Execute will be toggled and start the process. The SignalTap captures the process based on the sample space. The result of the 8’h33 XOR 8’h55 operation on SignalTap is shown in the pictures below. We can see that Reg A and B are storing the hardwired inputs correctly and the second picture successfully shows the result of the XOR function.

****

****

**Description of all Bugs (Post-lab Questions)**

For week 1 of this lab, the first unit that we started to build on the breadboard is the register unit, which includes two shift registers that weren’t familiar to us. We used the LEDs to detect the outputs from the two shift registers and see if the parallel loading works. After several tries on the 194 shift register, the LED will flash several times and before stabilizing. When the switch is flipped it takes a while for the LEDs to display the right outputs. After reviewing the lecture notes and more debugging, it is fascinating to see that when adding some resistors between the switch and the ground, the output of LED stabilizes and it will change immediately if the switch is flipped. Thus, the addition of resistors debounces the power floating between switches and registers.

After unit testing on the register unit and the computational unit and the routing unit, another problem occurred when building the routing unit. The cycle was discovered to have 0 output for S and the counter wouldn’t count when the execute switches to 1. However, when the execute switches to 0, the counter will count to four every few minutes. After recalculating the minterms and making sure the k-map works for S and Q, the Q is discovered to always turn to 1 immediately while the S turns to 1, resulting in the halt state immediately after reset state. After going to Office Hours and asking the professor, we discovered the bug, which was caused by the lack of use of a flip-flop to store the current state for the mealy machine. Commonly known, the finite state machine has to depend on the current state, and the flip-flop is used for latching the current state for one cycle and generating the next state. In this way, the next state Q+ can be different from the current state Q at a moment, jumping from reset to shift correctly.

For the second week of this lab, we had to use Quartus and SystemVerilog. There was not a lot of debugging to do as the code was mostly provided for us and only needed some adjustments. The most challenging part of this lab was the setup and learning how to operate around Quartus.

**Answer to Post-Lab Questions**

For a circuit that can optionally invert a signal, a 2-1 MUX is needed. The 2-1 MUX can accept two inputs, the one input signal and its inverted signal as the selection, and the other input as the select bit. The inverted signal can be generated from the original signal wiring to a NAND gate as the inverter. The select bit can decide whether the output should be equal to the input or the input inverted. This simple circuit can be very useful in the computational unit when we have to select one logic operation from the eight provided. It’s easy to discover that the 4 logic operations when F2 is 1 are exactly the inverted of the corresponding four operations when F2 is 0, at the situation when F1 and F0 are the same. Thus, the 8-1 MUX can be simplified by using only a 4-1 MUX and a 2-1MUX to optionally invert the signal depending on F2. Also, the 4-1 MUX uses the F1 and F0 as the select bit to select from four different operations. In this way, there are less logic gates required to build the other four logic operations and reduces the building time. Additionally, it has less logic gates and inverters to debug. However, personally it’s harder to debug inverters that haven't been used before, such as 4-1 MUX, the original circuit is built in real world events.

For the control unit, we learned beforehand that the Mealy machine not only depends on the current state, but also depends on the input. Meanwhile, the Moore machine only depends on the current state. Thus, using a Moore machine for this unit means using more states to loop through the four cycles and differentiating the hold state from the reset state. More specifically, more flip-flops are used to provide the next state bit. Compared to the Moore machine, the Mealy machine uses only one flip-flop for the current state and two counter inputs are used to replace the flip flops and shift to the next state. In the Mealy machine, only three states are required, and less need to create more than one current state.

The ModelSim can be generated without a FPGA board while the SignalTap needs to assign pin number on the board and download the program on the board in order to generate. Meanwhile, the ModelSim requires a testbench to operate the test on the system in order to see the result, while the SignalTap can input the result by using switches and see the result in LEDs immediately. In the situation when FPGA board is not presented or it’s dangerous to download the program on the FPGA in case the circuit doesn’t work, ModelSim is preferred to test out the circuit first virtually. After making sure the program won’t cause any potential dangers, the SignalTap should be used to simulate and function as a real circuit. SignalTap is more suitable to test if the circuit obtains any bugs that have not been tested from the testbench.

**Conclusion**

Lab 2 is designed to perform bit-serial logic operations on a real breadboard and on Quartus using SystemVerilog. Lab 2 prompted students to design and develop a circuit with two 4-bit shift registers, several multiplexers, and some type of counter. For this circuit, we load in the initial values using parallel loading on the shift register. The circuit runs for four cycles each time the execute switch is flipped, and will produce the desired operation and outcome before showing results on a LED chip. On Quartus, the code was provided and we had to make adjustments to extend the 4-bit logic processor to 8-bits using SystemVerilog. By doing so, students are able to intimately understand SystemVerilog and practice it themselves. The code operates in a similar way to the breadboard. Instead of using the chip with the switches, we use the FPGA board switches and produce the binary numbers as a hexadecimal LED instead. The lab also introduces students to testbench and SignalTap which is a great way to debug the code and see the operations happening to each input.