Digital Logic Design Project 2

Zidane Karim

November 18, 2024

Contents

1 Introduction

The assignment was to design a sequential logic circuit that can both write and read a byte into a RAM chip, to be displayed on an array of LED outputs.

In other words, the circuit should be able to take in an 8-bit input and store it in memory, and then be able to read that memory and display it on the LED array.

For example, if the input is 10101000, the LED array should display the following configuration: ↓

Figure 1: LED Output for 10101000

The heart of the problem is manipulating the data around the RAM chip, as it was a 1024×4 RAM chip, meaning it had 1024 memory locations, each with 4 bits of data.

However, recall that the input is 8 bits, so we need to split the input into two 4-bit chunks to store in the RAM chip.

The circuit should be able to write the first 4 bits into the RAM chip, and then write the second 4 bits into the RAM chip, and then read the data from the RAM chip and display it on the LED array.

This manipulation of data is the main challenge of the project, which is why we turn to a sequential finite state machine to solve the problem.

1.1 Outline

1.1.1 Methodology

The circuit is divided into three main parts: the input, the RAM chip, and the output. The essential issue in each was:

- Input: Automatically splitting the 8-bit input into two 4-bit chunks to write to the RAM chip in sequence
- RAM Chip: Keeping the data in memory and being able to read it back out without losing it/moments of gibberish
- Output: Continously displaying the data on the LED array

1.1.2 Implementation

The vast majority of the circuit was implemented using TTL chips. The only exception was the bidirectional register used to hold the data which was a CMOS chip, requiring additional buffering.

2 Methods

2.0.1 Theory

The circuit was designed as two finite state machines, one for read and one for write:

- The **write FSM** would take in the 8-bit input and split it into two 4-bit chunks, writing them to the RAM chip in sequence.
- The **read FSM** would read the data from the RAM chip and display it on the LED array.

Why do we use a finite state machine?

Because the circuit needs to be able to remember the state it is in, and then change states based on the input. This is the essence of a finite state machine. The practical relation to the circuit is that the circuit needs to know whether it is in the write state or the read state, and then change addresses in the RAM chip accordingly. based on the details of the state.

This comes from the fact that the RAM chip has a limited address width (10 bits for 1024 locations), which requires control over address selection to store and retrieve the data correctly.

In addition to addresses, the RAM chip requires a write-enable signal to write data to memory. This signal is controlled by the write FSM, which is responsible for writing the data to the RAM chip in the correct sequence.

The design of the RAM chip has certain time-intervals where between address changes, the data written to memory is invalid/incorect. Because of this, the write FSM must be designed to wait for the correct time to write the data to memory, and then change the address to write the next data.

However, the timing intervals on the 2114 TTL RAM chip is measured in nanoseconds, which our clocks generated from 555-timers cannot feasibly reach due to equipment availability. Thus, this timing was not of much concern considering our RAM chip would never be able to reach the speeds required to cause a problem. Because of this, we do not really care about an idle state in both FSMs as the RAM chip will never be able to write/read data fast enough to cause a problem. But in my original thinking, I designed the states with an idle in mind.

To summarize, the finite state machines are responsible for selecting the correct address and enabling the read or write operation as required by the process:

Why do we use external registers?

The RAM chip has a 4-bit data width, which means it can only store 4 bits of data at a time. However, the input is 8 bits, which means we need to split the input into two 4-bit chunks to store in memory.

To do this, we use two 4-bit registers to hold the data before writing it to memory. The first register holds the first 4 bits of data, and the second register holds the second 4 bits of data. Why do we use registers instead of reading from RAM directly?

This is because the RAM chip has a limited address width, which means we need to write the data to memory in the correct sequence. If we read the data from memory directly, we would not be able to control the sequence in which the data is written to memory. In other words, we want to view all the data at once, not just the 4-bits at a time. The external registers also store the values inside so we can change the values on the dip switches, yet the values on the

LED array remain the same since we have not written to the RAM.

