**ECE 385**

Fall 2019

Experiment #3

A Logic Processor

Jacqueline Jiang and Ji Ho Han

AB6 / Wednesday 12:00 - 2:50pm

Nicholas Cebry and Yujie Zhou

# Introduction

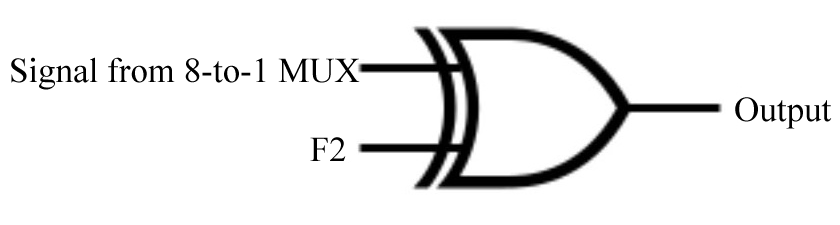
## Summary of the Circuit

The purpose of this experiment was to design an operating bit-serial processor that takes in two 4-bit words and performs one of eight distinct operations on them. The processor is then able to route the output of these operations in four different ways. Although the bit-serial processor only operates on one bit at a time, it will carry out the operation for a total of four times, thus altering four bits in each cycle.

## Answers to Pre-Lab Questions

A. The simplest circuit that can optionally invert a signal can be implemented by using the XOR gate (Fig. 1). We can have 2 separate inputs — one that is equal to the original value (specifically the signal from the MUX), and another input (F2) that will act like a select bit to determine whether the output should be equal to the original value or be the inverted value. When given a logic one, the select bit input will cause the XOR gate to output the inverted value while a logic zero will cause the XOR gate to output the original value.

B. Normally, we have to develop 8 different operations for the computation unit. However, by using a modular design, this process is drastically reduced down into 4 different operations with a single XOR gate. In general, modular designs are useful because it allows us to break down our design into smaller, simpler components, thus reducing development time. It also greatly improves testability because smaller components are easier to unit test, thus allowing us to debug faster when something goes wrong.



*Figure 1: Circuit Diagram of Simplest Signal Inverter*

# 

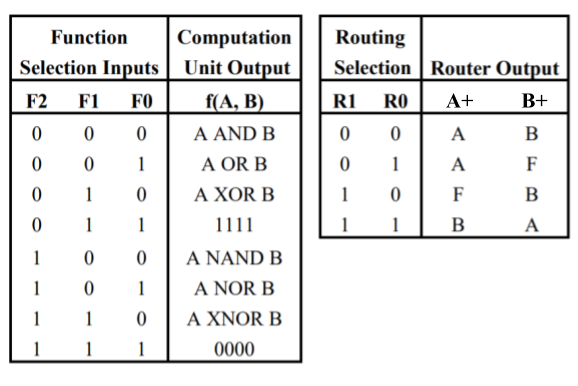
# Operation of the Logic Processor

## Sequence of Switches the User must Flip to Load Data into the A and B registers

In order to load the data into the shift registers RegA and RegB, the input data must first be chosen via the input of the register unit. This is represented as D3~D0 (Fig. 4), and is implemented as switches SB1~SB4 (Fig. 16). The Load A and Load B switches in the Control Unit then control which register will receive the data. The register(s) selected will then be parallel-loaded with the new data in the next clock cycle.

## Sequence of Switches the User must Flip to Initiate a Computation and Routing Operation

First and foremost, the operation to be carried out by the Computation Unit is selected through adjusting F2, F1, and F0 (Fig. 4). Using these selection inputs, a total of 8 different operations can be selected as seen in Figure 2. Next, by changing R1 and R0, the user can redirect the signals from the Computation Unit and from Registers A and B to become the new inputs for Registers A and B. Figure 3 also shows the possible routing outputs for the Routing Unit. After the selection inputs for both the Computation Unit and the Routing Unit are set, the Execute switch can be flipped to a logic one to initiate the operation. Note that Execute, R1, R0, F2, F1, and F0 are all implemented as switches in our circuit (Fig. 16).



