EECE 144
Fall 2011

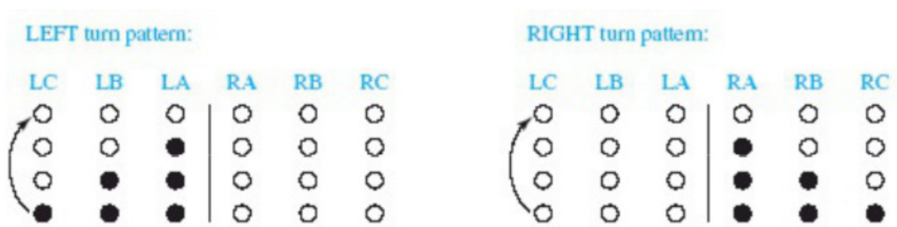
Lab Report #13
Section 4
12/07/2011

Submitted by: Jeremiah Mahler

Signature	Printed Name	Date
	Jeremiah Mahler	**Dec 7, 2011**
	Marvanee Johnson	**Dec 7, 2011**

1 Description/Objectives

The goal of this lab is to design a circuit in Verilog[1] which replicates the tail light operation of an older model Thunderbird.



> Design a Moore sequential circuit to control these lights. The circuit has three inputs LEFT, RIGHT and HAZ. LEFT and RIGHT come from the driver's turn signal switch and cannot be 1 at the same time. As indicated above, when LEFT = 1 the lights flash in a pattern LA on; LA and LB on; LA, LB, and LC on; all off; and then the sequence repeats. When RIGHT = 1, the light sequence is similar. If a switch from LEFT to RIGHT (or vice versa) occurs in the middle of a flashing sequence, the circuit should immediately go to the IDLE (lights off) state and then, start the new sequence. HAZ comes from the hazard switch, and when HAZ = 1, all six lights flash on and off in unison. HAZ takes precedence if LEFT or RIGHT is also on. Assume that a clock signal is available with a frequency equal to the desired flashing rate.
> [2, p. 547, prob. 16.27].

2 Procedure

The method used here builds a state diagram and state table for one side and then this is used as a guide to duplicate for the opposite side. Then it is programmed in Verilog using behavioral modeling. With behavioral modeling statements such as `if` can be used which make it similar to a regular programming language such as C.

One limitation of using behavioral modeling it is not easily translated to gates and flip flops. An alternative would be to design this using data flow modeling. It would be more difficult to design but it could be translated to gates and flip flops more easily.

From Figure 1 and Table 1, which describe the operation of the left turn signal, it can be seen in general how the system will operate. The operation of the right turn signal is similar but with the states/values of right and left reversed.

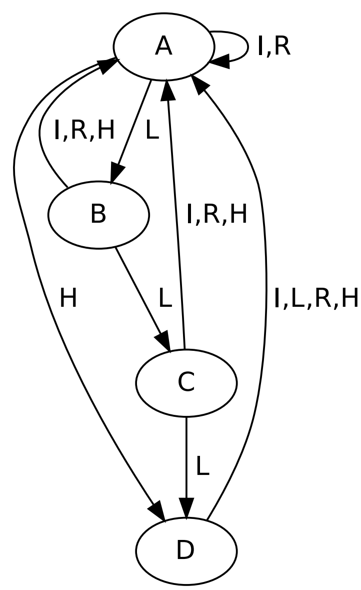


Figure 1: State diagram for left turn signal operation. State A has zero lights on (000), state B has one (001), state C has two (011) and state D has all three (111). Each transition is denoted with I as idle, R as right, L as left, and H as hazard.

The source code for the Verilog definition of the tail lights is given in Listing 1. The module is positive edge triggered similar to a flip-flop as shown on Line 13, The outer logic consists of a chain of `if`, `else if`, statements which decide among the possible X input values (H, L, R, and the catch all state I) as shown on Lines 14, 24, 40 and 56. For each

S	S^+				
	$X = I$	L	R	H	Z
A	A	B	A	D	000
B	A	C	A	A	001
C	A	D	A	A	011
D	A	A	A	A	111

Table 1: State table for LEFT turn signal. The states of X are all mutually exclusive.