2.0.2 State Transitions and Truth Tables

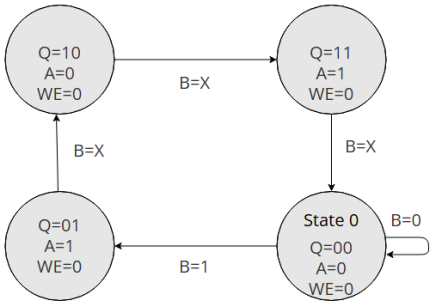

Figure 2: Read State Transition Diagram

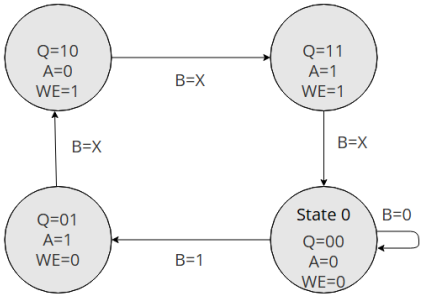

Figure 3: Write State Transition Diagram

Above are the two state transiton diagrams for read and write respectively. Notice that they almost exactly the same, except the write FSM uses the write enable signal to write to memory, while the read FSM leaves it off the entire time. These FSMs are Mealy Machines as outputs rely on the button presses as well as the original states. The essential logic of both FSMs is to stay idle until the button is pressed, which then cycles through the diagram. At each point, the machine switches addresses and either writes or reads.

Below is the truth table for the FSMs, which is used to create the Karnaugh Maps for the FSMs. The truth table is the same for both FSMs, except the write FSM has an additional output for the write enable signal. From this, we can derive logical expressions and then simplify them to create the FSM circuits.

	Input	State Variables		Output			Control Variables		State Output Variables	
State	Read Button	Q1^n	Q0^n	WE	Address	Mux Select	D1	D0	Q1^n+1	Q0^n+1
1	0	0	0	0	0	0	0	0	0	0
2	0	0	1	0	1	1	1	0	1	0
3	0	1	0	0	0	0	1	1	1	1
4	0	1	1	0	1	1	0	0	0	0
1	1	0	0	0	0	0	0	1	0	1
2	1	0	1	0	1	1	1	0	1	0
3	1	1	0	0	0	0	1	1	1	1
4	1	1	1	0	1	1	0	0	0	0

Figure 4: Read Truth Table

	Input	State Variables		Output			Control Variables		State Output Variables	
State	Write Button	Q1^n	Q0^n	WE	Address	Mux Select	D1	D0	Q1^n+1	Q0^n+1
1	0	0	0	0	0	0	0	0	0	0
2	0	0	1	0	1	1	1	0	1	0
3	0	1	0	1	0	0	1	1	1	1
4	0	1	1	1	1	1	0	0	0	0
1	1	0	0	0	0	0	0	1	0	1
2	1	0	1	0	1	1	1	0	1	0
3	1	1	0	1	0	0	1	1	1	1
4	1	1	1	1	1	1	0	0	0	0

Figure 5: Write Truth Table

2.0.3 Boolean Expressions

Read:

- $D_1 = Q_1 \oplus Q_0$
- $D_0 = (R_B + Q_1) \cdot \overline{Q_0}$
- $OE = \overline{WE}$

Write:

- $D_1 = Q_1 \oplus Q_0$
- $D_0 = (W_B + Q_1) \cdot \overline{Q_0}$
- $WE = Q_1$

Both:

- $A_0 = Q_0R + Q_0W$
- $MuxSelect = Q_0R + Q_0W$

The tri-state buffer controllers are connected to the write enable signal, because logically the RAM should only receive the inputs when write enable is on.

The mux select is connected to the address signal, because we want to select the proper address from the mux based on the state of the FSM.

Output enable does not actually exist on the RAM chip but it does in Logisim. It is connected to the inversion write enable signal, because we only want to read the data when the write enable signal is off. Since WE is off by default, when we are not writing, we are reading.

