CS M152A

Lab 3: Stopwatch

Bryan Wong

805 111 517

Teammates: Austin Zhang, Natasha Sarkar

Date: February 19th, 2020

TA: Srishti Majumdar

Section: Lab 2

1. Introduction and Requirements

For this lab, we were tasked with using Verilog to design a basic stopwatch module for our Nexys3 Spartan-6 FPGA board. In its most basic functionality, the stopwatch counts upwards and displays the elapsed time in a 'MM:SS' format on the board's 7-segment display. Additionally, the stopwatch has a debounced pause button that freezes/unfreezes counting and an asynchronous reset button that resets the stopwatch time to '00:00'. Once the stopwatch reaches '59:59', it is unable to count any further and is automatically paused.

There are also switches for the 'adjust' and 'select' states. If the 'adjust' switch is turned on, the stopwatch rapidly increments at a rate of 2 digits per 2hz tick. If 'select' is set to 1, the seconds are incremented; if 'select' is set to 0, the minutes are incremented. By operating these two switches, the user can quickly adjust the stopwatch's time to a desired value. Additionally, when the adjust mode is turned on, the field being adjusted (minutes or seconds) should blink at a rate of 4hz.

In terms of practical applications, our stopwatch can perform many of the same functions that any commercial stopwatch can perform. For example, it could be used to time athletic events such as track and field or horse racing. However, there are some limitations; since it is unable to keep a log of past times, it would not be suited for functionalities such as the recording of lap times in a race.

2. Design Description

Our top module, named 'nexys3.v', is responsible for reading input from the switches and buttons, debouncing the button inputs, and connecting each of the 3 submodules. It asynchronously senses the rising edge of the 'reset' button press, as to minimize latency between the button press and the resetting of the stopwatch's counter. The top module also uses a clock divider to sample the 'pause' button at a lower rate, and freezes the stopwatch's counter upon a rising edge. We made sure that the 'pause' button was very responsive so that the stopwatch could be frozen precisely. The top module then takes these inputs and processes them using the following submodules: 1) the 'clock' module to divide the base 100 Mhz clock into various different clock rates, 2) the 'counter' module to keep track of the elapsed time, and 3) the 'display' module to use the counter to activate the corresponding sections of the 7-segment display. Once all 3 submodules are used, 4-bit 'an' and 8-bit 'seg' outputs are used to display the stopwatch's counter on the 7-segment display.

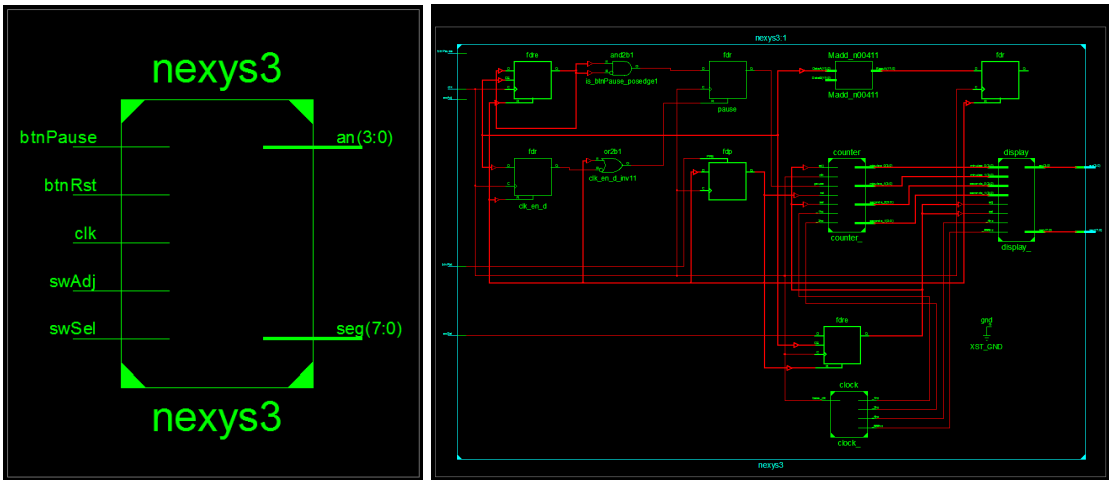

Figure 1: Stopwatch Top Module Circuit Diagram. 2 buttons, 2 switches, and an internal clock are used as inputs in order to output 4 7-segment displays.