*Figure 2, 3: Tables Representing the Operations from the Computation and Routing Units*

# Written Description, Block Diagram, and State Machine of Logic Processor

## Written Description

Before constructing the actual units, all the inputs — D3~D0, F2~F0, R1, R0, Load A, Load B, and Execute — were connected to the switches.

The Register Unit was implemented using 2 shift registers. This unit holds two purposes, storing the operands to be computed and displaying the final output of the operation. The switches D3~D0 were connected to both of the registers as parallel inputs, and the outputs of both registers (A3~A0 and B3~B0) were connected to LEDs. The Clock was connected to both of the shift registers, so that both registers operated on the same clock cycle. The select logic for the input mode of these registers were calculated in the Control Unit.

The Computation Unit was built using a 8-to-1 MUX combined with a 2-input XOR gate. The Computation Unit performed 8 different operations based on the select inputs F2, F1, and F0. Figure 2 shows all of the operations offered by the Computation Unit. Because the last 4 of the 8 operations were just the inverted versions of the first 4, we only really need to implement 4 operations. To optimize the design, we chose to use a lookup table to reduce the amount of gates used within our circuit. As such, the signals F1, A, and B were used the select bits to our lookup table, allowing us to perform the 4 distinct operations. To get the other 4 operations, we simply used an XOR gate to invert the output from the lookup table when necessary. The logic behind the XOR gate can be seen in Part A of the Pre-Lab.

The Routing Unit was implemented with two 4-to-1 multiplexers. This unit redirected the output of the Computation Unit and the outputs of the Registers A and B back to the Register Unit to become the new inputs for the registers. The routing operation was chosen by the select bit R1 and R0, shown in Figure 3.

The Control Unit outputted the select logic for the mode of the Register Unit. As such, it determined when the Register Unit should begin shifting to perform an operation. It is implemented with a Mealy state machine. The Control Unit uses a flip-flop synced with the Clock to maintain the state of the Mealy machine. A counter chip was also used to ensure that all bits in the registers were operated on once.

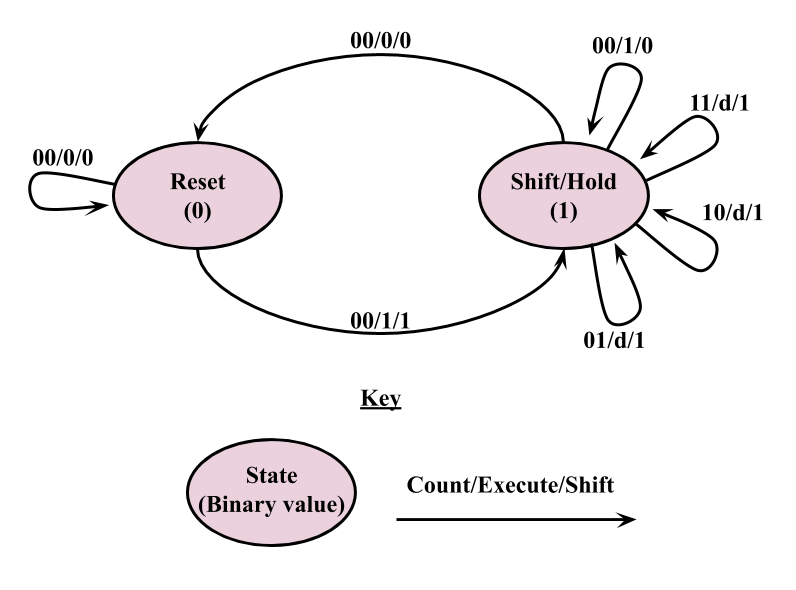
## High-Level Block Diagram

# 

*Figure 4: Block Diagram of Logic Processor*

## State Machine Diagram

We used a Mealy machine in our circuit. A diagram of this can be seen in Figure 5. In our circuit, the reset state is represented by a logic zero in our flip-flop, and the shift/hold state is represented by a logic one.



