ECE 385

Fall 2021

Experiment #2

A Logic Processor

Michael Grawe, Matthew Fang
ABE
Abigail Wezelis

Index

Introduction

At a high level, the function of the logic processor is to perform one of eight possible operations on a bit stream from two different registers, and either store that result in one of the registers, return the original values to the registers, or swap the values of the registers. The different operations that our circuit can perform are the AND, OR, XOR, NAND, NOR, XNOR, 0000, and 1111 where 0000 and 1111 store all zeros or all ones in the desired register. These operations were chosen because they are strictly bitwise (i.e., the result stored in the second bit depends only on the value of the values of the second bits in both registers.) The physical circuits from 2.1 can operate on only 4 bits, while our extended design from 2.2 can operate on 8 bits. Our 2.1 circuit displays the contents of both our registers, Register A and B, are displayed by LED's, with a lit LED representing a logical 1, and an off LED representing a logical 0.

Operation of the Logic Processor

To load values into A and B for our logic processor, we had four Input Value switches that corresponded to the bits that would be parallel loaded into Register A or Register B. Which register was loaded with the corresponding values dictated by these four switches was controlled by two switches called Load A and Load B. After a bit sequence was manually put into the Input Value switches, flipping Load A on would load that bit sequence into our Register A. Similarly, flipping Load B on would load that same sequence into Register B.

After data has been loaded into the Register Unit, it's time to perform an operation on it. We had 3 Function Switches corresponding to the three bits used to choose which function the ALU performed. The specific bits required to implement each function are listed below. We also had two Routing Switches that, much like the Function Switches, correspond to the two routing bits necessary to choose whether F, A, or B is routed to each of the two registers. Specific selection is listed 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

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

Written Description of Circuit

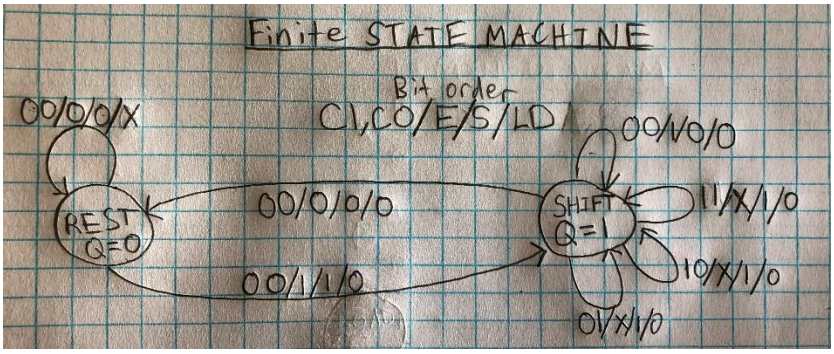
The Computation Unit for our circuit consisted of combinational logic and an 8:1 MUX. For each of the operations given in the Function Selection Table, we routed the lowest bit of our Registers A and B into a series of a 7400 Quad NAND chip, a 7404 Hex inverter chip, and a 7486 Quad XOR chip. These chips were used to create 8 different outputs for f(A,B). These 8 different outputs were then each hooked up to their own pin on the 8:1 MUX. Our Function Switches were placed in the select bit pins on the 8:1 MUX such that the correct series of Function Select Inputs corresponded to the right Operation being performed on A and B. It is important to note that we only operated on the lowest bits of Register A and Register B.

The Routing Unit's function was to store either f(A,B), A, or B into either register A or B. We implemented this using a 4:1 MUX and a series of switches for R1, R0. The output of our computation unit, the lowest bit of Register A, and the lowest bit of Register B were inputs to our 4:1 MUX. The select bits were R1, R0. We made sure to connect them in such a way such that the above Routing Selection would be implemented.

Our Register unit consisted of two Bidirectional 4-bit Shift Registers, Register A and Register B. The input pins to both registers were hooked up to our 4 Input Value Switches. The Clk pin for both was connected to our clock from the DE-10. The S1 and S0 pins were used to determine whether we wanted to load data into the register or shift data in the register. Our implementation used a left shift, so we connected the signal Shift OR Load to S1 and the signal Load to S0. Shift is the output of our Control Unit, and Load is just the switch signal of either Load A or Load B corresponding to the correct register, as described above.