a. clock.v

The 'clock' module takes in the base 100 MHz clock as an input and outputs 4 different clocks: 1) a 1 hz clock for incrementing the counter normally, 2) a 2 hz clock for incrementing the counter during 'adjust' mode, 3) a 4 hz clock for flashing the display in adjust mode, and 4) a 666.66 hz clock for flashing the 7-segment display fast enough where it appeared constant to human eyes. This was accomplished by creating counter registers and inverting the divided clock output once the counter reached a certain value. For example, to generate the 1 hz clock, a 26-bit counter register was used. The counter is incremented once every 100 MHz clock cycle. Since a 1 hz clock is 100000000 times slower than the base clock, the _1hz output is inverted once the counter is equal to 50000000. At this time, the counter resets and begins incrementing itself for the next inversion of the 1 hz clock. The same process is used for the _2hz, _4hz, and _666hz outputs, with different counter sizes and reset values. Functionally, the divided clock signals outputted by this module can now be used by the 'counter' and 'display' modules.

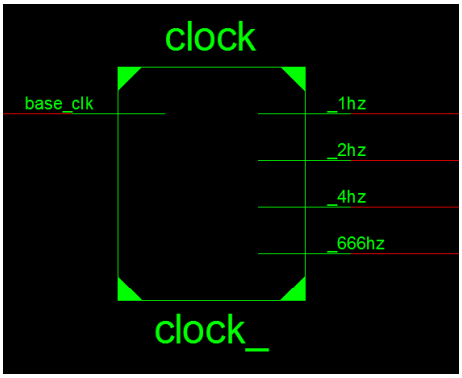

Figure 2: Clock Submodule Circuit Diagram. Several counters are used to convert the base 100 Mhz clock into 1 hz, 2 hz, 4 hz, and 666hz clocks.

b. counter.v

The ‘counter’ module is responsible for managing the internal counter that represents the time elapsed. It takes in rst, pause, the base clk, _2hz, _1hz, sel, and adj as inputs and outputs the 4 counter digits to be shown by the 7-segment display. The 2-bit ‘_1hz_buffer’ and ‘_2hz_buffer’ and 1-bit ‘_1hz_valid’ and ‘_2hz_valid’ registers are used to sense the rising edge of the inputted clock signals. In its normal functionality, the rising edge of the 1 hz clock allows the least significant seconds digit to be incremented by 1 every second. If the new value of a digit overflows the digit (10 for the least significant digit, 6 for the next least significant digit), a carryout is used to increment the next most significant digit. If the counter reaches its maximum value of ‘59:59’, it is unable to increment any further and is automatically paused.

If the adjust mode input ‘adj’ is on, the counter behaves similarly but increments the region specified by ‘sel’ at an incrementation rate of 2 once every half second. If ‘sel’ is 0, the minutes are incremented once per rising edge of the _2hz signal and if ‘sel’ is 1, the seconds are incremented instead. A notable difference between the ‘adjust’ mode incrementation and the regular incrementation is that upon reaching the maximum value, a modulus operation with the max value is used so that the digit can be incremented properly. For example, if the least significant digit is a 9, it becomes 11 once incremented and becomes 1 once it is modulo’d with the max value of 10. Also, if the most significant minutes or most significant seconds digit reaches its max value, it is reset to 0 without affecting any other digits. This allows the adjust mode incrementation to wrap around once it hits its maximum value. Finally, the ‘counter’ module allows the pause state to be toggled, and will not allow the counter to be incremented while paused, effectively freezing it. If the reset button is pressed, the module will reset all of the counter’s digits to 0.