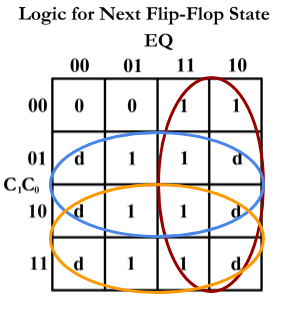
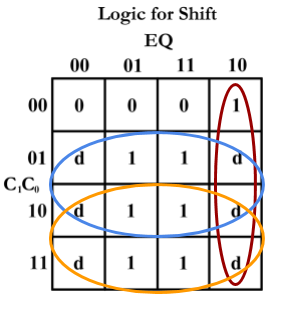
*Figure 5: Mealy State Machine Diagram of Logic Processor*

# 

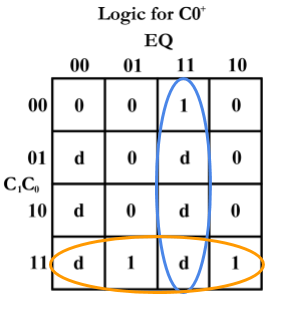
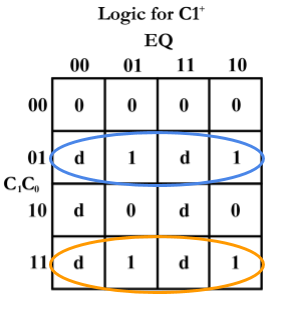
# Design Steps Taken and Detailed Circuit Schematic Design

## Written Procedure of the Design Steps Taken

For our Control Unit, we used a Mealy state machine in our design. A flip-flop was used to hold the current state of the state machine. We calculated the logic for the next state (Q+) of the flip-flop and the Shift signal using K-maps shown in Figures 6~9. In the K-map, “E” represents Execute, “Q” represents the state of the flip-flop, and “C1C0” represent the counter bits from the counter chip. The counter chip we used in our Control Unit took care of the logic for advancing C1 and C0 to their next values, so we only needed to worry about creating the logic for Shift and the next state of the flip-flop. In total, this logic needed a total of 3 NAND gates to implement, two 3-input NAND gates and one 2-input NAND gates. To reduce the total number of chips in our circuit, we decided to implement 2-input NAND gate using a 3-input NAND gate by connecting the extra input to high. A logic diagram for this can be seen in Figure 14.

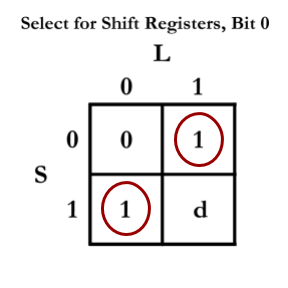
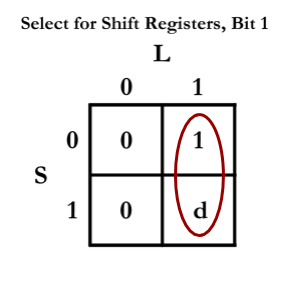


