Lab Report #2

Aryan Shah (aryans5), Dev Patel (devdp2)

University of Illinois at Urbana Champaign

ECE 385 - Digital Systems Laboratory

Professor Zuofu Cheng, T.A. Gene Lee

September 17, 2023

Introduction

The purpose of this lab was to implement a bit-serial logic operation processor. The circuit contains 2 stored register values, each with a 4-bit input. The 8 operations that can be performed include AND, OR, XOR, NAND, NOR, XNOR, a logical high, and a logical low. The circuit will then route the operations in 4 distinct ways, which include maintaining both initial register values, replacing the final operation value's in either register, and swapping the register's initial values. All this logic was made possible by the design of a register unit, computational unit, routing unit, and a control unit, all of which will be further described below.

Operation of the Logic Processor

Sequence of switches the user must flip to load data into the A and B registers

Our physical circuit made use of manual switches to input into our 2 4-bit registers, given the names 'A' and 'B'. Our switches, labeled 'Load A' and 'Load B', were responsible for inputting the binary 1s and 0s into our target register. In essence, once a user inputted the necessary 4-bit value by using our switches, they then toggled a Load A or Load B switch to incorporate it into its respective register. The 4-bit value was loaded in with the use of 4 other switches, labeled D0, D1, D2, and D3. A user would perform the following steps: if they were to input the value '1001' into register A, they would first toggle the 'D#' switches, and then flip the Load A switch to store it into the register. Then, if they were to input the value '0100' into register B, they would do the same procedure by toggling the 'D#' switches and then flipping the Load B switch to store it into its register, leaving register A untouched.

Sequence of switches the user must flip to initiate a computation and routing operation

The computation unit was intended to handle the 8 bitwise operators: AND, OR, XOR, NAND, NOR, XNOR, a logical high, and logical low. To signify which operator was meant to be chosen, a series of 3 switches labeled F0, F1, and F2 were included in our switchbox (F0 as our least significant bit and F2 as our most significant). The function select inputs were detailed in Table 1 of our Experiment 2 document: 000, 001, 010, 011, 100, 101, 110, and 111 translated to AND, OR, XOR, logical high (1111), NAND, NOR, XNOR, logical low (0000) respectively. Our routing unit controlled the way in which the final computation was stored back into our registers: maintaining the initial register values, storing the computed value in either A or B, and swapping the values. To specify this, another series of 2 switches labeled R0 and R1 (R0 as our least significant bit and R1 as our most significant) were also included in our switchbox. The routing selection was also detailed in the same Table 1 of our Experiment 2 document: 00, 01, 10, and 11 translated to maintaining our initial values (A* to A and B* to B), loading our computed value into register B and keeping A's value the same, loading our computed value into register A and keeping B's value the same, and finally swapping the 2 initial values (A* to B and B* to A), respectively. After loading the values into the register unit, the computation and routing operations must also be specified with these switches prior to flipping the execute switch.

Written general description, block diagram, and state machine diagram of logic processor

Written general description for the register unit

The register unit was designed to hold the 4-bit values we wished to input into registers A and B and store them for the next steps. The unit could also shift bits according to our control unit's logic operations, as to indicate the resulting values in our register after the necessary computations were complete.

Written general description for the computation unit

The computation unit was responsible for the AND, OR, XOR, NAND, NOR, XNOR, logical high, and logical low operations. A set of switches was used to indicate the bitwise operations that were to be performed and the resulting values were stored back into the register following the routing unit.

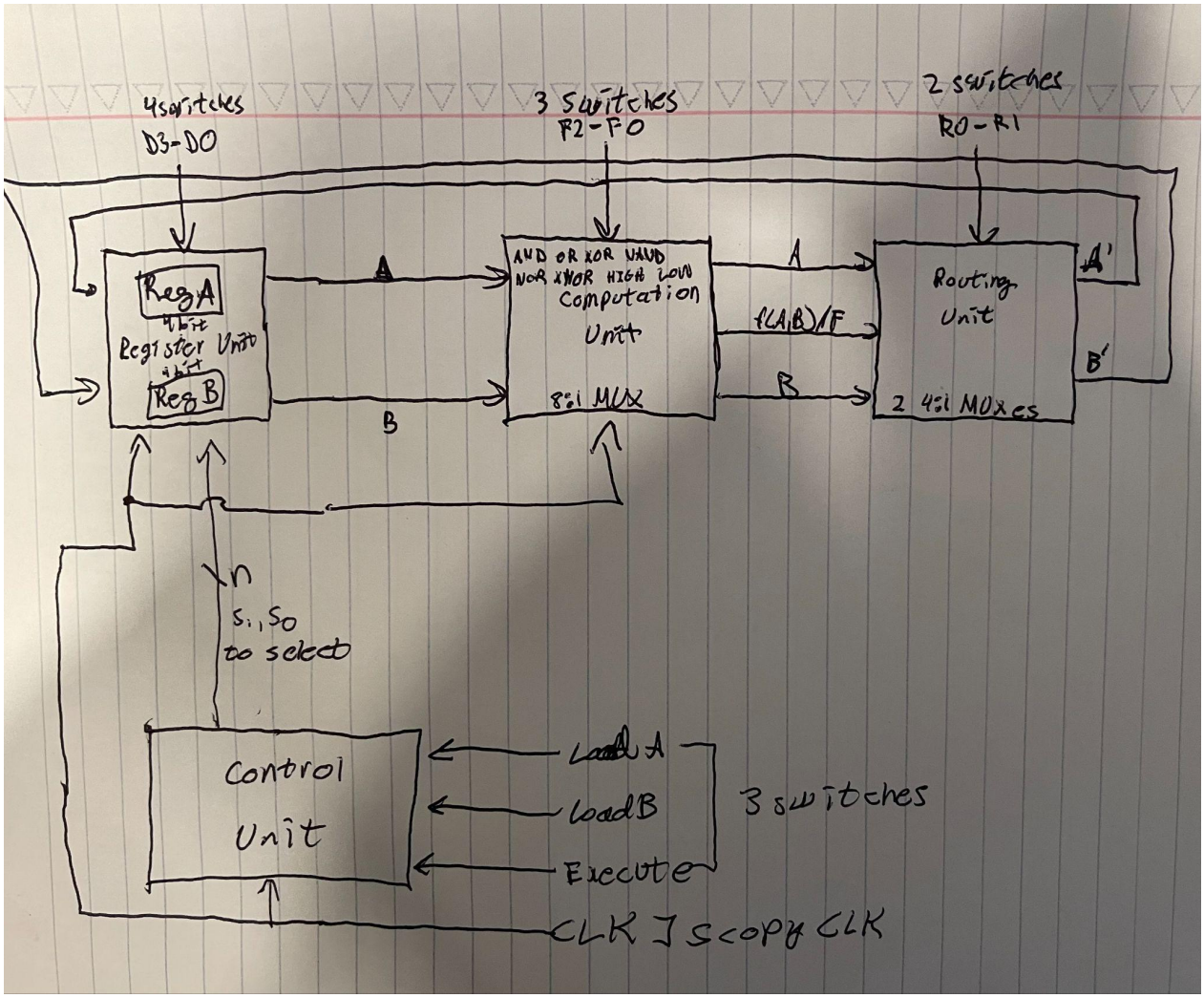
Written general description for the routing unit

The routing unit was responsible for the 4 ways that the values could be stored back into our registers A and B. These included maintaining the initial values, swapping the initial values, or storing the computation unit's final values in either register A or B. For its design, we made use of 2 4-1 multiplexers. 2 select bits determined by our switches handled the logic to determine which values should be stored and where.

