Lab 7: Reaction Timer

Introduction

In this lab a sequential circuit will be designed to test reaction speed.

1 Prelab

Figure 1 shows the overall circuit that you will design and build. You will use a clock (`Clock`) to drive a two-digit counter; the input control to the mux (`w`) will be a pulse that lights the LED. The LED will turn off as soon as the switch (connected to the AND gate) is pressed, and the signal `Reset` can be used to reset the counters, the display, and get the circuit ready to start over.

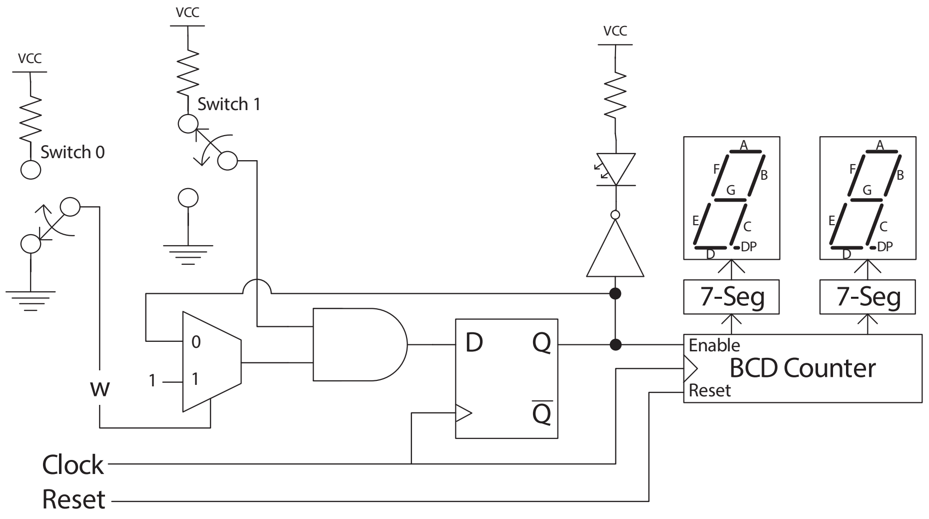

Figure 1: Circuit for Testing Your Reaction Speed

1.1. We want a minimum resolution of measuring your reaction time to be 1/100 of a second, but your board has a 8 MHz clock. **How can you use a counter to generate a clock with a frequency of 100 Hz?**

1.2. Now that you have a 100 Hz clock, you can use that to count the time it takes you to press the switch after the LED goes on and display that on two 7-segment displays. To accomplish this you will need to write a code that will increment the least significant digit and every time it reaches the number 9 you reset it and increment the most significant digit. Write Verilog code to implement the two digit BCD counter shown in Figure 1.

1.3. Write Verilog code to implement the entire circuit as shown in Figure 1. Use one of the LEDs on your board to test the code.

2 Lab

2.1. Simulate your Verilog program and make sure it works as planned.

2.2. Wire your double 7-segment display and test your counter code.

2.3. Wire two debounced switches. One will be used as the input to `w` and the other will be the one you depress once you see the LED light on.

2.4. Experiment with the resistor attached to the LED try both 1 kΩ and 390 Ω resistors. **Note at least two differences.**

2.5. **Ask the instructor or TA to test the circuit and record how fast you are.**

The game is played as follows:

- The switch connected to the Mux is depressed by the TIMER and released.
- The LED in illuminated, and the player presses the other switch as fast as possible.
- The LED ceases to be illuminated; the total time (in 1/100$^{\text{th}}$'s of a second) is displayed.

3 Supplement: Verilog 4

So far the statements you have been using that are encapsulated by always were blocking type, i.e., they are executed sequentially. For this lab you will need to use a non-blocking procedural assignment. Non-blocking assignments are executed concurrently. To differentiate between the two, let's look at the following two examples:

Listing 1: Blocking Assignments in Verilog

```
B = A;
C = B + 1;
```

and

Listing 2: Non-blocking Assignments in Verilog

```
B <= A;
C <= B + 1;
```

The first case (Listing 1) is the blocking one, where the statements are executed sequentially, therefore the first statement stores `A` into `B`, and then 1 is added to `B` and stored into `C`. At the end a value of $A+1$ is stored into `C`. On the other hand, the second case (Listing 2) is non-blocking. In this case the assignments are made using `<=` instead of a `=` sign. Non-blocking statements are executed concurrently, so values of the right hand side are stored in temporary locations until the entire block is finished and then they will be assigned to the variables on the left. For this case `C` will contain the original value of $B+1$ rather than $A+1$ as in the first case.

Use blocking assignments when you must have sequential execution. If you are modelling edge-sensitive behavior use non-blocking assignments. In synchronous sequential circuits, changes will occur based on transitions rather than levels. In this case we need to indicate whether the change should occur based on the rising edge or the falling edge of the signal. This behavior is modeled using the two keywords `posedge` and `negedge` to represent rising and falling edge, respectively, for example:

Listing 3: Edge Sensitive `always` block

```
always @(posedge clock or negedge reset)
```

This will start executing on the rising edge of clock or on the rising edge of reset. If your circuit has an asynchronous reset, you may need an `if-else` statement to indicate whether you are resetting or triggering the circuit, in this case the very last statement of the non-blocking assignment needs to be the one related to the clock. For a D flip-flop with asynchronous reset, example code is provided in Listing 4.

Listing 4: An example of a D flip-flop with asynchronous reset

```
module D_Flip_Flop(output reg Q, input D, clock, reset);
        always @ (posedge clock or negedge reset);
                if(~reset)
                        Q <= 1'b0;
                else
                        Q <= D;
endmodule
```