The final puzzle to solve is the register clock. The register is not hooked up to the 555-timer clock, because then it would read on every clock, which we do not want. We only want to update the register's value to the LED when we press the read button. Since the Q_0 of Read controls the address, we can connect the output of Q_0 to the first register and the inversion/negation of this signal to the other register. This way, when we press the read button, the register will update to the value of the LED array.

2.0.4 Functional Block Diagrams

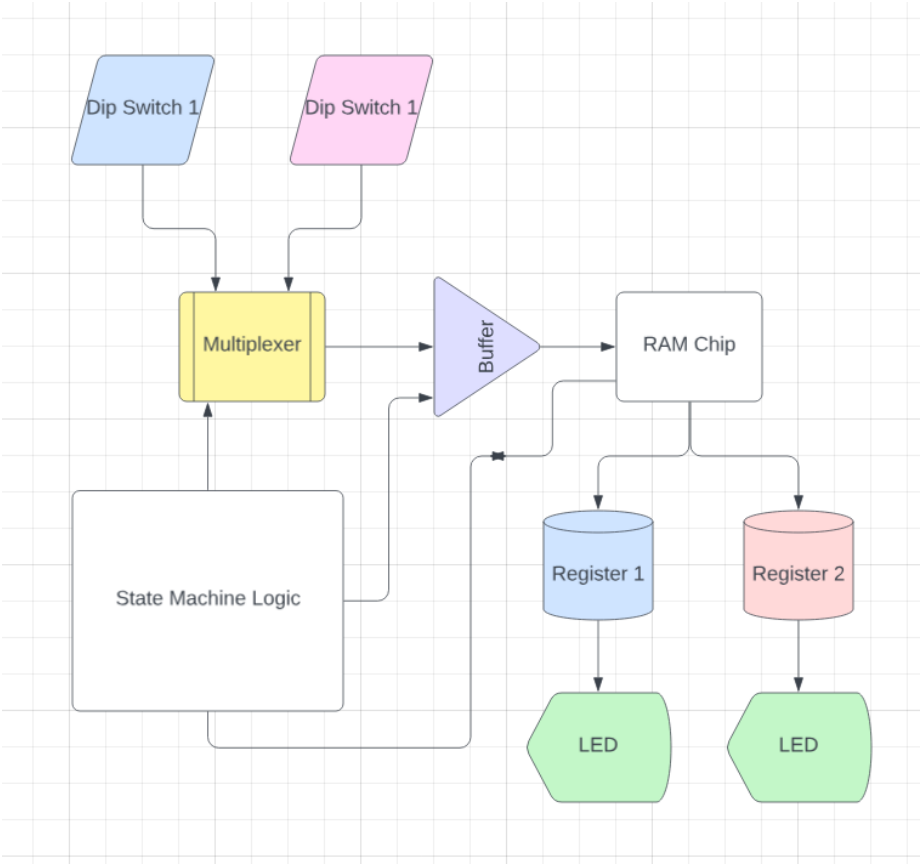

Figure 6: Functional Block Diagram
This follows the logic of the entire circuit. The input is split into two 4-bit chunks, which are then written to the RAM chip. The RAM chip is then read and displayed on the LED array through storage on the registers.