Our Control unit's main function was to determine when to shift the data left in the registers. This shift corresponded to the Shift signal. Inputs to the control unit included Execute, a debounced switch from our DE-10. The chips used in our control unit were a 4-bit Binary counter, a D Flip-Flop, a 7400 Quad NAND, a 7404 Hex Inverter, and a 7427 three-input NOR. The internal signals within the control unit include the lower two bits of the counter, the Execute signal, and the current state of the control unit finite state machine called Q. Q is represented by the output of the flip-flop, and the next value of Q is modified by combinational logic: $Q+ = E + C0 + C1$ where E is Execute and C1, C0 are the lowest two counter bits. Shift signal is also determined by combinational logic: $S=C0 + C1 + (E\ Q')$. The counter and the flip-flop in the

control unit are both clocked with the DE-10. Our counter was set up to stop counting after four clock cycles so that only four shifts were performed, as desired. This was achieved by hooking the ENP and ENT pins, the count enable bits, to our Shift signal. This way, as soon as we wanted to stop shifting, our counter would stop. The state machine used to implement our control unit, as well as a block diagram for our entire circuit is detailed below. We used a Mealy state machine for simplicity.



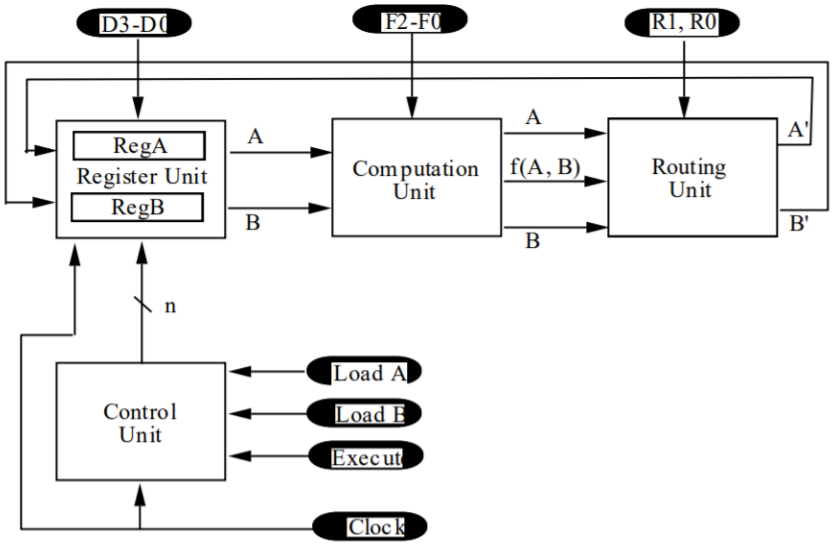


Figure 1: Block Diagram

Design Steps

The first step we took when designing the circuit was choosing which shift-register to use. We chose the bidirectional register because we wanted some freedom to decide which way to shift later in the project. We then implemented the computational unit, connecting it to the lowest bit on the shift registers. We opted to not use exclusively NAND gates because breadboard neatness was a valuable commodity in this lab. By using an inverter and an XOR, we were able to have less wires going between the combinational logic chips. This made debugging the computation unit much easier.

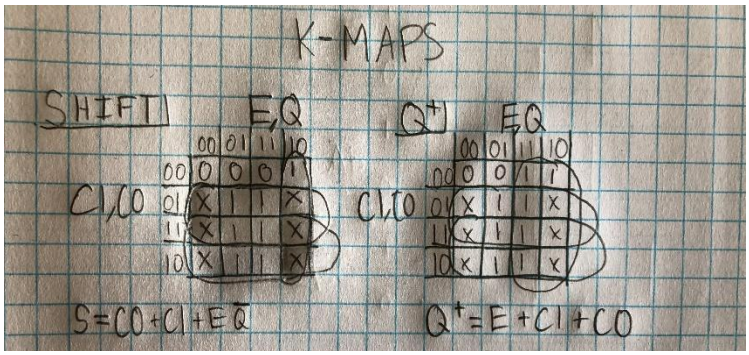
When deciding how to implement the routing unit and the computational unit, we made the decision to use MUXes to choose what routing and function type to implement. This was a decision we came to quickly, as it seemed clear that it was the easiest option to choose between a variety of inputs since that is what a MUX is designed to do.

