Zhihao Wang (zhihaow6), Laurenz Nava (lfnava2)

Introduction

We built a simple calculator supporting eight kinds of logic operations from two registers, A and B. The computing unit side it can do, AND, OR, NOR, XOR , NAND, XNOR, 0000, 1111 output, and using a routing unit gives us choice how to store the original A, B and commuted output getting from computing unit back to the A and B registers. It can operate two 4-bit inputs and compute 4-bit outputs, while the SV code in Vivado could support up to 8-bits.

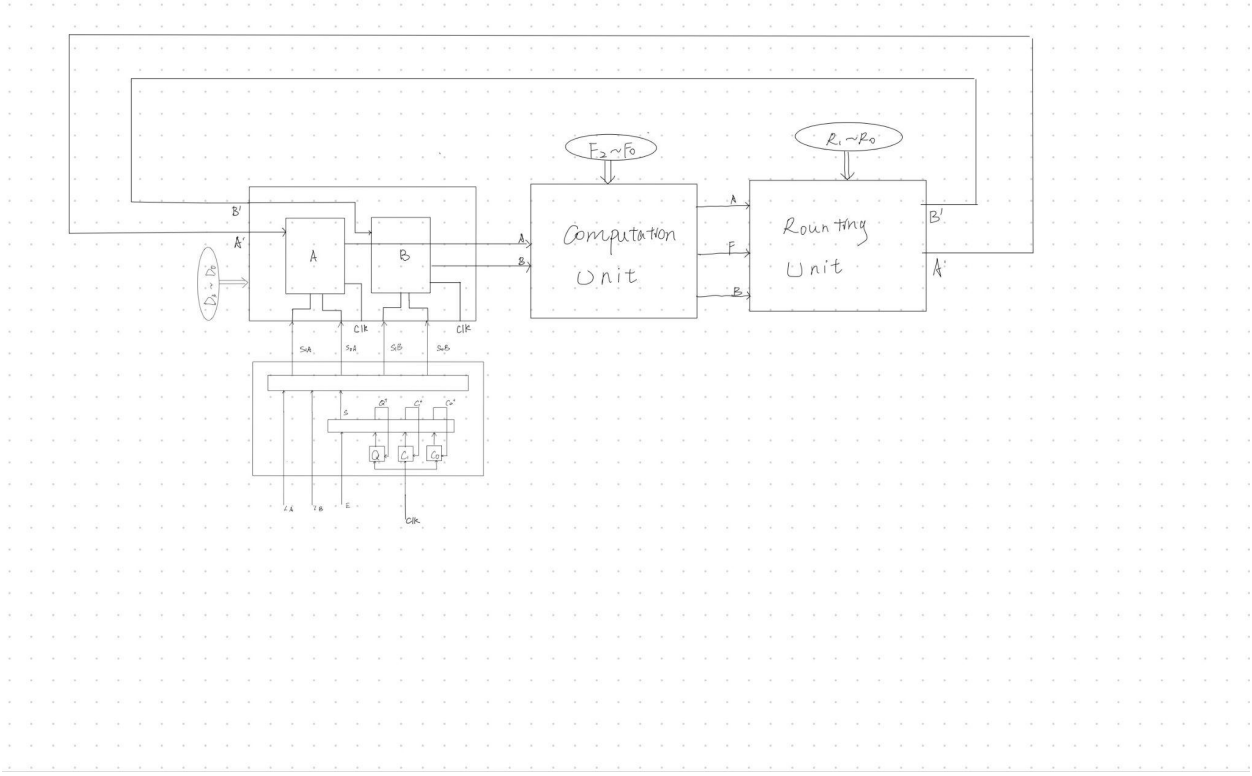
Operation of Logic Processor

a. Use switches to indicate 4-bit input of A, and then press the loadA button to load A. Use switches to indicate 4-bit input of B, and then press the loadB button to load B.
b. Use switches to indicate the computing kinds, ‘000’ for AND, ‘001’ for OR, ‘010’ for XOR, ‘011’ for 1111, ‘100 for NAND’, ‘101’ for NOR, ‘110’ for XNOR, ‘111’ for 0000. Use switches to indicate the storing logic, ‘00’ for storing Original A to A, Original B to B, ‘01’ for storing Original A to A, computed result to B, ‘10’ for storing computed result to A, Original B to B, and ‘11’ for storing Original B to A, and Original A to B.

