Lab 2 Report

Noah Conner & Adam Naboulsi
Section: ABP

Introduction:

On the highest level, the circuit performs bitwise logic operations on two 4-bit numbers, then routes the result to a specified destination register. The operations that the processor supports are AND, OR, XOR, NAND, NOR, XNOR, CLR, SET, and SWAP. The user has control over what logic operation is being performed, where the result is being routed, and when the processor executes the operation.

Operation of the logic processor:

In order to perform a logic operation with the processor, the user must understand the inputs to the processor. The user-controlled inputs are the following: a 3-bit function control, 2-bit router control, load A, load B, and execute. The function control governs what logic operation is to be performed. The router control determines which register will store the result of the logic operation, whether the registers will swap values, or whether the registers will retain the same value. Load A and load B are used to load values into registers A and B. Finally, the execute signal triggers the start of an operation. In order to load data into the A and B registers, there are four switches connected to either the A or B register (depending on which is hardwired). Once the desired 4-bit number is entered on the switches, the user must toggle the load A or load B switch for one clock cycle. After one clock cycle, the value will be parallel loaded into the desired register. To initiate a computation and routing operation, the user must press the execute button for one clock cycle.

Description of block diagram and state machine diagram of logic processor:

Overview

The register, computation, routing, and control units work together to create a functional 4-bit logic processor.

Register Unit

The register unit is connected in series with the computation unit and routing unit. The unit is controlled by the shift, load A and load B signals and receives data from the parallel in switches and the router. The outputs include the shift data and parallel data out.

Computation Unit

The computation unit is connected in series with the register unit and routing unit. The unit is controlled by the 3-bit function control input and receives serial data from the register unit. The serial output feeds into the routing unit.

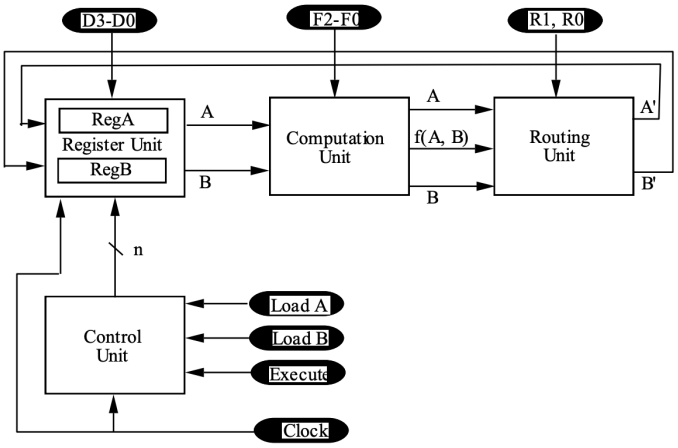
Routing Unit

The routing unit is connected in series with the register unit and computation unit. The unit is controlled by the 2-bit router control input and receives serial data from the computation unit. The serial output feeds into the routing unit.

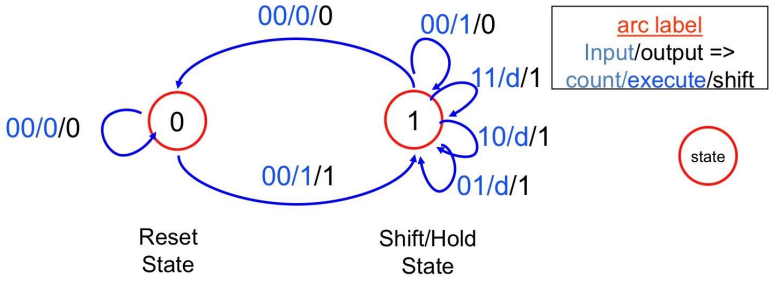
Control Unit

The control unit is connected to the register unit. The unit is controlled by load A, load B, and execute signals. The outputs include shift, load A and load B signals that feed into the register unit.

Block Diagram:



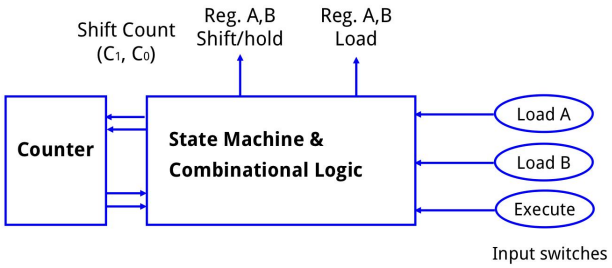
State Machine Diagram: Below is the Mealy machine we used to design our state machine.