Connecting the register unit to the Load A, Load B, and Input signals required us to decide whether to use the debounced switches on the DE-10 or the DIP switches on the breadboard. We chose the DIP switches because all the signals in the circuit would be clocked. Therefore, debouncing would only be a problem for the Execute signal. This also applied for the switches for our Function Selection and Routing Selection.

We spent the bulk of our time in this lab working on implementing the control unit. Our decision to use a Mealy finite state machine was greatly influenced by the ease of implementation that a Mealy machine creates. Our first step in designing the Mealy machine was creating the K-maps for our Next State (Q+) and Shift (S) signals. We did this from the given state transition table. Both the truth table and the K-maps are shown below. We then implemented the combinational logic for our Q+ and S signals, using the output of our chosen counter chip. For the counter, we needed to make the counter stop after four shifts, so we connected the enable bits to S since S went low after 4 cycles. Finally, we determined the logic necessary to pass to the register unit to control when the register unit would perform a shift or a load.

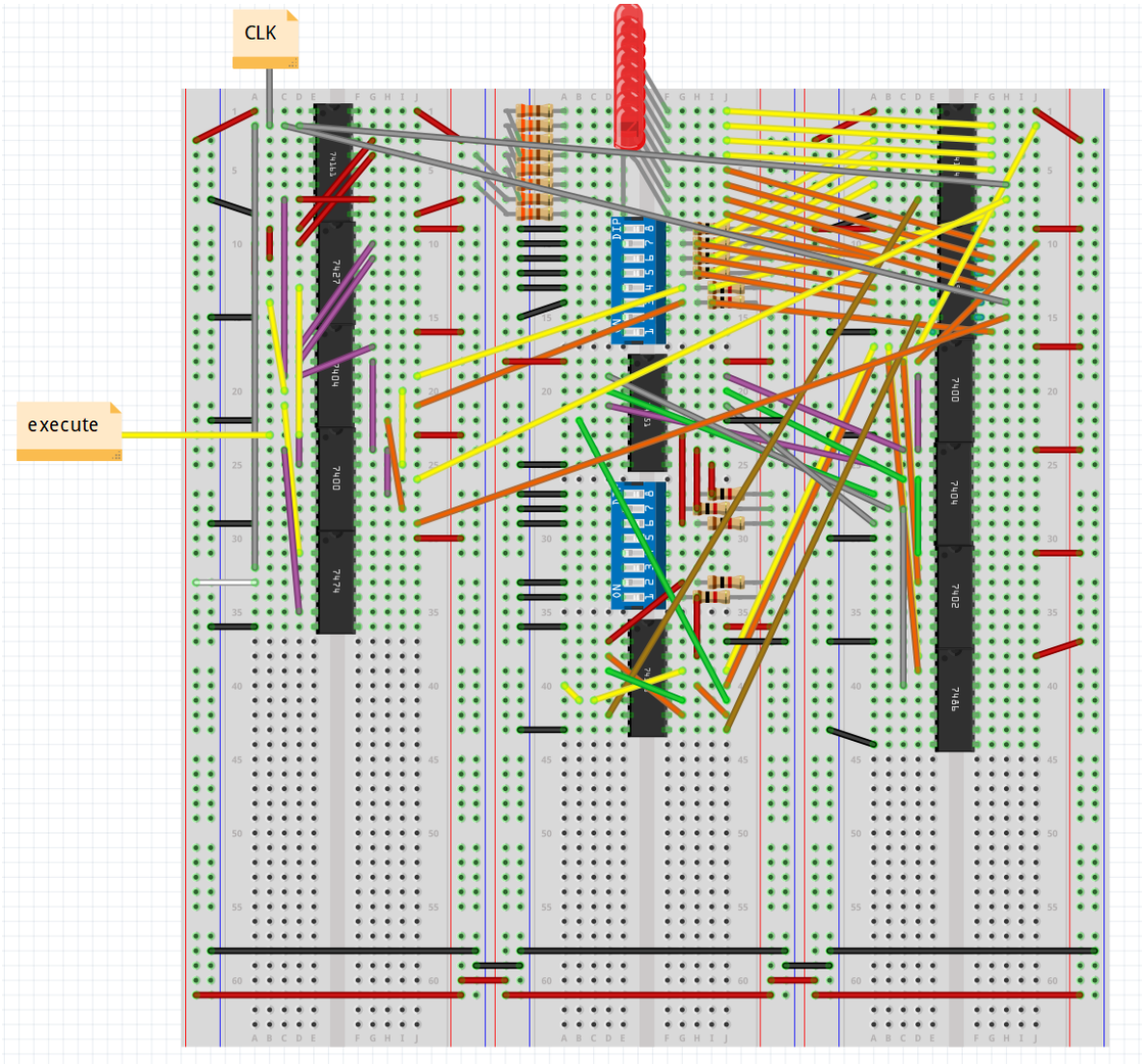
TABLE 1: Control unit state transition table using the Mealy state machine

