The Impossible Game

ECE 150 Digital Logic Design Final Project

Toyou Kim: tkim@cooper.edu
Yash Sharma: sharma2@cooper.edu

Spring 2015

1. Abstract

For our final project, we made a game called The Impossible Game. In the game, the player moves vertically, i.e. jumps and falls, while the map shifts leftward with each clock cycle. This produces the effect of the player moving rightward relative to the still map. While in our initial proposal, we intended to use the Shift Registers only for displaying the map, we ended up using the Shift Register to resolve most of the issues. A push button was used to make the player "jump," i.e. moves up and down, and Shift Register with logics was used to make sure that the player jumps only when the condition was met – that is, when the player was either on the ground or on a block. An EEPROM was used to store the map, and a 12-Bit Up Binary counter is used for the address selector pin of EEPROM to shift the map leftward every clock cycle. 64 LEDs soldered like a squared grid format (Resembling the appearance of 8x8 LED matrix, but not using the common cathode/anode way) was used to display the game. We also had a 7-Segment Display used to display the number of attempts, and a reset button that will allow you to reset the game. To display the number of attempts, a counter was used, and the clock to the counter was the output of our logic for registering a collision. This logic, done by the use of logic gates and a monostable pulse, was sent to reset pins on all of the devices that govern the appearance of the player and the map.

Through this project, we learned how to utilize shift register to do numerous things, and was amazed by the versatility and powerfulness of the Universal Shift Register. Apart from that, it was an opportunity to ponder about timing elements, since the player had to jump on the block exactly on the right time and so on. Our project demonstrated the usage of memory, counting, creation of state machines, soldering and constructing display abilities, and used every device we have talked about in class except for multiplexers and demultiplexs, as using logic gates ended up being more efficient as there the multiplexer/demultiplexer logic was easily done by maximizing the number of gates used on every logic chip. By strategizing how to complete this project, going through the debugging, and tackling the long hours required for the construction of the board design and display design, we truly feel that this project has given us a greater understanding of how to utilize the devices taught to us to create a functioning, complex project. We thank you for giving us this opportunity.

2. Design Narrative

2.1 Map

The map was stored in one AT28C64B, a 64K (8K x 8-Bit) Parallel EEPROM. While Random Number Generator (RNG) was also considered an option at the planning stage, an EEPROM was used instead for the following reasons: 1. Limitations to randomness – In order to make it possible to clear the game, randomization had to be restricted in certain ways, e.g. at most one obstacle in any given column, spikes only on the ground. 2. Increasing difficulty – It was considered overly complicating, actually seemed impossible, to set up the RNG so that the difficulty of the generated map would progressively increase.

In order to access the map data in the EEPROM, a 12-Bit Binary Up Counter, set up by daisy-chaining three CD4516BE, 4-Bit Binary Up Counter, was used. $Q_0Q_1Q_2Q_3Q_4Q_5Q_6Q_7Q_8Q_9$ Q_{10}, the output pins on the counter were connected to $A_0A_1A_2A_3A_4A_5A_6A_7A_8A_9$ A_{10}, the corresponding address pins on the EEPROM. This would enable the map to shift leftward by one bit, for every clock cycle that operates the Binary Counter. The counter was also used to stop the map shifting after the player completes the game. (See *2.6 Miscellaneous*)

For the interaction between the map and the player, and for the display, 8 8-Bit SIPO (Serial-In, Parallel-Out) Shift Registers, set up by daisy-chaining two 4-Bit Shift Registers, were used. (In this project, 74HC194E, 4-Bit Bidirectional Universal Shift Register, was used for everything, so this documentation will specify how the shift register was used, so the reader could reconstruct the project with substitute chips) Parallel-Out functionality was required because parallel access to all eight outputs was needed for all the interaction mechanism. Each shift register accounted for a row in the 8x8 LED grid. For the n^{th} row, I/O_n of the EEPROM was connected to the D_{SL} (Data Shift Left) pin. S_1 was connected to power, and S_0 to ground, for shift left. More details on the usage of data output from each Q pin will be addressed in the related sections.

2.2 Player

The player was configured as a Logic High in an 8-bit Bidirectional PIPO Shift Register, set up by daisy-chaining two 74HC194E. This configuration was possible because in the actual game, the player will only move vertically, i.e. jumping and falling, as the map moves leftward, thus simulating the player moving right relative to the map.

In the LED Grid, the player is present on the 2nd column to the left, and it interacts with the map presented in the same column. In the data array $P_0P_1P_2P_3P_4P_5P_6P_7$, where P_0 is the bottom bit of column and P_7 the top bit of column, the default position of the player was P_1, because the row P_0 belongs to is the "ground layer," crucial part of the design for collision interaction. (See *2.5 Collision*) Because a Logic High in P_1 bit of the array is not initially present in the Shift Register, we needed the Parallel-In functionality to set up a value of $P_0P_1P_2P_3P_4P_5P_6P_7$ = 01000000 for initialization and after every collision. (See *2.5 Collision*) Parallel-Out functionality was required for all the interaction mechanisms, i.e. display, jump, fall, collision. The bidirectional feature was an essential function required for jumping and falling of the player, which was the main part of this project.

The data shift direction of the chip is dependent on the inputs in the S_0 and S_1, thus setting up the adequate logics for these two inputs was the most important part of the project. (Because the S_0 and S_1 of every other Shift Register used in the project are connected to either power or ground, we will use S_0 and S_1 to denote the pins on the player Shift Register) The logical expressions of S_0 and S_1 can be expressed as follows: $S_0 = S^*_0$ OR Collision Clock Pulse (CCP), and $S_1 = S^*_1$ OR CCP. S^*_0 and S^*_1 are logics required for the jumping and falling, while CCP is required for initial and post-collision parallel loading of $P_0P_1P_2P_3P_4P_5P_6P_7$ = 01000000. (See *2.3 Jump, 2.4 Fall, 2.5 Collision*) As an additional feature, MR' (Due to technical limitation, we will use X' for 'not X', using apostrophe instead of overbar. Here, apostrophe indicates that function is activated when the input is Logic Low), Master Reset pin which resets all the data value to 00000000, was connected to Q'_4 of Collision Counter output, which will be discussed in Collision part.

