**Lab 7 Report**

Michael Dorsey, EE457 Tuesday – 11/20/2018

**The Game**

1. **Introduction**

The game is on. Reflexes are here to stay and creating warriors with reflexes capable of taking down any opponent is necessary for a successful army these days. The goal of this lab aims to test and hone the reflexes of an individual to bring up to par with the reflex champions of the world. This lab will include a stopwatch, seven segment controllers, a counter and a top-level design for our board to function properly. The stopwatch will count divisions of seconds and display the output of how quickly the stop button is pressed to determine how long it took for the simulation to end. Let the games begin!

**Theory of Operation**

**Requirements:**

1. Upon execution, the design will wait one second, and begin counting until user input occurs.
2. Once the stop button is hit, the board will display an output of how many seconds were awaited.
3. This initial number of seconds will be used to determine a pseudo-random number that will be generated.
4. The random number generated will be used to determine how long the board should wait before testing user reflexes.
5. Once the user is ready, if the start button is pressed again, the board will count up to this pseudo-random number.
6. Upon reaching this number, the board will then turn on a red light to show that the game will now start counting how quickly it takes for a user to press the stop button.
7. During the counting phase of the pseudo-random number, display will be turned off. Once the red light turns on, display will begin counting.
8. If the user stops the stopwatch after the red light has come on, then a display of how many division of seconds have occurred since the red light has come on. This display is how long it took for the stop button to be pressed.
9. If the user presses the stop button, then they will have lost one attempt.
10. If the user has a failed attempt, but reaches a successful attempt, then numbers of attempts will be reset.
11. If two continuous attempts occur, then the board will output “FAIL” showing that the user does not have what it takes to be champion of reflexes in this grueling day and age.

**Description of the design:**

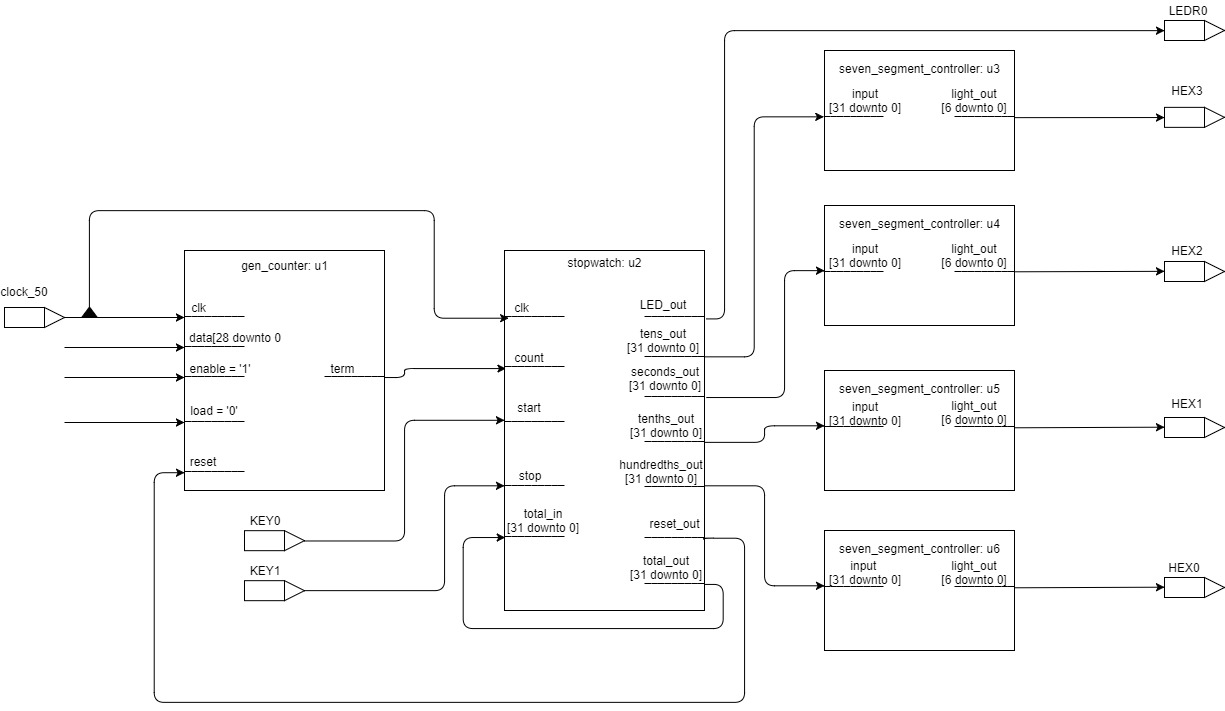
Overall, the design consists of 11 states. These states will consist of a set of states used to determine this *pseudo*-*random* number, and another set of states to test the reflexes of the user. The first set of states are described as such…