*Figure 6, 7: Logic for Shift and Next Flip-Flop State*



*Figure 8, 9: Logic for C1+ and C0+*

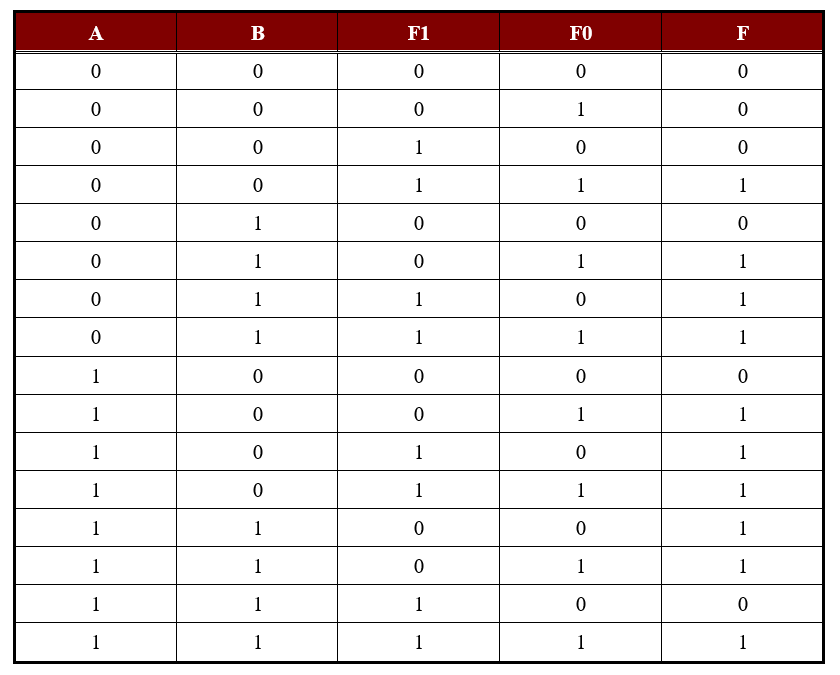
For the select bits for the mode of the shift registers, we calculated the logic need using the K-maps shown in Figures 10 and 11. The “L” represents Load A and Load B signals, and “S” represents the Shift signal. When either Load signal is toggled on, the corresponding register will parallel load D3~D0 in the next clock cycle. Otherwise, the shift registers will be in a NOP or shift right state depending on the state machine.



*Figure 10, 11: Bitwise K-map of Select for Shift Registers*

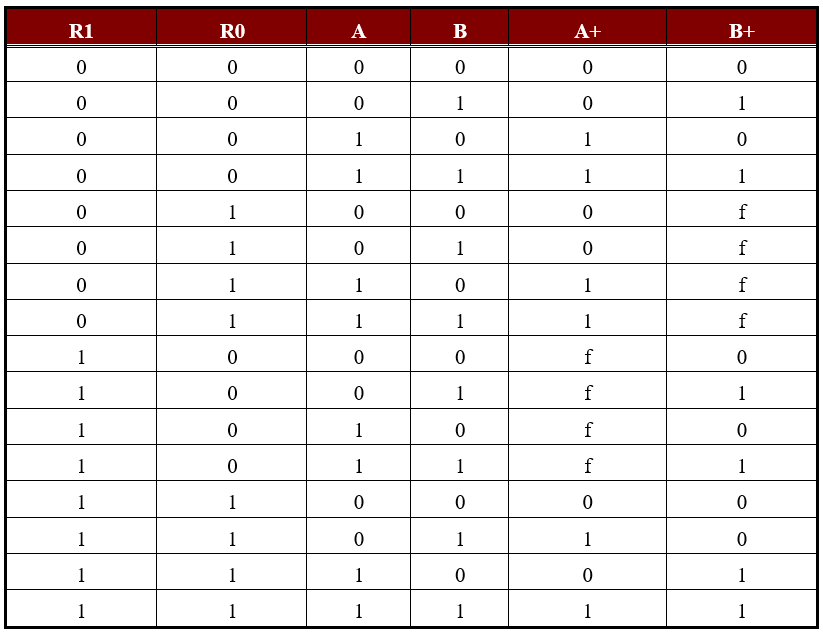
For the Computation Unit, the first thing we were debating was the selective inverter circuit mentioned in Part A of the Pre-Lab. Because the first 4 operations of the Computation Unit were just inverted to produce the last 4 operations of the Computation Unit, finding the simplest design for this can reduce the complexity of our circuit. At first we were considering using a NOT gate and 2-to-1 MUX with F2 as the select bit to choose between the inverted and original value. However, this used many chips and was not the simplest design possible for the task at hand. After thinking for a bit, we realized that we could use an XOR gate to achieve the same result. As such, we implemented design with the XOR gate because it reduced the complexity of the circuit.

The next decision choice we debated was whether we wanted to use the Computation Unit design shown lecture, which contained an AND gate, an OR gate, an XOR gate, and a 4-to-1 MUX. However, due to the sheer amount of chips necessary for this idea we discarded this idea. Instead, we decided to implement a lookup table with a 8-to 1 MUX. The lookup table has many benefits over the design shown in class such as less chips and faster calculations since there are a minimal amount of gates. The only drawback of using a lookup table is that it masks the actual operations being done in the Computation Unit because it quickly grabs the corresponding output instead of calculating the output with actual gates. The lookup table was implemented using A, B, and F1 as the select bits, and a truth table for it can be seen in Figure 12.