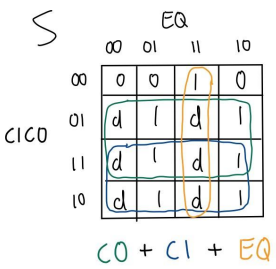
Design steps and circuit schematic diagram:

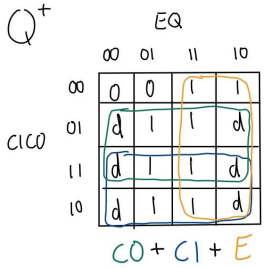
Procedure

In designing our bit-serial logic processor, we first implemented the control unit following the diagram shown below. In designing it, we determined the necessary combinational logic for the state machine using the provided truth table to create K-maps for the outputs S+ (next shift signal value) and Q+ (next state) shown below. We used a 4-bit synchronous up-down counter provided in our lab kit and a single JK flip flop synchronous with the CLK signal that took in Q+ as an input (with an output that is fed back into Q of the state machine). Instead of feeding Load A and Load B into the state machine (as depicted in the diagram), we fed them into the registers.



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

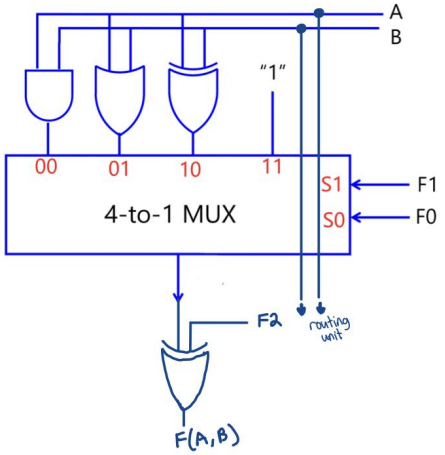




Next, we implemented the register unit. To implement our register unit, we used two 4-bit bi-directional shift registers. These shift registers had two inputs S1 and S0 whose values determined whether the registers were in a hold state, a shift right state, or a parallel load state. To ensure that the registers were in the right state at the right time, we fed the load signal of each register into their corresponding S1 input pin, and OR'd the shift signal (S) with the load signal of each register into their corresponding S0 pin.

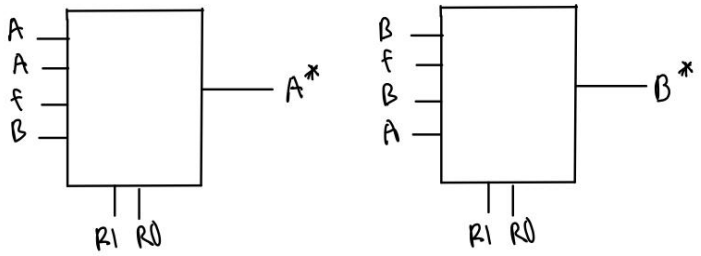
Next, we implemented the computation unit which took as inputs the least significant bit outputs from RegA and RegB (after right shifts) of our register unit and the signals F2-F0 which determined the desired logical computation. Its outputs were the F(A, B), the output bit of the desired logical computation, and the least significant bit inputs from RegA and RegB which were passed through to the routing unit of our design. In order to implement the 8 functions of the

computation unit (AND, OR, XOR, 1111, NAND, NOR, XNOR, and 0000), we used a 4-to-1 MUX whose output was XOR'd with F2 (this was done due to the fact that the inverted versions of the operations occurred when F2 was high according to the truth table shown below). A gate-level diagram of our computation unit is shown below.



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

Lastly, we had to implement the routing unit which took as inputs the outputs from RegA and RegB (after right shifts of the registers) and F(A, B) from the computation unit along with R1 and R0 which determined the destination of A+ and B+, the outputs that will be fed back into the register unit. To implement the routing unit, we used two 4-to-1 MUXs with the select bits R1 and R0 as shown below.



Design Considerations/Trade-offs

The first implementation of our control unit combined the state logic and counter logic into our state machine, needing only the addition of a few flip flops to synchronize the counter logic with the clock. After realizing the number of chips that we would need to implement this design, we redesigned the state machine to use a separate counter chip. This way, the state logic, and counter are separate modules, resulting in a more organized design. Additionally, the use of fewer chips lowers the power consumption of the control unit. For the computation unit, a simple approach would be to use an 8:1 multiplexer (made up of two 4:1 multiplexers) to control which of the eight functions to perform. However, since half of the functions are just negations of the other half, we were able to use only a 4:1 multiplexer and an XOR gate. The XOR gate allows the user to optionally invert the functions. As with the control unit, this made for a sleeker design and, ultimately, fewer wires. The routing and register units had no optimizations to be made as they are fairly simple units.

