**ECE 385**

Fall 2019

Experiment #2

Data Storage

Jacqueline Jiang and Ji Ho Han

AB6 / Wednesday 12:00 - 2:50pm

Nicholas Cebry and Yujie Zhou

# Introduction

The purpose of this lab was to create a simple data storage unit of four 2-bit words that can read and write. The data will circulate through the shift registers until a new command is given by the control logic. The available commands for this circuit are fetching from memory (FETCH), reading from memory (STORE), and loading data into a buffer (LDSBR). It is useful to understand circuits like the one we made because devices such as Charge-Coupled Device camera sensor are inherently organized as shift registers.

# Operation of the Memory Circuit

Our storage unit can store four 2-bit words, using two 4-bit shift registers. Each of these shift registers holds one bit of each word in the storage unit. As such, we will refer to these registers as Bit1 and Bit 0. Flip-flops are used for the Storage Buffer Register (SBR), which is a buffer register that holds words that are to be written or have been read. We will refer to the bits of SBR as SBR1 and SBR0. The data within SBR is represented by LEDs to allow users to quickly and easily read the data stored in the buffer.

In order to implement memory addressing, we have a Storage Address Register (SAR) and a counter. The address in SAR is controlled by the switches SAR1 and SAR0, and the register dictates which address in memory the user wants to access. The counter is used to keep track of the current location in memory. Because we are using shift registers for memory storage, we can sync the shift registers with the counter using the clock. With this, each clock cycle will both shift the bits in the registers to the right by one and increment the clock up by one. As such, to address a certain place in memory, the address in SAR and the counter will first need to align before any operations like reading or writing is done to the memory.

To perform the write operation on memory, we would first load in the value we want to write into SBR. This is done by first setting the input switches DIN1 and DIN0 to the desired value and then turning on the LDSBR switch. The data to be written will be loaded within SBR in the next clock cycle. Once the new data is loaded in SBR, we can switch off LDSBR and write this data into memory using the STORE operation. To do this, you will need to set SAR to the desired memory address by using the input switches SAR1 and SAR0. After the address is set, flipping on the STORE switch will write the data within SBR to memory. However, because of how our storage unit is designed, the STORE command may take up to four clock cycles to actually write to memory because it will need to shift over to the correct memory address first before writing the word to the shift register. During these extra clock cycles, the circuit just circulate the old data until it reaches the desired memory location.

To perform the read operation, we would first set the address we want to read by using SAR1 and SAR0. After setting the address, we can flip on the FETCH operation. This operation will wait for the shift register to reach the address appointed by SAR and then transfer the desired bits to SBR. The FETCH command can also take up to four clock cycles to shift over to the desired memory address in order to read its data.

# Written Description of Memory Circuit Implementation

In order to simulate memory storage, we use two 4-bit shift registers to hold data. Each shift register represents a single bit of the words we are storing in memory. To support the STORE command, we used two 2-to-1 multiplexers, one for each bit, to decide whether we want to continue circulating the old data or to write the data in SBR to memory. The select bit to control which option the circuit chooses is decided by the control unit.

For the FETCH and LDSBR commands, we used two 4-to-1 multiplexers, one for each bit, to decide what value to load into SBR. A 4-to-1 multiplexer was chosen because we needed to support three commands (fetching old data, loading in new data, and preserving the current value of the register), and the 4-to-1 MUX was the next smallest multiplexer that was available for us to use. The select bits to these multiplexers were controlled by the control unit. To properly maintain SBR and sync it with the rest of our circuit, we used two flip-flops, one for each bit, to ensure that we are able to preserve the current value of SBR when necessary and to ensure that the value within SBR only changed with the clock cycle.

The control unit consisted of a counter, comparator, and four NAND gates. The counter synced with the clock and was used to keep track of the current memory address in our storage unit. The comparator was used to check if the current memory address, of which the counter kept track, was the same as the address in SAR. If the comparator found these two values the same, the FETCH and STORE operations were allowed to access memory for reading and writing. The four NAND gates in our control unit were used to calculate the values of the select bits for our multiplexers.