2.0.5 Logic Diagrams

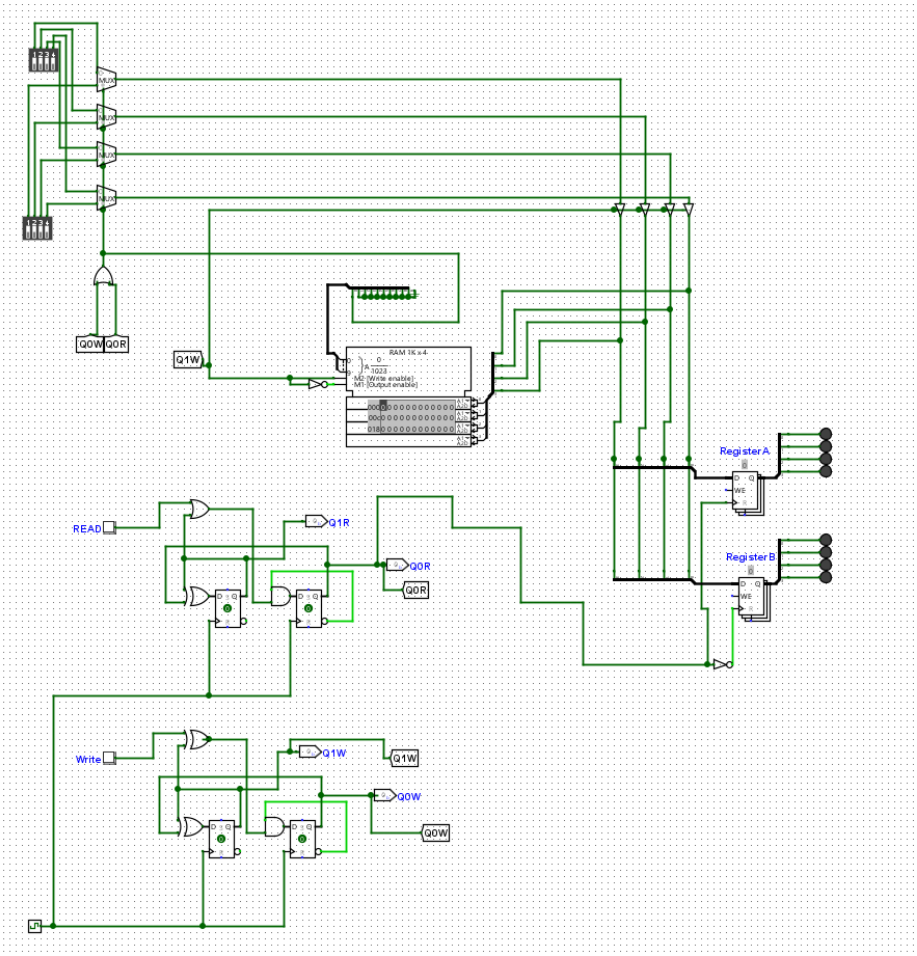

Figure 7: Entire Circuit

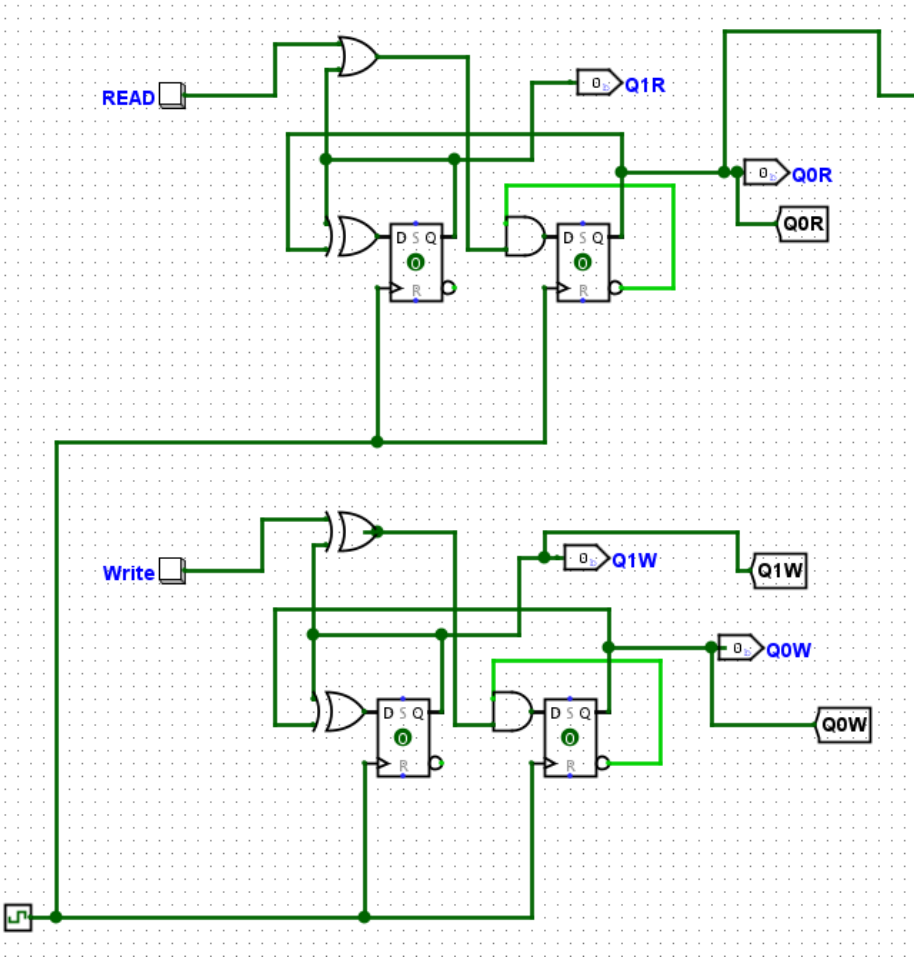

Figure 8: Read and Write FSM Circuits

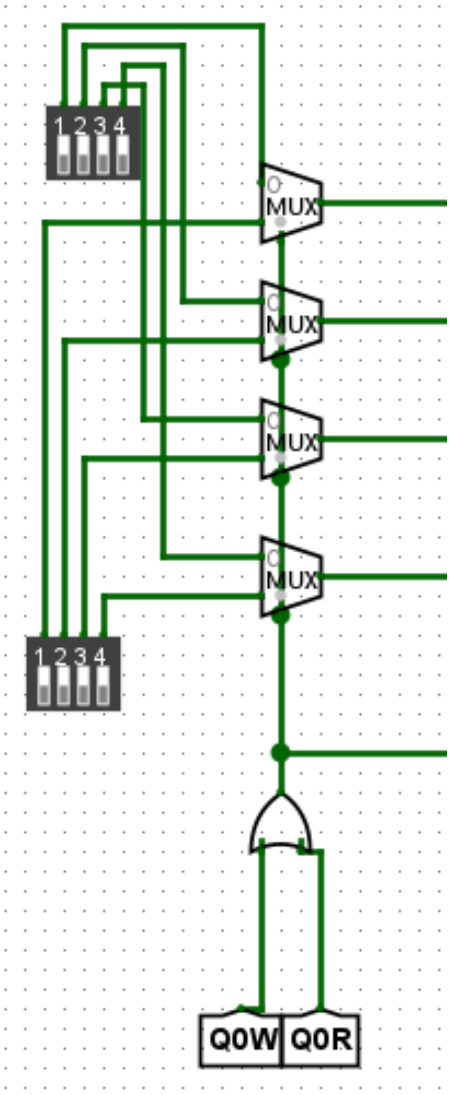

Figure 9: Muxes and Inputs

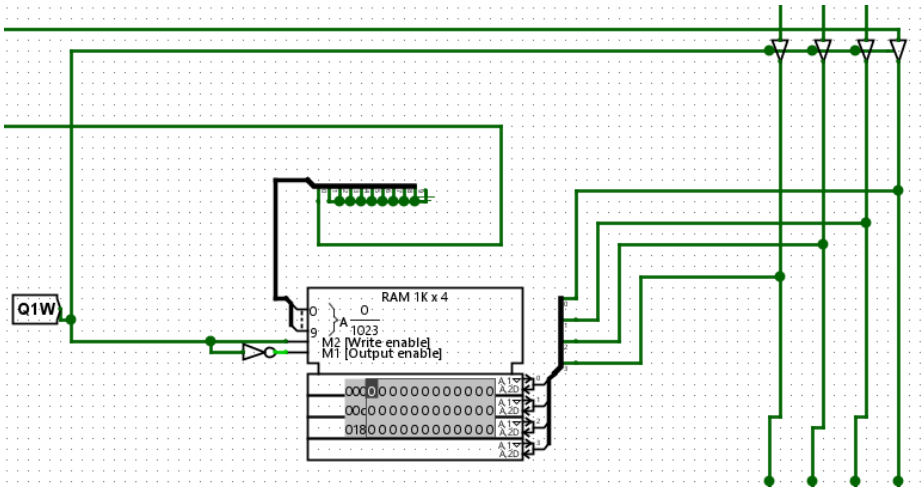

Figure 10: RAM Chip

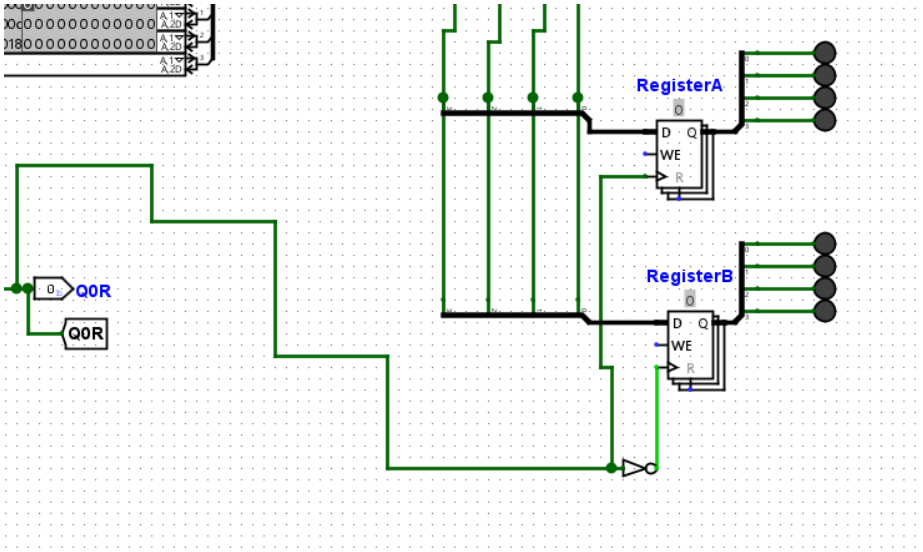

Figure 11: Registers and Outputs

3 Implementation

3.0.1 Components

The circuit was implemented using the following components:

- 2× D-Flip Flops (TTL)
- 1 Quad XOR gate (TTL)

- 1 Quad OR gate (TTL)
- 1 Quad AND gate (TTL)
- 1 Schmitt Trigger (TTL)
- 2× 4-bit registers (CMOS)
- 1 2114-IC RAM chip
- Hex buffer w/ open collector (TTL)
- 1 Quad 2:1 Multiplexer w/ built in buffer (TTL)
- 2× Dip Switch
- 2× button
- 555-timer (TTL)
- 8× LED
- 2× Speaker

3.0.2 TTL-CMOS Interfacing

Every chip used was TTL, except for the 4-bit registers. The registers were CMOS, which required additional buffering to interface with the TTL chips. This was done using a hex buffer with open collector outputs, which allowed the CMOS chip to interface with the TTL chips.

Why do we need this buffering?

The TTL-chips operate at a lower/more forigiving voltage threshold as compared to the CMOS chips. So if a TTL signal goes into a CMOS chip, the CMOS chip may not recognize the signal as a high or low, because the voltage is not high enough. The buffer fixes this problem, converting the TTL signal to a CMOS signal that the CMOS chip can recognize properly.