The *idle* state is a standby state, waiting for the user to start the game. A one second delay occurs before the red *LED* turns on and the counter begins generating our number used to determine the pseudo-random number. After completing one second, the board begins counting hundredths of a second, tenths of a second, seconds, and ten seconds. During this state of *running*, the board is not displaying any output, so the number that appears when stopped will be somewhat random to begin with, but after stopping, the board displays the number of divisions of seconds waited. After getting this random set of seconds, the board will then multiply each set of *seconds*, *hundredths*, *tenths*, *ones*, and *tens*, by values of *1000*, *100*, *10* and *1* respectively. The board will then select a set of four vector bits and this will become our *pseudo*-*random* number. This will end the first set of states, where our last state in this set is *stopped* where our red *LED* is off, which is similar to the *idle* state, waiting for the user to continue to the next set.

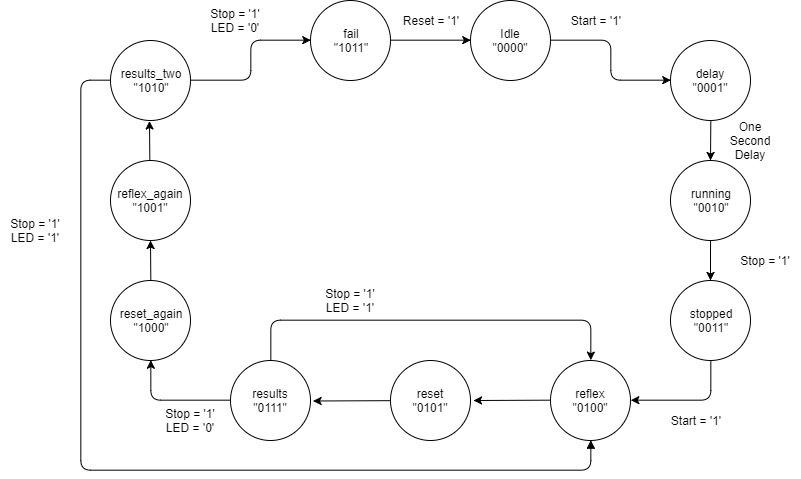
The second set of states begins after the user has started the actual game. We deem this state *reflex*, and it will begin with user our *pseudo*-*random* number to count the number of divisions of seconds to wait until our red *LED* turns on. Once the light turns on, the game is on. The user can wait a continuous number of seconds until pressing the stop button, but for the sake of our design, we can only count to *99:99*, before our clock rolls over into *00:00* and continues counting. The goal of the game is to get the lowest number of split seconds, which we can consider the lowest “score”. If the stop button is pressed after our red *LED* turns on, then the board will display the results, in the *result* state, to show how quickly the button was pressed. The number of attempts at this state are infinitesimal, so long as the user is pressing the stop button after the *LED* has turned on. Pressing the start button will begin the in the second set of states again, and the previous seconds awaited are used to generate another *pseudo-random* number, so our reflex test is as random as it can be. On the other hand, if the red *LED* does not turn on, and input is received to stop, then the user will have one strike against them. Upon completing a successful round, then the number of attempts the user gets will be reset, but two continuous strikes against the user, and the *fail* state is reached, where the user has failed to successfully test their reflexes.

This can be demonstrated by the normal *GYR* sequence. Consider the states *weight\_check\_ns, green\_a, yellow\_a* and *weight\_check\_EW.* During *weight\_check\_ns* both sets of lights are red. Let’s assume there is no weight, so we proceed to *green\_a*. This means that North-South will be given green, while East-West will remain red. The same occurs during *yellow\_a*, but North-South will become yellow. Once *weight\_check\_EW* occurs, we make all lights red, to give a buffer to drivers who have a tendency to run close to the edge of red lights when the light transitions from yellow to red. In order to maintain safety, there’s a brief overlap of red lights in all four directions to ensure drivers have come to a complete stop at the light.