# 

# High-Level Block Diagram of Memory Circuit Implementation

# 

*Fig 1: Block Diagram of Memory Circuit Implementation of the Data Storage*

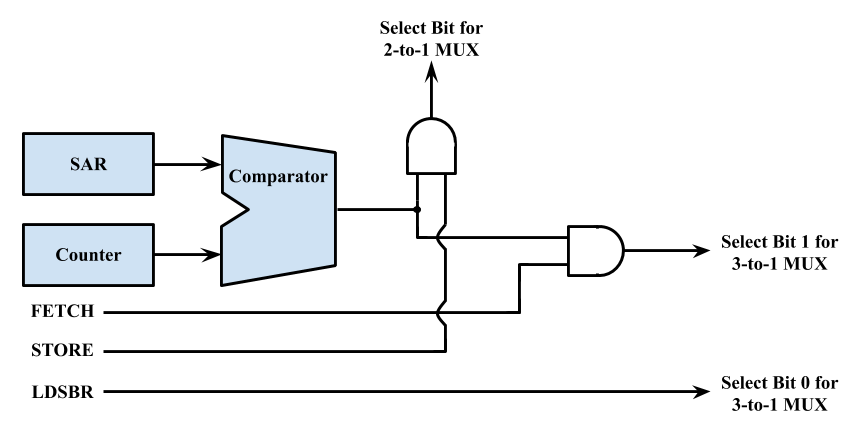
# Written Description of Control Unit

The major components of the control logic are implemented using a counter, a comparator, and NAND gates. This control logic determined the select bits of the multiplexers used.

Since the 2-to-1 MUX must allow new data to come into the register when STORE is high, the select of the 2-to-1 MUX depended on the STORE input and the output of the comparator. If the value of the counter matches that of the value inside the SAR, i.e., the comparator sends out a logic one, and if the STORE switch is on, the select value of MUX would change into a logic one, thus storing the new data from SBR into the memory. Otherwise, our circuit will just circulate the current data in memory.

The FETCH and LDSBR operations were only relevant to the 3-to-1 MUX. We decided to set the DIN input switches as the 01 input to the MUX, which will be chosen when LDSBR is high. The data from the shift registers was set as the 10 input and is chosen when FETCH is switched high and the comparator indicates that the memory address match. Because we used 4-to-1 multiplexers in our implementation of the 3-to-1 MUX, both the 00 and 11 inputs to the MUX were set to the current value in SBR. This allows the circuit to both maintain its current SBR value if no inputs are given and safeguard SBR if the user mistakenly flips on both LDSBR and FETCH at the same time.

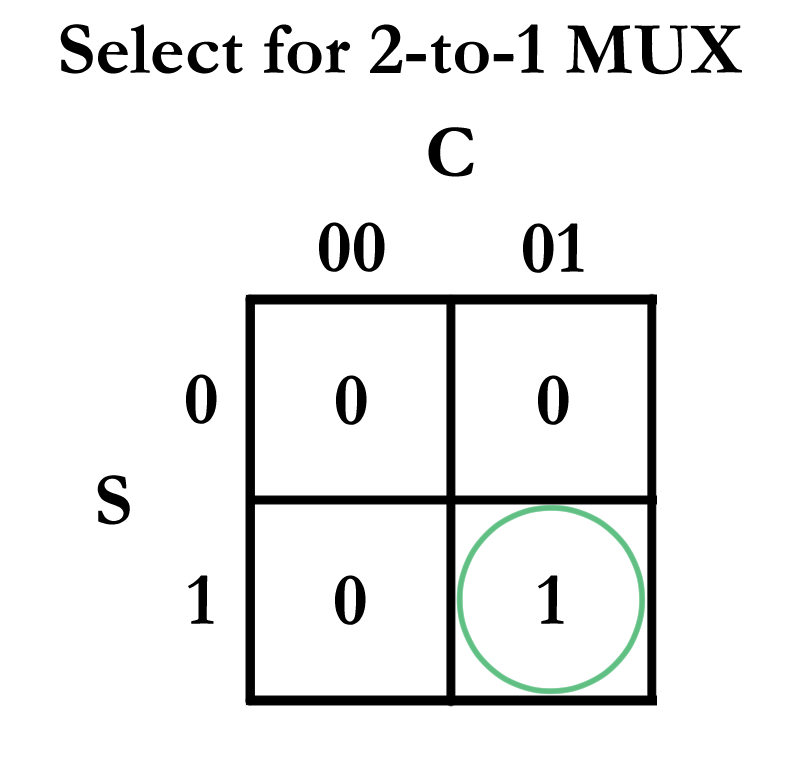
# High-Level Block Diagram of Control Unit

