# Thunderbird Taillights: Lab 7 Report

Riley Ruckman (Partner: Chida Rijal)

TCES230A, Autumn 2019

December 5, 2019

## Introduction

The taillights of a Ford Thunderbird work as follows: each side is split into three different lights with a total of 6. When the left-turn blinker is turned on, the 3 lights on the left side of the car turn on in a specific order: the inner-most light turns on first, then the middle light turns on, and lastly, the farthest light turns on, with all three lights on at the end of the sequence. This sequence works the same for the right taillights when the right-turn blinker is turned on. When the hazard is turned on, all lights on both sides blink on and off in sync, and the hazard will override either left-turn or right-turn blinker if they're turned on at the same time as the hazard. Whenever the taillights switch to another behavior, like from turning left to turning right, the new behavior always starts from the idle state (when all lights are turned off).

In this scenario, like an actual physical car, we'll assume the left-turn and right-turn blinker cannot be on at the same time. A picture of a Ford Thunderbird's taillights is shown in Figure 1.


**Figure 1***: A picture of the taillights of a Ford Thunderbird. In the picture, the car is in the middle of the left-turn sequence.*

With these requirements, a Moore machine is to be designed to model the behavior of the lights. To start off the design, tables and expressions are needed.

## Circuit Design

The behavior of the car lights needs to be converted to a set of inputs and outputs that can be used in a state table. Since there are 6 different lights (3 lights for each side), they can be labeled as *L3*, *L2*, *L1*, *R1*, *R2*, and *R3*, going left to right. The behaviors of the lights are controlled by the inputs of a hazard, left-turn blinker, and a right-turn blinker, which are labeled as *H*, *L*, and *R* respectively.

Since this design is a state machine, a specific amount of memory is required to allow every possible combinations of lights. 8 cases can be determined from the behavior of the lights: 1 case when no lights are on (the *idle* state), 1 case when all lights are on (the hazard is turned on), and 3 cases for each side.

Each side requires three different cases because: 1 case when the inner-most light is on, 1 case when the inner-most and middle lights are on, and 1 case when the all lights on one side are on.

Because there are 8 different cases, 3 memory units need to be used to allow 8 different states, which will be labeled as *x*, *y*, and *z*. The memory used in this circuit will be D Flip-Flops, which will be referred to as DFF's from now on. With the amount of memory determined, a truth table can be made to map each state of the memory to a specific output of the lights. Table 1 showcases a truth table that will be used to model the rest of the design. The specific structure of the truth table is an arbitrary choice: if the relationship between each state and the outputs are consistent throughout the whole circuit design, the choice of one state mapping to a specific output is up to the circuit designer.

| *States* | *x* | *y* | *z* | *L3* | *L2* | *L1* | *R1* | *R2* | *R3* |
|---|---|---|---|---|---|---|---|---|---|
| *Idle* | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| *Turning Left* | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| *Turning Right* | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| *Hazards* | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**Table 1***: Truth table showing each state and the resulting output. Groups of states have been formed for specific behaviors*

A good way to visualize the state transitions of this machine is to use a state diagram. A state diagram involves visually drawing the connection from each state to another, with each connection requiring a specific combination of inputs to transition. Designing an arbitrary truth table before the state diagram proves to be useful for this: the truth table shows all connections for each state. For example, the left-turn sequence should follow this sequence of states based off the truth table: 000, 001, 010, 011. Therefore, most of the work for the state diagram is outlined in the truth table. The state diagram for this circuit is shown in Figure 2.