.

value the current state of the tail lights (TL, TR) is examined and a decision is made as to what state to assign next.

```verilog
/*
 * Replicate the operation of tail lights on old Thunderbirds.
 *
 * L:   left signal
 * R:   right signal
 * H:   hazard signal
 *
 * TL:  left tail light
 * TR:  right tail light
 *
 */
module taillights (input clk, L, R, H, output reg [2:0] TL, TR);
        always @(posedge clk) begin
                if (1 == H) begin
                        if (TL == 3'b000 && TR == 3'b000) begin
                                TL <= 3'b111;
                                TR <= 3'b111;
                        end
                        else begin
                                TL <= 3'b000;
                                TR <= 3'b000;
                        end
                end
                else if (1 == L) begin
                        TR <= 3'b000;

                        if (TL == 3'b000) begin
                                TL <= 3'b001;
                        end
                        else if (TL == 3'b001) begin
                                TL <= 3'b011;
                        end
                        else if (TL == 3'b011) begin
                                TL <= 3'b111;
                        end
                        else begin
                                TL <= 3'b000;
```

```
            end
        end
        else if (1 == R) begin
            TL <= 3'b000;

            if (TR == 3'b000) begin
                TR <= 3'b001;
            end
            else if (TR == 3'b001) begin
                TR <= 3'b011;
            end
            else if (TR == 3'b011) begin
                TR <= 3'b111;
            end
            else begin
                TR <= 3'b000;
            end
        end
        else begin
            TL <= 3'b000;
            TR <= 3'b000;
        end
    end
endmodule
```

Listing 1: Verilog source for the tail lights.

2.1 Compiling Verilog source and running GTKWave

Once all the code has been defined it can be compiled and run. The test bench is given in AppendixA.

To compile the source code using Icarus Verilog[1] under Linux the following command can be run.

`iverilog test.v`

This will produce a filed named 'a.out'. Under Linux this can be executed directly.

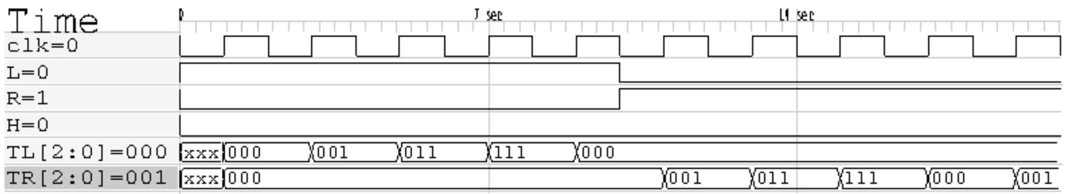
`./a.out`

Alternatively the `vvp` command can be used. This method works under Linux or Windows.

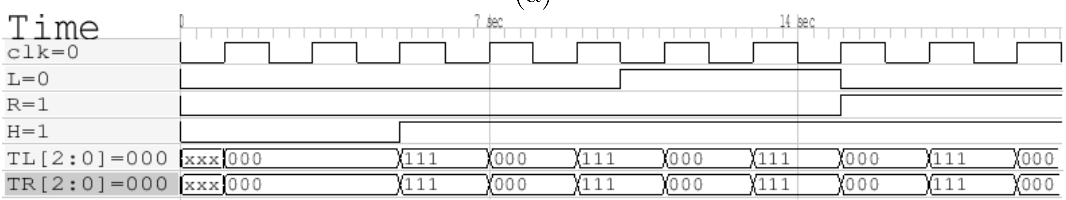
`vvp a.out`

Because the `$dumpfile` and `$dumpvars` have been added to the test bench (Appendix A) it will produce an output file suitable for GTKWave[3]. The file should have the extension '.vcd' and can be run as shown below.