Detailed Circuit Schematic

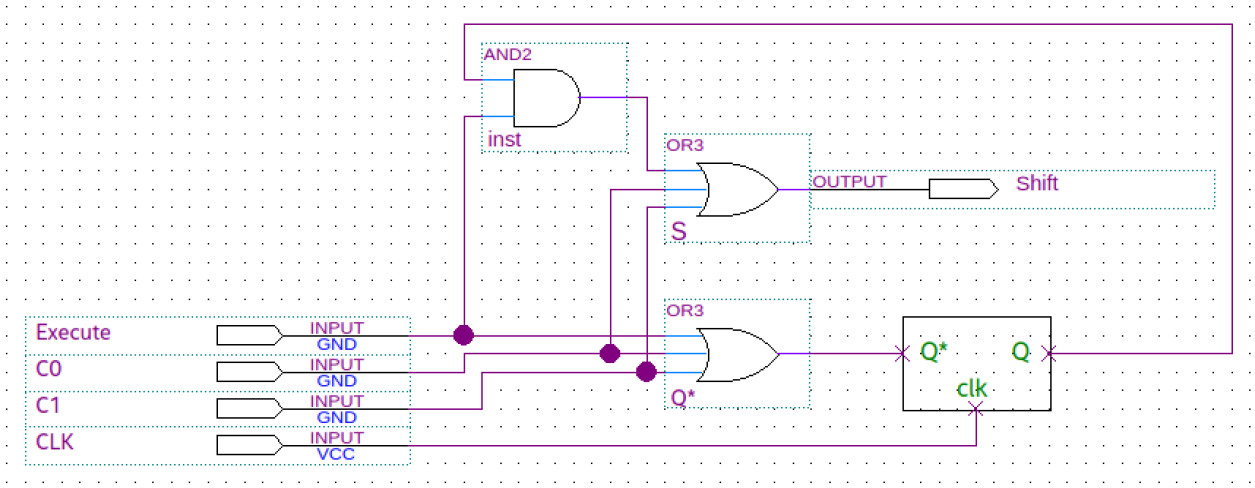


Figure: Gate-Level of Control Unit

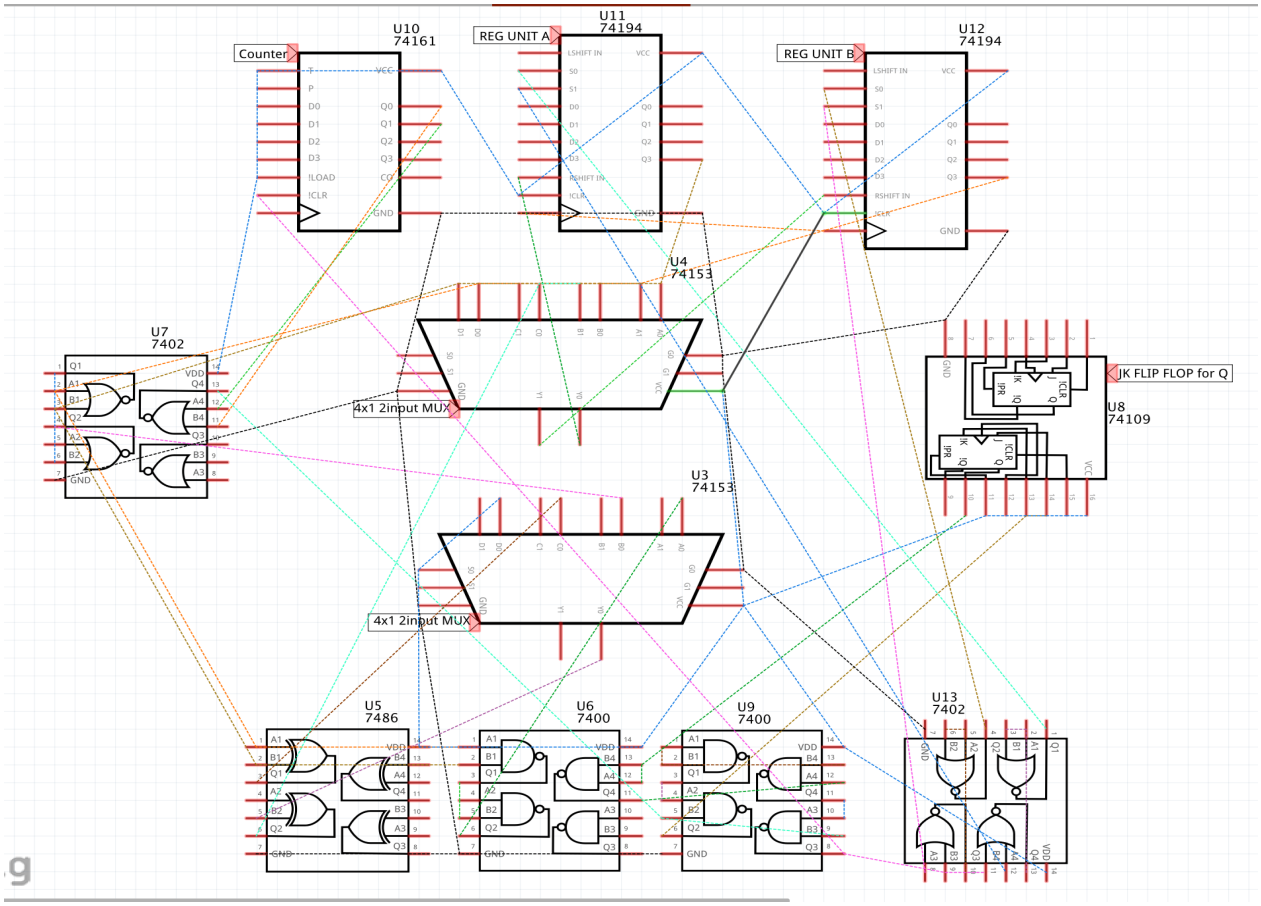


Figure: Processor Top-level Schematic

BreadBoard View:

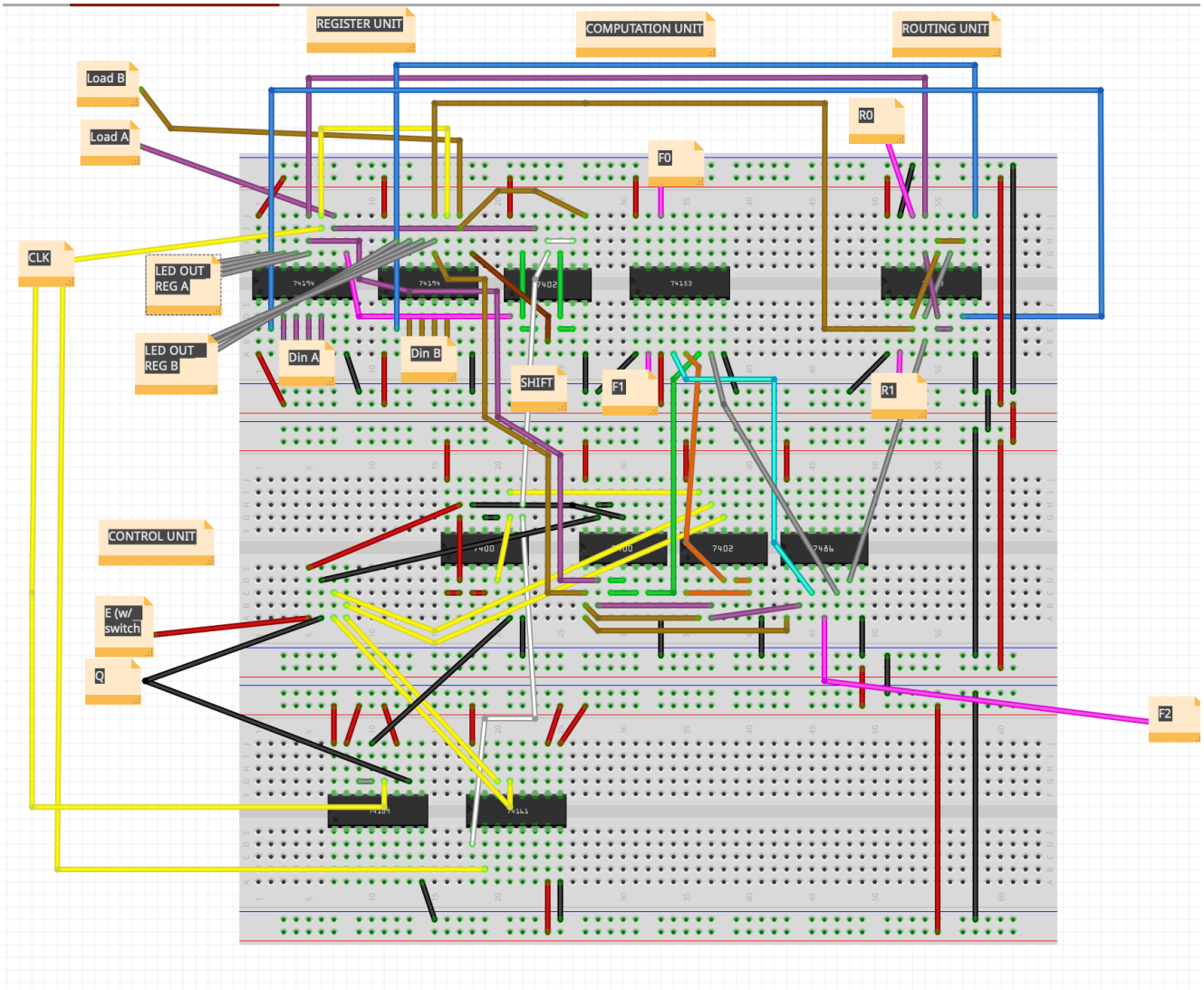


Figure: Breadboard Schematic from Fritzing

The 8-bit logic processor on FPGA:

Module: Synchronizers.sv

Inputs: Clk, d

Outputs: q

Description: The synchronizer module outputs a clock-synchronized signal of an asynchronous input signal. Includes 3 sync modules, one with no reset for switches and buttons, one with reset to 0, and one with reset to 1.

Purpose: This is used to synchronize all of the user-inputted signals.
Modifications Made: None were needed.

Module: Router.sv

Inputs: [1:0] R, A_In, B_In, F_A_B
Outputs: A_Out, B_Out

Description: Depending on the value of the 2-bit router control bits, the router assigns outputs A and B to either register A, register B or F(A, B).

Purpose: To route the inputs of the shifted (out) register bits and the computation done in the computation unit (F) to the desired outputs (A_Out and B_out).
Modifications Made: None were needed.

Module: RegisterUnit.sv

Inputs: Clk, Reset, A_In, B_In, Ld_A, Ld_B, Shift_En, [3:0] D
Outputs: A_Out, B_Out, [3:0] A, [3:0]

Description: Contains two register instances. Tells the registers when they should shift, reset, or parallel load. Also transmits the data out of the registers as an output.

Purpose: The register unit module acts as a signal interface to register A and register B

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

Module: Reg_4.sv

Inputs: Clk, Reset, Shift_In, Load, Shift_En, [3:0] D
Outputs: Shift_Out, [3:0] Data_Out