**Block Diagram**

******

**State Diagram**

******

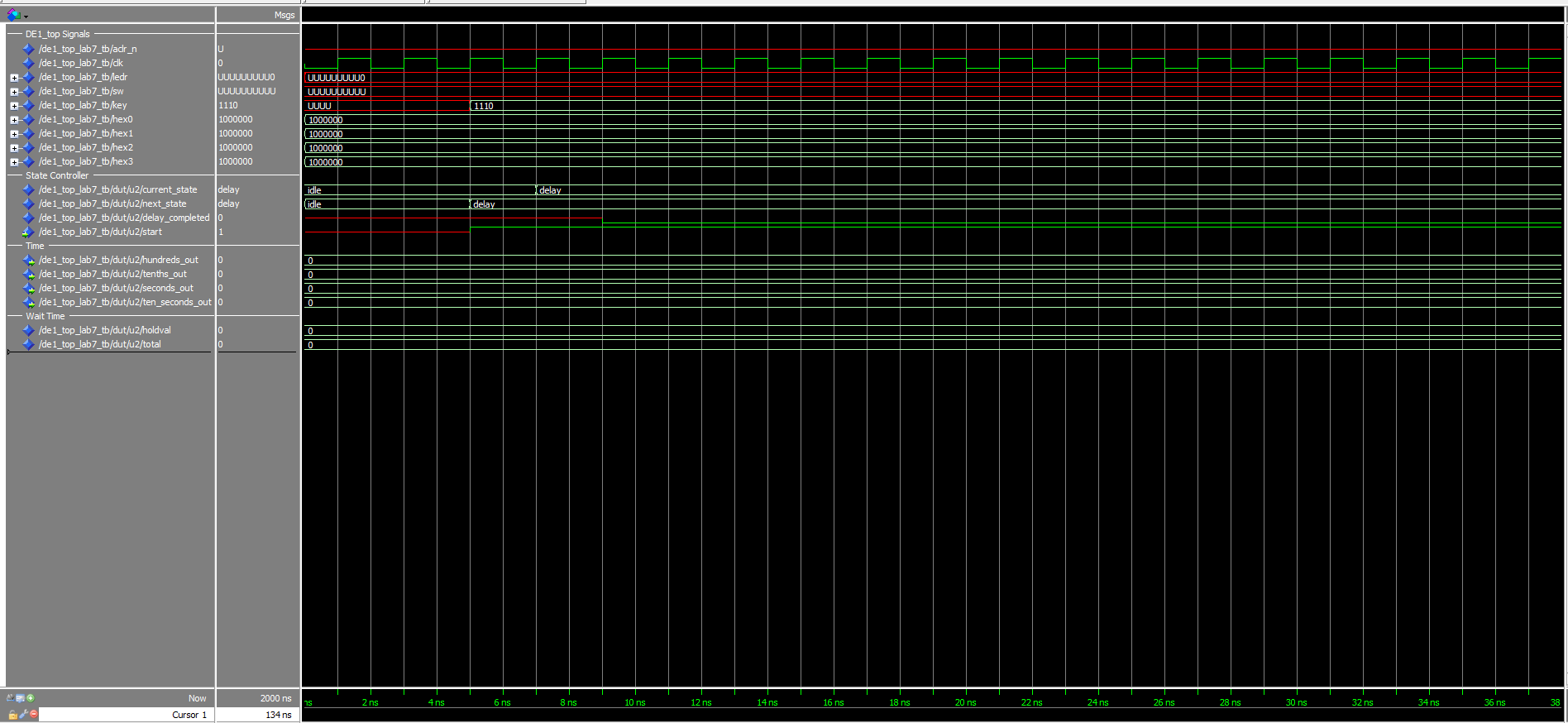
1. **Verification**

**Test Plan:**

To ensure that our design is running correctly, there are a few guidelines that must be followed to ensure that reflexes are being tested optimally.