Description, Block Diagram, State Machine

a. Register unit will load the initial value from the switch when the corresponding load button is pressed. The register unit is a shift register, which will output one bit and accept one bit in each clock cycle. The behavior of the register, whether to clear itself or store the incoming bit, is controlled by S1, S0 which is from the control unit. The computation unit will accept two bits from A and B each clock cycle and do corresponding operations indicated by F2-F0. The routing unit will choose which of the output back to the register, that is, which two of three values, operational output, Original A bit or Original B bit. For the control unit, it will contain three flip-flops which indicates the internal state. Each time it will use the outside signal E, and internal flip-flop output, Q, C1, C0 to compute the control signal, which is S(Shift), C+, C1+, C0+. Shift bit will combine with Load to get the actual control bit, S1, S0 for each shift register.

b.



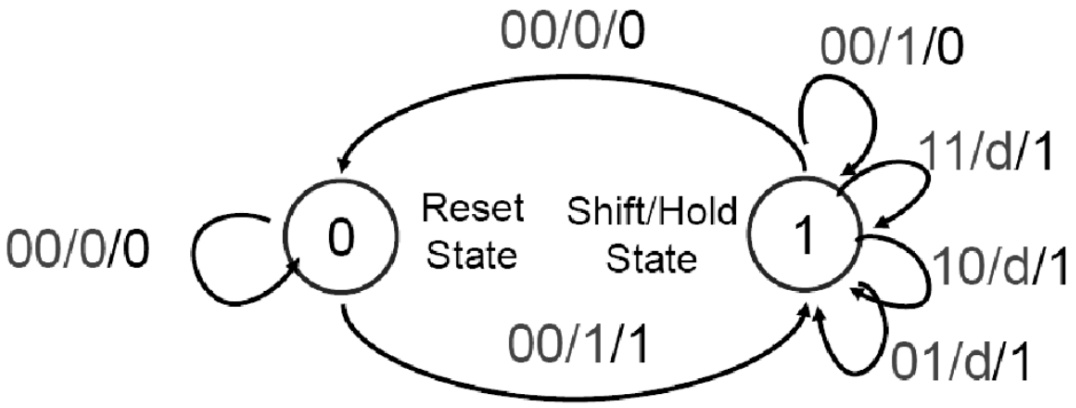


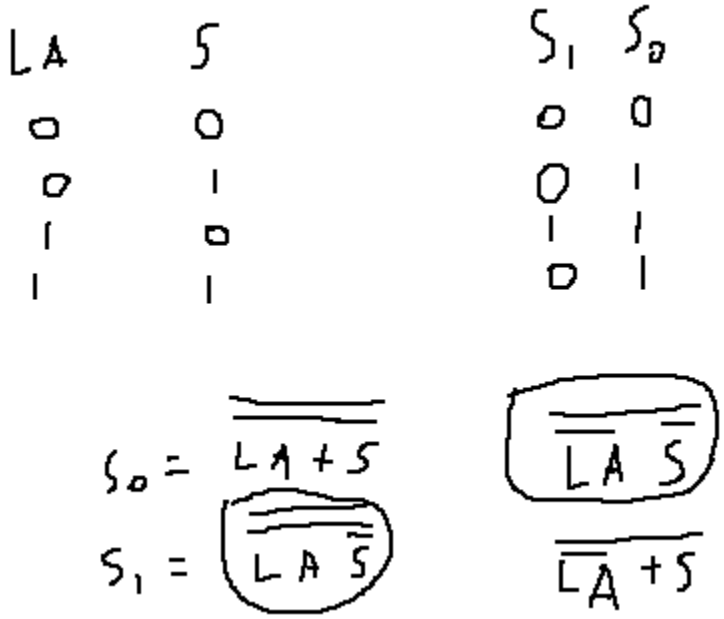
Figure 2: State Diagram

c.