Description: This is a positive-edge triggered 8-bit register with a synchronous reset, synchronous load, and synchronous shift. When Load is high, data is parallel loaded from Din into the register on the positive edge of Clk. When Reset is high, Data_Out is set to 0. When Shift_En is high, it concatenates the Shift_In bit with the previous three most significant bits stored in the register. Assigns Data_Out[0] as Shift_Out.

Purpose: The reg module acts as a simple 4-bit register with parallel load and shift abilities. It is used to instantiate registers A and registers B.

Modifications Made: Had to change the input D and Data_Out to 8 bits; adjust shift procedure to concatenate the 7 leftmost bits.

Module: Processor.sv

Inputs: Clk, Reset, LoadA, LoadB, Execute, [3:0] Din, [2:0] F, [1:0] R,
Outputs: [3:0] LED, [3:0] A_Val, [3:0] B_Val, [3:0] AHexL, [3:0] AHexU, [3:0] BHexL, [3:0] BHexU

Description: Instantiates the modules within the serial-bit processor design and 'wires' the outputs of certain modules to inputs of other modules (such as the shifted bit outputs of the register unit to the inputs of the computation unit). It also uses the sync modules to make the asynchronous inputs of the processor synchronous with the clock signal. Lastly, it instantiates the Hex Drivers to display the desired output bits of the registers.

Purpose: Serves as the top-level module of the serial processor, connecting the various modules.

Modifications Made: Had to change Din, A, B, and Din_S to 8 bits ([8:0]). Also had to instantiate two more hex drivers for the new upper 8 bits of the two registers.