2.3 Jump

One 4-Bit SIPO Shift Register was used to implement the jump mechanism. Before start building, operational definition of "jumping" was set to identify the exact conditions to fulfill. Jump was defined as follows: A jump is followed by transmission of user input, which occurs by pressing a push button. Jump input is accepted only when the player is "on the ground"; that is, jump input is not accepted when the player is in the process of jumping, or in the process of falling, which may happen after jumping depending on player's orientation to ground or flat obstacle, i.e. block. With every input, the player shifts "upward," which is shifting rightward within the player Shift Register, by two bits relative to its current location in the Shift Register. After shifting upward by two bit, the player is going to "stay in air," i.e. does not change its location within the Shift Register, for one clock cycle, which terminates a jump sequence.

It was immediately obvious that controlling the S_0, S_1 of the player Shift Register was the key to solution. Moreover, since the jump input was dependent on the current state of the player – on the ground, it could be deduced that S_0, S_1 were going to be included in the logic expression. The main difficulty was finding the way to hold onto the temporary Logic High value that occurs with push button, which was resolved by using Shift Register. Afterward, we created a state table of S_0, S_1 over a jump sequence.

	Q_n	Q_{n+1}	Q_{n+2}	Q_{n+3}	Q_{n+4}
S_1	0	0	0	0	(Falling)
S_0	0	1	1	0	(Falling)

Table 1 State Table for Jumping Mechanism,

When the player is "on the ground," it does not move – which means both S_0, S_1 = 0. Afterward, since the data shifts rightward two bits, for two clock cycle S_0 = 1. Lastly, before the sequence terminates, the player stays in its location for one bit – S_0, S_1 = 0. Logic for this state table is provided in the figure below, colored black.

D_{SL} pin receives an input through the AND gate for one clock cycle, which will travel across the shift register, and returns the adequate logic values through NOR/OR gates for given amount of clock cycles. Therefore, it is as important as the logic itself to make sure that D_{SL} receives a Logic High for only the first clock cycle when the user initially presses the push button. Red colored part is responsible for the task, which will be discussed in detail later. (See *3.2 Encountered Problems*)

2.4 Fall

As it is already mentioned above and written down in the state table, when the jump sequence ends after the player reaches the top of jump, its action afterward is governed by falling mechanism. Only the S_1 required attention for falling mechanism, since Logic High of S_0 will results in either parallel load or shifting right, or upward viewed from the display, as neither is expected. Thus, value of S_1 is affected by different parts of logic expression at different intervals.

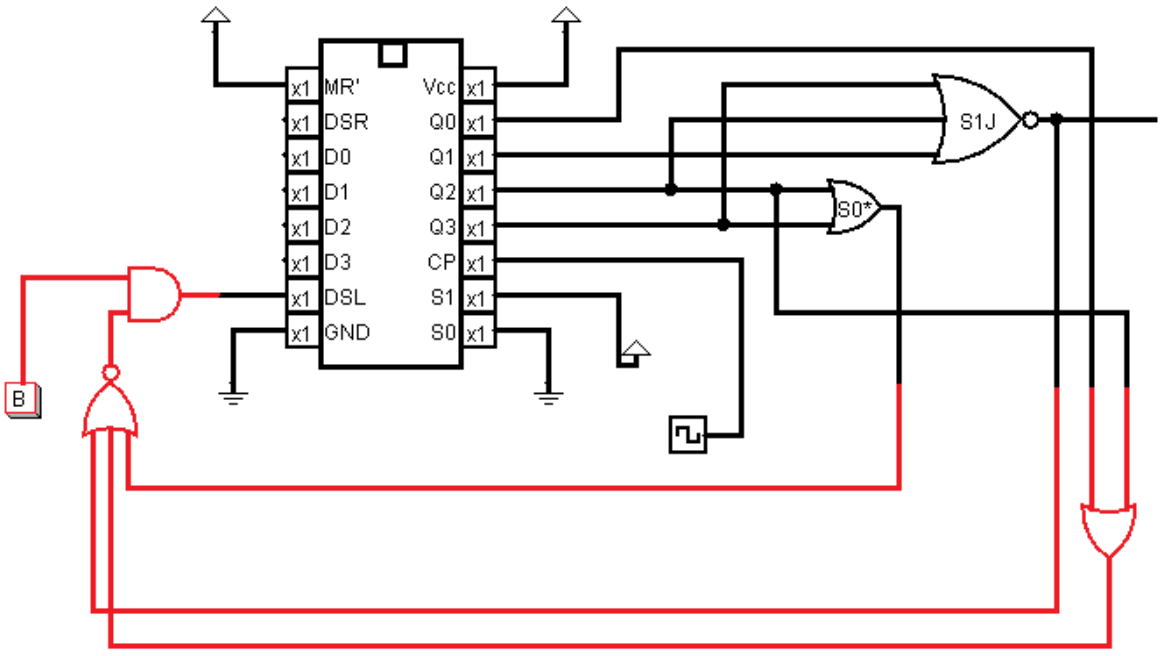


Figure 1. Logic Diagram for the Jumping Mechanism. Black part accounts for the logic for the desired S_0, S_1 value based on the state table. Red part works like a buffer, which does not allow D_{SL} (Data Shift Left) to get Logic High, even if the player is keep pressing on the push button.

In the Fig.1 above, it can be seen that the OR gate is labeled as "S_{1J}," which indicates the part determines the value of S_1 during the jump sequence. As it is seen in the logic, S_{1J} returns Logic High after the jump sequence, until another jump sequence is activated. Here, S_{1J} is connected with S_{1F}, which determines S_1 value during the falling interval, via AND gate. Further details are given in the figure below.