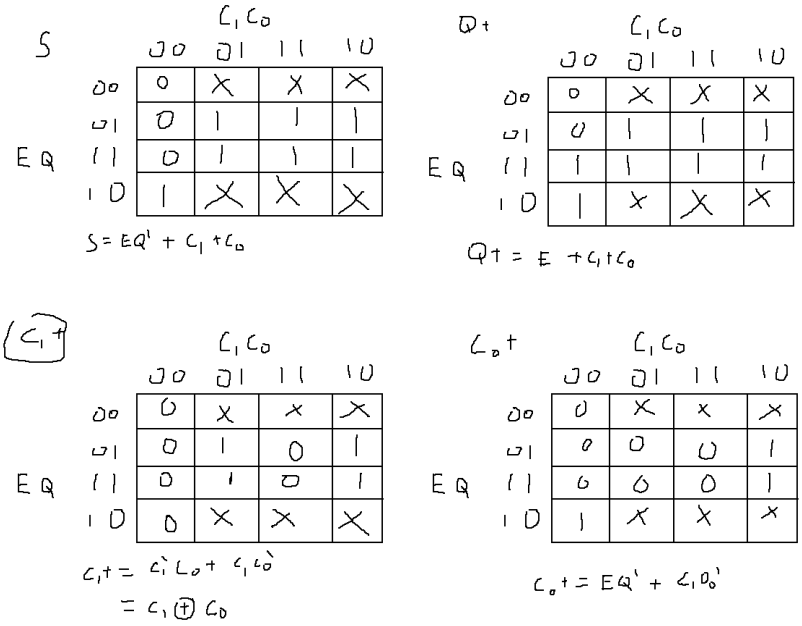
Design Steps

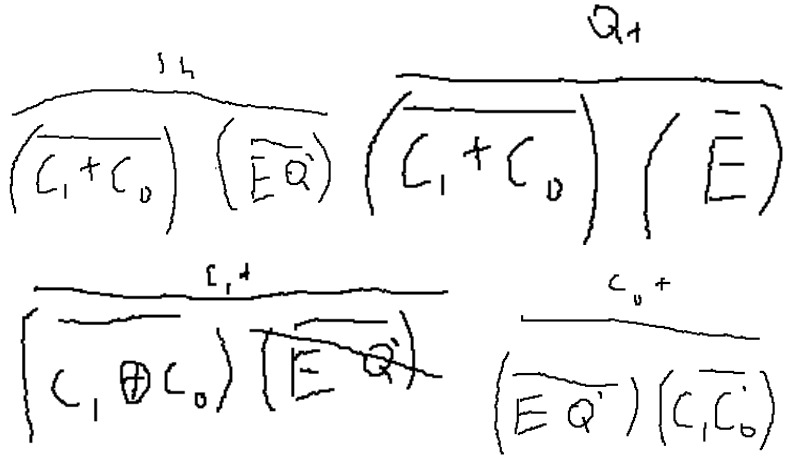
Written Procedure

Load A, Load B, and Execute were implemented with switches instead of buttons, because when we tried it with buttons, the results were not consistent.



Since there were two sources that needed to use the S1 and S0 of the CD74194 registers, there needed to be some minor combinational logic implemented inorder to accommodate both. First, a truth table where Load A only parallel loads if it is the only high input, and S overrides it to right-shift a bit into the register. The combinations circled above were chosen because there was available space on the chips already on the board, so it would not need any new chips added. There is a similar design for Load B.