Module: HexDriver.sv

Inputs: [3:0] In0
Outputs: [6:0] Out0

Description: Based on the inputs, a switch statement is used to map the outputs to the corresponding LED segments (7 total) on the HEX display of the FPGA board.

Purpose: Used to output bits of registers in the register unit to the HEX displays on the FPGA board.

Modifications Made: None were needed.

Module: Control.sv

Inputs: Clk, Reset, LoadA, LoadB, Execute
Outputs: Shift_En, Ld_A, Ld_B

Description: Defines the state values A-F and uses a switch statement to control transitions to next states; updates flip flop with each positive edge of the clock signal; checks for reset; and assigns outputs based on the current state.

Purpose: Module that implements the state machine for the control unit in the serial processor design.

Modifications Made: Had to add 4 shift states (now ranging from A-J) due to the modification of the serial processor to 8-bits.

Module: Compute.sv

Inputs: [2:0] F, A_In, B_In
Outputs: A_Out, B_Out, F_A_B

Description: Takes shifted bits from registers within the register unit and computes desired logical computation corresponding to the F input bits. Uses a switch statement with the cases as the different F inputs to decide which logical computation must be performed on the shifted bits from RegA and RegB (outputs the computed bit along with the originally shifted bits that are passed through to the routing unit).

Purpose: Implementation of the computation unit of our serial processor

Modifications Made: No changes were needed.