****

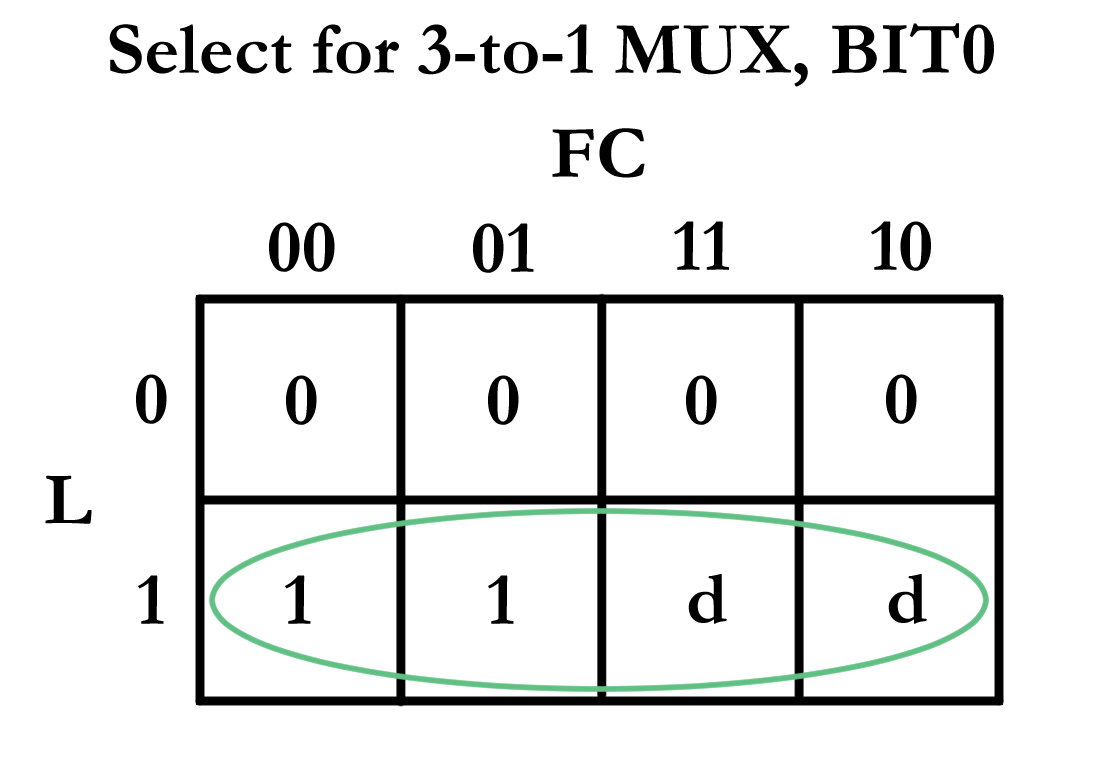
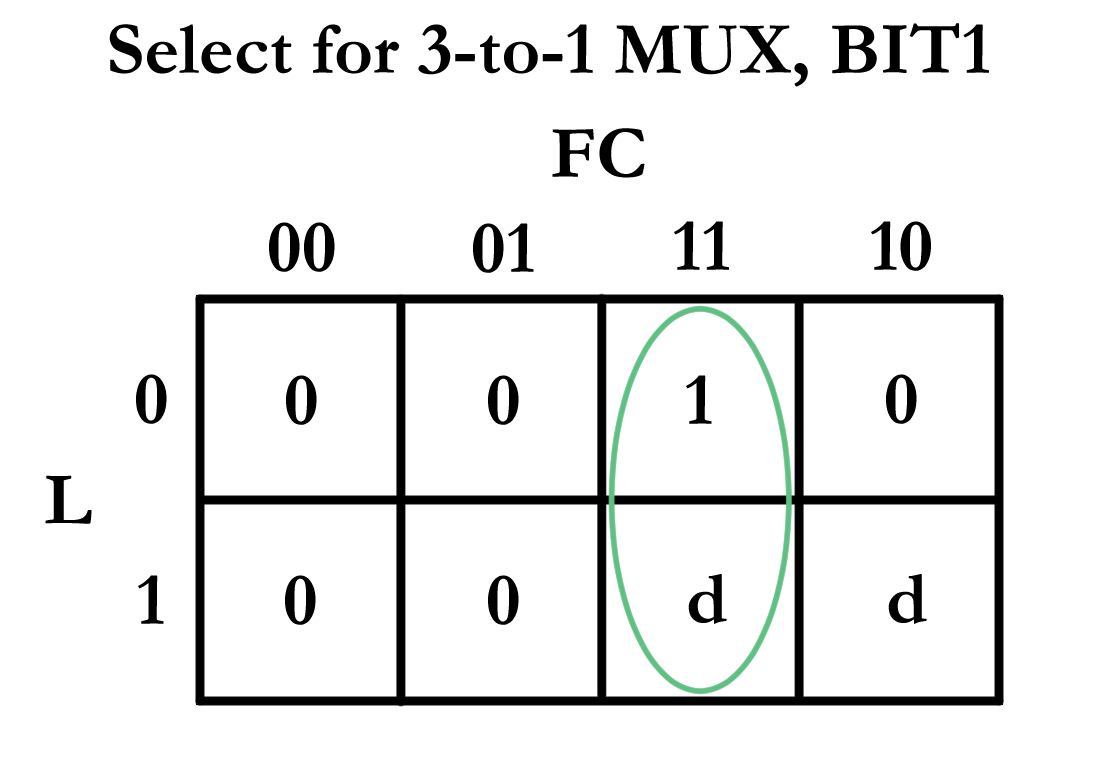
*Fig 2: Block Diagram of Control Unit of the Data Storage*