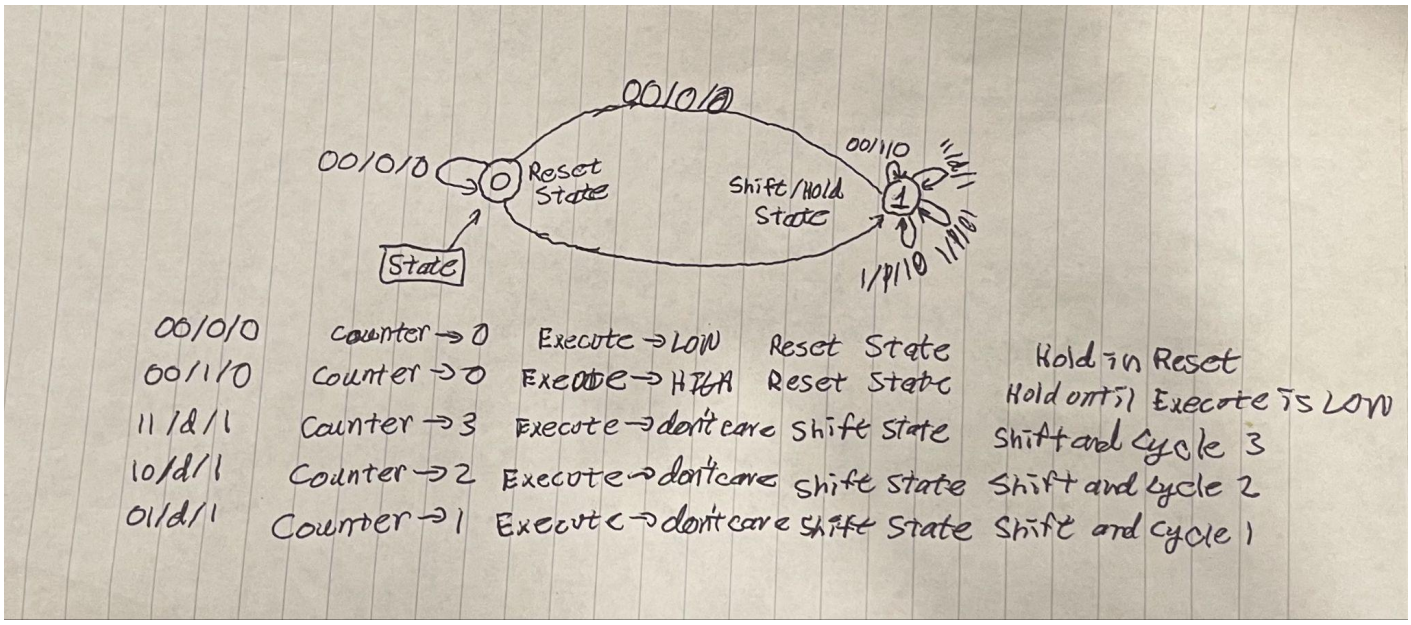
Written general description for the control unit

The control unit is responsible for the actual computation of the operations specified by the computational unit. As specified in the lab, we made use of the most appropriate state machine for this experiment: a Mealy Machine. Our outputs depended on the execute switch E, single-bit state Q (reset/hold), and our 2-bit count C1C0 (number of shifts to be executed).

High-level block diagram (can be used from lab manual if modified)



State Machine Diagram



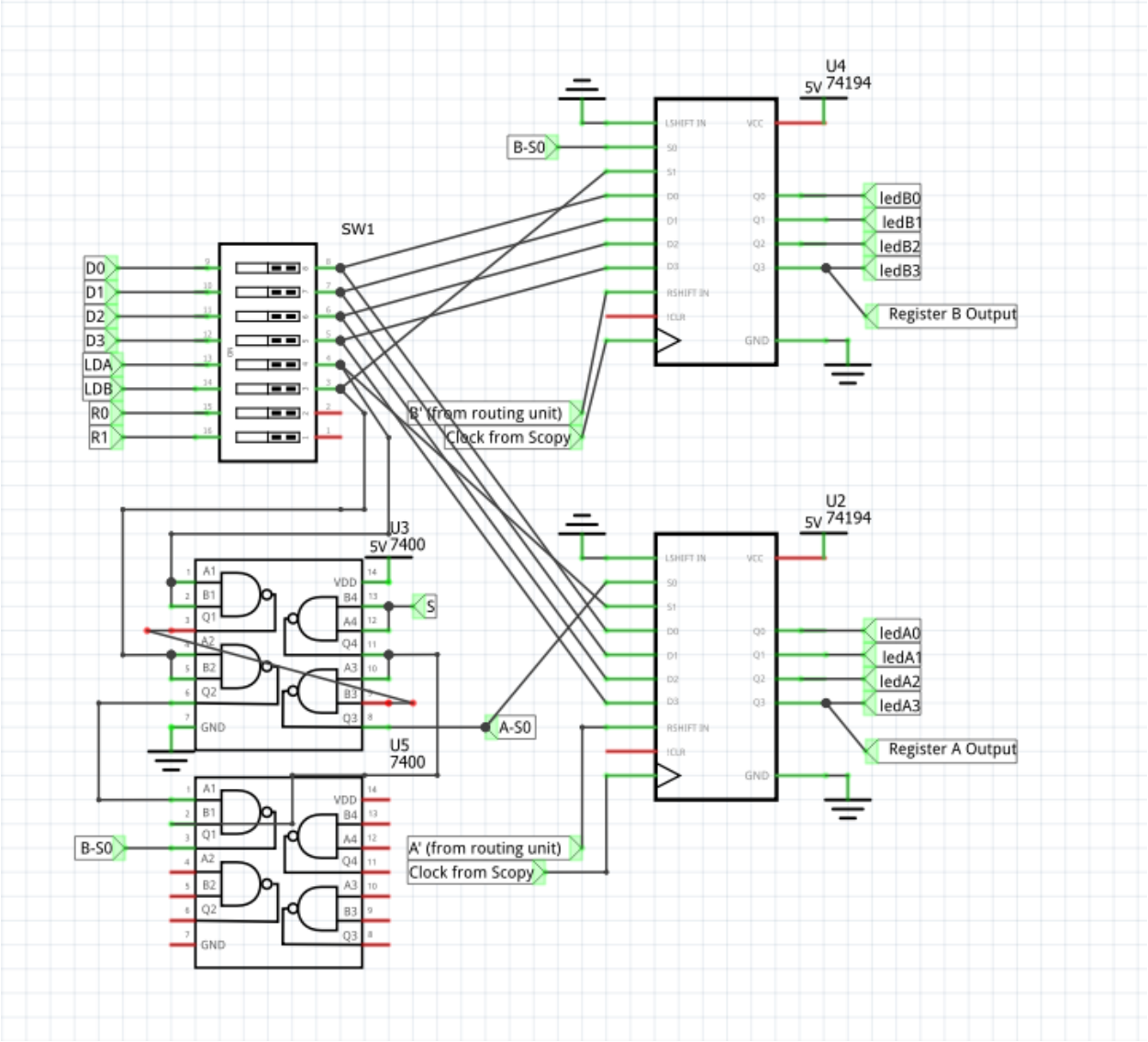
Mealy State Machine

The diagram above depicts the Mealy State Machine we used to drive our design for the control unit. The FSM depends upon the execute switch E, single-bit state Q, and 2-bit count C1C0. The circuit operates by starting in a reset state and will start shifting bits once the execute switch is triggered. Once triggered, there are 4 shifts of the initial register values that are then replaced with their new values. During the shifting process, our execute bit will be don't care so that the shifting will not halt in the middle. Once the process is complete, our execute switch turns off to signify we will be back in our reset switch, ready to start the process again.