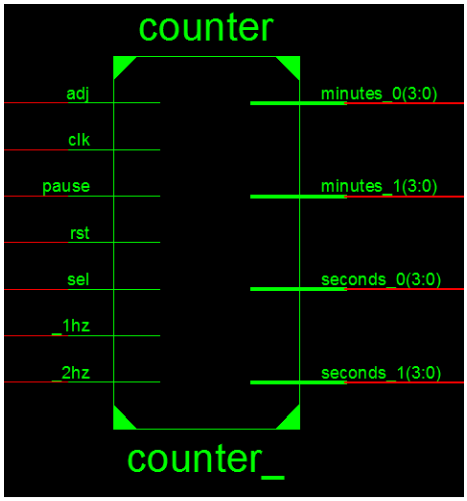

Figure 3: Counter Submodule Circuit Diagram. This module is responsible for recording the time elapsed, and managing pause/rst/adj functionalities.

c. display.v

The ‘display’ module is responsible for displaying the ‘minutes_1’, ‘minutes_0’, ‘seconds_1’, and ‘seconds_0’ inputs by turning on the corresponding segments in the 7-segment display. Since the Nexys3 board is unable to simultaneously display 4 different digits at the same time, we instead use a 666.66 hz clock to rapidly flash one digit at a time. This is accomplished by using a modulo 4 counter to select the digit to be displayed. We also used a set_display_segs function to convert an integer digit value into the segments to be displayed. This was quite confusing at first, since we did not realize that the 7-segment display logic was inverted; a 0 corresponded to a lit segment and a 1 corresponded to an unlit segment. The resulting effect is that the user perceives the 7-segment display as a 4-digit ‘MM:SS’ time. Additionally, if the adjust mode is on, the 7-segment display will flash the adjusted section at a rate of 4 hz.

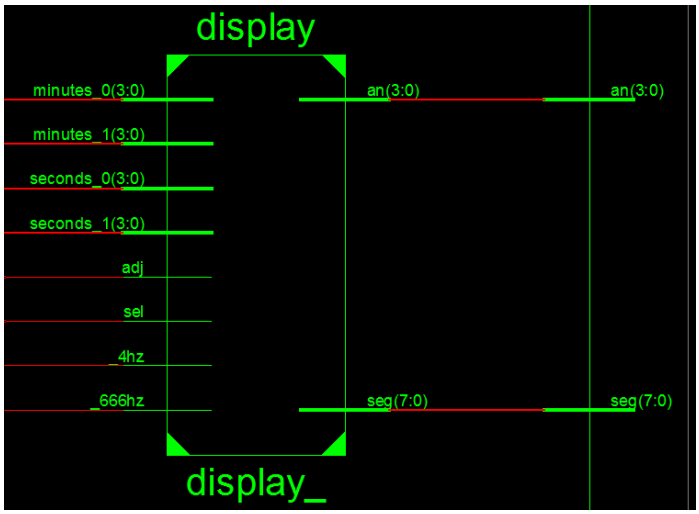

Figure 4: Display Submodule Circuit Diagram. This module takes in the 4 digit inputs and turns on the corresponding parts of the 7-segment display. If adjust mode is on, it also blinks the adjusted digits at a rate of 4 hz.