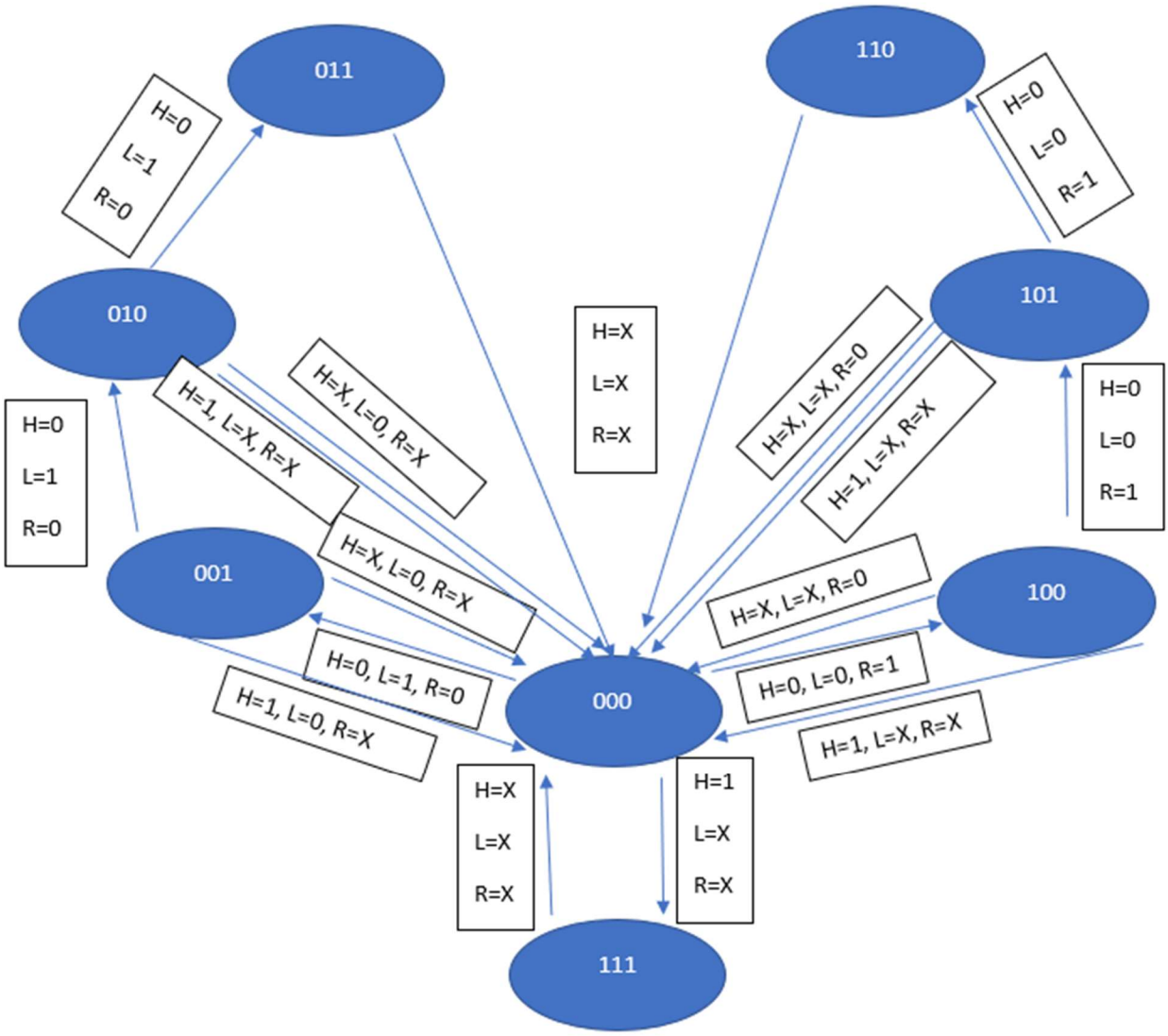



**Figure 2**: *State diagram made to visualize state changes due to specific combinations of inputs.*

Looking at the state diagram, there are a couple scenarios where one or more of the inputs will be labeled with a "X", which is a *Do Not Care* case. This is due to the requirement that any of the states need to be able to return to the *idle* state 000 any time. This requirement causes simplifications to pop up. For example, states 001 and 110 both return to the *idle* state in any situation because 1) they're the last state in their own respective sequences and 2) they must be able to return to the *idle* state when any of the inputs change. Therefore, it doesn't matter what the inputs are when the machine is in either state 001 or 110, since they'll always return to *idle*.