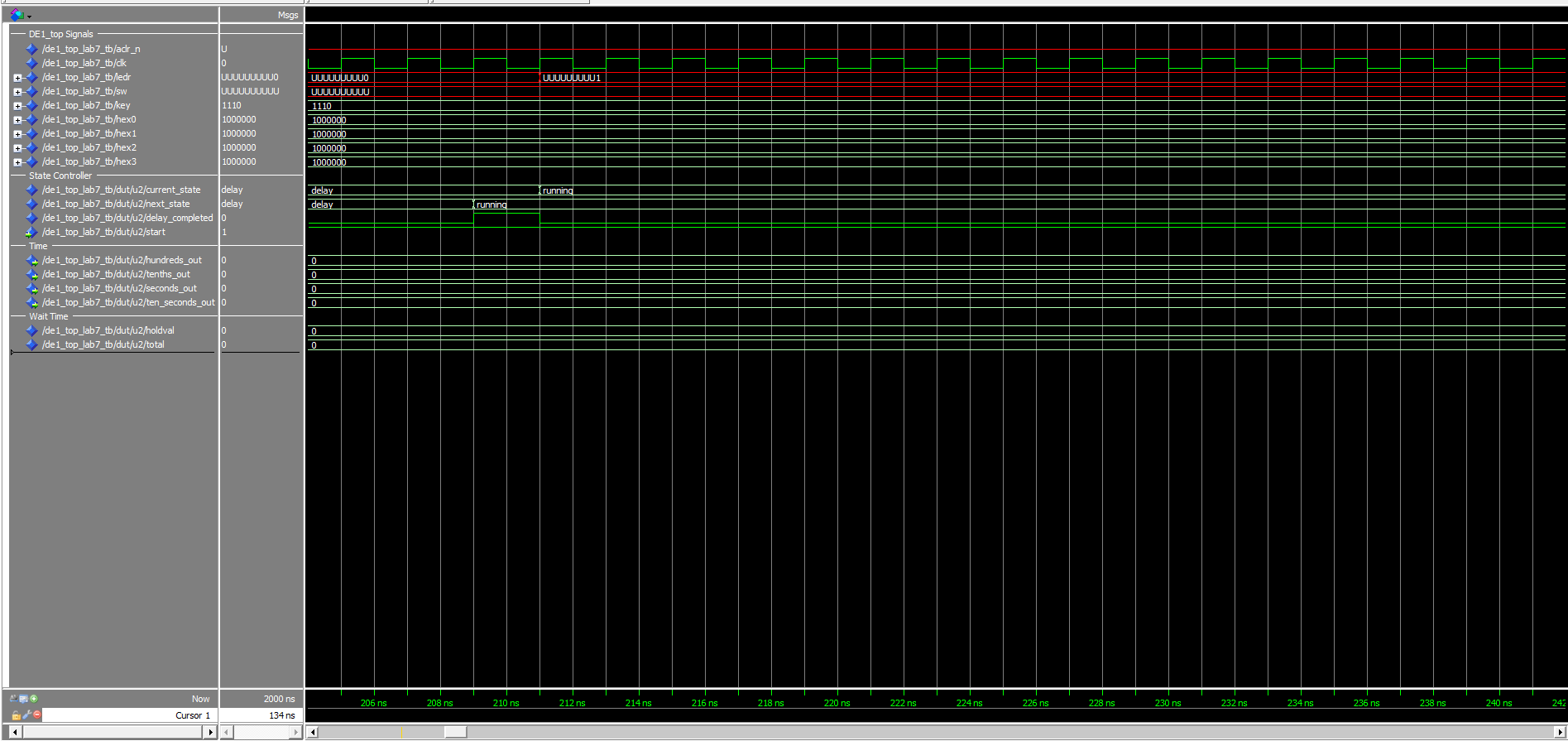
1. Starting the application, a one second delay must occur
2. During this delay, the LED should be turned off until one second has been reached, then it must turn on
3. Once the application has begun, no output will be displayed to the user until the stop button has been pressed. The LED will remain on during this time.
4. After the stop button is pressed, display the output to the user, and wait until user input occurs again. Once the button is pressed, turn off the LED light.
5. Before testing the reflexes, ensure that a random number is generated to be used to count down from.
6. Application must wait until user input has been received to start the reflex test.
7. Once the application has received the “random” number, LED remains off until the counter has reached the random number.
8. Once the random number has been reached, turn red light on, show that user output was successful if stop button was pressed.
9. If failure to wait for light to turn on, penalize user.
10. If failure occurs twice in a row, fail the user.
11. If user is successful at testing their reflexes, return to reflex state and start the application from the point where our random number was reached.