Design Steps Taken and Detailed Circuit Schematic Diagram

Written procedure of the design steps taken

Register Unit

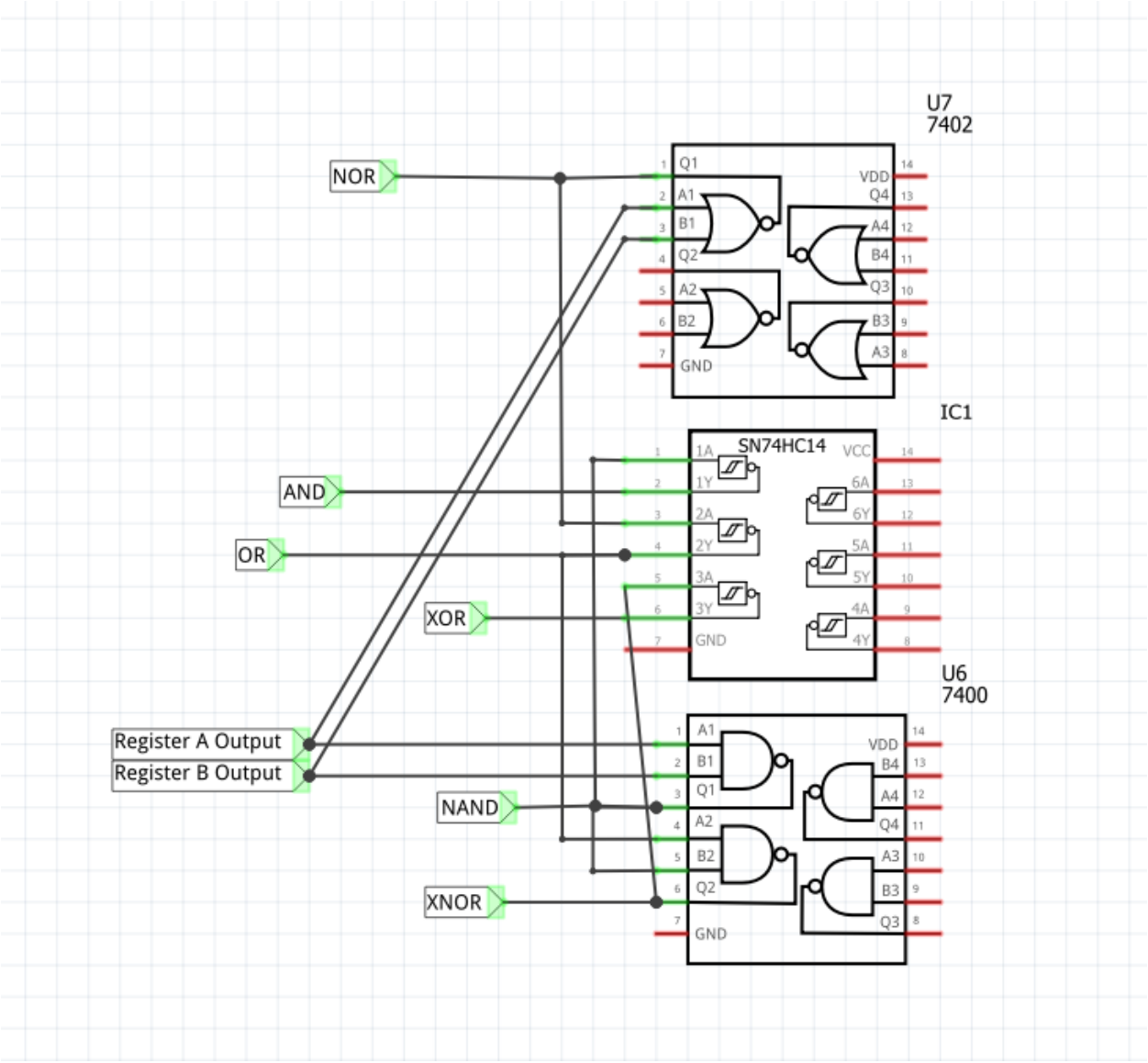


The chip we used to design our register was the provided CD74HC194E, a 4-bit bidirectional shift register. We used 2 of these chips, one for register A and one for register B. The D0, D1, D2, and D3 pins of the chip are connected to switches for both registers A and B. The signal Q3 serves as the output and will then be sent into the

computational unit as its input. The output of our routing unit was used as our DSR pin so that it could detect the new values that will be displayed on the registers. The S1 and S0 select pins transition between the shift/hold processes of the registers and are handled by the control unit. The diagram below outlines the shift register's chip operation that we used to direct our logic for S1 and S0 shifting. S1 comes from the LoadA or LoadB switch, while S0_A is LoadA OR S, and S0_B is LoadB OR S. This is also reflected in the gate level schematic above.

OPERATING MODE	INPUTS							OUTPUT			
	CP	$\overline{MR}$	S1	S0	DSR	DSL	D_n	Q_0	Q_1	Q_2	Q_3
Reset (Clear)	X	L	X	X	X	X	X	L	L	L	L
Hold (Do Nothing)	X	H	l	l	X	X	X	q_0	q_1	q_2	q_3
Shift Left	↑	H	h	l	X	l	X	q_1	q_2	q_3	L
	↑	H	h	l	X	h	X	q_1	q_2	q_3	H
Shift Right	↑	H	l	h	l	X	X	L	q_0	q_1	q_2
	↑	H	l	h	h	X	X	H	q_0	q_1	q_2
Parallel Load	↑	H	h	h	X	X	d_n	d_0	d_1	d_2	d_3

Computation Unit

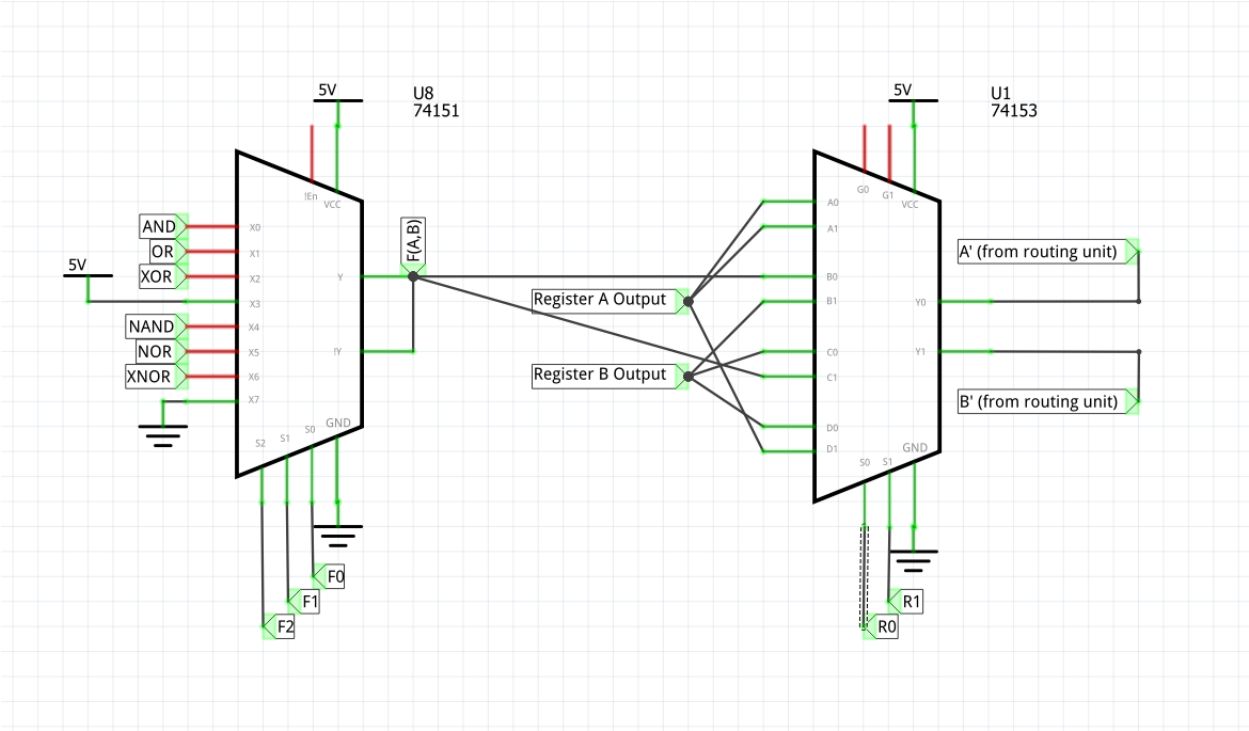


The computation unit was responsible for handling the AND, OR, XOR, NAND, NOR, XNOR, logical high, and logical low operations. However, the provided contents of our lab kits only contained NAND and NOR logic gates along with hex inverters. Rather than use multiple NAND gates to invert the outputs for intermediary operations, we utilized the hex inverters to simplify the design of our computational unit. The Quad NAND CD74AC00E, Quad NOR CD74AC00E, and Hex Inverter CD74AC04E chips

were used for our design. Additionally, we included an 8-to-1 multiplexer (SN74HC151N chip) to make it so that only 1 operation would be computed and outputted. The truth table below shows the different combination of selection inputs routed to the separate operations we were to complete as proposed by the experiment document.