# Karnaugh Maps and Truth Table of the Design

Karnaugh Map (K-map) has been drawn for the MUX and control components. In the following K-maps, F represents FETCH, S represents STORE, L represents LDSBR, and C represents when the comparator indicates that the counter matches the address in SAR. Because we do not have to consider the case when more than one input switch is turned on, these cases are marked with a “d” for don’t cares because these scenarios should never occur. In order to simplify the logic within our K-maps and truth tables, we only included the essential inputs within these diagrams.

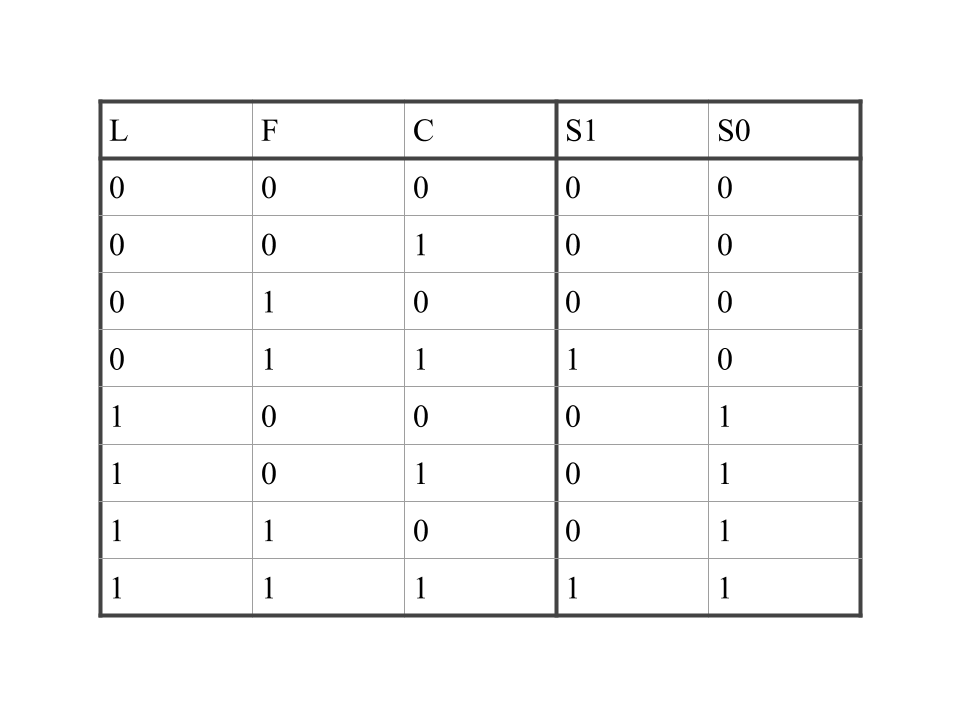


*Fig 3: K-Map for Select of 2-to-1 MUX*



*Fig 4, Fig 5: K-Map for Select of 3-to-1 MUX*

Following the decisions we made in the K-maps, truth tables were also drawn for each of these logics.

****

*Fig 6: Truth table of the logics for the LDSBR, FETCH, and Comparator*

# 

*Fig 7: Truth table of the logics for the STORE and Comparator*

# 

# Written Description of the Design Considerations Taken

The first thing that has to be considered was “how to define the address, and how to store the data to the given address”. Since the shift register right shifts the stored data once each clock cycle while SAR was implemented with switches, this process was harder than it sounds since we had to figure out when the address given by SAR matched the current address of the shift register. It was necessary to design some type of logic to keep track of current location in the shift registers that we can compare with the address in SAR. To resolve this problem, we used a synchronous counter to provide actual numbers that could be used as addresses.

The counter and shift registers will run using the same clock, thus allowing the counter to represent the address of the current register. When the value of the counter is equal to that of the SAR, the comparator will send out the signal, thus providing the circuit with the information necessary to determine whether the current iteration of shift registers are what we desire.

The second aspect we had to consider was about the integration of whole circuit. Implementing each operation as individual components seems easy, but integration everything together into one circuit was a different story. We received inspiration from the high-level diagrams in the lecture — using the multiplexers in order to choose between different operations. The method of how to implement the 3-to-1 MUX was also part of this consideration. We could have used three 2-to-1 MUX chips in order to implement this, but it would be inefficient to use several 2-to-1 MUX chips when we can use a single 4-to-1 MUX chip to do the same job.

Our other considerations for the multiplexers was which operations corresponded to which inputs on multiplexers, and the logic behind the select bits. We realized that not all of the input switches were necessary to choose the output of the multiplexers. For example, the 2-to-1 MUX only depended on STORE and not on LDSBR nor FETCH since only one switch should be on at a time. As such, this factor greatly simplified our logic for our select bits. As for what we should use as inputs to our multiplexers, we decided that it would be best to go with what logically made sense. Although it uses more gates, we chose this design because it was better convention. Because of this decision, for the 2-to-1 MUX, we made the 0 input as the old data within the registers and the 1 input as the new data from SBR. For the 4-to-1 MUX, we made the 00 input as the current data in SBR, the 01 input as the DIN switch, and the 10 input as the output data from the shift registers. The last input, 11, was something that should not be achievable by the constraints given by the lab, but we chose to make this input to also be the current data in SBR in order to cover all of the inputs to the MUX and to ensure that nothing will mess up the circuit in case the user inputs an invalid command.