Table 1 can also be used to determine the logic expressions for each output. These expressions are shown in Figure 3 and are found using Sum of Products (SOP) expressions.

| *Output Expressions* |
|---|
| $L3 = yz$ |
| $L2 = \bar{x}y + yz$ |
| $L1 = \bar{x}y + yz + \bar{x}z$ |
| $R1 = x$ |
| $R2 = xy + xz$ |
| $R3 = xy$ |

**Figure 3**: *Expressions for all outputs.*

With Figure 2 as a reference, a state table can be created to determine the sequence of states for each combination of inputs. A state table would consist of three different sections: the current state of the machine, the next state of the machine based off the different combinations of inputs into the state machine and the current state, and the output of the machine for any of the current states. Since there are 3 inputs, a total of 8 different sequences need to be determined. The state table is displayed in Table 2.

Notice how the cases where *L* and *R* are both 1 are *Do Not Care* cases and are marked as X's. The left-turn and right-turn blinkers cannot be on at the same time, so the scenarios where they're both on don't matter in the designing of this circuit. This holds true in a real car, where only one blinker can be turned on at a time. Most of the space in the state table is the state 000 because whenever the machine changes to another sequence/behavior, the machine should always return to the *idle* state.

| *Current State (k)* | | | *Next State (k+1)* | | | | | | | | | | | | | | | | | | | | | | *Output* | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | *000 (HLR)* | | | *001* | | | *010* | | | *011* | | | *100* | | | *101* | | | *110* | | | *111* | | | | | | | |
| *x* | *y* | *z* | *x* | *y* | *z* | *x* | *y* | *z* | *x* | *y* | *z* | *x* | *y* | *z* | *x* | *y* | *z* | *x* | *y* | *z* | *x* | *y* | *z* | *x* | *y* | *z* | *L3* | *L2* | *L1* | *R1* | *R2* | *R3* |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | X | X | X | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | X | X | X | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | X | X | X | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | X | X | X | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | X | X | X | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | X | X | X | 0 | 1 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | X | X | X | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | X | X | X | 1 | 1 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | X | X | X | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | X | X | X | 0 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | X | X | X | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | X | X | X | 0 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | X | X | X | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | X | X | X | 0 | 0 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | X | X | X | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | X | X | X | 1 | 1 | 1 | 1 | 1 | 1 |

**Table 2***: State table based off the state diagram from Figure 2.*

With this current state table, handwriting any of the expressions for any of the next states would take a long time. Because of this, the website 32x8.com was recommended. 32x8.com is an online logic circuit simplification tool which takes a user-defined truth table and creates a Karnaugh map, a circuit from the truth table, and most importantly, a simplified expression for the table. For this specific circuit, 6 variables control the next state for each DFF: *H*, *L*, *R*, $x_k$, $y_k$, and $z_k$, so the 6-variable option was chosen. On the website, the truth table has *A*-*F* as the variable names, so the 6 variables for each next state had to be mapped to the 6 variables on 32x8. This mapping was chosen: *A* = *H*, *B* = *L*, *C* = *R*, D = $x_k$, *E* = $y_k$, and *F* = $z_k$. The mapping simplifies entering all the values into 32x8 because the significance of *H*, *L*, and *R* allows us to read down each column one by one instead of skipping across to different columns. The process went like this: for example, if solving for $x_k$, we'd first find the column in the next state part of the state table

where $H = 0$, $L = 0$, and $R = 0$. Starting from the top of the column, we'd enter the value of $x_{k+1}$ until we reach the bottom, which fills in the rows 0-7 on 32x8. The next column would then be identified as where $H = 0$, $L = 0$, and $R = 1$, and the values of $x_{k+1}$ would be entered, filling in rows 8-15 on 32x8.

Continuing this process until the whole truth table is filled results in a simplified SOP expression for the next state of one of the DFF's. To find the simplified SOP expressions for the next state of the other DFF's, this process needs to be repeated for each DFF. The expression given by 32x8 is in terms of *A*-*F*, so using the mapping created earlier, each expression was put into terms of the circuit's variables *H*, *L*, *R*, $x_k$, $y_k$, and $z_k$. The resulting expressions for each DFF is shown in Figure 4.