RTL Block Diagram

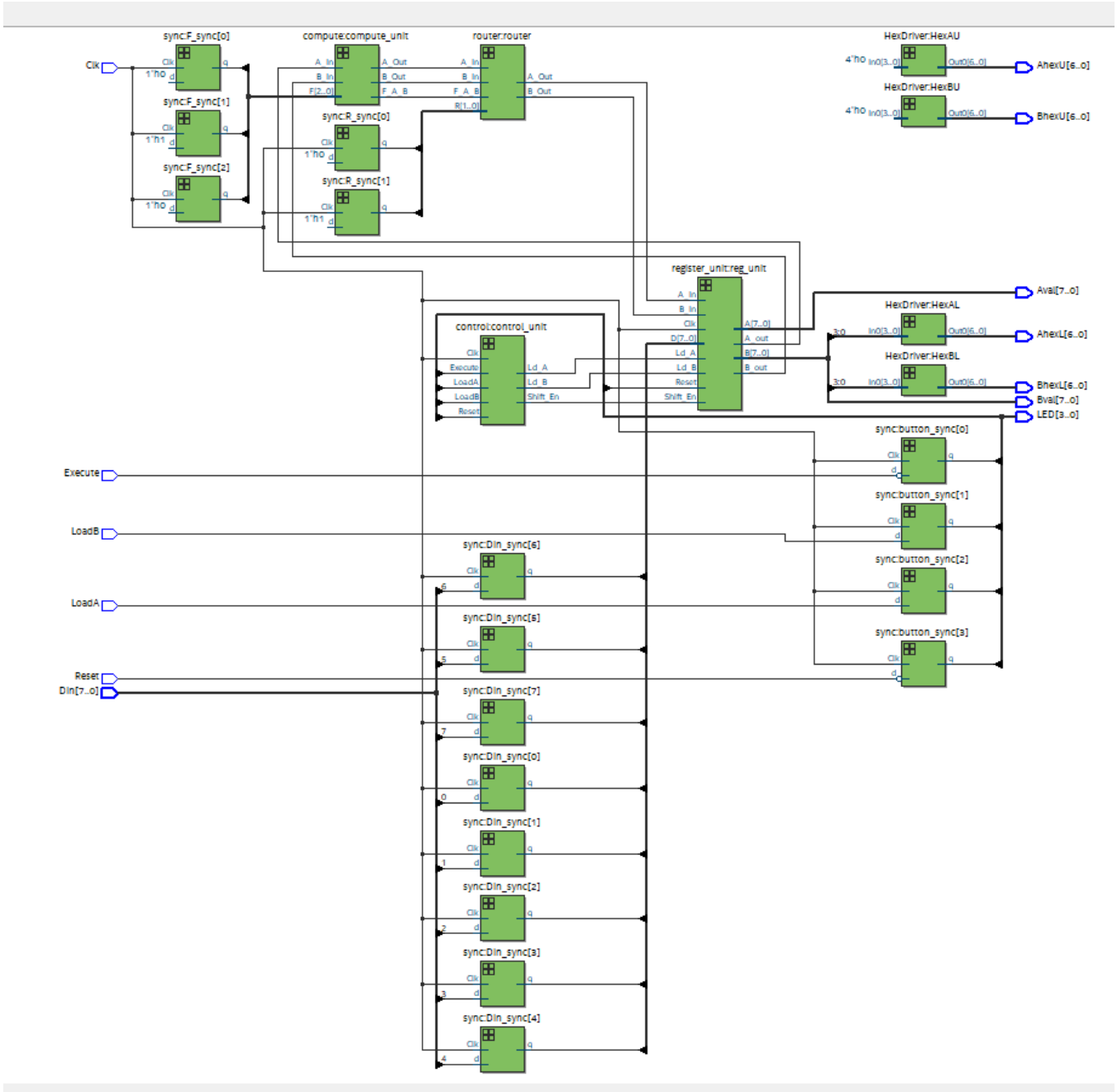


Figure: Processor RTL Diagram

Processor Simulation

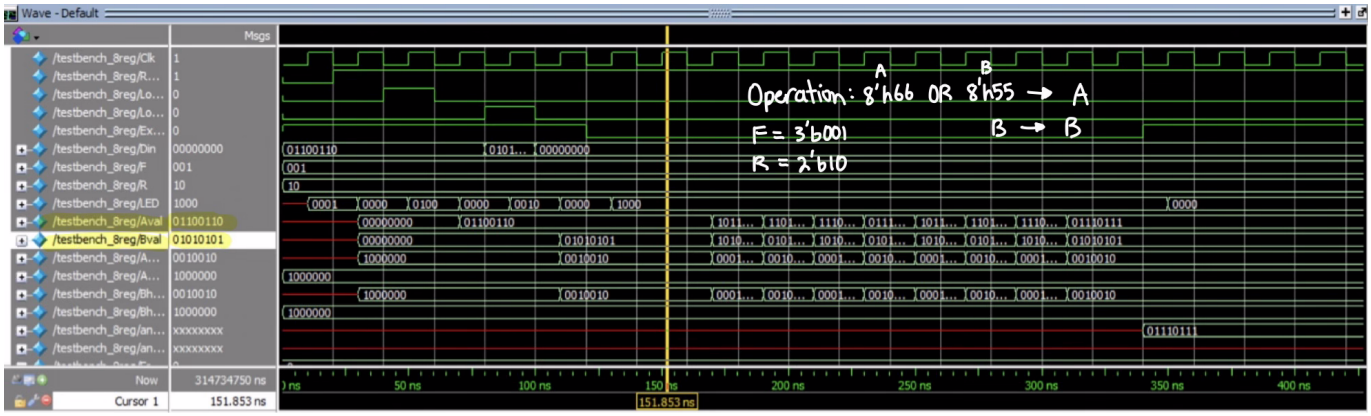


Figure: Simulation of Operation h66 OR h55 w/ Testbench

SignalTap Procedure and Results

Step 1: Open SignalTap and (in the setup tab) add Reg A data out, Reg B data out, and execute nodes; set execute trigger to falling edge. Set clock to correct clk input, then compile code.

Step 2: Compile code, then program the FPGA device.

Step 3: Click on SignalTap instance and run analysis. Status should display "waiting for trigger".

Step 3: Set data in switches to desired value for reg A (in my case, 8'h33) and flip loadA switch up and down.

Step 4: Repeat step 3 for reg B using loadB switch (in my case, 8'h55).

Step 5: Push execute button and the data – collected overtime of reg A, reg B, and execute – should display on SignalTap in data tab.