# 

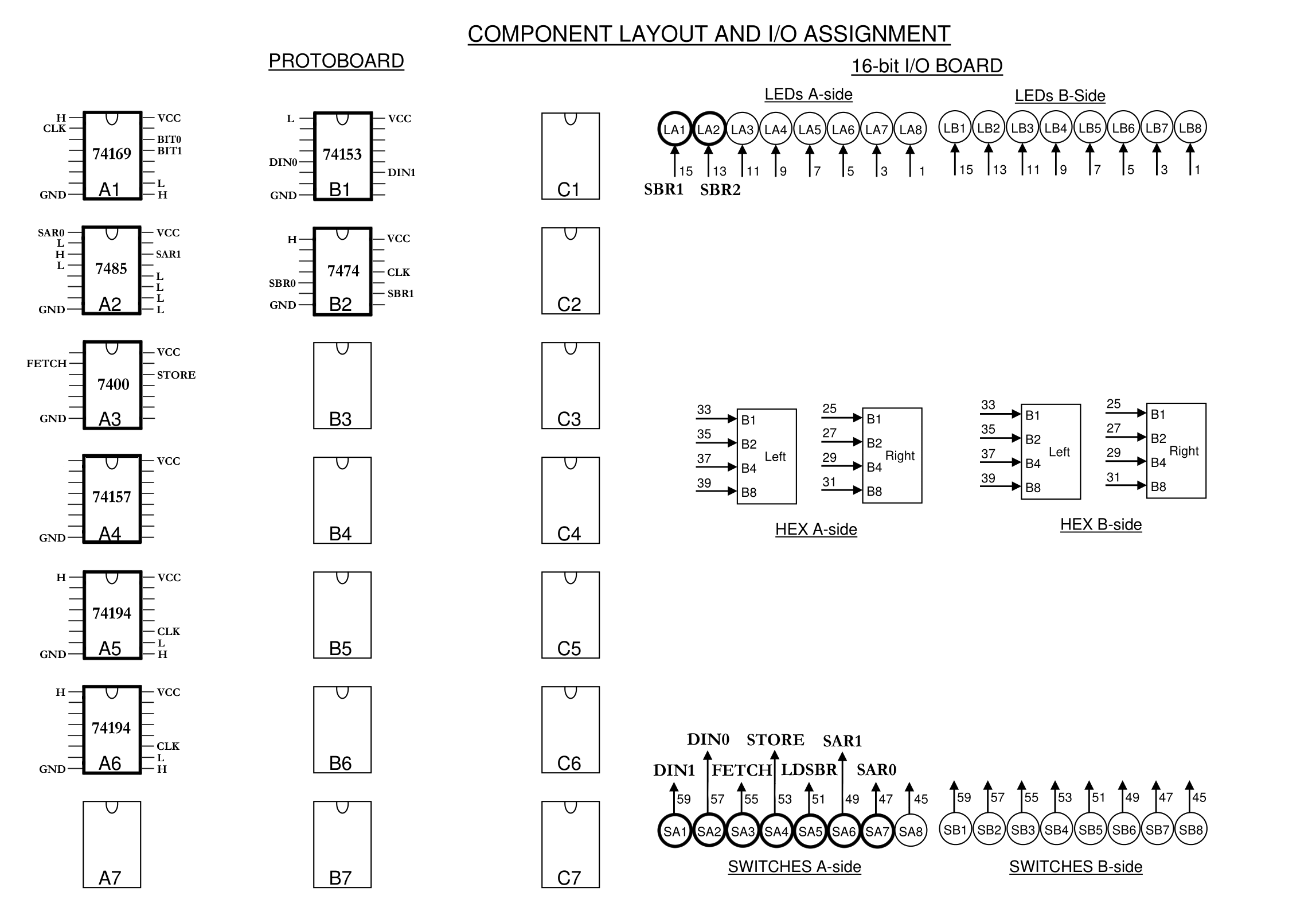
# 

# Detailed Circuit Schematic

# 

*Fig 8: Detailed Circuit Schematic*

# Component Layout



*Fig 9: Component Layout of Data Storage*

# Description of all Bugs Encountered and Corrective Measures Taken

The major bug that we had when implementing this lab was that one of our shift registers outputed only logic zeroes, regardless of what inputs we gave. At first, we thought that we miswired something but after careful examination, we saw that both shift registers were wired similarly, so we ruled out the possibility of miswiring the chip. Our next speculation for the cause of the issue was that the chip broke when we were incorporating it into the circuit because it worked when we unit tested it; however, after separating it from the circuit and unit testing it again, we found that the chip still work. We discovered our issue when reinserting and rewiring the chip. The cause of the issue was a broken wire connecting the clear pin to logic one. The connection stable enough for the wire to report a logic one when we were examining the inputs and outputs of the chip with a wire connected to an LED, but it was just unstable enough to clear our shift register every time our circuit did anything. We corrected this issue by replacing the wire with an unbroken one.