****

*Figure 12: Truth Table of Computation Unit*

Our Routing Unit was implemented with two 4-to-1 multiplexers with R1 and R0 as select bits. The truth table for the Routing Unit can be found in Figure 13. The “f” in the truth table represents the output from the Computation Unit.



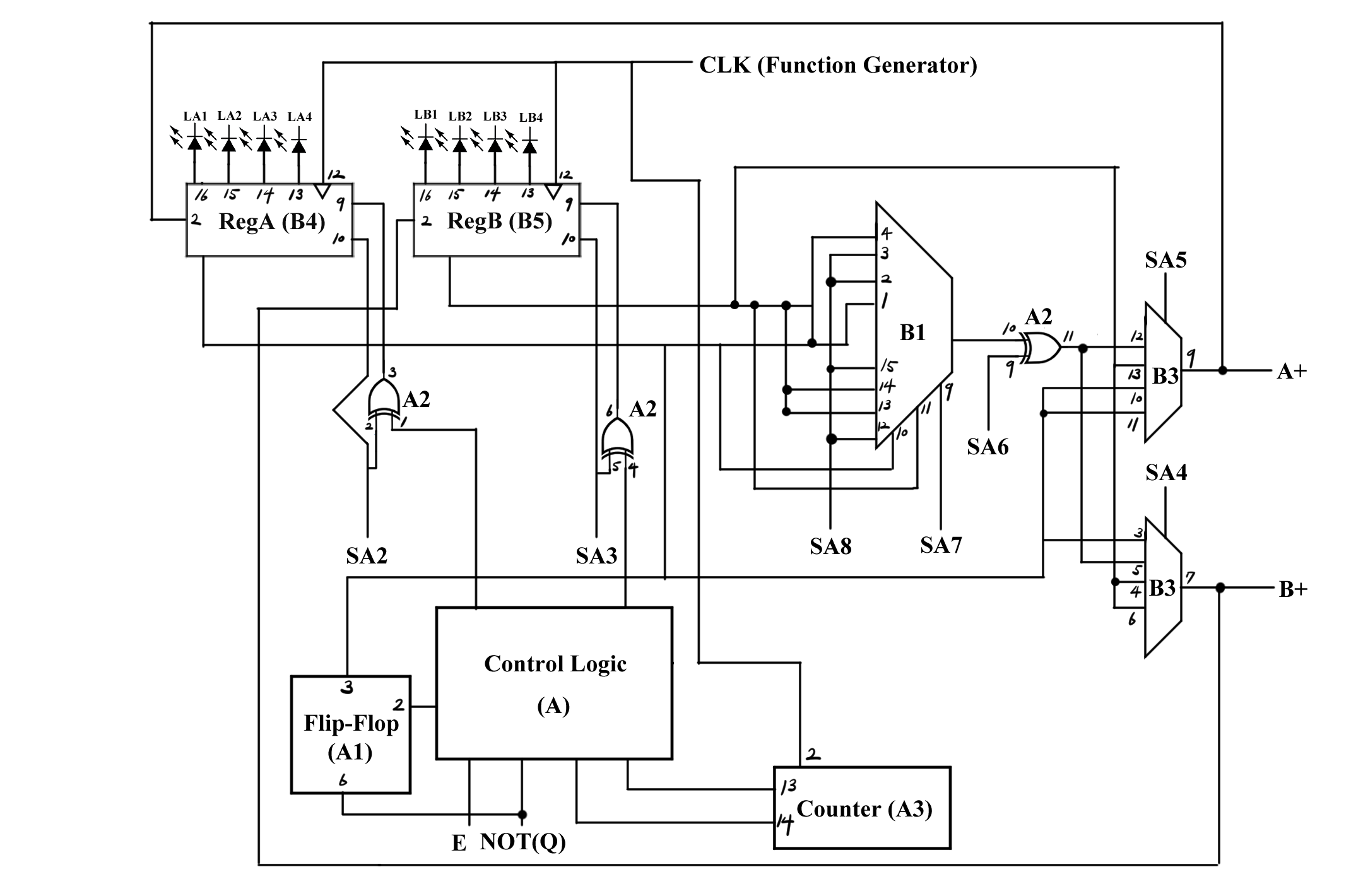
*Figure 13: Truth Table of Routing Unit*