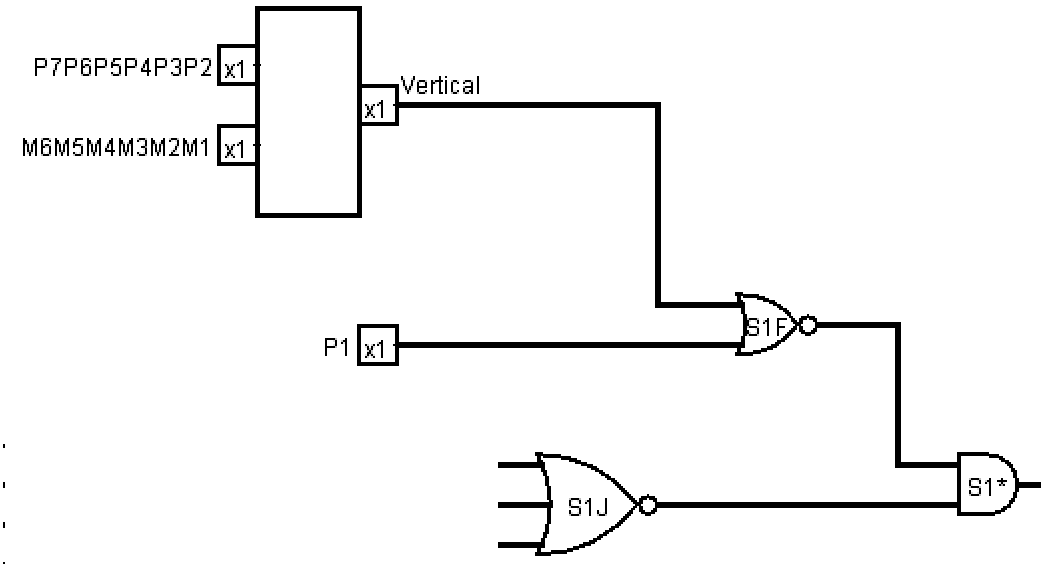


Figure 2. Logic Diagram for the Falling Mechanism. Comparator was used determine whether the player was standing on a block. P_1 should be high when the player is on the ground. Hence, either value being 1 (They actually can't be 1 simultaneously) indicates the player is "on the ground." By connecting two values with NOR gate, it only returns a 1, which means the player is "in the air", thus falling until it is on the ground."

One 6-Bit Comparator, set up by connecting the A = B outputs of two 4-Bit Magnitude Comparators, CD4585B, with AND gate was used to verify if an obstacle was right under, i.e. one bit under, the player. This was achieved by setting up the comparator so that value of each bit of player Shift Register would be compared to the data of map of 7^{th} column that is one bit under. Comparator was added to the design after the initial proposal, in order to reduce on the size of unnecessary logic that is not directly relevant to the purpose of the project. However, this usage of comparator instead of pure logic produced one restriction in map.

2.5 Collision

Collision element is extremely important, as the output that becomes Logic High when collision occurs is inputted into multiple pins to ensure complete resetting of the game. In the game, there are two types of collision: horizontal collision and vertical collision. Horizontal collision occurs when the Logic High in player Shift Register that accounts for the position of player is identical to that of the map. For this logic, a 6-Bit Comparator similar to the one used for falling was used. However, in this case the data of player Shift Register was exactly compared to that of Map Shift Register that accounts for 2^{nd} column to the left. ($P_6P_5P_4P_3P_2P_1$ compared to $M_6M_5M_4M_3M_2M_1$) Both obstacles – block and spike – will produce horizontal collision.

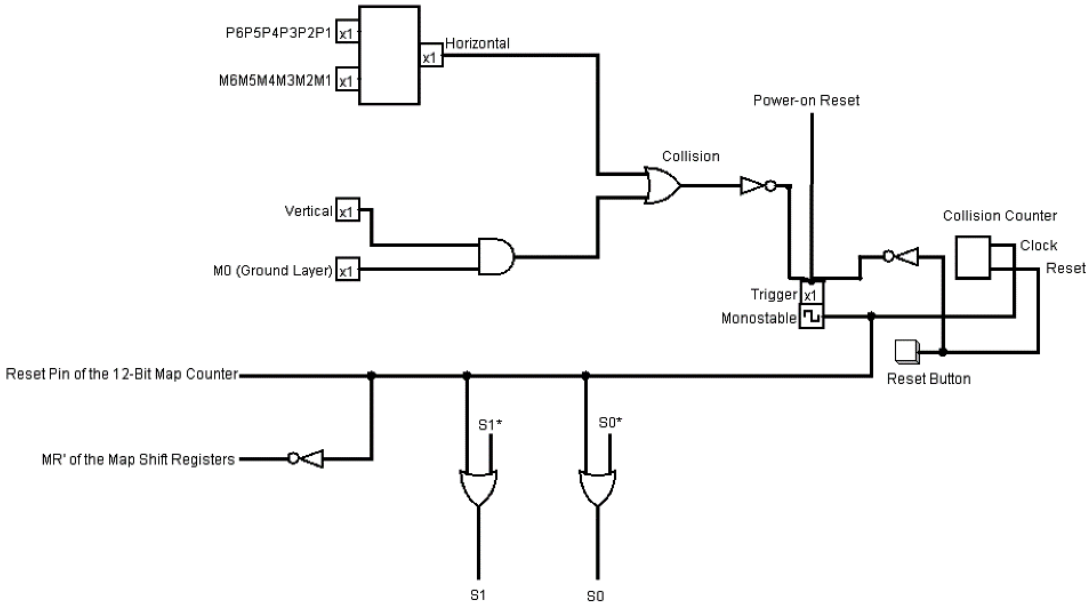


Figure 3. Logic Diagram for Collision Mechanism. In order to avoid timing element problem, instead of direct application, NOT of collision output is connected to the trigger pin, which will produce a clock pulse for every collision. The monostable clock pulse is connected to Reset Pin of the Map Counter, resetting the addresses of EEPROM, thus restarting the map. Similarly, NOT of the clock pulse is connected to all the MR' of the Map Shift Registers, resetting everything in the Shift Register to 0.

The second type of collision, vertical collision, occurs only between the player and spike. The ground layer, whose original purpose is to differentiate between a block and a spike, become very useful for the task. The result from the vertical interaction between an obstacle and the player obtained for falling mechanism indicates that an obstacle is right under the

player if the A=B output is Logic High. Since the obstacle is a spike when the ground layer is lit up – That is, the data output at the ground layer row is Logic High, Vertical AND M_0 = 1 can be interpreted as vertical collision. Collision is then given by Horizontal OR Vertical. After acquiring the logic for collision, it is also as important to set up the system that utilizes this value to reset the whole game. The details are provided above. (Fig. 3)

It should be noted that the clock pulse produced on collision and S^*_0, S^*_1 are connected via OR gate, respectively. This is to have the player Shift Registers parallel load in initialization and post-collision. Apart from that, a reset button is also configured, so the player can reset the whole game, including the number of attempts. Collision counter is used to indicate the number of attempts the user made so far in the game, and at same time to challenge the user by limiting the number of attempts.