**Initial Phase Idle to Delay**

****

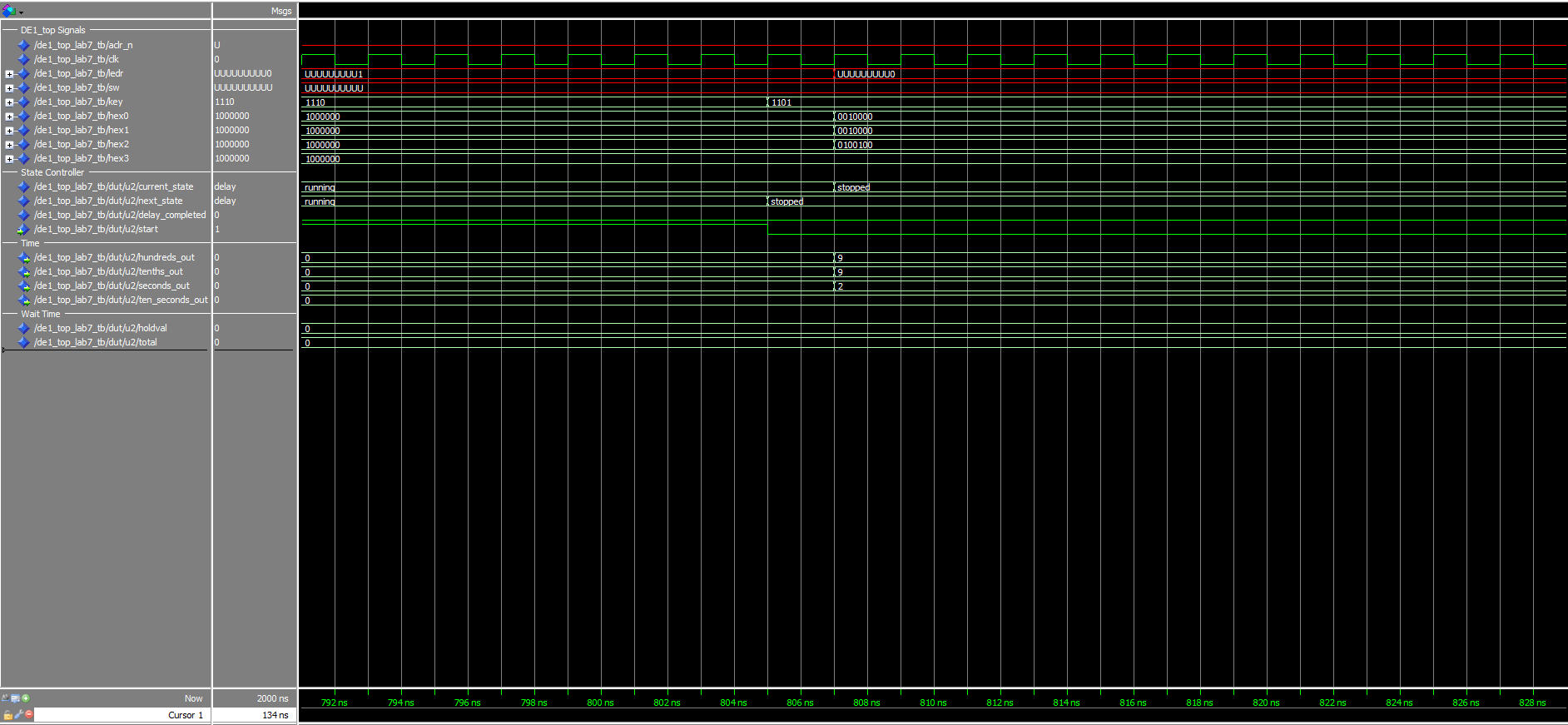
Application awaits for start Key “1110” to be pressed. Once pressed, a one second delay occurs. After delay occurs, running has begin and the user sees no output of numbers.

**Delay to Running**

****

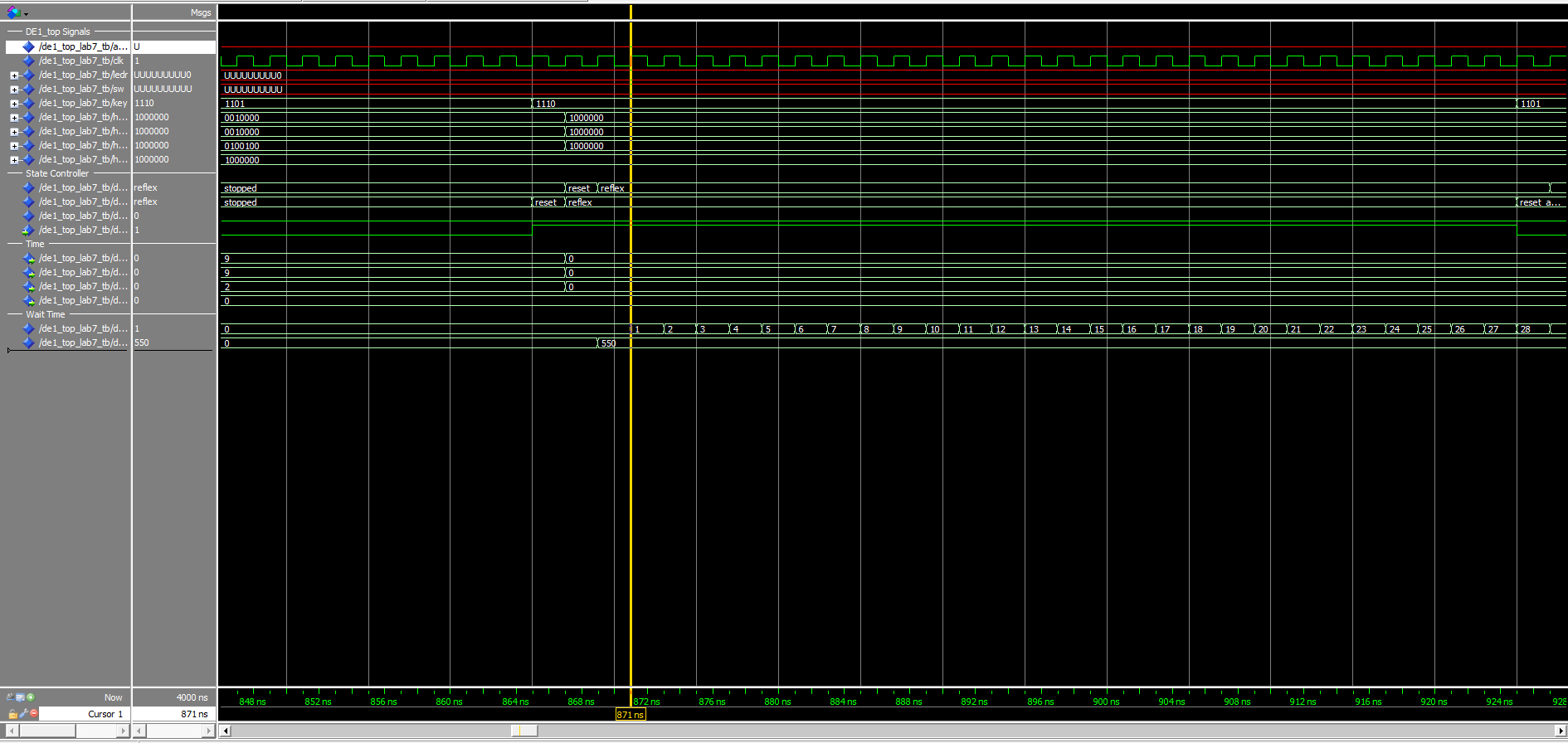
Application waits until key “1101” has been pressed and the application stops.