| *Next State Expressions* |
|---|
| $x_{k+1} = \overline{H}R\overline{y_k}\,\overline{z_k} + \overline{H}Rx_k\overline{y_k} + H\overline{x_k}\,\overline{y_k}\,\overline{z_k}$ |
| $y_{k+1} = H\overline{x_k}\,\overline{y_k}\,\overline{z_k} + \overline{H}Rx_k\overline{y_k}z_k + \overline{H}L\overline{x_k}\,\overline{y_k}z_k + \overline{H}L\,\overline{x_k}y_k\overline{z_k}$ |
| $z_{k+1} = \overline{H}L\,\overline{x_k}\,\overline{z_k} + H\overline{x_k}\,\overline{y_k}\,\overline{z_k} + \overline{H}Rx_k\overline{y_k}\,\overline{z_k}$ |

**Figure 4**: *Next state equations for the DFF's.*

Now that expressions have been created for the next state of each DFF and each output, a prototype was simulated to verify a working circuit that matches the behaviors of the Thunderbird taillights.

## Simulation

Two different simulation applications were done for this circuit: Logisim and ModelSim. Logisim is used to visually create circuits using drag-and-drop objects like logic gates, inputs, clocks, and memory units. Logisim has an option to declare variables and define each variable through an expression. Using this tool, the next-state logic and output logic were automatically created by Logisim by typing in the expressions in Figure 3 and Figure 4. A screenshot of the complete circuit can be found in Appendix: Logisim Simulation.

Testing each input combination of the circuit resulted in the expected behavior. Specifically, keeping *L* or *R* on for at least 4 clock cycles resulted in a complete execution of the left-turn sequence and right-turn sequence respectively. When *H* was turned on at any time, the circuit would reset to the *idle* state then proceed to flash all the lights on and off. Turning on *H* in the middle of the left-turn sequence or the right-turn sequence made the circuit immediately return to the *idle* state then start the *hazard* sequence. When no inputs are on, the circuit remains in the *idle* state, and when all inputs are turned off in the middle of a sequence, the circuit immediately returns to the *idle* state as well. All outcomes are what is expected, so the current design for the circuit is a working prototype.

A clock signal was sent to each DFF in the simulation to test real-time changes to each input and output instead of manually turning off and on an input to simulate a clock. The Logisim simulation also helps outlines a structure for a physical build of the circuit by visually showing connections from one part of another and displaying the number and kinds of gates required to build the circuit. This simulation isn't the simplest way to create the circuit but works as a functional prototype.

The ModelSim simulation uses Verilog code to program the circuit into a file which can be ran and analyzed using built-in tools, such as a timing diagram. A supplied file for the DFF's was given by the instructor, so not much coding was needed to create the complete circuit in ModelSim. The code used

and a couple timing diagrams are shown in Appendix: Verilog Code & Timing Diagrams. The typical troubles with coding, like misspelling expressions or forgetting a function, occurred and halted the simulation until they were fixed. Overall, the ModelSim simulation again verified that the current design behaves like the Ford Thunderbird's taillights.

## Experimental Results

My lab partner and I decided to build a simplified version of the circuit shown in Appendix: Logisim Simulation. A picture of the physical circuit is provided in Appendix: Physical Circuit.

## Conclusions

The process of designing and testing a circuit from a simple word problem is satisfying and has helped me understand the process and function of a state machine. The difference between a Moore and Mealy machine was also made very apparent in this lab: the inputs *H*, *L*, and *R* were never directly wired to any of the output logic, therefore the state machine created in this lab was a Moore machine.

Starting from a truth table and state diagram to visually display the desired behavior of the state machine was very helpful for the process of creating a state table and have proven to be vital steps in designing any kind of state machine. There is interest in a Mealy design that could reduce the complexity of the entire circuit while keeping the desired behavior, but that is out of the scope of the lab.