2.6 Miscellaneous

When the user clears the game, the screen will not move leftward anymore. This is done by connecting the Q_{11}' of the 12-Bit Map Counter to the reset of the astable 555 Timer that operates the 12-Bit Map Counter, since Logic High on Reset pin will stop the 555 timer.

For the collision counter, which counts the number of attempts, is also used to suspend user from trying too many times, thus making the game more challenging. This is done by connecting the Q_4' to the A12 (Address 12 Pin) of the EEPROM. This leads the user to go to a region he cannot normally reach, since the game ends within 1024 bit. The map will be filled over with "GAME OVER" continuously, until the user uses the reset button to reset the whole game.

For switch debouncing, while we proposed to use the Schmitt Trigger to do the work, we ended up using the capacitors, due to various reasons, including the space on the breadboard.

Color Scheme used for the project was as follows: Green wires were used to set up the Map Shift Registers, orange wires were used for setting up the logical expression of S_1, purple

wires for the logical expression of S_0. Brown wires were used to set up the collision logic, and blue wires were used to set up the astable 555 timer, the counter that selects the addresses of EEPROM, and the push button where the player inputs the jump. Finally, white wires were used to connect the 12-Bit Map Counter to address pins of the EEPROM.

Inventory

Chip Number	# of units used	Chip Name	Purpose
CD74HC194	19	High-Speed CMOS Logic 4-Bit Bidirectional Universal Shift Register	Shifting of the Map, Storing of Player, Logic for Jumping and Falling
CD4516B	3	CMOS Binary Presettable Up/Down Counters	Addresses for EPROM
ICM7555	2	General Purpose CMOS Timer	Astable for timing, Monostable for Reset Logic
CD4071B	3	CMOS Quad Dual-Input OR Gate	Logic for allowing Map and Player in same column, Jumping and Falling Logic, Collision Logic, Logic for removing Button Press Interference while Jumping and Falling feature is enabled
AT28C64B	1	64K (8K x 8) Parallel EEPROM with Page Write and Software Data Protection	Storing of Map
CD4069UB	2	CMOS Hex Inverter	Reset Logic
CD4585B	4	CMOS 4-Bit Magnitude Comparitor	Jumping and Falling Logic, Collision Logic
CD4081B	2	CMOS Quad Dual-Input AND Gate	Jumping and Falling Logic, Collision Logic, Logic for removing Button Press Interference while Jumping and Falling feature is enabled
CD4025B	1	CMOS Triple 3-Input NOR Gate	Jumping and Falling Logic, Collision Logic, Logic for removing Button Press Interference while Jumping and Falling feature is enabled
CD4518B	1	CMOS Dual BCD Up-Counter	Counting of # of Attempts
CD4543B	1	CMOS BCD-to-Seven-Segment Latch/Decoder/Driver For Liquid-Crystal Displays	Display of # of Attempts
Total Number of ICs		**39**	

Other Items Used and Purpose:

Extra Large Perfboard Solder Prototype Board (1)- Storing 8x8 LED Matrix in an aesthetically pleasing manner while also allowing for each LED to be individually controlled

Ribbon Cable 4 Wire (16)- Most neat method of connecting leeds of the LEDs to the outputs on the breadboards

IDC Header (8)- Neatly connecting Ribbon Cable from LED matrix to outputs on Breadboard

Button (2)- Used to trigger jumping, as well as reset of the game when GAME OVER

Seven-Segment-Display CC (1)- Display of number of attempts

LED (64)- Used to display the game

Resistor (71)- 64 connected to LEDs, 7 connected throughout the circuit for timing and such

Capacitor (5)- 2 connected to the 555 timers for period, two connected to push buttons for debouncing, and 1 used as a decoupling capacitor.

Insulated Wire- Used to create our game

Rosin Core Solder- Used to attach LEDs to the perfboard, attach resistors to the LEDs, attach ribbon cable to the resistors, attach ribbon cable to IDC headers