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



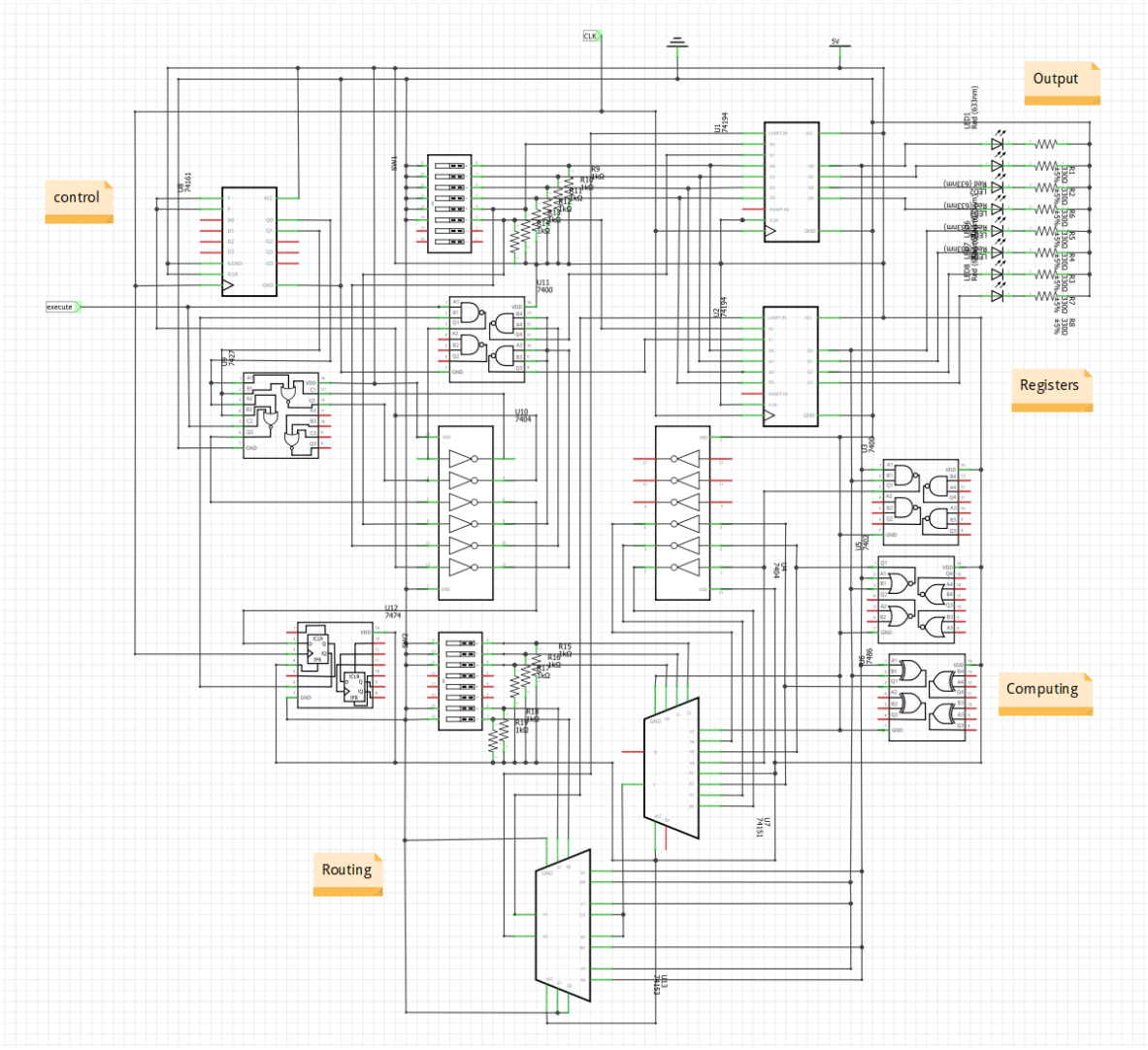
Component Layout

The following is the breadboard layout of the correct implementation of the Lab.



‘

The top four LEDs and bottom four LEDs show the registers A and B respectively. The top four switches control the inputs to the two registers while the fifth switch and sixth switches load A and B. In the second 8-DIP switches, the top 3 control the computing unit while the bottom two control the routing unit.