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

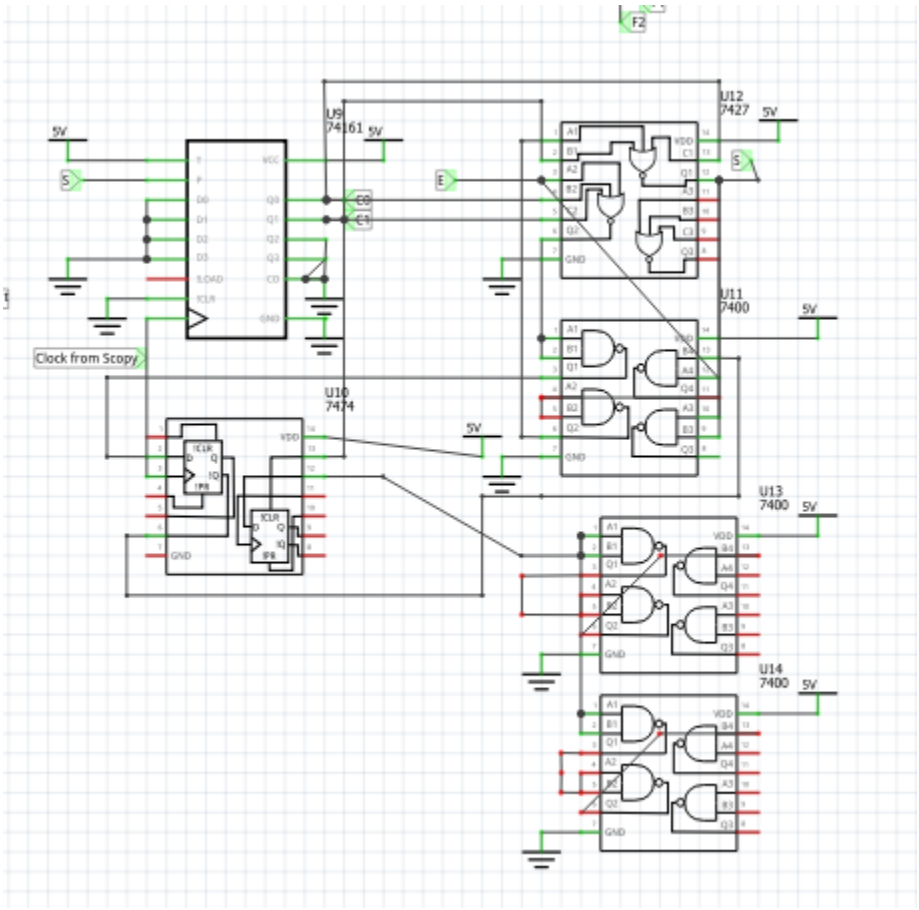
Routing Unit



The purpose of the routing unit was to indicate the final values that were to be stored in registers A and B following the computation process. These values include maintaining A and B's initial values, storing the computation result in either A or B, or swapping the initial values of A and B. The value 'F' stands for function and stores the value of the output of one of the specified 8 operations. In our design, we included the SN74HC153N Dual 4:1 MUX. R1 and R0 are our select inputs that control the value of the next states for A and B (A* and B*). To translate this both MUXes (2 4-to-1 multiplexers), R1 and R0 were switches and our inputs were A, B, and F (from our computation unit). The diagram below is our truth table taken from the experiment document. The 8-1 MUX shown in the above diagram collects the correct output of the computation unit based on the values of F2,F1,and F0.

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

Control Unit

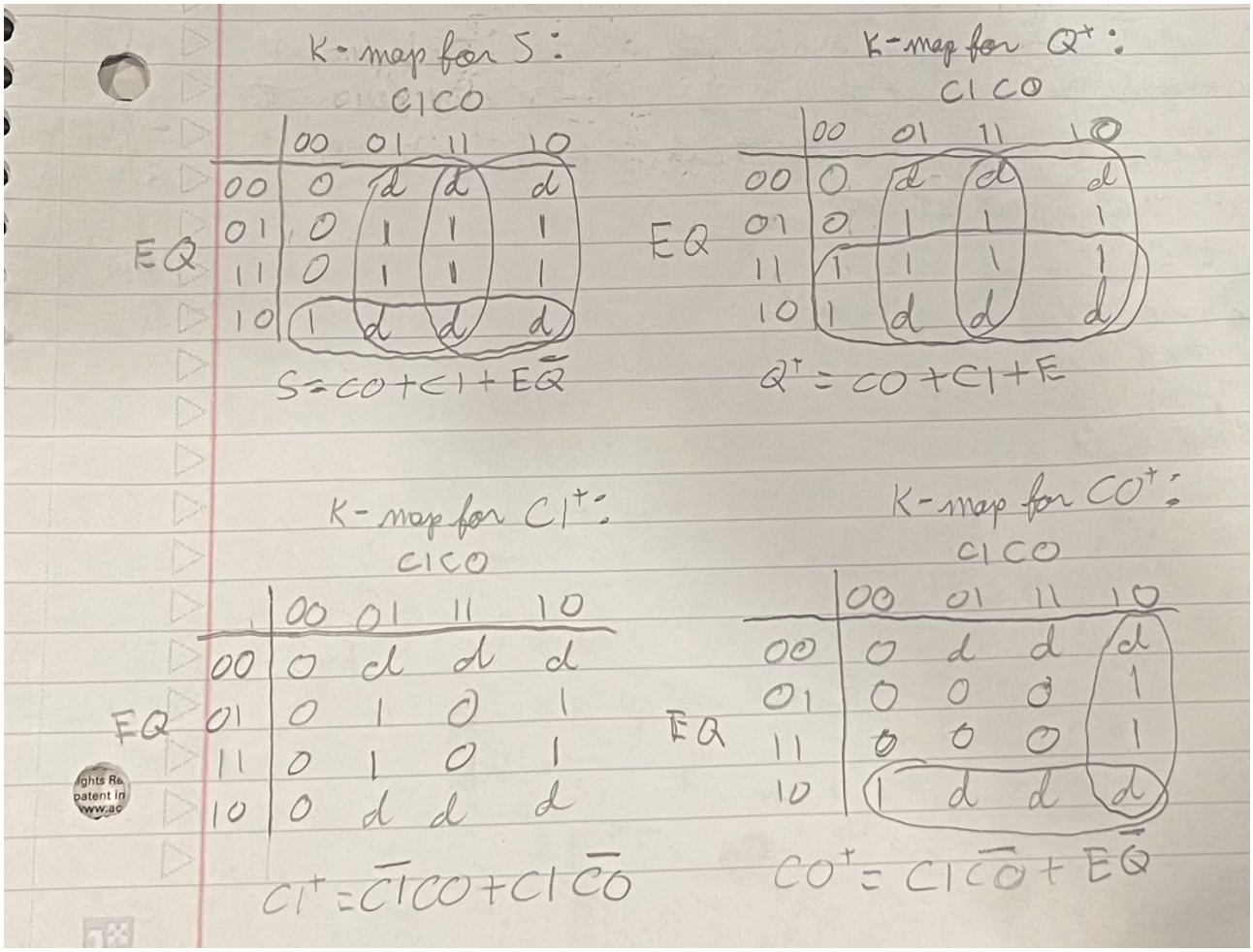


The control unit was designed to manage the shifting, loading, and computation of the values stored in our registers. It started in the reset state, handled by flipping an execute

switch, indicating to the circuit that the cycle of its operation should begin, shifting the computed 4 bits into its intended registers. After the execute switch is then flipped to indicate that the operation is complete, it then transitions back into its reset state ready to start the process again. We began by analyzing the logic outlined in our provided truth table. From this truth table, we created K-maps to find minimal SOP expressions for S, Q+, C1+, and C0+. The value of S controls whether the circuit should remain in a shift (1) or hold (0) state. It also directs the value of S1 and S0 for each register.