Breadboards (13)- Used to create our game

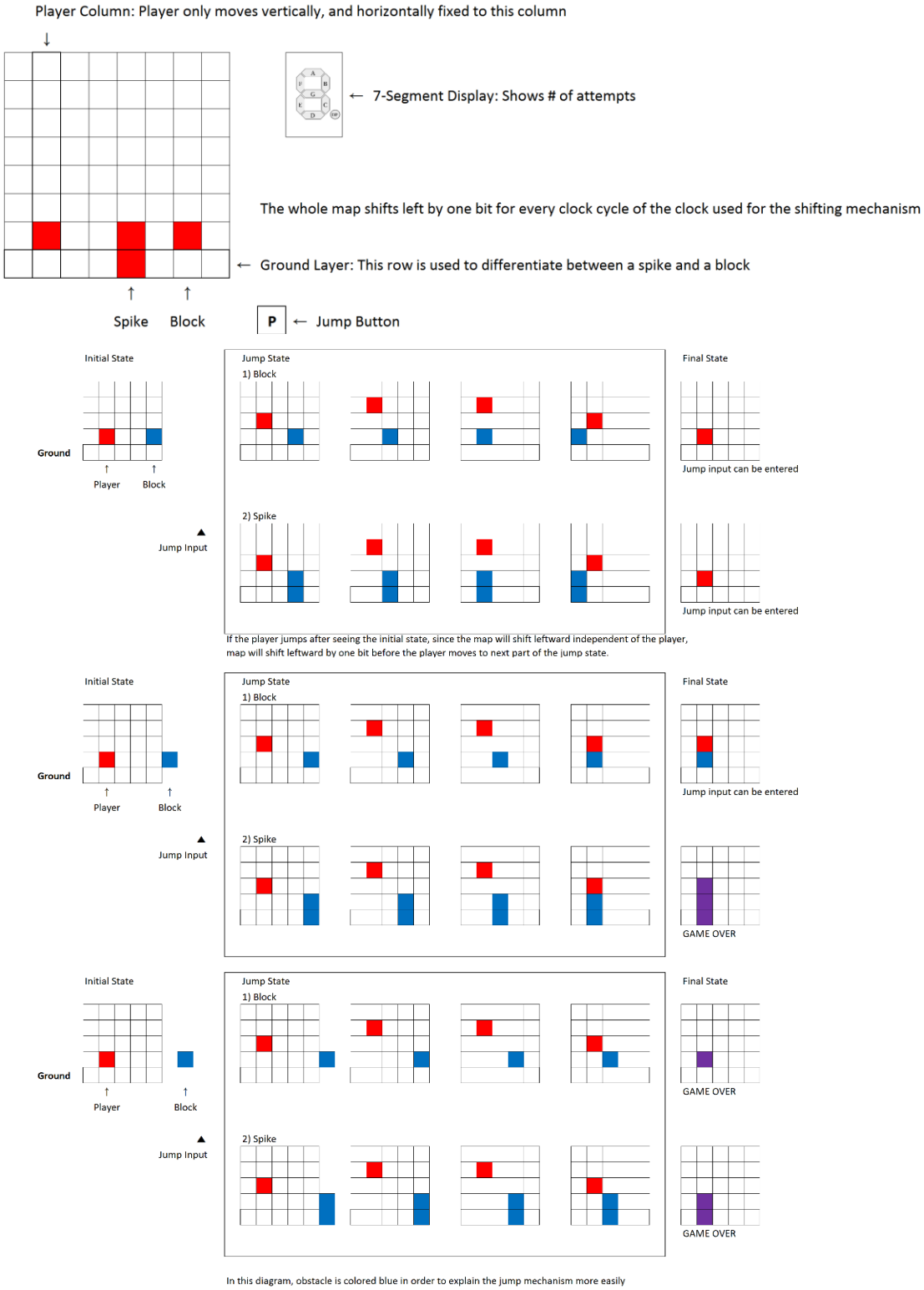


Figure 4. User Interface.

3. Encountered Problems

3.1 Push Button

Since the player was not allowed to jump when in the air, we had to configure the D_{SL} that goes into the Shift Register that configures S_0 and S_1 during the jump sequence. However, we encountered the problem where the jump input is activated right one state before the falling sequence was completed. Careful analysis and observation revealed that S_1 became 0, simultaneously as the player Shift Register was completing its jump sequence. Therefore, pressing on the push button consistently produced some kind of race condition, which was the cause of the problem. In order to solve this issue, we tried to use propagation delay, i.e. double NOT the output, putting it through Flip-Flop, but it was no use because the activation of push button was completely dependent on the S_1 turning 0. After constant failure, we deviated from propagation delay, and instead focused on slowing the activation of push button. And this solved the issue. By including the last, Q_3, of the Shift Register in the NOR gate, there was one clock cycle delay after S_1 turned 0 to input the next jump input. This was very critical problem, so we were very satisfied with the solution, and we also felt the power of thinking outside of the box.

3.2 LED Display

As neither group member was experienced at soldering, the creation of the LED display took much longer than expected. Our first prototype for our display failed as, through the design of the perfboard, we mistakenly soldered the cathode and anode of each LED together. Our second prototype for our display had our cathodes and anodes dangerously close, so the display would not light up properly, and electrical tape was not fine enough to fix the problem properly. Also, many of the LEDs were fried due to poor soldering technique. In addition, we did not educate ourselves on the advantages of using ribbon cable and soldered 64 individual wires to our LEDs, making properly placing each wire into the board difficult, as well as making debugging for issues nightmarish. We finally carefully toiled on our 3rd prototype to solve all of

the issues the previous prototypes had, and after many long hours obtained a functioning and relatively neat LED display.

3.3 Defective Breadboard

While master reset of Shift Register should turn all the data outputs of the Shift Register to 0, one of the Shift Register instead held a value of 0111, which was apparently abnormal. At first, we thought the issue was within the chip, so we tried to use new chips, and interchanged it with a functioning chip that was placed at a different position. Interestingly, the functional chip showed the same issue at the specific location, and the seemingly malfunctioning chip worked perfectly fine when moved to a different location. Initially we considered changing the breadboard, but luckily moving the chip by one bit on the breadboard solved the issue.