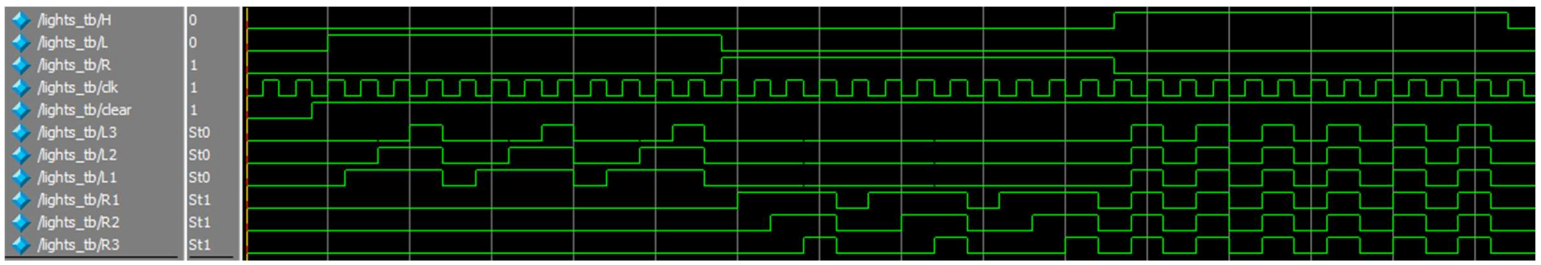
This lab has cemented the ideas behind logical circuit design, from word problems to state tables to expressions and finally to a working prototype. Every step of the design process stacks on top of one another and missing one step can cause a cascade of problems.

## Appendix: Verilog Code & Timing Diagrams

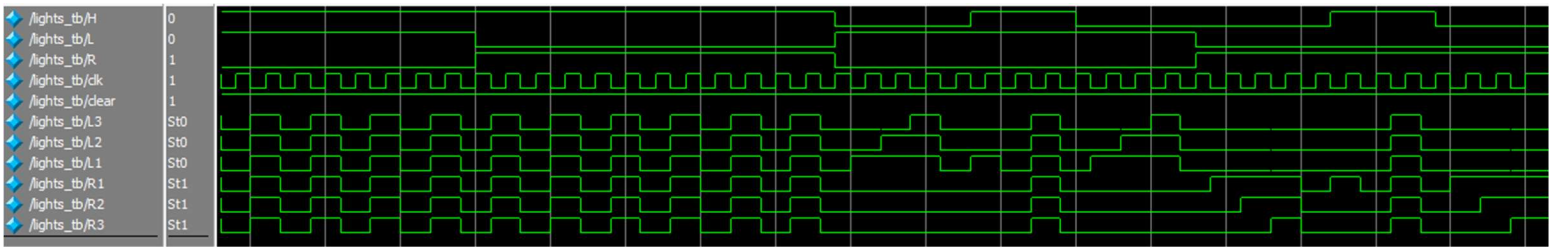
The Verilog code used in ModelSim. The Testbench file was not included due to its length. (Imagine a lot of *clk = 0; clk = 1;* statements.)

```
// Lab 7

module lights(H,L,R,clk,clear,L3,L2,L1,R1,R2,R3);

// Inputs and Outputs
input H,L,R,clk,clear;
output L3,L2,L1,R1,R2,R3;

// Internal Wires;
wire xn,yn,zn;
wire x,y,z;
wire x_bar,y_bar,z_bar;

////////////////////////////////////////////
// Next-State Logic

assign xn = (H&~x&~y&~z)|(~H&R&~y&~z)|(~H&R&x&~y);
assign yn = (H&~x&~y&~z)|(~H&R&x&~y&z)|(~H&L&~x&~y&z)|(~H&L&~x&y&~z);
assign zn = (H&~x&~y&~z)|(~H&R&x&~y&~z)|(~H&L&~x&~z);

////////////////////////////////////////////
// Flip-Flops

dff dff_x(xn,clk,clear,x,x_bar);
dff dff_y(yn,clk,clear,y,y_bar);
dff dff_z(zn,clk,clear,z,z_bar);

////////////////////////////////////////////
// Output Logic

assign L3 = (y&z);
assign L2 = (~x&y)|(y&z);
assign L1 = (~x&y)|(y&z)|(~x&z);
assign R1 = x;
assign R2 = (x&y)|(x&z);
assign R3 = (x&y);

endmodule
```

The timing diagram created with ModelSim. As seen, every sequence for the left-turn blinker, right-turn blinker, and the hazards works as expected.