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

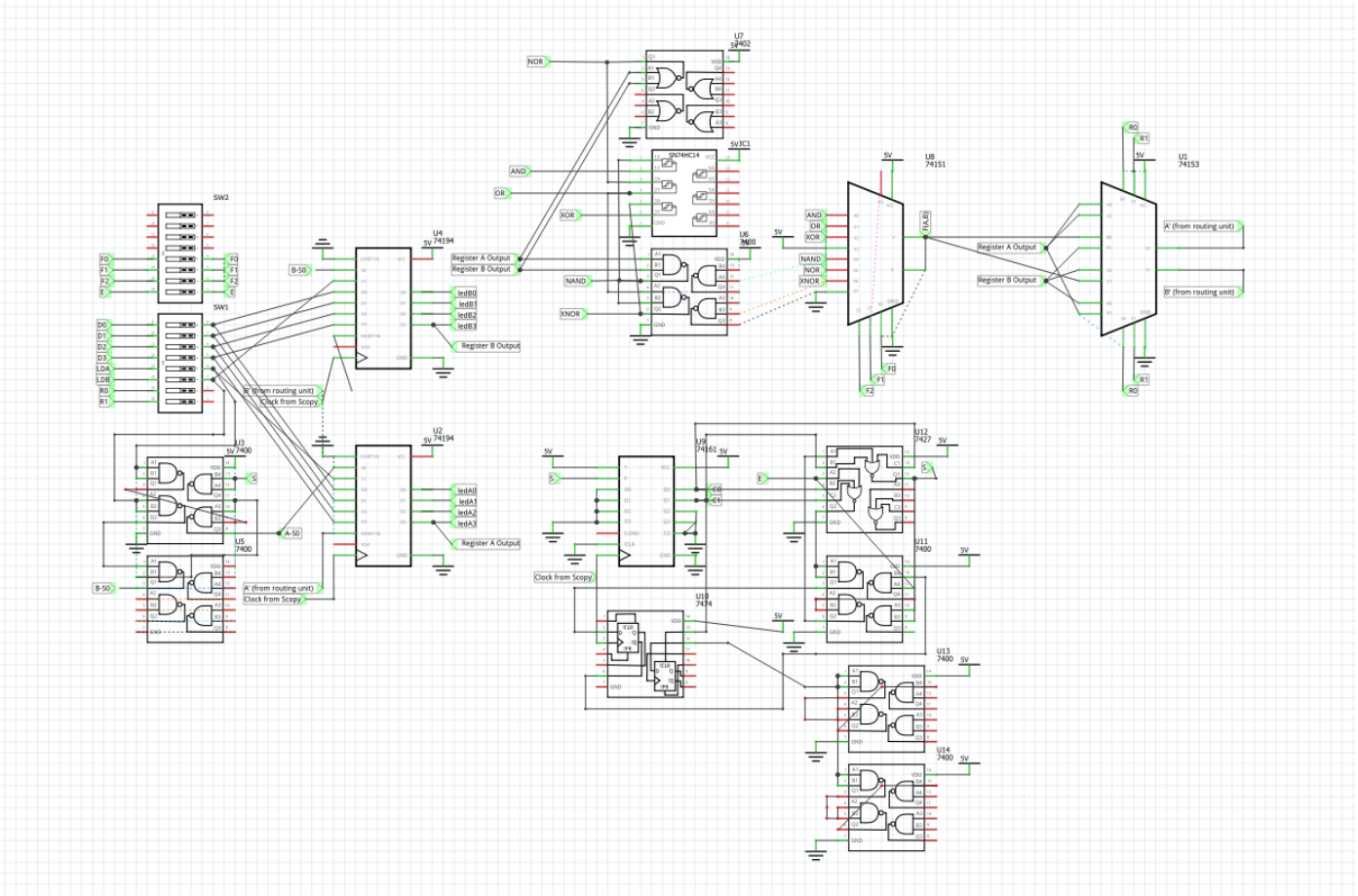
Mealy FSM Truth Table



K-maps for S, Q+, C1+, C0+

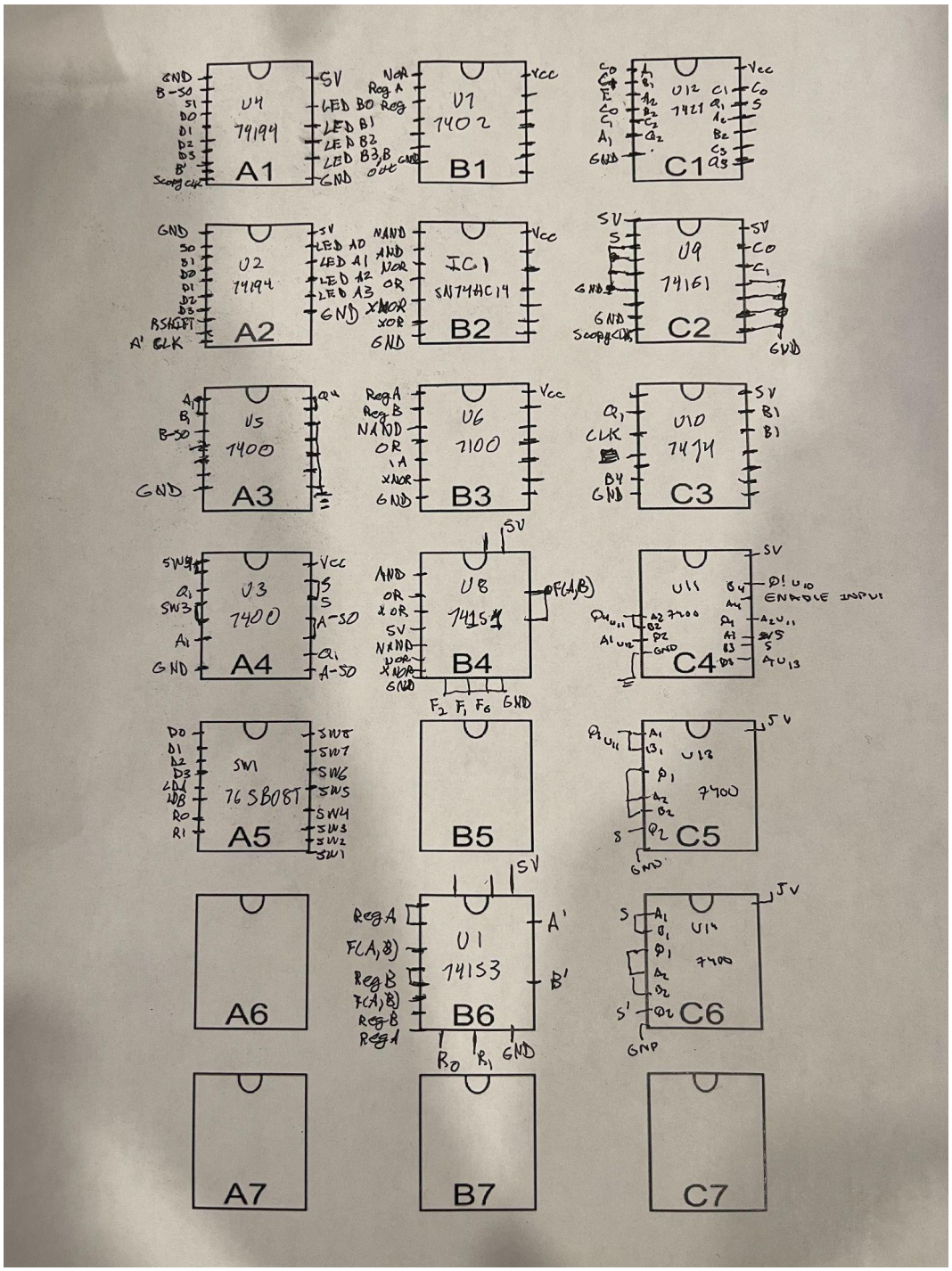
Detailed Circuit Schematic

Gate level schematic (must use a CAD tool)