3. Simulation Documentation

In order to make sure that each of our submodules was working properly before proceeding, we created a testbench for each one. These testbench modules are explained in more detail below:

a. clock_tb.v

The clock submodule was pretty simple and straightforward to test. We artificially created a 100Mhz base clock in our testbench code and verified that each of the outputted divided clock signals was accurate. We encountered no bugs for this simulation. Waveforms can be found below:

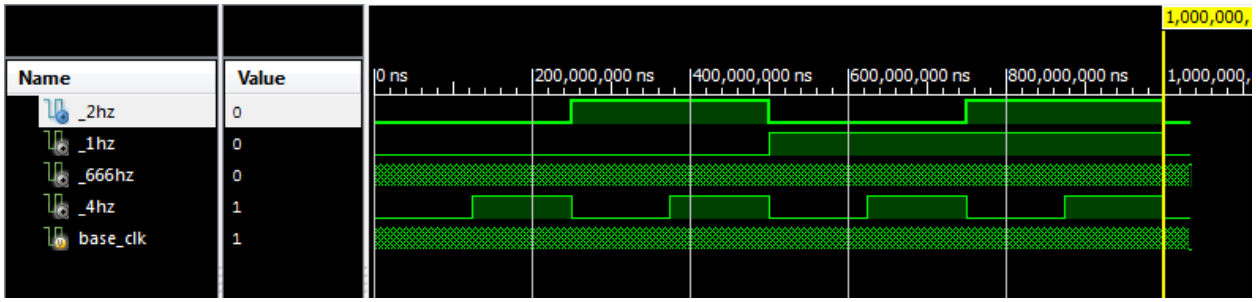

Figure 5: clock_tb.v Simulation Waveforms. Pictured is 1 second of this simulation's waveforms. Each of the outputted signals is divided properly; for example, the _1hz signal has a period of 1 second.

b. counter_tb.v

In order to test the counter submodule, we had to account for a variety of test cases. In all cases, we generated a base 100 Mhz clock, 1 hz clock, and 2 hz clock in the code and verified that the counter behaved correctly. Our first test case was making sure that the base functionality of the stopwatch (with adjust mode off) worked properly. Our simulation revealed a bug where the counter failed to increase since we did not initialize our posedge buffers or pause status, causing the counter to stay frozen at '00:00'. Another bug that we found after uploading the module to the board was that the pause status was set to 0 when the reset button is pressed. The intended behavior is for the pause status to be preserved in this case. The corrected simulation waveform can be found below:

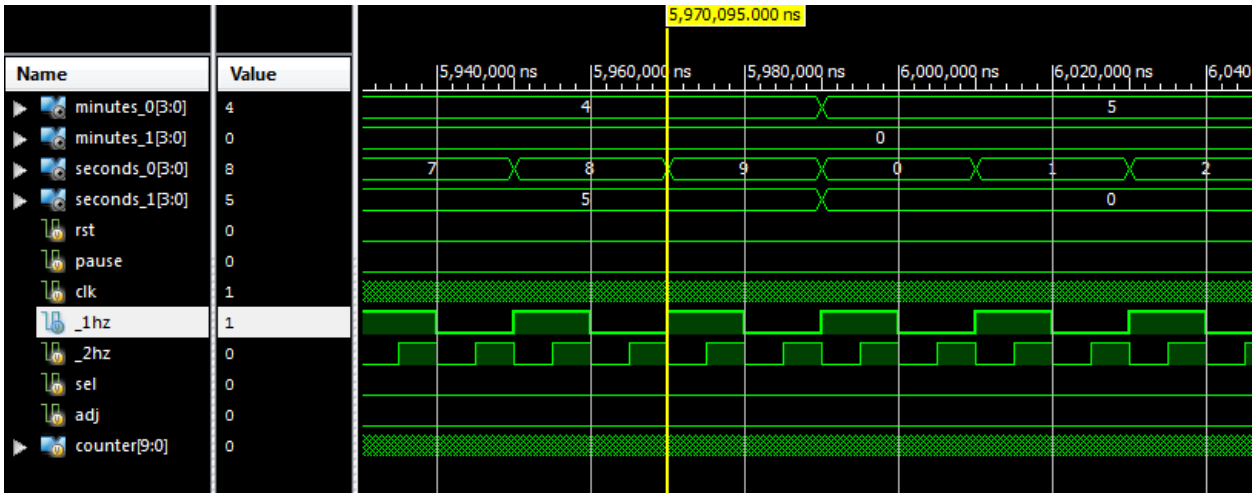

Figure 6: Regular counter_tb.v Simulation Waveforms. As we can see, the seconds_0 output is incremented once for every positive edge of the _1hz clock signal. The carryout also works properly, allowing us to go from '04:59' to '05:00'.

