Lab 3 Report:
Stopwatch

CS M152A, Potkonjak
Lab 3, Gu
Spring 2017

By,
Eddie Huang, Lucas Jenkins, and Jason Less

INTRODUCTION AND REQUIREMENTS

Background:

For this lab, we were tasked with designing and implementing a digital stopwatch using the seven-segment display of the Nexys 3 Spartan-6 FPGA board, in addition to two of the board's switches and buttons respectively. The stopwatch consisted of four seven-segment displays; the left two displays representing the minutes and right two displays representing the seconds. Both the minutes and seconds field of the stopwatch begin at 00 and increment up until they reach 59 (at which time they will be reset back to 00).

Moreover, the stopwatch consists of two modes: normal mode (in which the stopwatch increments upwards once per second using a 1 Hz clock) and adjust mode (in which either the minutes or seconds fields of the stopwatch increment upwards twice per second using a 2 Hz clock). By default, the stopwatch begins in normal mode, but in order to switch back and forth between normal and adjust mode, an adjust switch (ADJ) is used. When the adjust input is set to logic high (i.e. the switch is flipped to its up position), the stopwatch is in adjust mode, and when the adjust input is set to logic low (i.e. the switch is flipped to its down position), the stopwatch is in normal mode. To select which field of the stopwatch is in adjust mode, a select switch (SEL) is used. When the select input is set to logic high, the seconds field is in adjust mode, while the minutes field is frozen at its current value, and vice versa for when the select input is set to logic low.

ADJ	Action
0	Stopwatch behaves normally
1	Stopwatch stops and *'Selected'* increases at 2Hz

Figure 1: Adjust mode chart. If ADJ is not selected, the stopwatch behaves normally. If it is selected, it goes into adjust mode. On the basis of the select bit, either the minutes or the seconds increase at 2Hz while the other remains constant. See Figure 2 for how SEL is used to determine the adjust mode.

SEL	Selected
0	Minutes
1	Seconds

Figure 2: Select mode chart. If select is 0, (with ADJ set to 1) the watch will be in minutes adjust mode. If select is 1, the watch will be in seconds adjust mode.

Furthermore, the stopwatch supports two additional features: reset and pause (which are implemented using two of the board's buttons). When the reset input is at logic high, the value of the stopwatch is reset to 00:00. In addition, when the pause input is at logic high, the current value of the stopwatch is paused (or frozen in place) to its current time.

Lastly, the stopwatch implementation supports four different clock inputs: the 1 Hz clock (for normal mode), the 2 Hz clock (for adjust mode), the 4 Hz clock (for blink mode), and the 400 Hz clock (to display the stopwatch time using the seven-segment display). The 1 Hz clock and 2 Hz clock were previously discussed. However, the 4 Hz clock is used during the adjust

mode in order to have the time field that is being adjusted (either minutes or seconds) blink. By having the adjusted time field blink, the user of the stopwatch is able to identify which time field is being adjusted. The 400 Hz clock is used to display the actual values for each time field in the seven-segment display. By cycling through each of the four seven-segment displays very quickly, the segments are able to display the time of the stopwatch.

Design Requirements:

To implement the stopwatch, our group designed a modular system that consisted of a single top module and seven sub-modules each having their individual tasks. First of all, to create the four clock frequencies we created a clock divider (clk_div.v) module. This module created the four different clocks (i.e. 1 Hz, 2 Hz, 4 Hz, and 400 Hz) using the FPGA board's master clock (100 MHz).

Next, a debouncer module (debouncer.v) was used to debounce the two buttons and switches used. To prevent bouncing of the switches, a counter variable was used to record the state of the button or switch (i.e. high or low). After waiting a certain period of time for the signal to become stable and to register that the button or switch was actually enabled (and a glitch didn't occur), the state of the signal was confirmed, and was set to its appropriate state. Moreover, to deal with the metastability problem, two registers were used to synchronize the state of the signal to the frequency of the clock. By doing this, the register input was made stable for a minimum period of time before and after the clock edge, and the possibility of unpredictable outputs to the other modules connected to the signal were reduced.

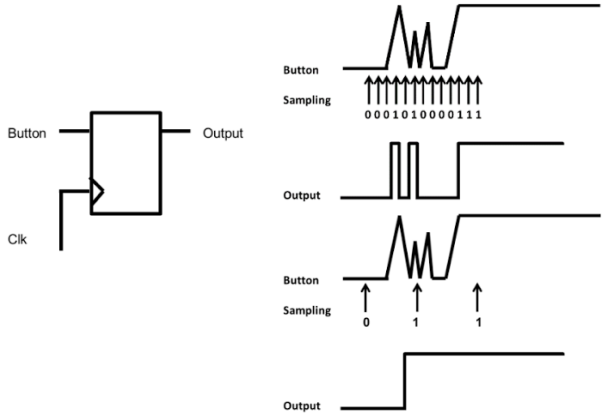

Figure 3: Debouncing button input. At a high sample rate, a shaky initial button signal can result in multiple clock signals being incorrectly registered. By searching for multiple consecutive signals from the button to eliminate the possibility of noise, one can ignore the noise generated due to poor metal contact, and produce the intended clock signal.