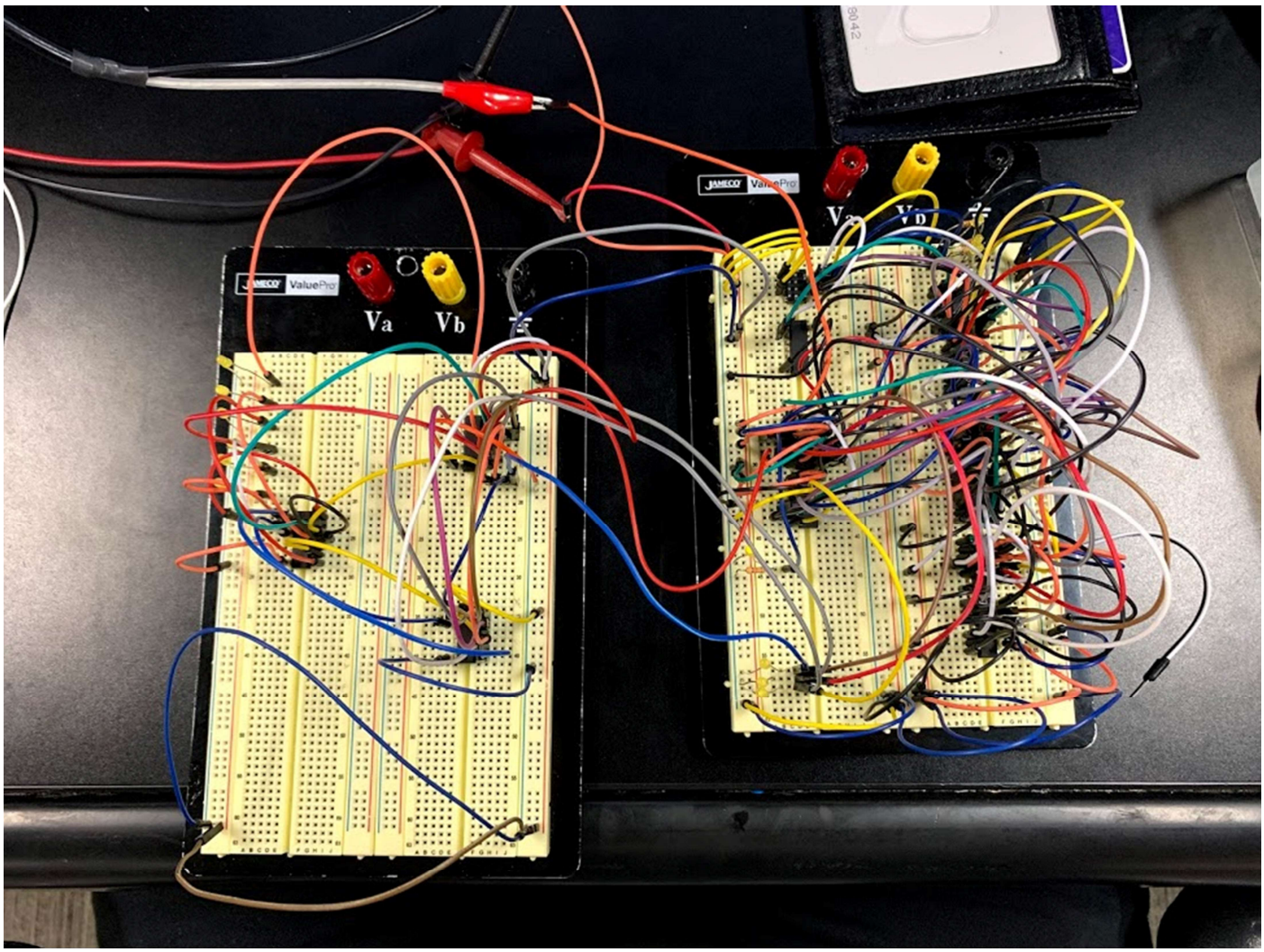


Another timing diagram showcases the return to the *idle* state whenever the machine changes sequences. For example, when *H* = 1 when *L* = 1, the left-turn sequence immediately returns to the *idle* state first and then proceeds to perform the *hazards* sequence.




## Appendix: Physical Circuit

A physical build of a simplified version of the Logisim diagram. The right breadboard holds the inputs, next state logic, and the DFF's. The left breadboard holds the output logic and lights.



## Appendix: Logisim Simulation

A screenshot of the circuit built in Logisim. The left side of the circuit is the next state logic, and the right side is the output logic.