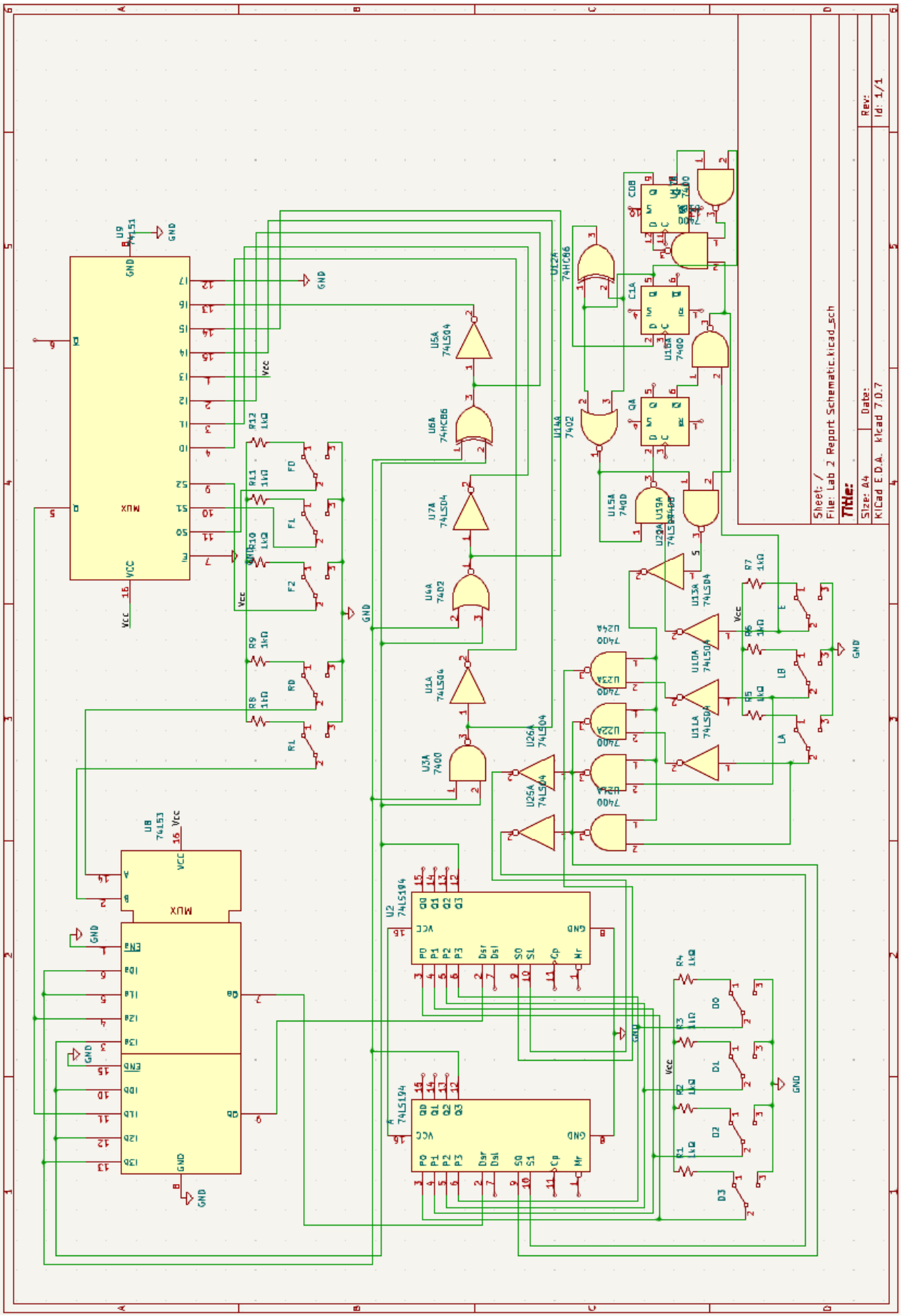




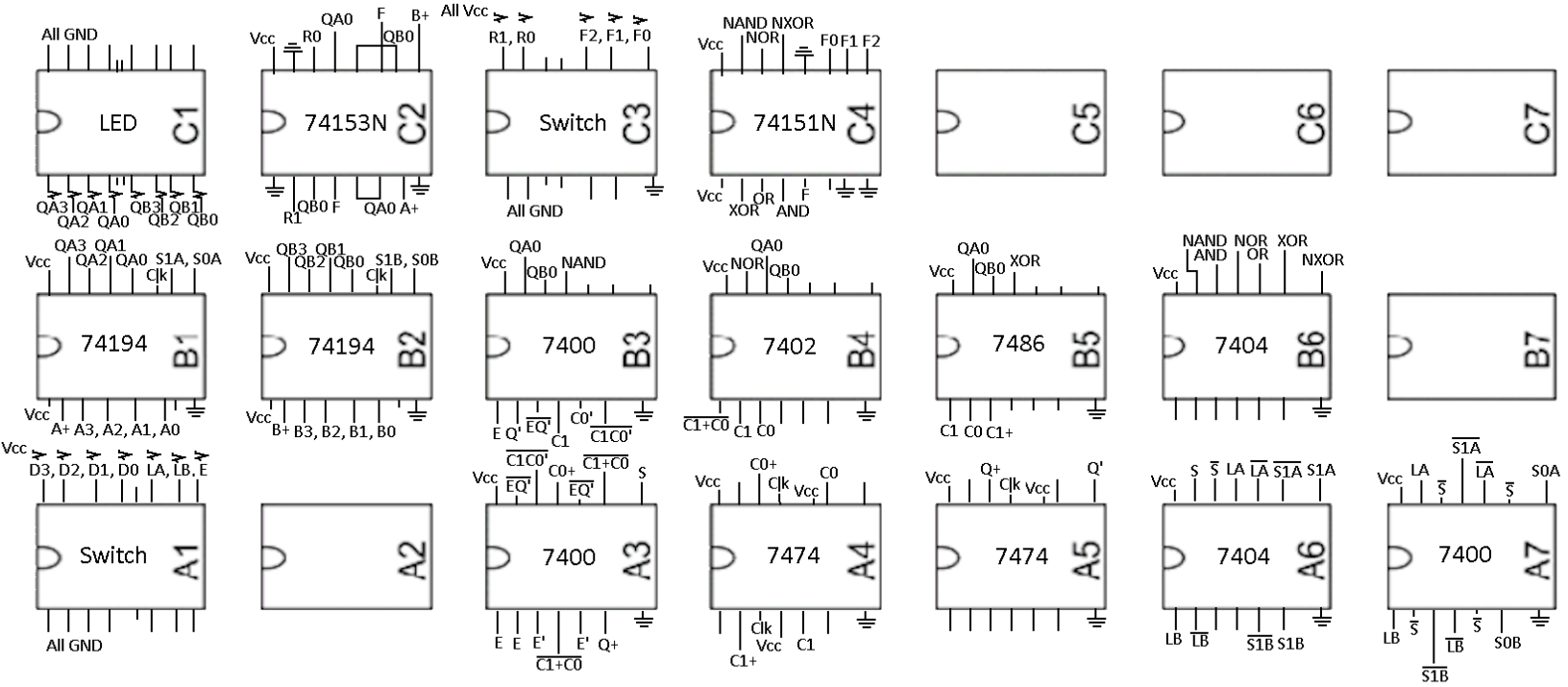
It was recommended to us to use internal state variables stored in SN7474 Flip-Flops instead of a 4 bit counter, which we agreed since it seemed simpler as we could not quite understand the 4 bit counter datasheet. The above are K-maps and subsequent actual implementation of the next state variable from the current state variables. The $\overline{EQ'}$ was crossed out due to an earlier K-Map that erroneously included that term.
A design choice we made to simplify wiring was to use one of the power buses as a data line, one for A_0 and one for B_0. This made it easier to connect the A and B inputs to the combinational gates of the computation unit and the MUX inputs of the routing unit since the distance between the pins was shorter compared to a single lane on the main body.