**Stop running, get the seconds it took to stop**

****

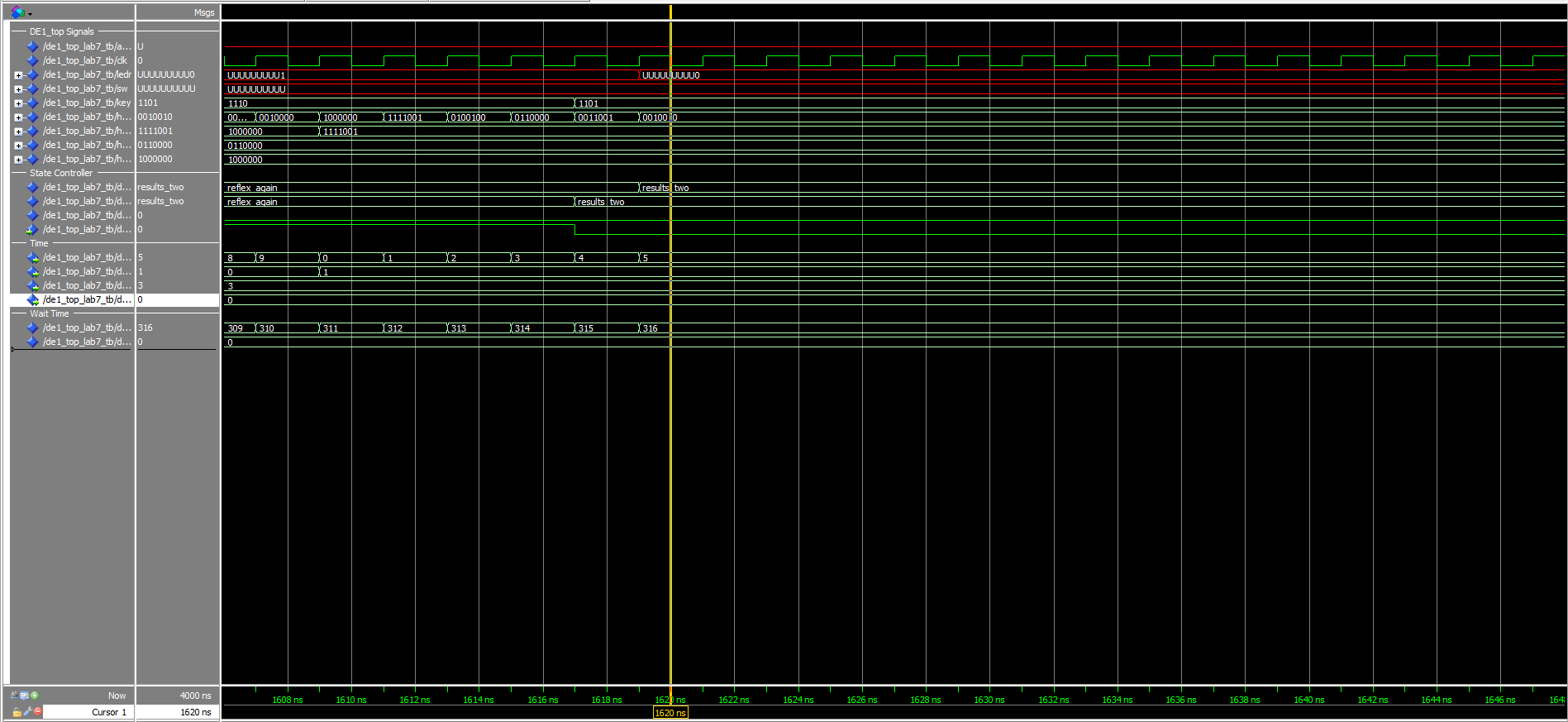
User input received Key “1101” and the number of seconds it took to stop are displayed.