## Detailed Circuit Schematic

# 

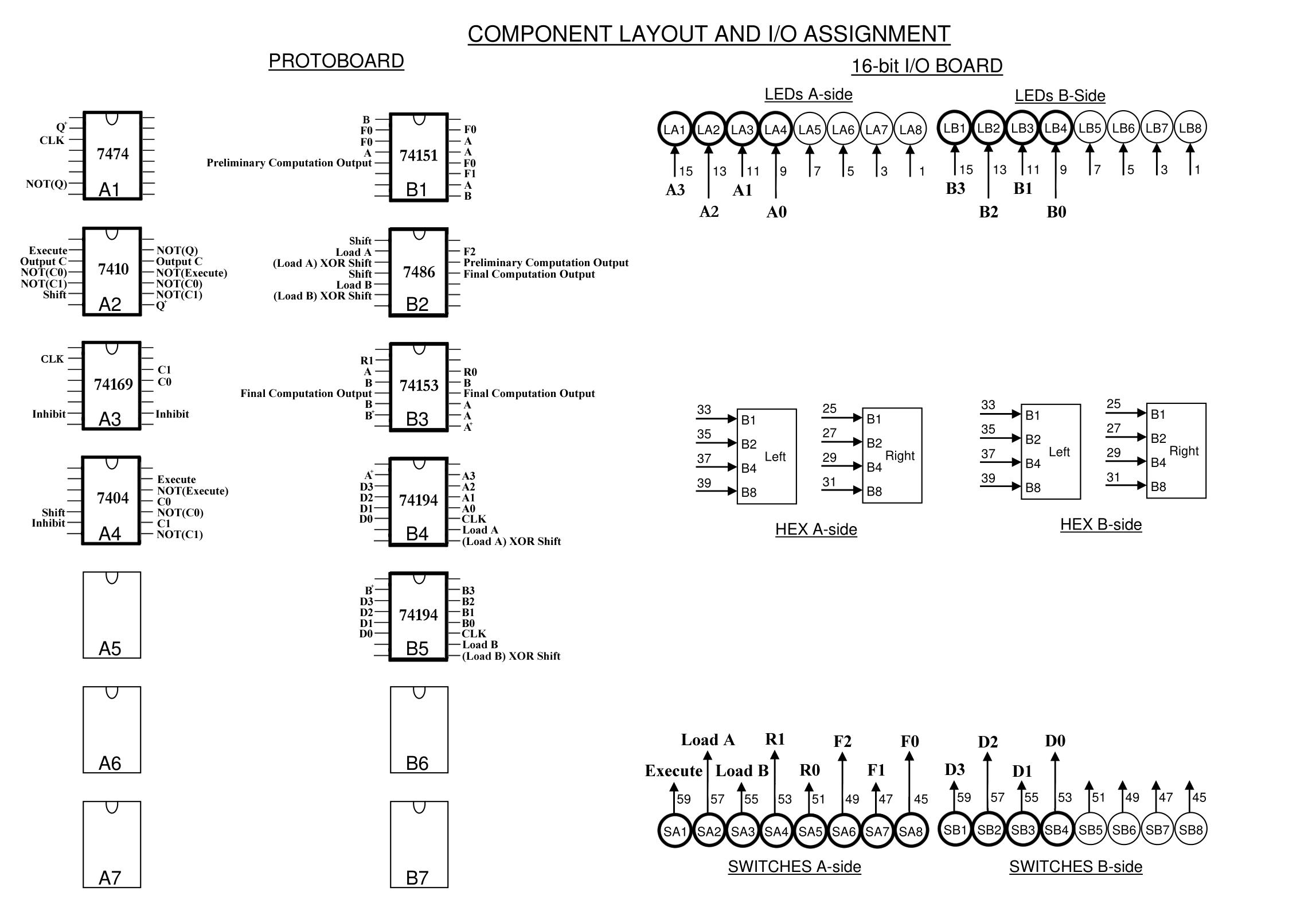
*Figure 14: Circuit Schematic of Control Logic*

Figure 15 represents our detailed circuit schematic.