The routing unit was implemented with an SN74153 4-to-1 MUX and the computation unit was implemented with an SN74151 8-to-1 MUX. The reasoning was that it was the most straightforward way to implement both. The computation unit could have been implemented using a 4-to-1 MUX and XOR inverter. The XOR inverter takes a value input and a control input. If the control input is low, the value input equals the output, since it acts like an OR. If the control input is high, the output will be high if the value input is low, and the output will be low if the value input is high, meaning it is inverted. This is a useful way to simplify the computation module, depending on how the rest of the circuit was implemented. I did not implement it that way because the 8-to-1 MUX was already implemented and worked, as well as that both designs add the same amount of chips (the XOR already in place).

Design Schematic



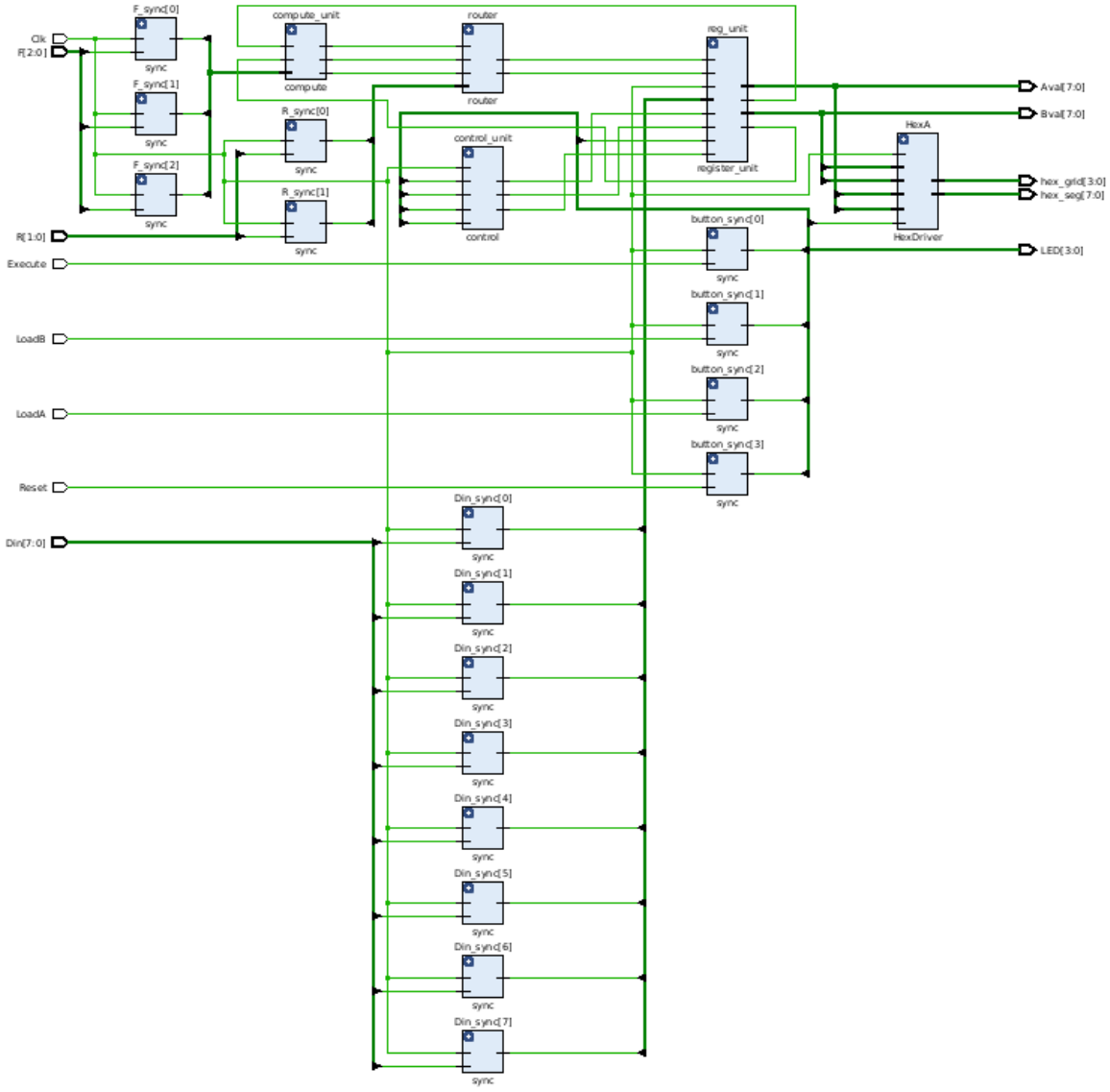
Breadboard View, Layout



8 bit Logic Processor

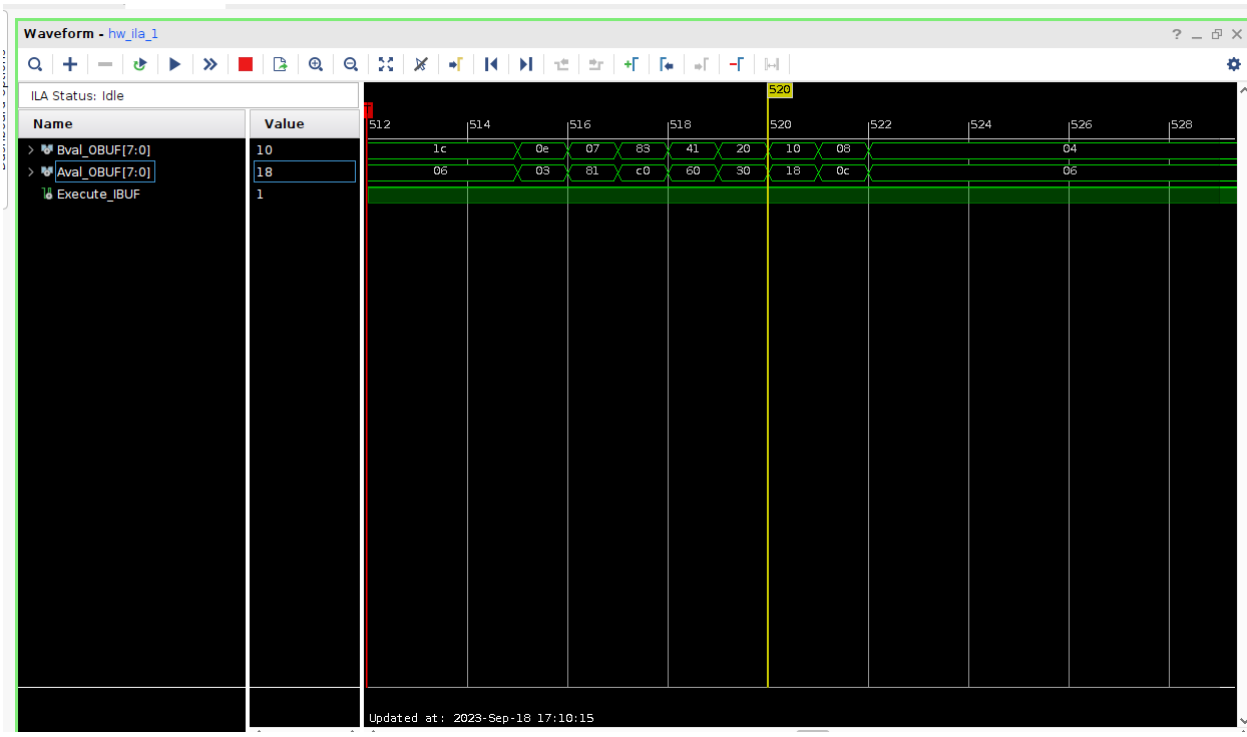
a. In the control module, it accepts five outside input clk, reset, loadA, lodB, and execute. And generate three three control signal shifts, Ld_A, Ld_B. Inside the module, it has 10 different states, which was used to describe each stage of the computation. In the initial state, it will check execution to determine whether to begin computation or not. In the last state, also check execution to determine whether to get back to the initial state. Only at initial state, the Ld_A and Ld_B are controlled by the input loadA, loadB, that is, only at that state, we have the right to load data.
b. In the hexDriver module, it accepts the input and displays them. It also has clk signal to synchronize.
c. Processor module is the top level module, inside it will instantiate the register unit for A and B, the computer unit, router unit, control unit and the hex driver. Also, it will use sync module to synchronize all processor module's outside input, for example, Reset_SH, LoadA_SH, LoadB_SH, Execute_SH, Din_S, F_S and R_S
d. Reg_4 module is used to simulate the shift register, each time it will accept either initial input which will be used for initialization, or for the single bit during computation. It will shift out one bit per cycle for the computing unit for doing calculation. Also it will output the current data inside the register for hex driver showing the data.