**Start Counting and wait for user to stop**

****

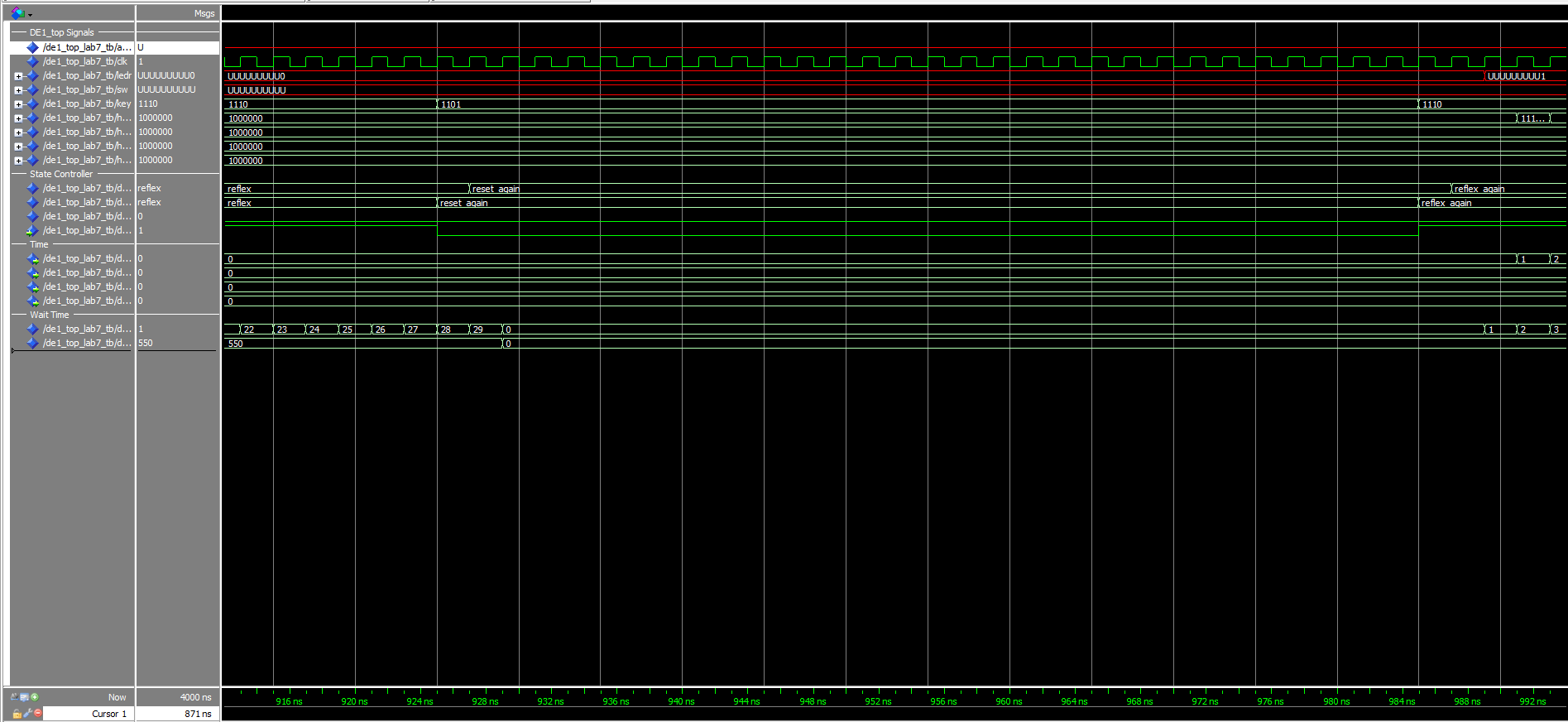
User will continue to see nothing, but the “total” is an integer representing the number of seconds to wait until LED is on and the user can successfully stop. Holdval is the value that counts up. Once Holdval equals total, LED will come on.