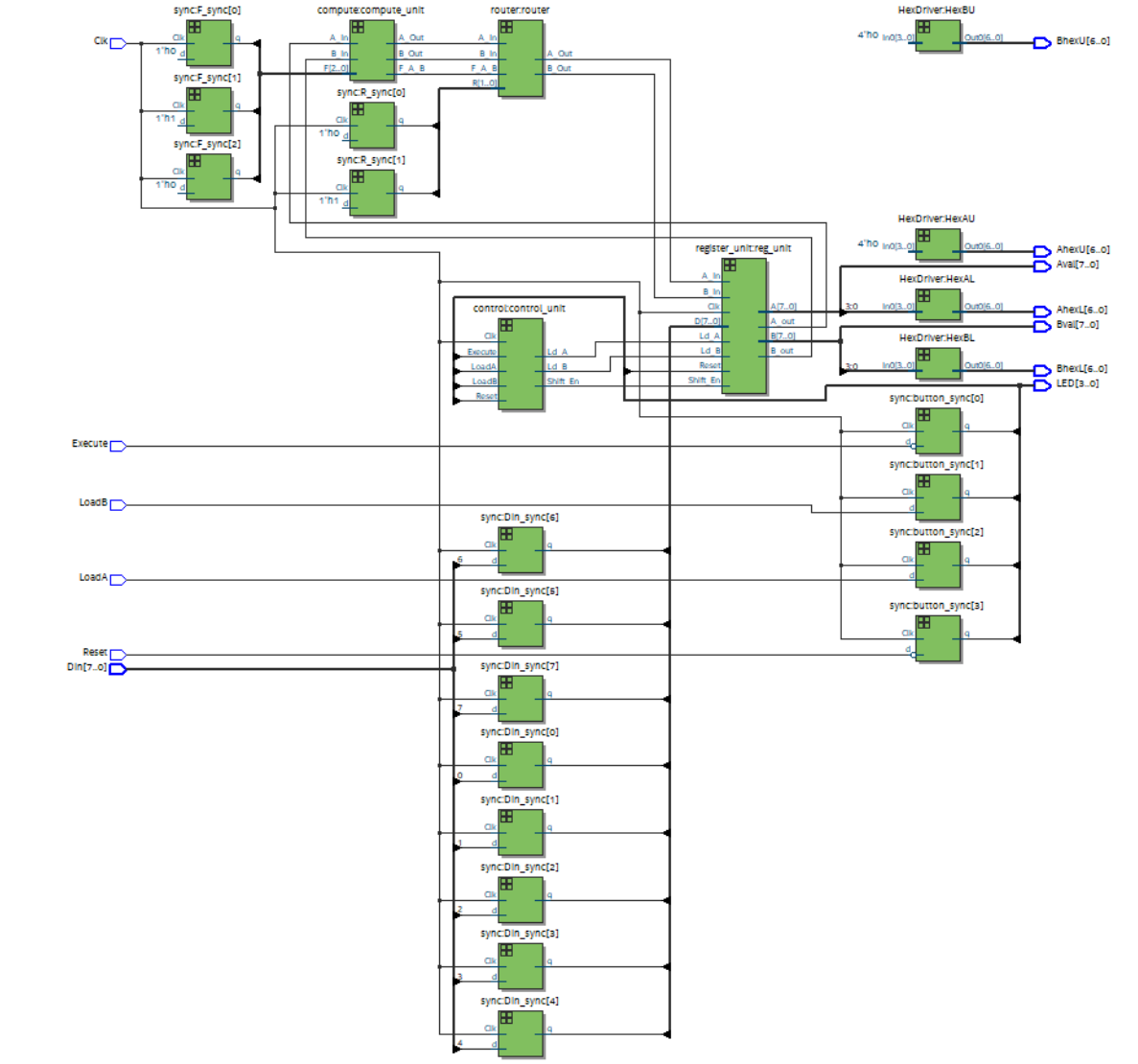


8-bit logic processor on FPGA

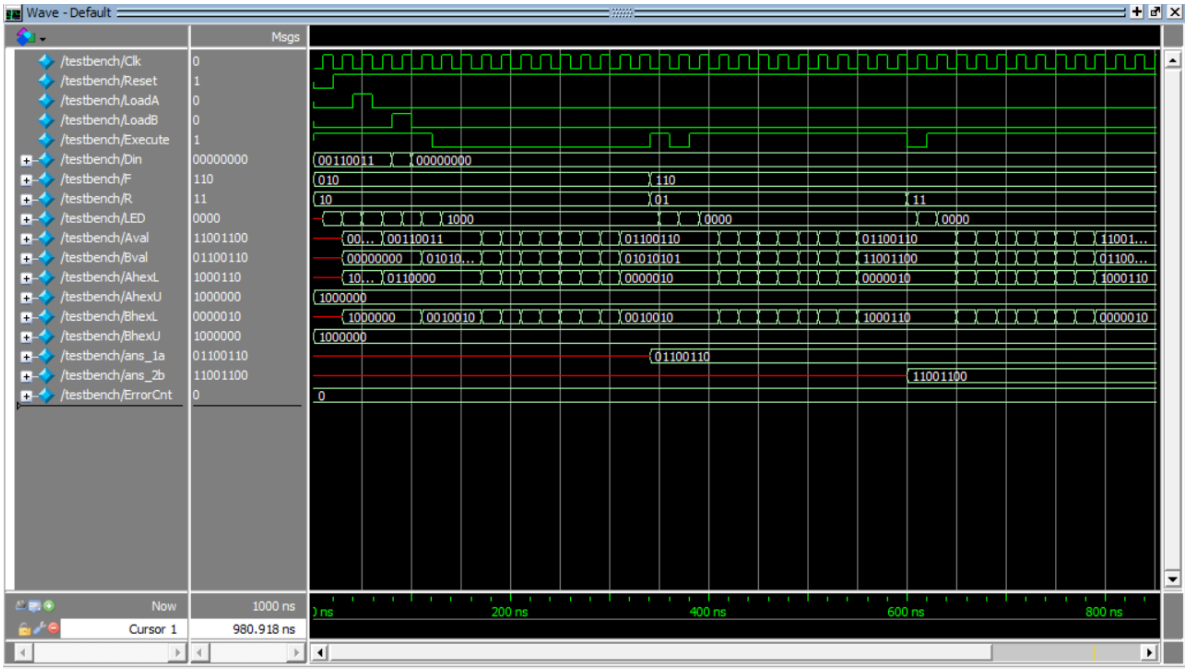
To create an 8-bit logic processor on an FPGA we built off the SV files of a 4-bit logic processor. The ‘processor’ file is the top level module used for both logic processors. This file calls the other files to do the actual logic processing. In order to account for the additional bits, we need to change Din, Aval, Bval, and Din_sync to be logic values that take in 8 bits instead of 4 bits. The ‘register_unit’ and ‘reg_4’ files are used to load registers when shift is enabled, both of which are set on the FPGA board. In order to allow the system to process 8 bits instead of 4 bits, we increased the size of the registers to fit 8 bits instead of 4 bits. The ‘computes’ file does the combinational logic based off of F values. Since there were not enough switches on the FPGA, the F values were initialized in the file. The ‘router’ file determines the outputs of each