Breadboard View/Layout Sheet

Use either the manual layout sheet or the Fritzing breadboard view (page 2.11 explains rules in more detail



8-bit logic processor on FPGA

Summary of all .SV modules and the changes made to extend the processor to 8-bits. Describe even provided modules that you did not create.

Compute.sv - We did not make any changes to this file. Its intent was to select the logic operation for the computation unit.

Control.sv - This file was modified so that we can increase the number of states. By increasing the number of bits to 8, we had to add 4 additional states to account for the 4 new bits.

HexDriver.sv - No changes were made to this file. Its intent is to program the 4-digit hex display on our FPGA board.

Processor.sv - We had to expand ‘Din’, ‘Aval’, and ‘Bval’ from 4 bits to 8 bits. We also had to add more hex drivers to view the upper nibbles of our registers.

Reg_4.sv - To modify this file for our 8-bit logic processor, we had to extend ‘D’ to 8 bits from 4. Additionally, we had to modify ‘Data_Out’ to contain 2 hexadecimal numbers, which translates to 8 binary bits. The size of our ‘Data_out’ was changed to 7 to account for the bit shifting (once a bit is shifted, another one takes its place so we would need 1 bit less than the 8 bits held in our register).

Register_unit.sv - We had to modify this file to allow for ‘D’, ‘A’, and ‘B’ to each contain 8 bits rather than 4. Essentially, we made it so that our logic processor now has 8-bit inputs and outputs rather than 4.

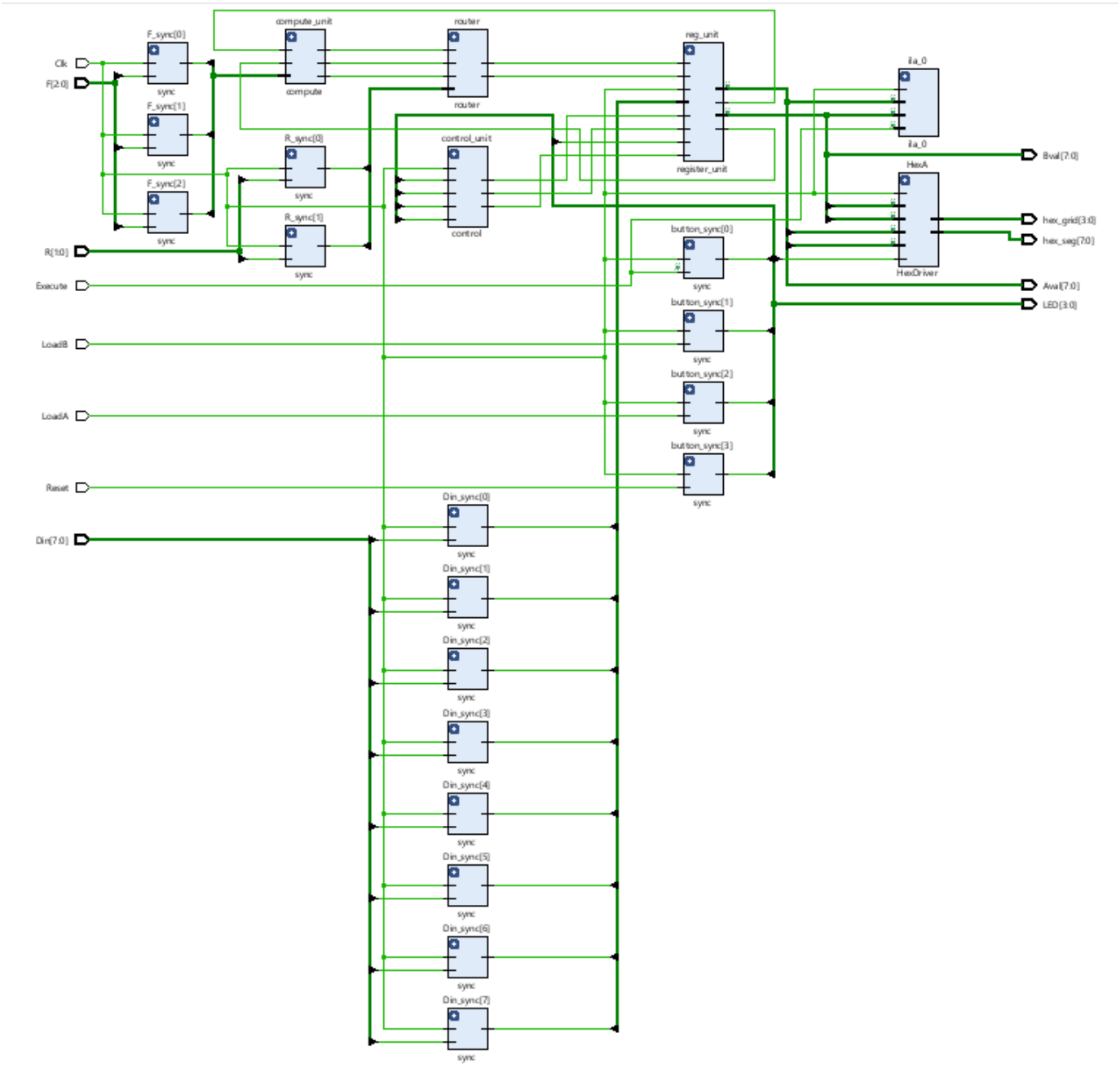
Router.sv - No changes were made to this file. Its intent is to determine what values will be routed back into our registers A and B.

Synchronizers.sv - No changes were made to this file. It is intended to allow for asynchronous signals into our FPGA and synchronize them with its built-in clock.