**Successful stop, LED is on, HoldVal has reached Total**

****

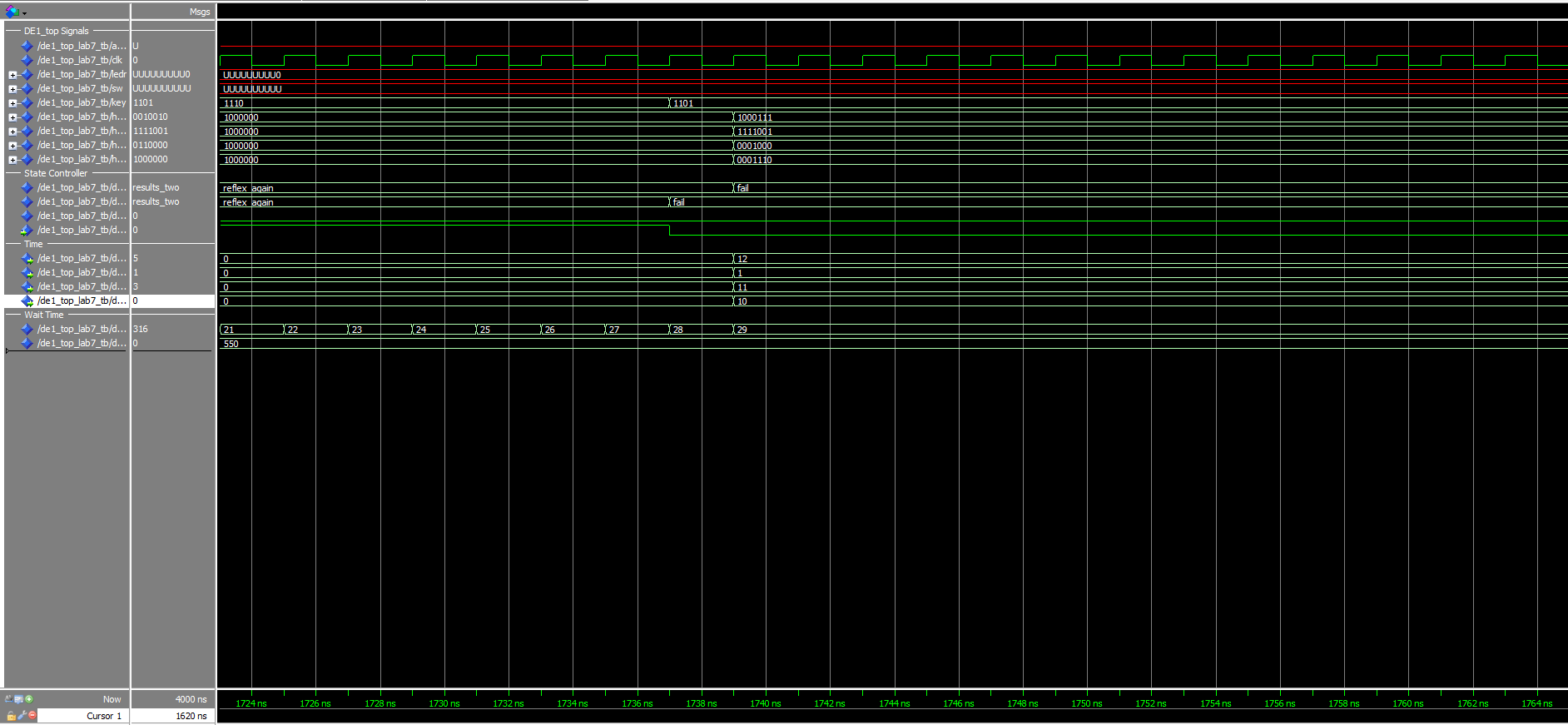
User input received Key “1101” and the number of seconds it took to stop are displayed. After 1 second, and 2 tenths of a second, the light turns on. The user took 3 seconds, 1 tenth of a second, and 5 hundredths of a second to turn the light off.

**First Unsuccessful stop, LED is off, HoldVal has not reached Total**

****

User has pressed the button before the light has turned on, causing the application to fail once.

**Second Unsuccessful stop, LED is off, Fail is reached**

****

User has unsuccessfully stopped too soon once again, causing the application to reach the “fail” state, where the output to HEX0 -> HEX3, to display FAIL.

1. **Conclusion**

Overall, the result of the lab was successful, but I generally believe that the route that I took was a very inefficient, complicated mess to get there. Originally I had planned on using 4 separate counters, but my application timing was constantly off. Using one counter works, and I am able to demonstrate the ability to use a stopwatch, generate a pseudo-random number, count down from it and receive user input to determine if their reaction was too fast, or was after our random number was reached.

The struggles came in from understanding how the counters worked originally. I tried many attempts to use proper counters to determine the amount of time it took to run the application and receive user input. Once I was able to figure out my implementation, I figured I would just stick with it, even though at this point my understanding of counters is greater than it was when I started the application. If I were to attempt the implementation again, I would definitely use separate counters and make use of the modulus function inside of Quartus.