registers depending on R. Similar to F, because there were not enough switches on the FPGA, R was initialized in the file. Both the ‘compute’ and ‘router’ files did not need to be updated when creating an 8-bit logic processor. The ‘control’ file keeps track of the states of the finite state machine and controls shifting. With 4 bit processing, only six states were needed. But, in 8 bit processing, an additional 4 shift states were implemented to account for the extra bits. The ‘hex driver’ files are used to display what’s stored in the registers. To account for an additional 4 bits, the input had to be changed to take in more bits and additional LEDs were utilized to represent the extra bits. The ‘synchronizers’ file was used to account for the asynchronous signals sent by the FPGA and synchronized the signals for smoother FPGA use. This file did not need to be updated to account for the additional bits.

The RTL block diagram is shown below. It should be noted that the hex drivers are not connected in the block diagram because the hex drivers were not required in the demonstration of the circuit.



Model Simulation



This simulation begins with Aval and Bval being set to 8'h33 and 8'h55, a short delay after loadA and loadB are flipped. A falling edge for 'execute' can be seen at 120ns and shortly after, Aval and Bval begin to change. This is the shifting as the new values are being shifted in. The new values once the shifting ends are Aval as 8'h66 and 8'h55. This is expected because you can see F was set to 3'b010, a XOR function as defined above, and R was set to 2'b10, where the function result will be pushed into Aval and the Bval will be pushed into itself. The program is tested again but, this time F was set to 3'b110, XNOR, while R was set to 2'b01, where Aval is pushed into Aval, and the function result is pushed into Bval. Execute is also set to have a falling edge while the program is running to test whether or not the program would still yield the correct results. After the 8 shifts have completed, we see Aval is 8'h66 and Bval is 8'hAA, which is what we expected from the F and R combination we inputted.

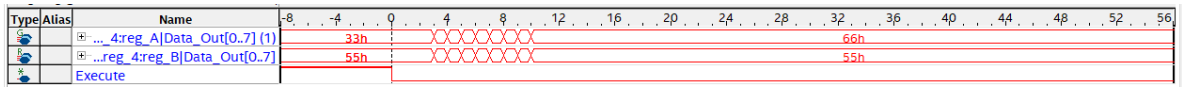
SignalTap trace procedure

Step 1: Go to the 'Processor' file and set F and R depending on the computation and routing you want (refer to operation of logic processor for controls). In this example F and R are set to 3'b010 and 2'b10 respectively in order to have register B stay the same and register A take on the value of A XOR B.

Step 2: Open SignalTap and select the desired device. Find and add reg_a|Data_out[0:7] and reg_b|Data_out[0:7] nodes as well as the Execute node. Make sure to group all the reg_a|Data_out values as well as all the reg_b|Data_out values.

Step 3: Program your FPGA on quartus and click the scan chain and autorun analysis on SignalTap. You can then input values onto the FPGA switches.

Step 4: Use switches[0:7] to input a bitwise value into the FPGA. Then switch 9 inputs the value into register A while switch 8 inputs the value into register B. Key 0 can be used to reset both registers and Key 1 can be used to execute the program. In this example we set register A to 8'h33 by flipping switches 0, 1, 4 and 5 and leaving them high before flipping switch 9 to load register A. We then set 8'h55 by flipping switches 0, 2, 4, and 6 and leaving them high before flipping switch 9 to load register B. Once key 1 is flipped the following result will be shown.