3.0.3 Circuit Pictures

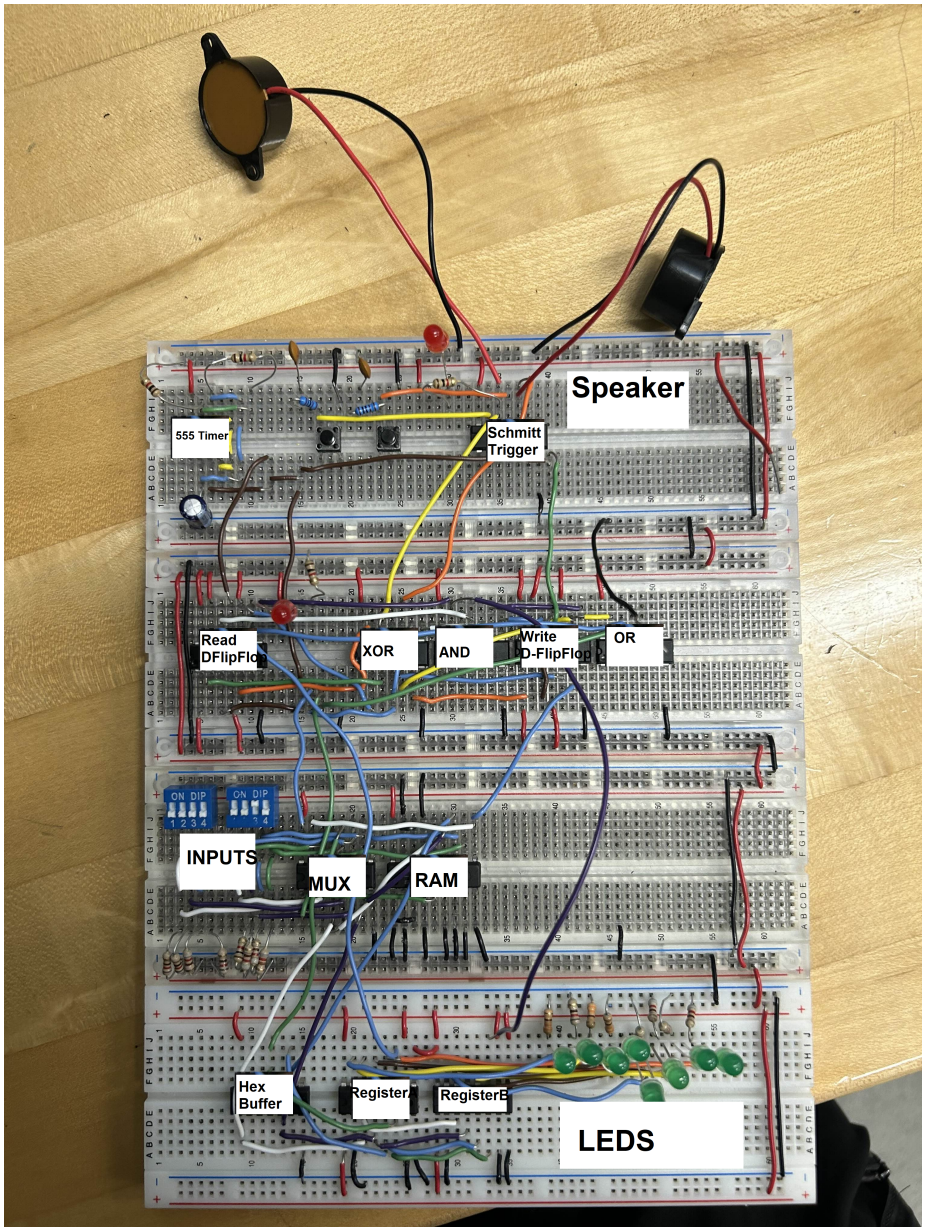

Figure 12: Labeled Circuit

4 Conclusion

4.1 Results and Summary

The circuit worked well, with outputs not changing even if the dip switches changed (unless I wrote first.) Something I noticed was that the LED array would display the values of the dip switch on the first connection to power, but as soon as I wrote/read proper values, it worked as noirmal.

Once again, the initial question was to design a circuit that could write and read data to a RAM chip, and display it on an LED array. It should be able to take in an 8-bit input, split it into two 4-bit chunks, write them to memory, and then read them back out, without losing information or displaying gibberish. This was achieved through the use of two finite state machines, one for write and one for read, which controlled the address selection and write enable signal for the RAM chip. The data was stored in two 4-bit registers before being written to memory, and then read back out and displayed on the LED array.