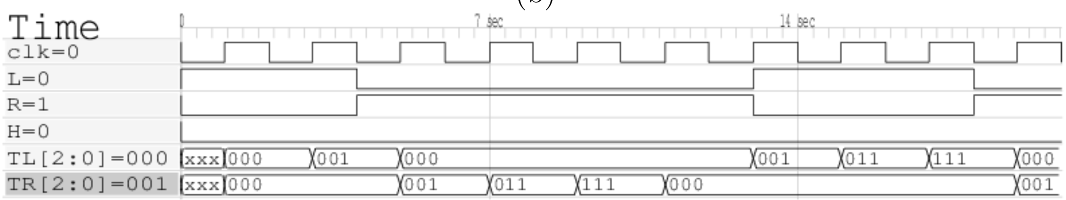
`gtkwave output.vcd`



(a)



(b)



(c)

Figure 2: Wave forms for of the tail lights in various situations. In (a) it goes from from left and then switches to right. In (b) it starts at idle, then the hazards are turned on, then left turn signal is engaged, and then the right turn signal is engaged. Notice that the hazard has priority and that the left and right have no effect. In (c) it is switched back and forth from left to right to show the transitions when switch midway through a cycle.

And from within GTKWave the different variables can be selected and displayed.

3 Observations

The output for various operations of the turn signal are given in Figure 2. It can be seen that when it is in either the left or right position it progresses through the proper tail light sequence. It also handles transient situations where it is changed from left to right mid way through a sequence. And the hazard operation alternates from all on to all off as it is unaffected by either left or right being actuated at the same time.

4 Conclusion

This lab was a complete success in implementing a design for old Thunderbird tail lights in Verilog. All design requirements were met and the tail lights behaved as expected. This design did not lend itself to an obvious translation in to hardware but this would be a worthwhile addition for a future lab.

5 References

[1] S. Williams, "Icarus verilog." `http://iverilog.icarus.com/`, 2011.

[2] C. Roth Jr., <u>Fundamentals of Logic Design</u>. Cengage Learning, 2009.

[3] T. Bybell, "Gtkwave." `http://gtkwave.sourceforge.net/`, 2011.

A Verilog Test Bench

```
/*
 * Test bench tail lights.
 *
 * To compile this file run:
 *
 *   iverilog test.v
 *
 * Run the executable:
 *
 *   ./a.out
 * OR
 *   vvp a.out
 *
 * Because $dumpfile and $dumpvars have been
 * added it will generate data for gtkwave
 * in gtkwave-output.vcd.
 *
 * This output file can then be shown with Gtkwave.
 *
 *   gtkwave gtkwave-output.vcd
 *
 * And from within Gtkwave you can pick and choose
 * variables to see their waveforms over time.
 *
 *
 * This project was completed as part of lab 13 in the
 * class EECE-144 taught by Kurtis Kredo II at Chico State
 * during the Fall of 2011.
 *
 * Author(s): Jeremiah Mahler <jmmahler@gmail.com>
 *                    Marvanee Johnson
 */

`include "taillights.v"

module test;
    reg clk, L, R, H;
    wire [2:0] TL, TR;

        taillights tl1(.clk(clk), .L(L), .R(R), .H(H), .TL(TL), .TR(TR));

    initial begin
        $dumpfile("gtkwave-output.vcd");
        $dumpvars(0, test);

        // initialize
            L = 0;
            R = 0;
            H = 0;
```

```
        clk = 0;

                // configure how to run the test

                /*
                // Left
                L = 1; R = 0;

                // Right
                #10 L = 0; R = 1;
                */

            /*
                // Hazard
                H = 0;
                #5 H = 1;
                // engaging the left or right should not change anything
                #5 L = 1;
                #5 L = 0; R=1;
                */

                // back and forth from left to right
                H = 0;
                L = 1; R = 0;
                #4 L = 0; R = 1;
                #9 L = 1; R = 0;
                #5 L = 0; R = 1;

                #2 $finish;
    end

    always begin
        #1 clk = ~clk;
    end
endmodule
```