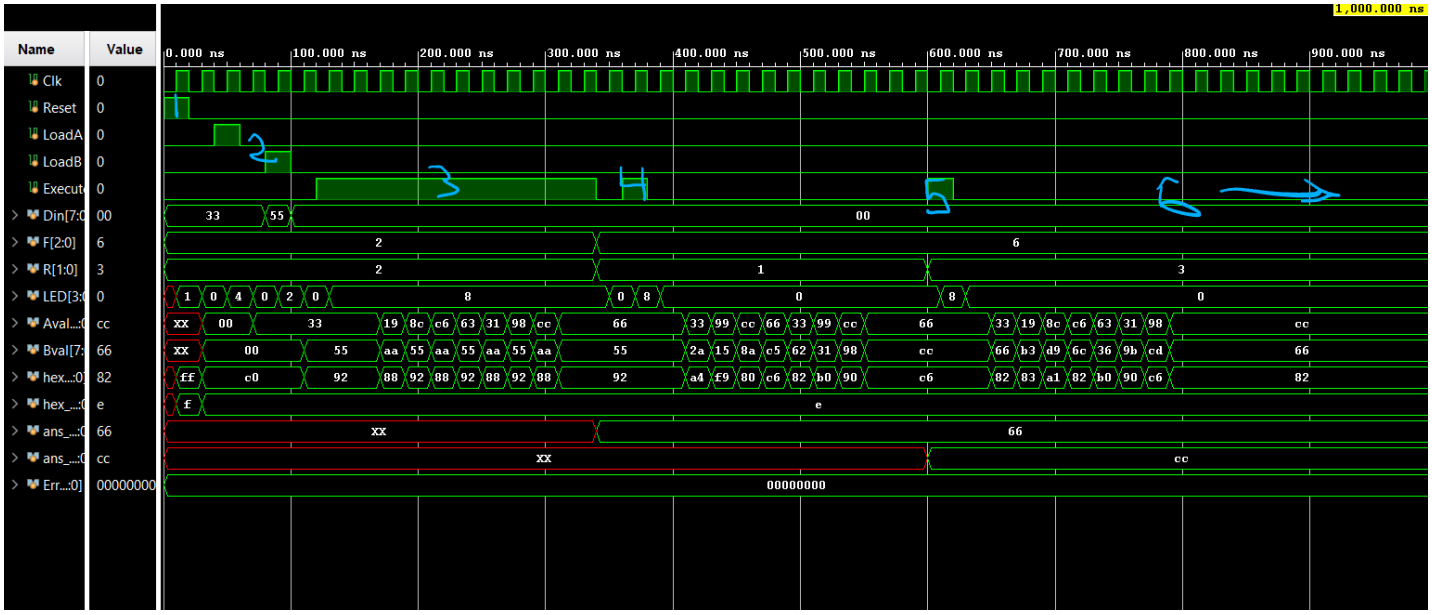
Testbench.sv - No changes were made to this file. It serves as a test bench when running the behavioral simulation for our 4-bit logic processor.

Testbench_8.sv - No changes were made to this file. It serves as a test bench when running the behavioral simulation for our extended 8-bit logic processor.

RTL Block Diagram



Simulation of the processor that is annotated with information such as what operation is being performed, where the result was stored, etc.



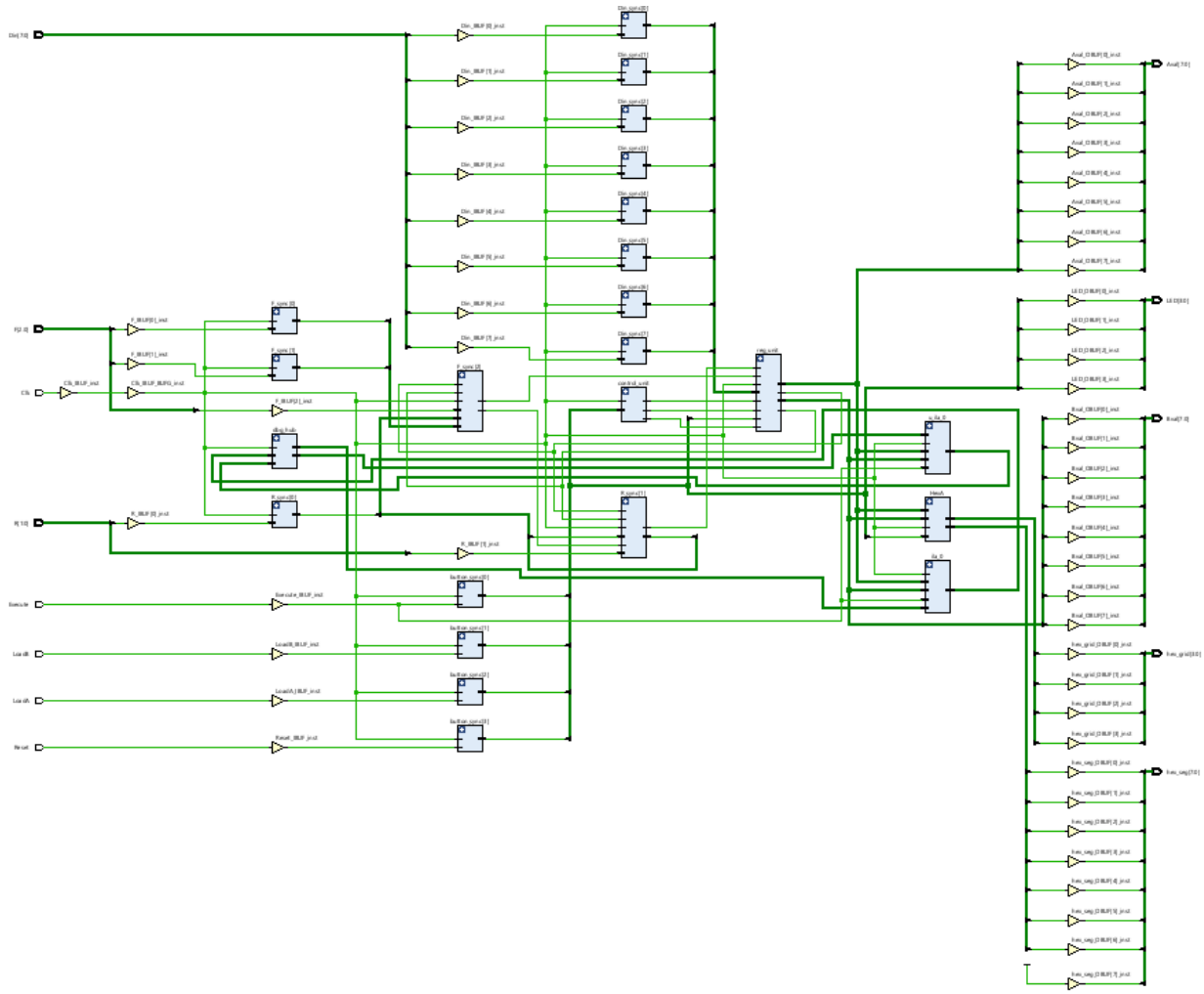
1) **Step 1**
    a) Reset signal set to 1
    b) Load A, Load B, and Execute set to 0
    c) Initialize Din, F, and R
2) Step 2
    a) Reset toggled to 0 after a delay of 2 time cycles
3) Step 3
    a) Load A toggled on and off after 2 time cycles
    b) Load B toggled on and off after 2 time cycles
    c) Din changed to hex 55 during Load B's toggle, and then back to hex 0
4) Step 4
    a) Execute toggled on and off after 2 time cycles, first execute cycle
    b) Aval computed as hex 33 XOR hex 55
    c) Bval stays as hex 55
5) Step 5

a) Execute toggled on and off after a 2 time cycles, second execute cycle
b) Aval stays the same
c) Bval is the output of cycle 1 XNOR hex 55