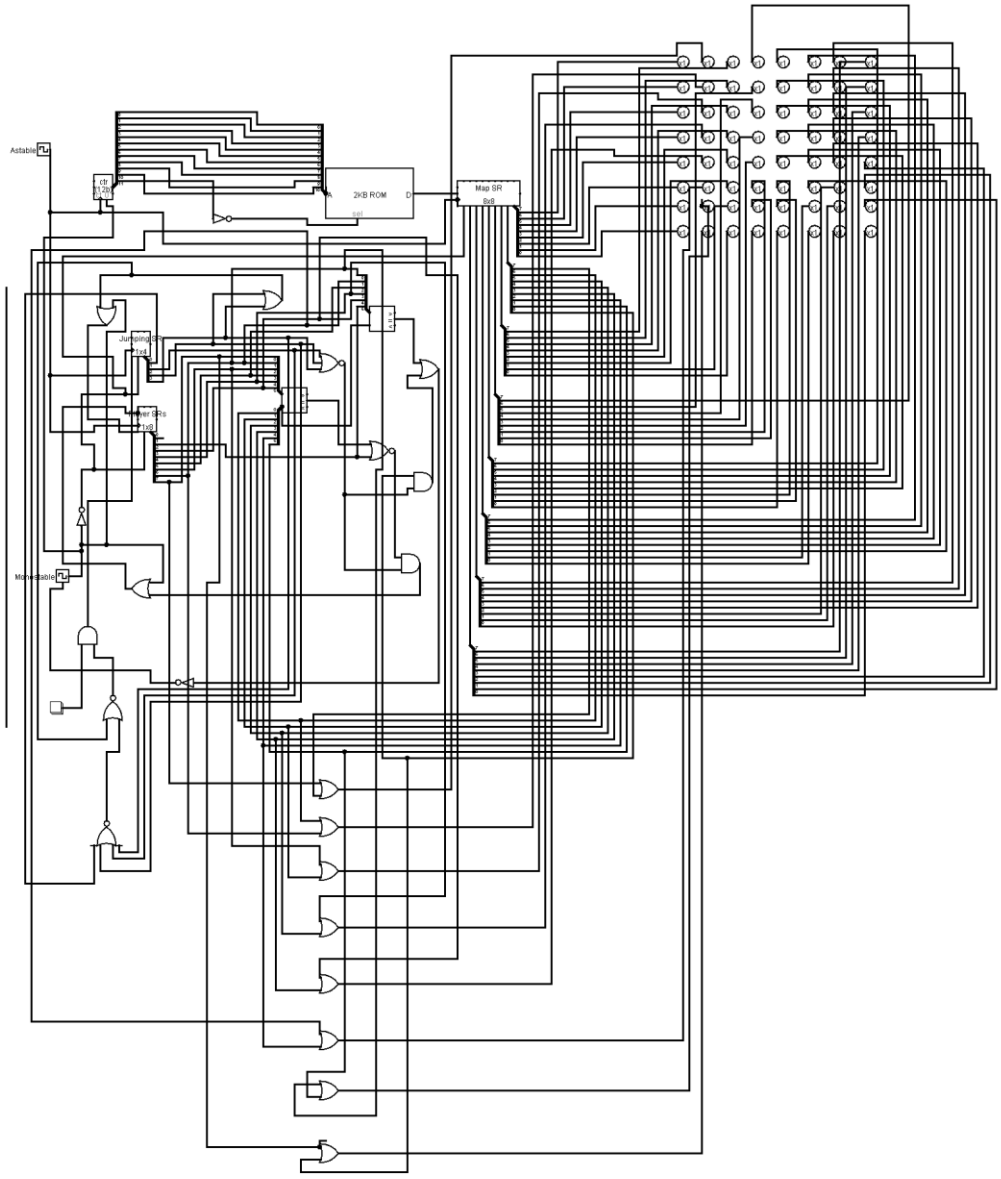
3.4 Minor Issues

Because it was a very huge project, there were numerous humane mistakes which caused the problems. For instance, by connecting a LED without resistor attached to it to an output of a chip, the current was dissipated to ground through the LED, thus causing a problem in the whole circuit. Without realizing this issue, it took us a while to figure out actually using a bare LED as a debugging tool was the problem.

Apart from this, there were some difficulties setting up the EEPROM, because it was the first time using it, and we experienced a power problem, due to numerous number of LEDs.