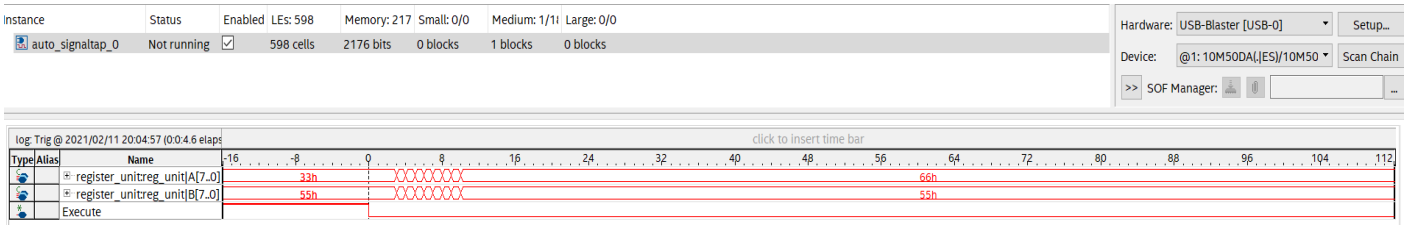


Figure: Data Acquired of h55 XOR h33 on SignalTap

Bugs Encountered and Corrective Measures:
The bugs we encountered during the breadboard implementation of the 4-bit serial processor boiled down to not pulling specific pins to gnd/power. For example, on the 4-bit sync up-down counter chip we used, we initially did not pull the ENP and ENT pins high (which had to be pulled high in order for the chip to count as stated on its datasheet). We narrowed down the issue to the control unit by monitoring the outputs of each component of our design using the signal analyzer on our Scopy 2 device provided to us in ECE 210. In doing so, we found that the outputs of the counter were not changing. Thus, we looked into its datasheet and found the error. A similar situation occurred with our JK Flip Flop for input Q (the current state of our finite state machine) in the control unit for which we did not pull its active-low reset pin (PRE) high.

In regard to bugs encountered with modifying the SystemVerilog implementation of the 8-bit serial processor on Quartus, we initially used one less state than we should have in the control module. Our code compiled, and through testing it with the testbench file and ModelSim, we saw that the registers were shifting one too few times. Thus, we added another shift state to control.sv and retested our code through ModelSim which fixed the issue.

Conclusion:
Summary
In this lab, our goal was to design and build a 4-bit serial processor capable of carrying out 8 logical operations. We used 4-bit shift registers, MUXs, a counter, and logic gates (NANDs, NORs, and XORs) along with a finite state machine serving as the control unit to build the processor. After implementing the processor on our breadboard using TTL chips, we learned to use SystemVerilog and modified the implementation of the 4-bit processor on Quartus to accommodate 8-bits.

Post-Lab
Q1: Document changes to your design and correct your Pre-Lab write-up, explaining any difficulties you had in debugging your circuit.

The most significant change to our design that we made during the breadboard implementation involved the counter mechanism. We initially designed the state machine such that the counter was internal; we created K-maps using the provided truth table for bits C1 and C0 and determined the necessary combinational logic to implement the counter bits. The need to use a few more chips along with two flip-flops for the counter bits made us reconsider our design. We instead decided to use a counter chip provided to us in the lab kit which significantly reduced clutter and allowed for a much more organized design.

Q2/3: Outline how the modular approach proposed in the pre-lab helps you isolate design and wiring faults, be specific and give examples from your actual lab experience. Explain how a modular design such as that presented above improves testability and cuts down development time.

Modularity is a very important aspect to consider when designing any system. Firstly, it makes the project easier to understand because you can break the system into modules. These modules can then be worked on and refined independently of the rest of the system. This drastically cuts down on development time because the tasks can be easily broken up and worked on separately. It is also invaluable when it comes to testability. In a good design, each module should be able to be tested and debugged separately from the rest of the system. This makes it much easier to pinpoint problems and fix them.

Q4: Discuss the design process of your state machine, what are the tradeoffs of a Mealy machine vs a Moore machine?

We chose to implement our control unit as a Mealy state machine because of the fewer number of states. In the case of a Moore machine, the state machine would have needed at least five different states: reset, three counter states, and halt. This would have required a much larger number of chips to implement than its Mealy counterpart. In choosing the Mealy machine, we were able to design a much sleeker, organized control unit. In the case of this project, I would say there were no negative tradeoffs in using a Mealy machine. However, in many cases, a Mealy machine could add more complexity to and, sometimes, negatively impact the design.

Q5: 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 and SignalTap are very different and have their own advantages and disadvantages. SignalTap is a tool used to collect actual signals present on the FPGA, just like a signal analyzer. ModelSim is different from SignalTap in that it is purely a simulation of the hardware and does not use the physical device. ModelSim is a great tool for quickly simulating a controlled set of inputs. Using a testbench, the user can automate a controlled sequence of inputs to the simulated hardware to see how the system behaves. This is very useful during the debugging process when you are working on a specific problem that may require a large number of user inputs to actuate. Where ModelSim lacks, SignalTap allows for a more flexible, accurate analysis of the hardware. Since SignalTap acts as a signal analyzer, you can observe how the system behaves in real-time on the actual FPGA. The user can input any combination of inputs without changing the code. This is useful when you are testing the system as a whole.