We also tested the automatic pause functionality by manually setting the counter to ‘59:57’ and running the simulation waveform. We verified that this worked properly, since the time becomes frozen at ‘59:59’. The waveforms can be found below:

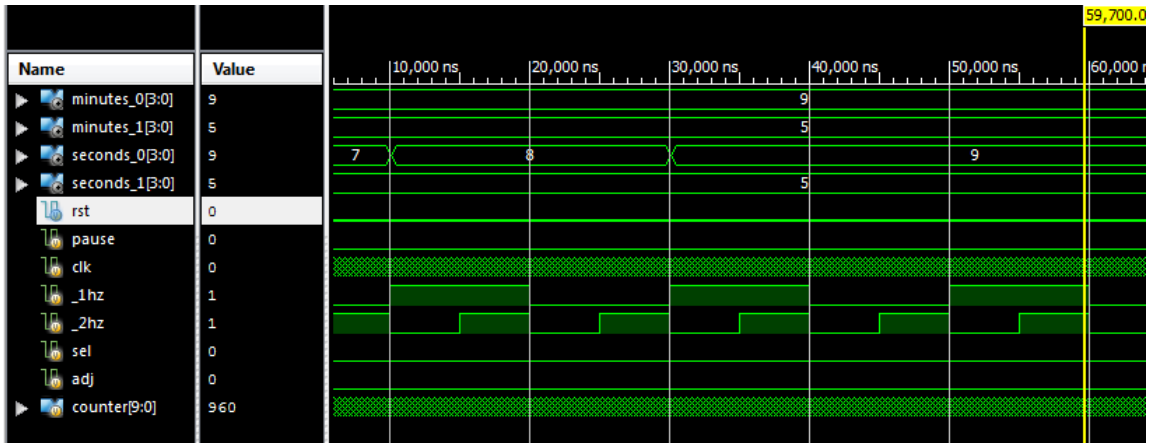

Figure 7: Auto-pause counter_tb.v Simulation Waveforms. Once the elapsed time reaches ‘59:59’, it no longer increments and the pause_status (not pictured) is set to 1.

Lastly, we used simulation waveforms to test that the adjust mode behavior of the counter worked. By setting the ‘adj’ option to 1 in our code, we were able to verify that the counter incremented by 2 at a rate of 2 hz. We also initialized the counter to an odd value (like ‘00:59’) to make sure that the modulo operation worked properly in adjust mode. A bug that we encountered was that our ‘sel’ logic was inverted. Our simulations can be found below:

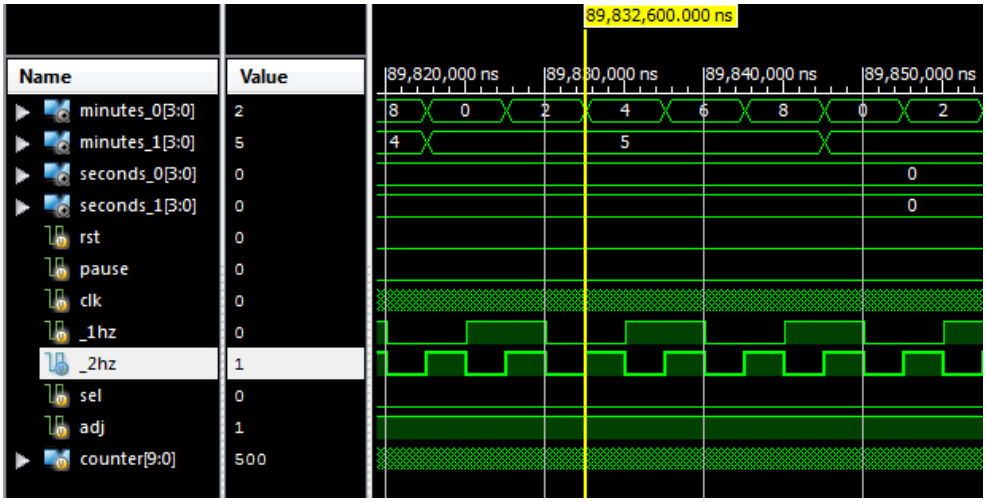

Figure 8: Adjust mode counter_tb.v Simulation Waveforms. As we can see, the ‘adj’ option being set to 1 causes the counter increments by 2 with the positive edge of the _2hz clock signal.

c. display_tb.v

In order to test our display submodule, we manually inputted various stopwatch times and ensured that the 7-segment display outputs were cycled properly. We also made sure that it blinked properly at a rate of _4hz when in adjust mode. Overall, we were not able to identify any glaring errors from our simulation waveforms, which are shown below:

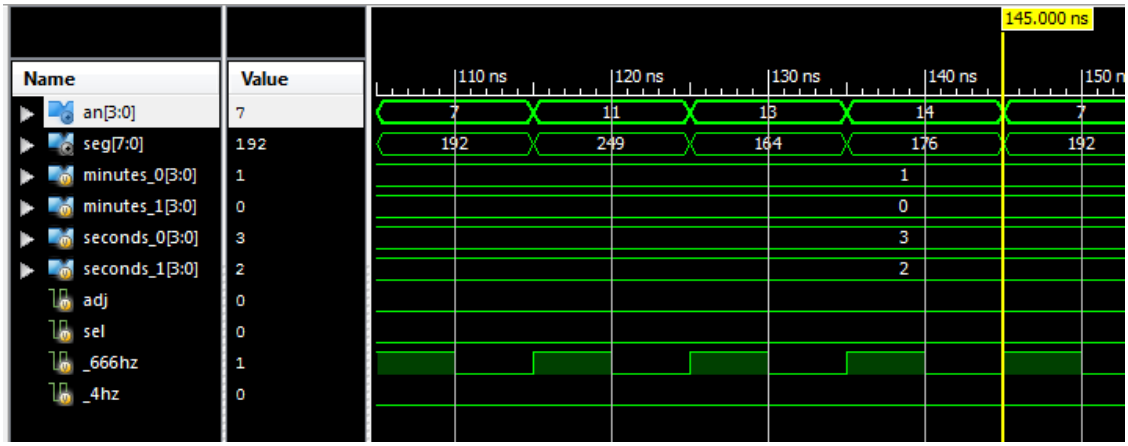

Figure 9: display_tb.v Simulation Waveform. As we can see, the four 7-segment display outputs are cycled to display a digit at a time, switching at the positive edge of the _666hz clock. In this case, '01:23' is displayed, so the digits 0, 1, 2, 3 are cycled.

One bug that we were unable to catch with our simulation, but did encounter later, was the fact that the 7-segment logic is inverted. For example, a '1' signals an unlit segment and a '0' signals a lit segment. We realized this when uploading the stopwatch code to the board, and were able to quickly correct it.

4. Conclusion

Ultimately, we were able to successfully implement a stopwatch module with full adjust, pause, and reset functionalities on our Nexys3 Spartan-6 FPGA. By using what we learned from previous labs and incrementally testing each submodule using testbench simulations, the process was relatively seamless. Bugs that we encountered were for the most part easily identified and corrected.

The most difficult part of the lab for us was implementing the display submodule, since we had to generate a function to convert digits to segments as well as manage the logic for cycling through each digit in the adjust/non-adjust cases. In addition, the testbench simulations were not particularly revealing since we had to upload the module to the board in order to test it. Overall, this lab was a great learning experience in developing our own modules from the ground up, and showed us the benefits of testing submodules as they are developed.