4. Supplementary Materials

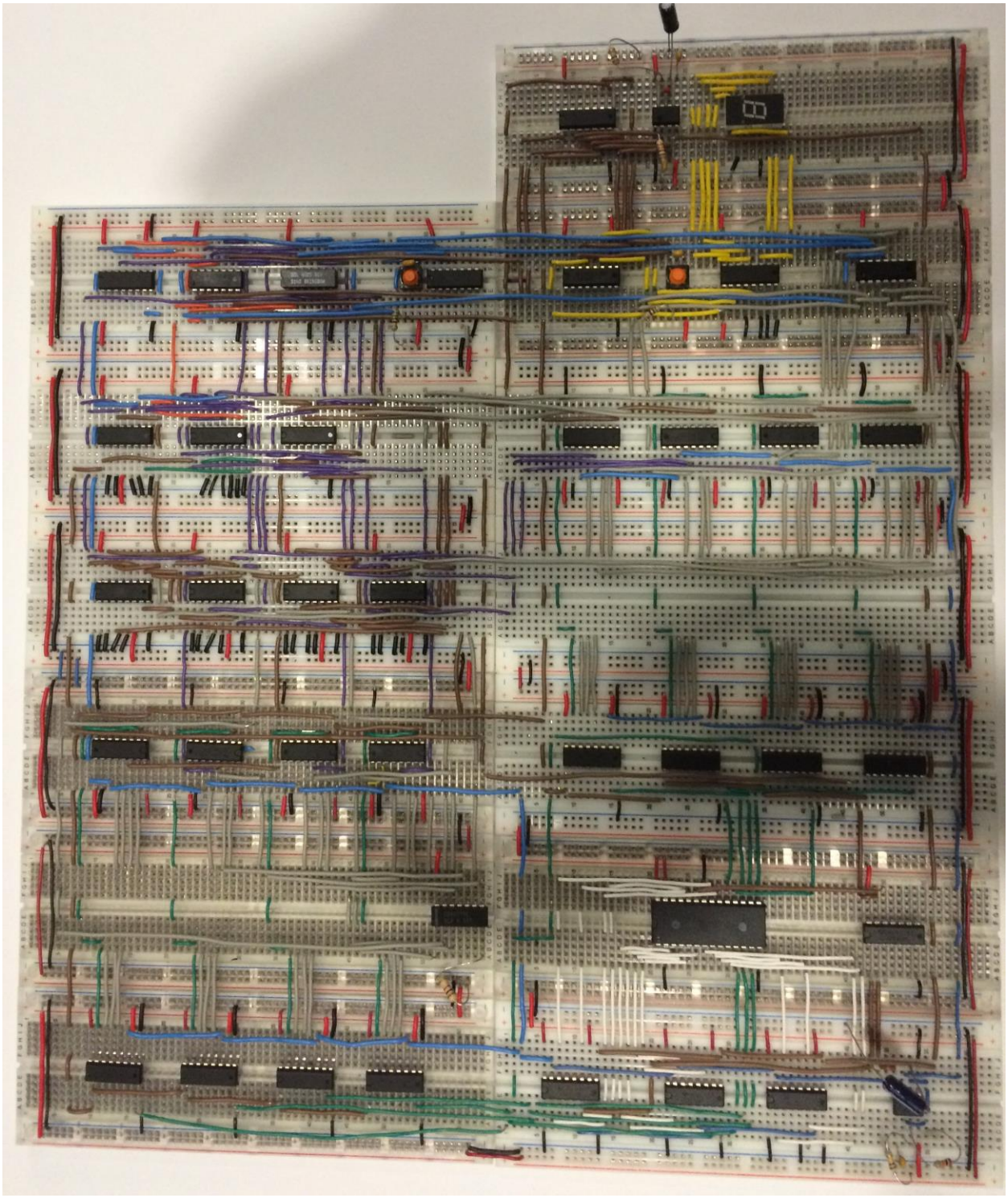
4.1 Logic Diagram



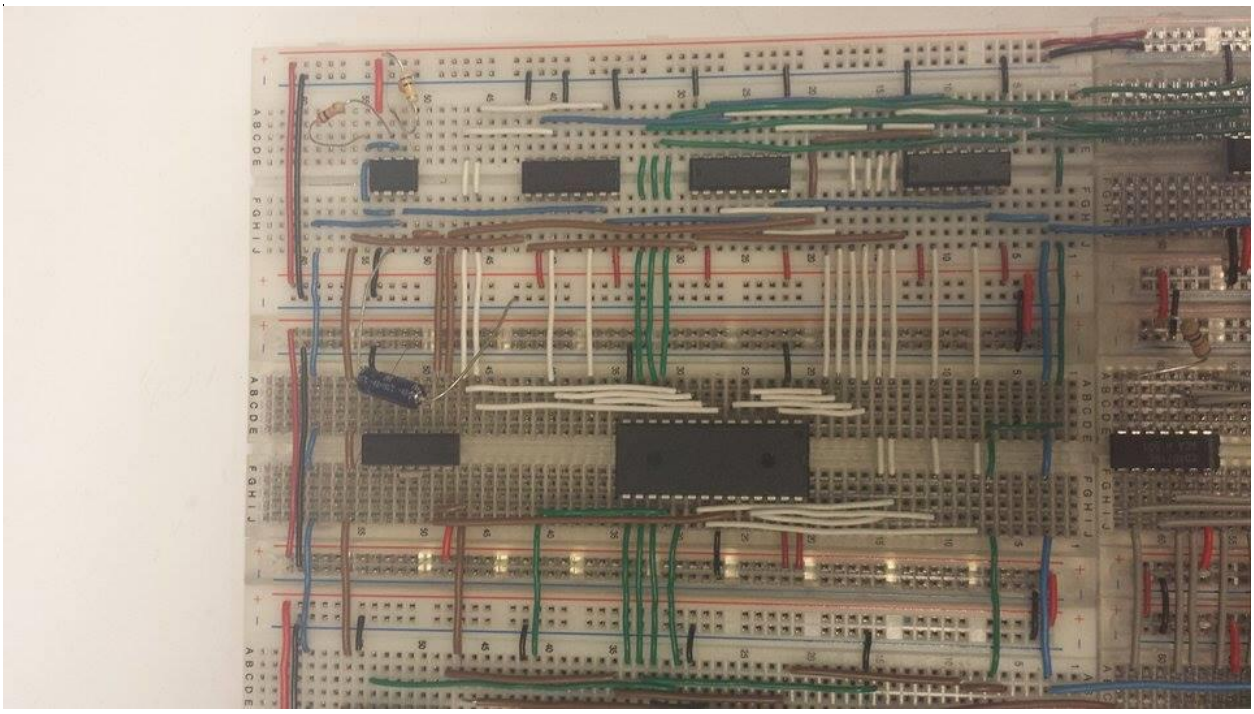
Our Logic Diagram tested on Logisim is brief and compact due to the fact that we used the Logic Diagram to purely test and easily see how our logic would work. This was not intended upon demonstrating each and every gate and chip we put on the board, as the schematic was best used for. Therefore, we have only one shift register for our map shift register as logisim gives us the ability to use an 8 stage 8 bit Shift Register, which we did with 16 4 bit Shift Registers in our project. Optimizations like this make our Logic Diagram simple, but it is still an accurate representation of our project.

