Lab 3 Report

ECE 385

Friday (ABB)

11:00AM-1:50PM

Logic Processor

Ritvik Avancha and Steven Phan

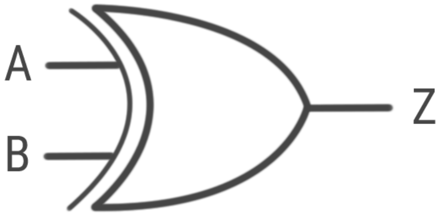
(rra2 and sphan5)

Table of Contents

Lab 3 Report

Introduction

The main purpose of this lab was to create a 4-bit serial logic operation processor that takes in two 4-bit inputs in Registers A and B and performs some bitwise operation on those bits. The output of this bitwise operation can be routed to register A or B or neither depending on the routing signals. This processor is able to perform 8 different operations: AND, OR, XOR, load "1111", and the remaining four are just NOT versions of the first four. By implementing a finite state machine (Mealy machine) to do the operation, the computation is guaranteed to complete in four cycles (even if EXECUTE is swapped to low during the operation).



Input		Output
A	B	A xor B
0	0	0
0	1	1
1	0	1
1	1	0

Figure 1.1: XOR Gate and Output

To simplify the circuit design in this lab, only four functions were implemented, and the rest was implementing use an XOR gate - the simplest (two-input one-output) circuit that can optionally invert an input signal A, given a select input B is an XOR gate. For Example, look at when A is 0. If the select B is low, the input is not inverted, and if B is high, the input is inverted.

When building this circuit, a modular design was kept in mind to decrease debugging time and simplify the overall design and look. The idea behind a modular design is to build the whole circuit with individual modules (i.e. components) that could be tested individually. In this lab there were four individually tested modules: the register unit, computation unit, routing unit, and the control unit that directs the aforementioned modules. By building them separately, they could be tested separately and it was easier to tell where a bug was when a bug showed up. In addition to easier debugging, development time was also cut down. Due to the physical separation in the design, there were less overall interconnections in the circuit, making for an easier build.

Operation of the Logic Processor

In order to load data into Register A and B, the switches D[3:0] should be flipped indicating what bits should be loaded into the registers. After setting the data, LOAD_A or LOAD_B (or both depending on what registers are to be loaded) are switched on in order to turn on parallel loading for the shift registers.

Once data is loaded as mentioned in the previous paragraph, in order to start a computation cycle, 3 sets of switches should be set. First, F[2:0] are set to determine what type of bitwise computation the circuit should do (AND, OR, etc). Then, R[1:0] should be set to determine where data is to flow. For example, if R[1:0] = 01, the output of the function goes into register B, and A retains its data. After those two sets of switches are set, the EXECUTE switch should be flipped to high to indicate the start of the computation cycle.

Description, Block Diagram, State Diagram

As mentioned in the introductory paragraph, this circuit is modular. It has four main components that work together to perform bitwise operations on four bits: register unit, computation unit, routing unit, and the control unit that dictates the previous three units.

The register unit consists of two 194 chips (universal bidirectional shift registers). D[3:0] was connected to parallel data inputs D, C, B, and A in order to allow the user to load in data they wanted. The control signals that shifts and loads data were LOAD (user set), and SHIFT (which comes from the control unit). These signals and their combinational logic were hooked up to S1 and S0. CLK was also hooked up to register in order to physically shift the data on the rising edge.

The computation unit consists of a 4-to-1 MUX, two NAND gates, two NOR gates, and two XOR gates. F[1] and F[0] are the select signals into this MUX (with F[1] being the more significant select bit), and the first four functions are implemented using the gates mentioned in the list. The remaining four functions are implemented using the output of the MUX XOR'd with F[2] (another control signal). The reason this works is mentioned in the introductory paragraph (the remaining half of the computations are just inverted from the first half). Overall, F[2:0] controls which operation is done on the bits.

The routing unit accepts three sets of inputs: f(A,B), Q (A), and Q (B) where Q is the most recent bit the computation unit did an operation on. There are two sets of 4-to-1 MUXes, one for A^{+} (the next bit serially shifted in RegA) and B^{+}. R[1:0] determines where data in the circuit flows. For example, if R[1:0] = 00 or 11, the function output f(A,B) is completely ignored, and so on. This part of the design dictates where data moves.

The control unit in this design takes four main signals: LOAD A, LOAD B, EXECUTE, and the CLK. Load A and B parallel loads data in set by switches D[3:0] into register A and B. After data is loaded and the user has set switches F[2:0] and R[1:0], EXECUTE is flipped high. Once this is high, the computation cycle begins and does not stop until the computations are done. Once EXECUTE is flipped, the control unit implements in a state machine that ensures exactly four cycles are carried out, regardless of whether EXECUTE is flipped back to low during the execution cycle.