**

*Figure 15: Detail Circuit Schematic of Logic Processor*

# Layout Sheet



*Figure 16: Component Layout of Logic Processor*

# Description of all Bugs Encountered and Corrective Measures Taken

Because we used a modular design, we were able to unit test along the way and did not encounter any bugs with our logic or wire. Nevertheless, on the demo day, we had an issue with connecting the circuit’s clock to the function generator. The circuit was working when we were using the debouncing switch for the clock however, the circuit gave inconsistent outputs when we started to use the function generator. At first, we thought one of our chips broke, but when we reattached our clock to the debouncing switch, we found that our circuit worked as expected. After careful examination of the function generator, we discovered that the offset of the function generator was not set. This resulted in lack of voltage even when the square wave was high, thus letting the circuitry to fail. This error did not occur when using the switches since switch directly connected the voltage which is high enough to power up the circuits. This bug was fixed by properly adding the offset to our function generator’s signal.

# Conclusion

## Summary of the Lab

In summary, the outcome of the lab was processed as expected. The data was successfully loaded to the registers, all 8 computations were performing just like our truth tables, and the resulting values were correctly redirected to the selected registers. Through this lab, we learned the importance of having a modular design especially when creating larger circuits. The modular approach allowed us to unit test each of our components. This is good practice because we will be able to catch bugs before they get too buried beneath all of the wires. We also had a refresher on Mealy and Moore machines through this lab, which will help us in future labs when we have to design more state machines in SystemVerilog.

## Answer to all Post-Lab Questions

The modular approach proposed in the pre-lab helped us greatly with ensuring the circuit worked as intended. Because we separated our design into separate components, we were easily able to unit test every component before attaching everything together. Although we did not have any major bugs while designing and wiring the circuit, this modular design was especially useful when testing our look-up table for the Computation Unit. Because of our modular design, we were able to test all possible combinations from the truth table before we connected our Computation Unit to anything else. This was useful because we could be confident that if an issue arises, the source of the problem would not be our Computation Unit. This further simplified our testing for our Routing Unit since we did not have to test every operation when seeing if our 2-to-1 MUX outputted the correct signals for functions.

The difference in a Mealy machine and Moore machine in the computation theory is pretty straightforward — the output of a Mealy machine is determined by both the current state and the current inputs, whilst the output of Moore machine is decided by only its current state. The benefits of using Mealy machines are its lower amount of states and its speed. Since Mealy machines generate their output based on the current input, they are faster than Moore machines because the output can be calculated asynchronously with respect to the clock. Mealy machines also need less hardware to implement because it has less states. On the other hand, the benefits of using Moore machines are its safety in synchronization and its simplicity of design. Because the output only depends on the state, Moore machines do not need any extra logic to calculate the output. Moore machines also change states based on the clock cycle unlike Mealy machines, whose state can change asynchronously, so the output generated will be synchronous with respect to the clock. Long story short, it’s all about speed vs stability.

For the logic processor, we used a state machine to implement the control logic. The state machine needed to include a shifting state for when we are evaluating each bit, a halting state to wait for Execute to be toggled off, and a reset state to restart this process when Execute is toggled on. The Moore machine would need a total of 6 states if a counter is not used to keep track of the amount of bits shifted, and if a counter chip is used, it would need 3 states to be fully implemented. On the other hand, the Mealy machine only needs 2 states in its state machine design, reset and shift, because we can combine the shift and halt states into one. Although the Moore machine is more stable and more simple in terms of the logic needed to implement it, it also consumes a lot more resources to implement all of the states. Because we were worried that more chips and wires would make our circuit cluttered and difficult to debug, we decided to use a Mealy machine within our design.