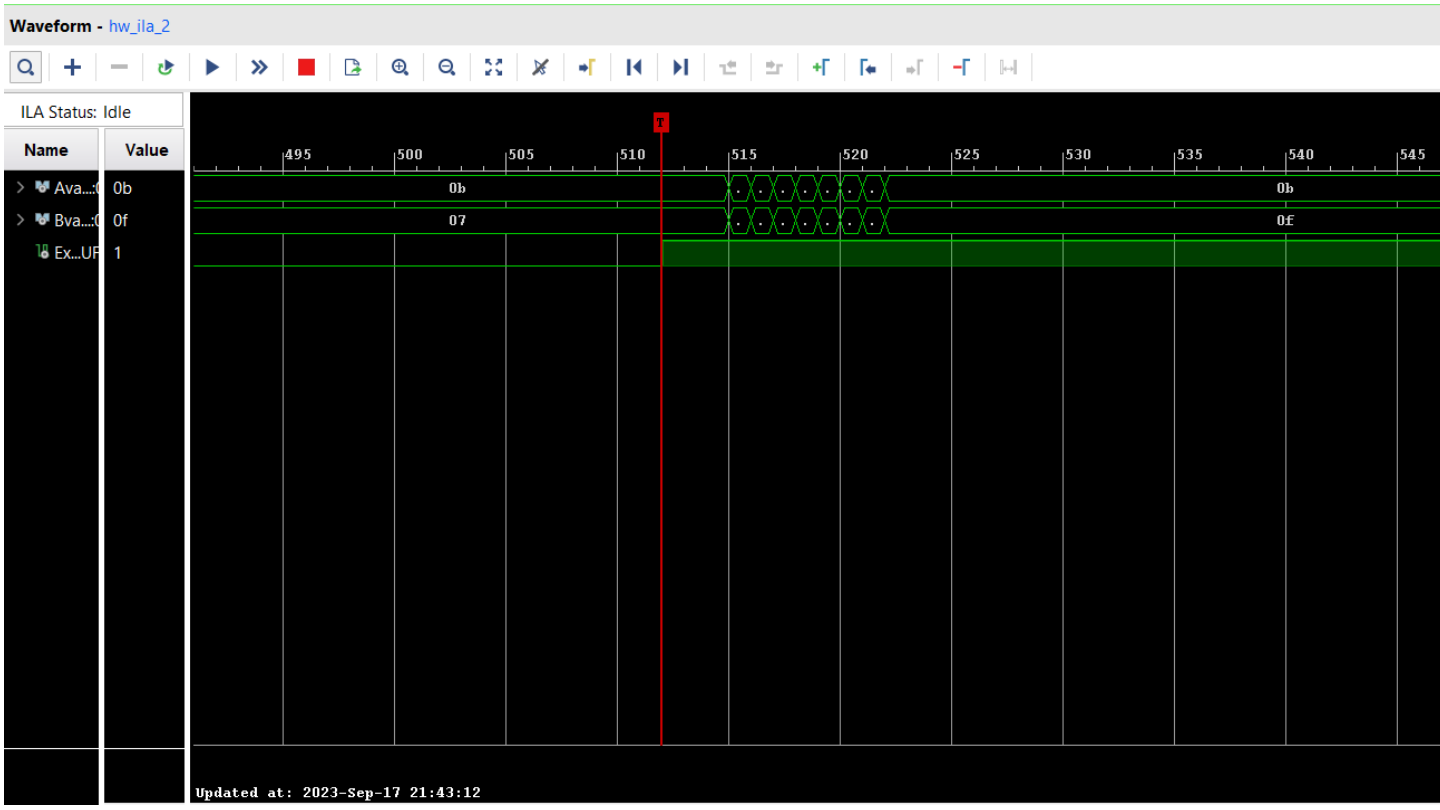
6) Step 6
    a) R is updated
    b) Execute toggles on and off after 2 time cycles
    c) Verify that Aval and Bval are swapped

Procedure to generate Vivado Debug Core trace as well as the result of such trace executing an example operation in steps

Under synthesis, you click set up debug. You then select the nets you want to debug (in our case, data for Aval and Bval, and a data and trigger for execute). Once you save the file, you can then open the elaborated design and run synthesis again. Under the synthesis schematic, you should see the ILA block created. You then modify the Processor.sv file to capture the outputs of Aval and Bval, and then run implementation again to see the changes in your ILA block. While running an operation on the FPGA, you can delay the waveform to capture its second half only after the trigger has been executed. You can then analyze the changes between Aval and Bval to ensure that they were transitioning as expected in the waveform.



Include output of the debug core trace performing an operation on the 8-bit logic processor



Description of all bugs encountered, and corrective measures taken

With the detailed description of our lab documentation, there were a minimal amount of bugs that we encountered. When it came to our circuit design, most of our issues came during the debugging stage. After realizing the complexity of our design and how it would translate onto the breadboard, we devised a color-coding and labeling scheme for certain parts of our design (e.x. same signals would have the same color and be taped together with a label, wires tracing within a unit would have the same color). This proved very useful, as debugging came much easier whereas previously it came to the point where we could not distinguish which wires handled what part of our design and thus an entire unit would have to be disassembled. When it came to

utilizing the Xilinx Vivado software, our testbench and the built-in TCL console proved useful in handling certain errors. We could detect syntax issues and simulate to ensure that our signals were holding and transitioning to intended values.

Conclusion

Re-summarization of the lab

Our intent with the lab was to design an 8-bit processor to handle 8 bitwise operations on 2 inputted 4-bit values. A register unit capable of holding and storing the 2 4-bit values, a routing unit to handle the 4 ways our result could be outputted, a computational unit to indicate the bitwise operations, and a control unit to perform the operations were all designed in order to achieve this. The logic was processed through the design of a Mealy State Machine, controlling the various states and timing of the operations. K-maps and a truth table were devised to translate the operations into those that could be handled using as minimal chips as possible from our provided toolkit. Following the design of our circuit, we sought to also implement this onto our provided FPGA. Using existing code for a 4-bit serial logic processor, we extended this to 8-bits and developed and simulated SystemVerilog code on Xilinx's Vivado software, building upon our physical breadboard design. Finally, we were able to execute tests by validating simulations against a testbench and performing tests ourselves and analyzing the results.

Answers to all post-lab questions

Describe the simplest (two-input one-output) circuit that can optionally invert a signal (i.e., one input determines if the output is equal to the other input or equal to the other input inverted). Explain why this is useful for the construction of this lab.

The simplest two-input one-output circuit that can optionally invert a signal is an XOR gate. One input would be your chosen value, and the second input would control whether or not the output should be inverted or not. This would be controlled by setting it high if you want your input to be inverted or setting it low if you want to leave it the same. This helped during the designing of our lab due to the fact that our first 4 operations (NAND, NOR, XNOR, logical high) were all the inverse of our last 4 operations (AND, OR, XOR, logical low). Using an XOR gate to simply drive the first 4 operations and optionally invert them saves us from a lot of extra designing for the second 4 operations, taking up less space on the breadboard, less chips, and making it easier to debug.

Explain how a modular design such as that presented above improves testability and cuts down development time

By splitting our lab into different units, thereby making it modular, allows for making our circuit more organized and therefore easier to analyze and debug. By separating the components on the breadboard, we can organize signals and logic within units and debug them separately. Instead of analyzing everything as a whole, there are 4 main origins where issues can be directed to. This makes the process of building, testing, and debugging is much simpler.

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

The logic of the FSM came from the given documentation for our control unit. It was devised with the idea that the execute switch shifts 4 bits only when flipped, and held in reset states when not triggered. A choice was to be made whether to utilize a Mealy or a Moore Machine. In Mealy machines, outputs are dependent on current states and its inputs. This allows for fewer states and faster output response. On the other hand, with Moore Machines, outputs only depend upon the states. This allows for better output stability and a simpler design in certain circumstances. However, in our case, since our logic requires the use of a shift and halt state, we would thus be able to make our design with fewer states using a Mealy machine.

What are the differences between vSim and Vivado Debug Cores? Although both systems generate waveforms, what situations might vSim be preferred and where might debug cores be more appropriate?

vSim allows us to test the behavior of our design at a functional level by simulating our RTL code. You can program different input conditions and analyze its response at each clock cycle. In this lab, we used it within our behavioral simulation to test it against our testbench for the 8-bit serial logic processor. On the other hand, debug cores allow us to monitor signals in real-time while the RTL code was running on the FPGA hardware.

This is also beneficial for debugging as we can see issues that may not arise in our software simulation. Utilizing vSim would be most beneficial during the early stages of the design process, to validate your RTL code prior to synthesizing and implementing the entire design. The Debug Cores, on the other hand, would be best utilized after the design has been synthesized and implemented. You can thus test timing and accuracy in real-time across all the behaviors that you expect and can be confident that the design works as intended.

Any parts of the documentation which were unclear or otherwise need attention

N/A, documentation explained the lab thoroughly.