Before time 0 and execute if flipped, register A holds 8'h33 and register B holds 8'h55. After 'execute' is flipped, after a short delay, register A holds 8'h66 and B still holds 8'h55, which is what we chose when setting F and R values.

Description of all Bugs encountered and corrective measures taken

For the circuit in 2.1, we encountered one main bug, this being the fact that our computation and register unit were not adhering to the 4 clock-cycle shift rule designed in the lab. In other words, our operations were being performed when Execute was flipped on, but would cease being performed if and only if Execute was flipped low. This was not desired behavior so to figure out what was going wrong, we decided to test each individual module of the circuit to see which part was malfunctioning. Our computation unit performed well, as did our routing unit, but both our register and control unit had issues. Our register unit did not shift the correct number of times, even when connected to our counter. This was due to a misunderstanding as to the meaning of some of the pins on the bidirectional shift registers. It was corrected by creating logic for S1 and S0 pins on both registers. Still, though, we weren't able to get our counter to stop counting after 4 cycles. This was rectified by connecting the shift enable pins on the counter to our Shift signal. After all of these issues were resolved, our circuit worked as desired.

For the lab in 2.2, there were not any bugs that we encountered. Modifying the given four-bit processor and extending it to 8-bits was fairly straightforward, and the largest issue we encountered was getting our signal-tap to work on our DE-10 board. This wasn't really a bug, it just took some careful reading of the introduction to Quartus and the layout of the DE-10.

Conclusion

The purpose of lab 2 was to introduce us to a basic logic processor and its implementation with both TTL logic on a breadboard and System-Verilog programmed FPGA. The lab required a lot of design decisions on our part and forced us to constantly test our designs for functionality. By thinking in a modular way within each lab, we will be better prepared for the modular structure of System-Verilog programming for the future labs. Overall, this lab did a fantastic job acting as

an introduction to the ECE 385 lab experience.

Post-Lab Questions

Simplest 2-input 1-output circuit that can optionally invert a signal: The simplest such circuit is an XOR gate. If A and B are both inputs to an XOR gate with an output Z, then Z=A when B=0, and Z=A' when B=1. Thus, B is the select bit to choose whether to invert A or not. It is very clear how we could have implemented this in our circuit, although we did not choose to do so because we didn't think of it at the time. In the function select part of the circuit, the different functions are AND, OR, XOR, 1111 and their inverses. Therefore, by hooking up the output of a 4-input MUX to an XOR gate with a select bit F2, we could have been able to implement our function select part of the circuit with only a 4:1 MUX. This would have saved some logic and breadboard space. Unfortunately, that is not the implementation we chose.

Explain how a modular design such as that presented above improves testability and cuts down development time: Having a modular design is extremely important. When something goes wrong in a large circuit, it can be very challenging and overwhelming to find out what the problem is. A modular design allows the problem to be located faster by testing each module to find out which one(s) contain the problem. Also, when designing the circuit initially, it allows for parallel development. This worked well for our partnership, as we were able to develop different modules individually and then verify each other's work. This made our design and debug process take less time.

Discuss the design process of your state machine, what are the tradeoffs of a Mealy machine vs a Moore machine: Our design process for our state machine was heavily influenced by the given state machine in the lab document. This was the Mealy design, and when comparing it to the corresponding Moore design, we noticed that there were significantly less states in the Mealy machine. This design allowed us to use less memory units to store the current state, saving space and keeping our breadboard organized. However, the simplicity of implementing a Moore machine would have been easier. One of the issues we had was the use of the counter to count the number of times we shifted and operated on the bits. This would not have been necessary with the Moore machine design. So, while we did save some space and memory, we also probably lost some design time to the Mealy machine.

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: The are a few key differences between ModelSim and SignalTap, the most important being that ModelSim is a simulation of the System-Verilog code written in Quartus, whereas Signal Tap is an implementation of that on the Breadboard. Because of this, we chose to use ModelSim initially to test our code and make sure that it worked, and then test using SignalTap. This way, we were able to make sure that our code was correct before adding in the variable of pin assignments and grouping on SignalTap. If a problem had occurred in SignalTap, we would have been able to rule out incorrect coding as one of the possible sources.