Other than the issue with the broken wire, the only other problem we had while implementing the lab was misaligning rows when wiring. However, this mistake was easily caught and fixed when we double checked our work after implementing each chip, so it did not turn into a major issue for us.

**Answers to all Lab Questions**

Answers to Pre-Lab

1. It is bad practice to gate the clock because doing so may introduce glitches to circuit components that depend on the clock. Additionally, since we want our registers to be shifting continuously on each clock pulse, we should not be gating the clock because this will cause our registers to miss some clock cycles.

The current operation in our circuit at any given moment depends on the state of our input switches. If one of the operation switches is flipped to a logic one, then the current operation of our circuit would to execute the given instruction. If no operation is selected, then the current operation is to circulate the data within the shift registers and wait for an input. When the command FETCH is given, we will load SBR when the desired address in SAR matches the current address in memory we are accessing because this will allow us to load SBR with the value we want. When the command LDSBR is given, we will load SBR at the next clock cycle because this command does not depend on any other factors, and only one command can be given at a time. Our control logic tells our MUX from which inputs to choose. For more details on exactly which select bit combination corresponds to which command, please refer to the description of the circuit above.

1. The only input that will need to be debounced within our circuit is our clock because we have many components within our circuit that are synchronized with the clock. These components need a stable clock input that does not have issues like contact bounce because we need our clock to produce consistent results. As such, we cannot have our clock progress an unknown amount of cycles after each flip due to contact bounce. Since the debouncer circuit solves this issue, we use a debounced switch for our clock.

Answers to Post-Lab

For our shift registers, in order to access a specific place in memory, it can take up to 4 clock cycles to reach the correct memory location. This is because our register shifts one bit at a time in order to get to the right address. However, because SRAM allows for random memory access, it is able to access a specific place in memory in O(1) time. Since our shift registers access memory in O(n) time where n represents the number of words our registers support, this means that SRAM has much better performance than that of our shift register memory.

While designing our circuit, we chose to use the SN74LS169A counter. We chose this counter because synchronously updates every flip-flop, so we will not have to worry about possible glitches that can occur from using ripple counters. We specifically chose to use the SN74LS169A counter instead of the SN74193 counter even though they are both synchronous counters because pins on the SN74LS169A chip is grouped in a more logical fashion. The SN74LS169A chip groups all of its outputs together on one side of the chips, and all of its inputs are grouped on the other side, so it is easier to wire and debug if something goes wrong. For our shift registers, we chose to use the SN74LS194A shift register chips because it is a more versatile chip. With the SN74LS194A chip, we have option to parallel load, shift right, shift left, and do nothing with very little setup. Additionally, getting more familiar with this chip will be more beneficial for us because we will have to use the chip in later labs like lab 3.

# Conclusion

In summary, we created the data storage that can store, load, and fetch the 2-bit data into 4 different addresses — each numbered as 00, 01, 10, and 11. The major challenge of this lab was to manipulating the exact memory, which is constantly changing same as clock cycle, while not differing other memories. We learned that in a situation like this, synchronous designs can be a good solution. At the same time, however, we also learned that gating the clock is poor practice for such designs.

Furthermore, we were able to analyze how this circuitry can be, especially in terms of its efficiency. As aforementioned in the Post-Lab, our circuit has the time complexity of O(n), which is not an ideal time complexity especially when we have a larger data storage unit. Memory implementations like SRAM can drastically reduce this time complexity into O(1), but it comes at a price, literally, in return.