4.2 Board Diagram

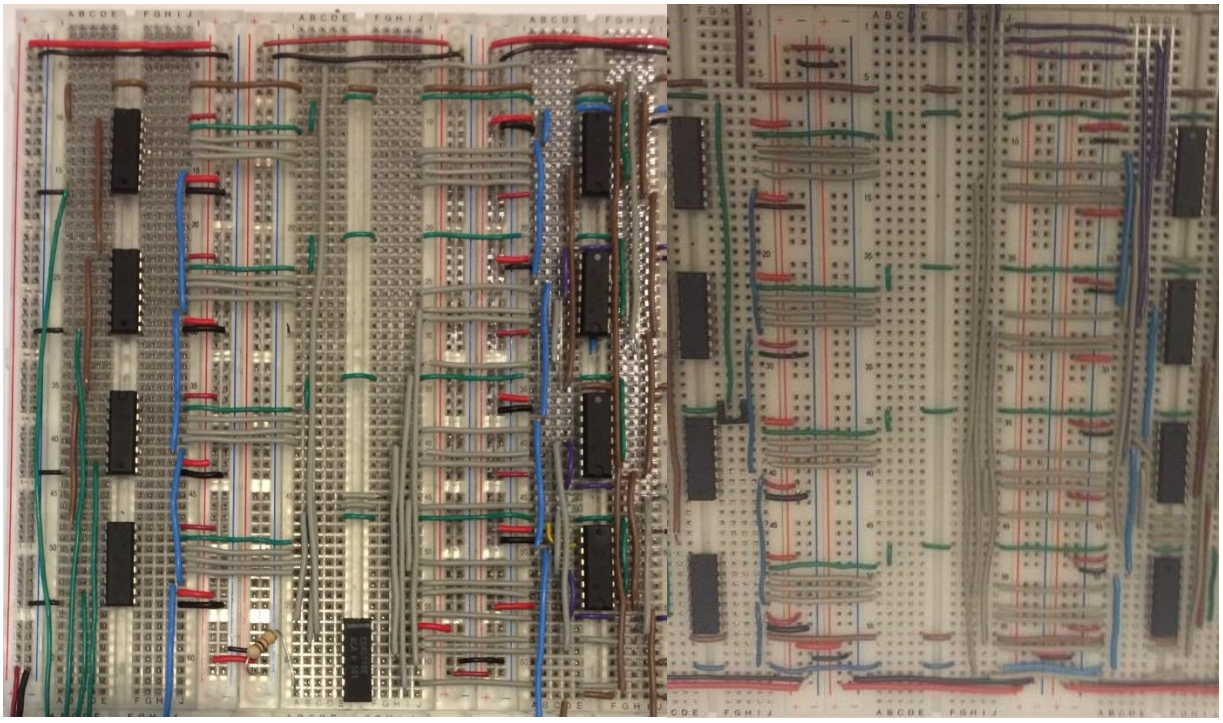
4.2.1 Full Board



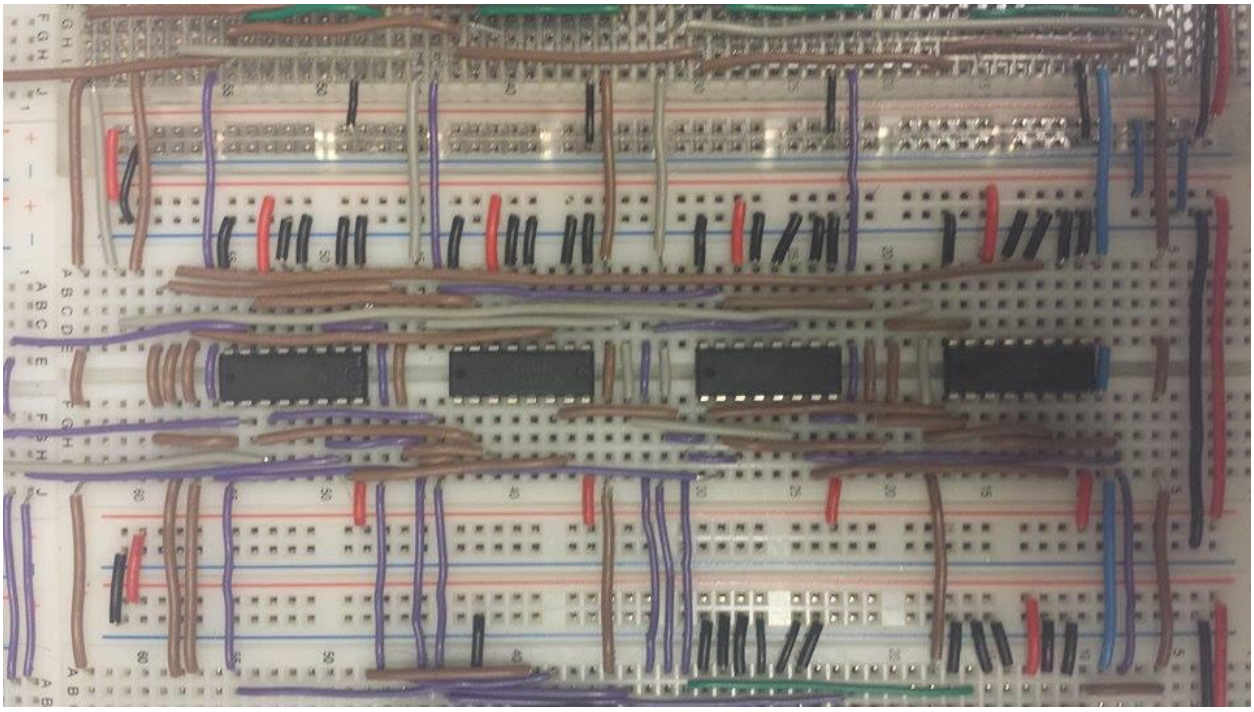
4.2.2 EEPROM



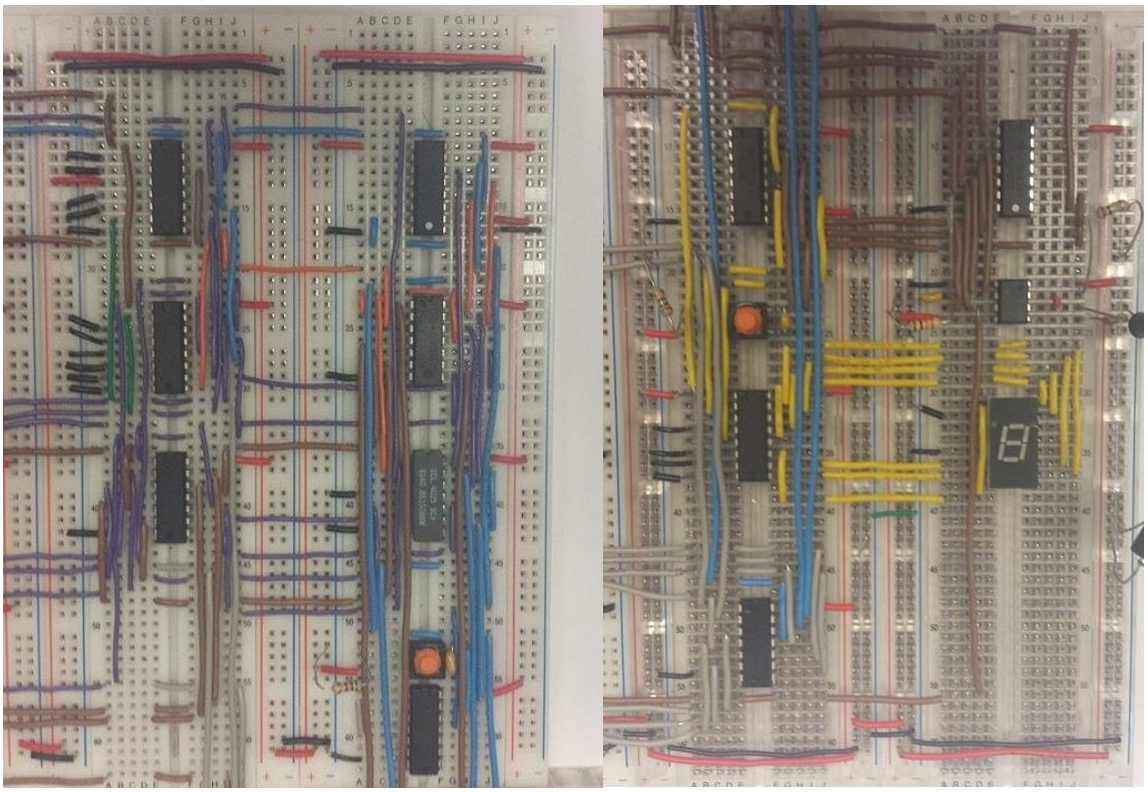
4.2.3 Map Shift Registers



4.2.4 Comparators for Collision and Falling Mechanisms



4.2.5 Player Shift Registers, Shift Register and Push Button for Jumping Mechanism. (Left) Collision Counter, Monostable 555 Timer for Collision, Reset Button. (Right)



4.3 Schematics Diagram