e. Register_unit is a module for packaging two reg_4 module
f. Router unit will route the original_A bit, original_B bit and computed bit to the register A and B based on the input R.
g. Compute module will use the incoming A bit and B bit to compute base on the input F.
h. The Sync module will synchronize the input base on the positive clk edge. It is used by the processor module which synchronizes all its outside input.
i. Inside the control unit, change the previous 6-stage to 10-stage to be compliant for 8-bit register operation. Add four more stages for calculating.
j. In the processor module, change the input Din to 8 bits, change the output of AVal and BVal to 8 bits. Change Din_S to 8 bits which is the synchronized D_in. Also change {B[7:4], B[3:0], A[7:4], A[3:0]} in hex for displaying the new two 8-bit values.



k.

l. Under the synthesis -> Set Up Debug -> Find_Nets_to_Add -> search for the value we want to trace -> select type (data or data and trigger). Later for displaying the data on screen, we need to set up the trigger after programming the device. Set operate to ==, Value to B(display both transitions).

m. 

Description of All Bugs Encountered

- BUG: The SN74153 4-to-1 MUX and SN74151 8-to-1 MUX did not seem to operate consistently

Solution: Ground the strobes, Pin 7 for SN74151 8-to-1, Pins 1 and 15 for SN74153 4-to-1 MUX

- BUG: Some of SN74151 8-to-1 MUX outputs were correct, some were incorrect

Solution: F2 and F0 were swapped

- BUG: Registers occasionally lost all data

Solution: Connect clear to Vcc to ensure a mistaken CLR signal does not go through

- BUG: Registers did not properly shift bits in

Solution: Add combinational logic between Load A, Load B, and the Shift bit to produce well-defined S1 and S0

- BUG: Buttons did not seem to output a consistent pulse when pressed, possibly due to faulty internal wiring or inconsistent pressing

Solution: Uses switches instead of buttons

- No notable bugs occurred will making the Vivado implementation

Conclucion

In this lab, we have constructed a machine that can take 4 bit (physical), or 8 bit (Vivado) inputs into registers, perform various bitwise operations, output to either register, or swap the register's contents. The modular design of this circuit aided in bugtesting since each module can be tested independently to determine more precisely what is working and what is not, rather than testing the entire thing and estimating what parts are faulty to cause the error. A Mealy machine uses both the state and current inputs, while a Moore machine transitions solely based on its inputs. A Mealy machine is able to have less states overall since the inputs can remove some intermediary states of a Moore machine, but because it may require inputs for a certain state transition, if a clock was also implemented in the Mealy machine, it may be difficult for a human to time it properly. A Moore machine is completely autonomous and can accomplish its tasks consistently, but it may not be the most efficient method, as if there were a shortcut it could take, another input would need to be added to indicate that, or an outside observer could manually direct it as a Mealy machine. The difference between vSim and Vivado debug cores is that vSim only interprets the code, generates an input waveform, and creates an output based on the code, whereas Vivado debug cores need to be implemented in the FPGA board, where the signals across the board are measured and its possible to set triggers to do your own inputs as well. vSim would be better testing if the concept is functioning as expected, while Vivado debug core is better for testing its physical implementation on the FPGA board. In future, I think it would be a great help if only the parts we have are in the datasheet to remove the clutter, and have each chip have a truth table to understand its function.