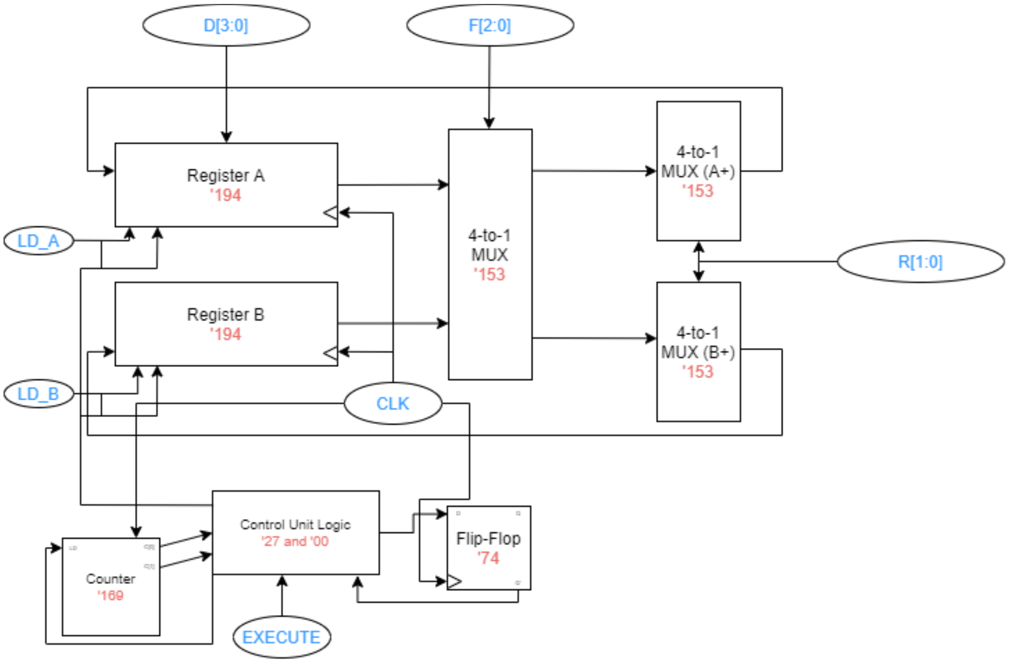


Figure 3.1: High Level Block Diagram

The way the control unit is implemented through a Mealy Finite State machine with two main states: Reset and Shift/Hold State. The transitions are dependent on the counter output, EXECUTE, and the SHIFT signal.

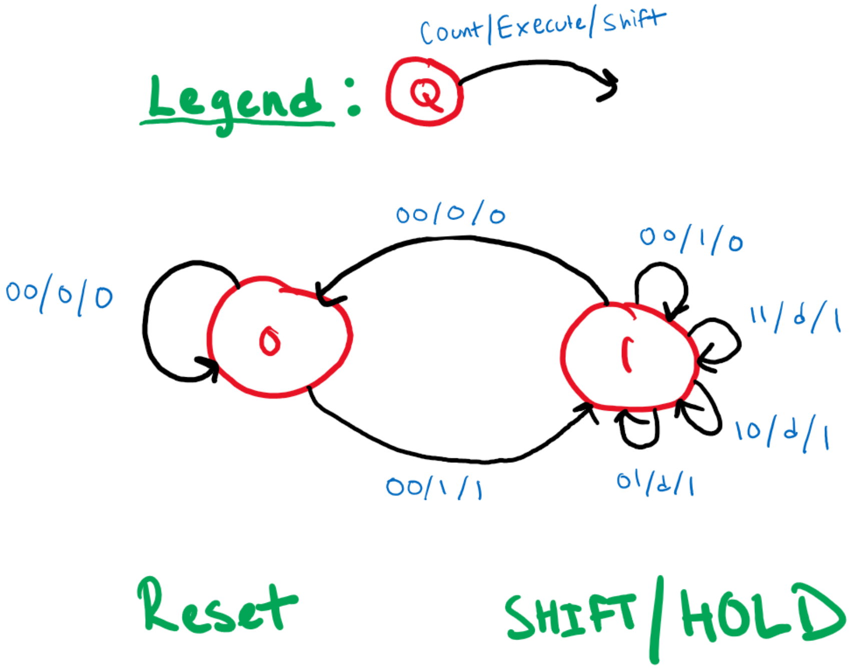
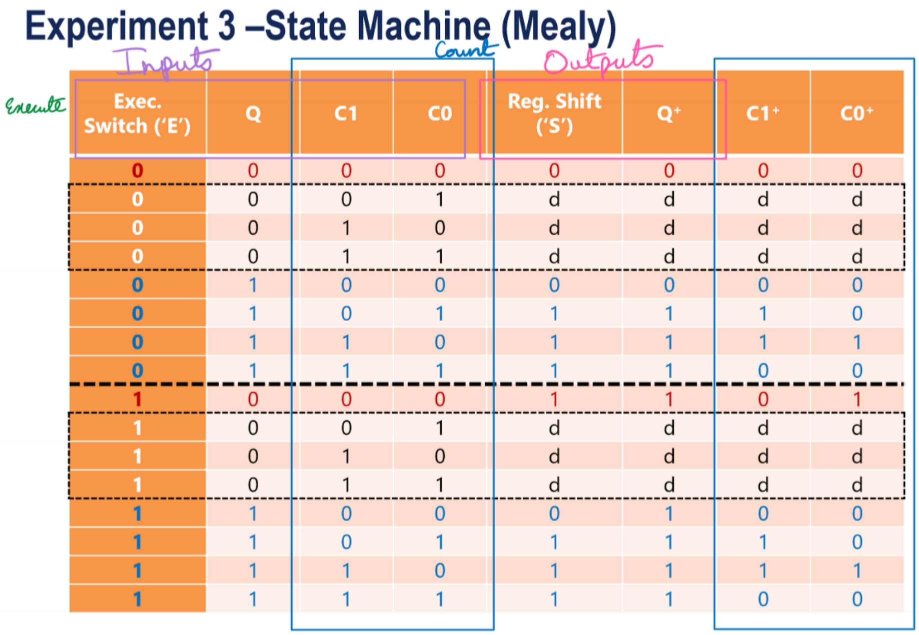


Figure 3.2: State Transition Diagram

<u>Design Steps, Detailed Circuit Schematic</u>

During the design of the implementation of the circuit, multiple designs were considered. One design implemented the state machine in Figure 3.2 using three D flip flops while the other design implemented the state machine using a 169’ counter and combinational logic depending on output S in Figure 4.1. The latter implementation allows for an advantage in space and complexity as the implementation required only a counter circuit and a few logic gates. The flip flop implementation was larger and more complicated but allowed for each state to be designed into the finite state machine. The implementation using the 169’ counter was chosen.



Experiment 3 –State Machine (Mealy)

Inputs

Count

Outputs

Execute

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

Figure 4.1: Truth Table

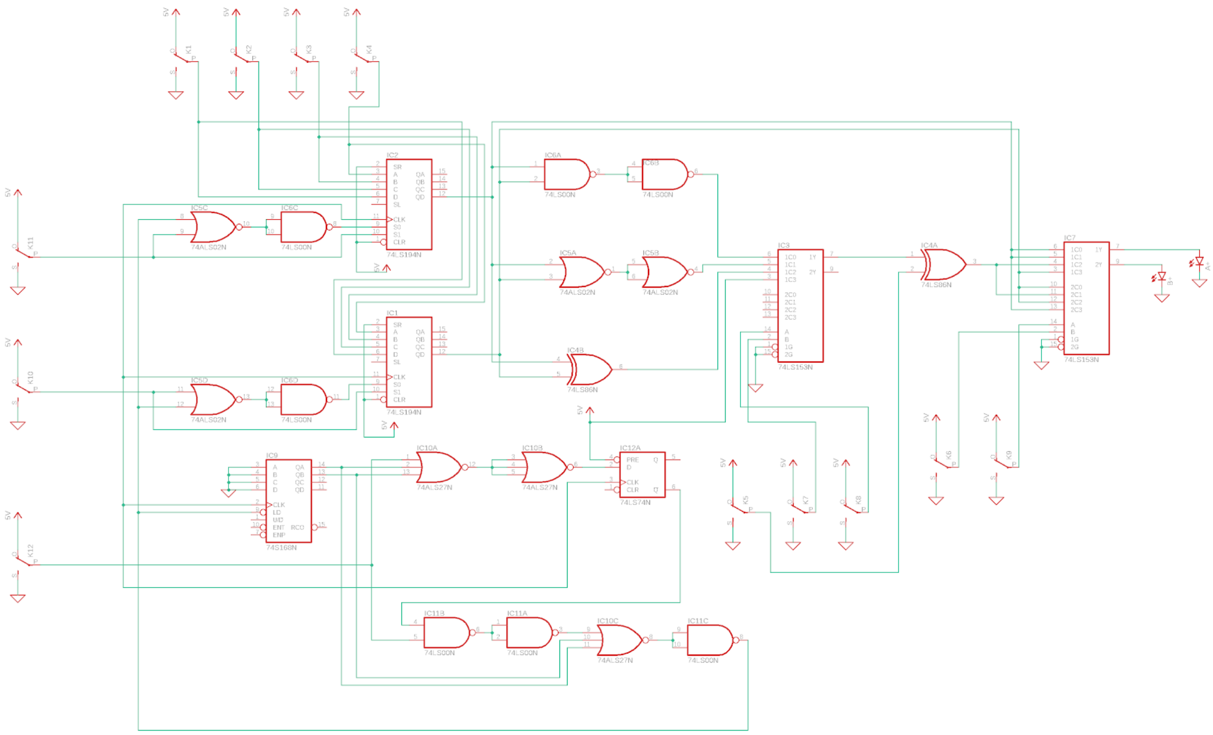


Figure 4.2: Detailed Circuit Schematic

Component Layout Sheet

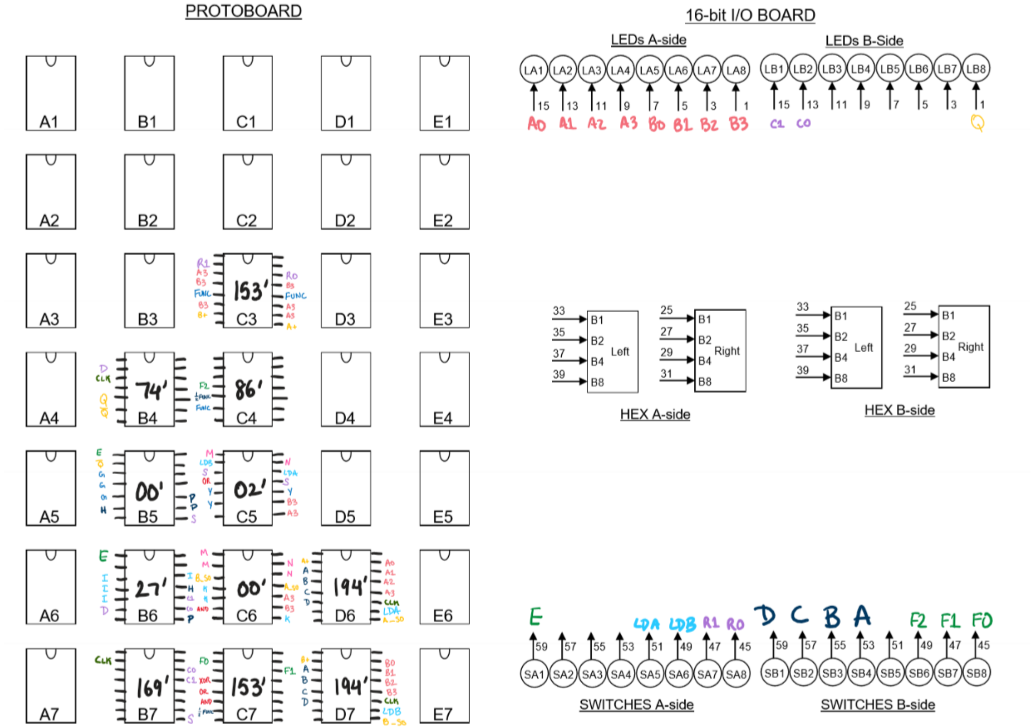


Figure 5.1: Component Layout Sheet

Bugs and Fixes

Since the design was implemented through modules as mentioned before in the report, there were no glaring issues while testing the circuit. Earlier on in the build, we had forgotten to connect the routing MUXes back to the shift registers, and we noticed that because whenever we executed the function, it would load all 1's into the registers. We had accidentally connected the shift register's inputs to high, and we forgot to connect the MUX outputs to the registers. Fixing this was a simple rewire. Another issue encountered while testing the XOR gates. In the data sheet, there are three variants of the chip, so we ended up using the wrong variant. This was fixed through unit testing all the possible variants to determine the correct layout.

Conclusion

In this experiment, a logic processor was built with the ability to compute bitwise operations, such as AND, OR, XOR, NAND, NOR, and XNOR, and stored in two 4-bit shift registers. Additionally, the circuit implementation allowed for a parallel load of data into the shift registers along with a swap of data between the shift registers. This design challenge was tackled with the completion of a high level schematic allowing for designated modules. A modular circuit allowed for segments of the circuit to be built and debugged independently. This approach reduced the compounding effects of multiple bugs and simplified the overall debugging process. During the construction of the circuit, the design allowed for a state module, a routing module, a computation module, and a storage module. Initially, the storage module was debugged alongside the computation module while the miswired state module was isolated and debugged later on. Our state module uses a Moore state machine implementation as the next state depending on both the current state and the inputs. The outputs in a Moore state machine depend only on the current state while the outputs on a Mealy state machine can depend on both the current state and the inputs. A Mealy state machine allows for reduced finite states thus reducing complexity of combinational logic. On the other hand, a Moore state machine is simpler to use as it only depends on the current state.