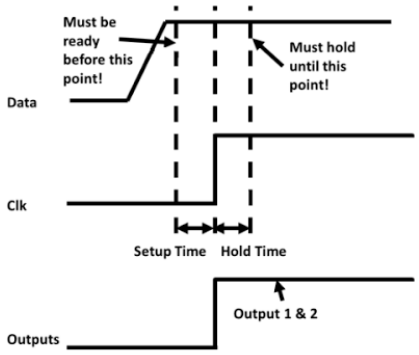

Figure 4: Solving metastability. By using two registers to synchronize the state of the signal to the frequency of the clock, it stabilized the register input and reduced the possibility of unpredictable outputs.

The main logic of the stopwatch was implemented in the counter module (counter.v), which contained a select and adjust module (sel_adj.v). The counter module incremented the time fields of the stopwatch at the frequency of the given clock (1 Hz or 2 Hz), and reset the fields accordingly when overflow occurred (e.g. seconds field goes from 59 to 00, and the minutes field is incremented by 1). The select and adjust module is used to select which mode (normal or adjust) and to choose the corresponding clock (1 Hz or 2 Hz) to use.

Lastly, three modules were used to create the seven-segment display: one module to separate the time fields into separate digits (i.e. 10's place for minutes, 1's place for minutes, 10's place for seconds, and 1's place for seconds) called separate_digits.v, one module to choose which cathodes corresponding to the four digits should be high called display.v, and another module to choose which anodes should be high and to cycle through the four anodes using the 400 Hz clock to display the stopwatch values via the seven-segment display called final_display.v

DESIGN DESCRIPTION

As described in the introduction, the functionality of our stopwatch was divided into a number of submodules. These modules can be divided into three different types: core functionality, user interface, and physical error handling. The core of the stopwatch is the logic which controls how the stopwatch counts up and how it displays the values as it counts up. The modules responsible for this functionality are the counter, separate_digits, display, and final_display modules. Some of these submodules controlled the user functionality - by pressing buttons or moving switches, the user could pause, reset, and adjust the value on the stopwatch. The modules that handled this functionality were the sel_adj and counter modules (as will be seen below, the counter module performs many duties). The last type of module deals with errors due to physical inputs - the debouncer module.

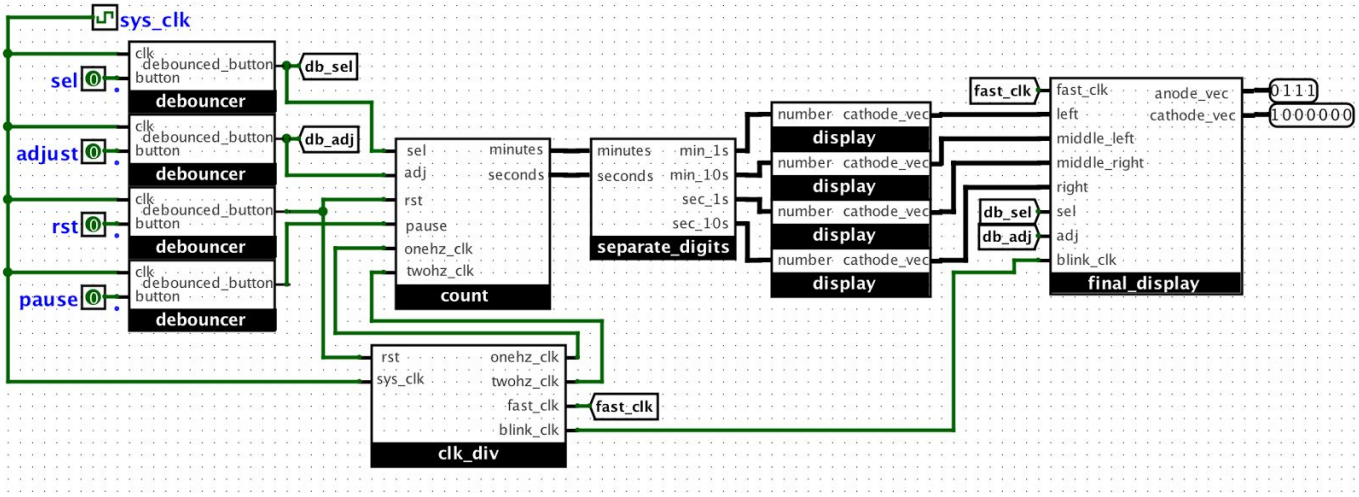

Figure 5: Top level design of our implementation: We divided up our stopwatch module into a number of smaller submodules in order to separate the logic for the many different functions of the stopwatch.

The top level design works as follows: The user inputs are passed through debouncing logic so that the signal received from them is consistent with what the user expects. This debounced signal is passed to a counter module. In addition to keeping track of the current value of the count, the counter module uses the user input to control the way in which the current value of the stopwatch is modified. The clk_div module divides up the system clock into a number of slower clocks, two of which are used by the above counter module. Because the counter module stores the minutes and seconds as full values, we created a separate_digits module to perform arithmetic on the minute and second values to extract their respective one's and ten's bits. Once we had a single value for the one's and ten's bits, we passed these values to the display modules, which output the values onto the FPGA board's seven-segment display.

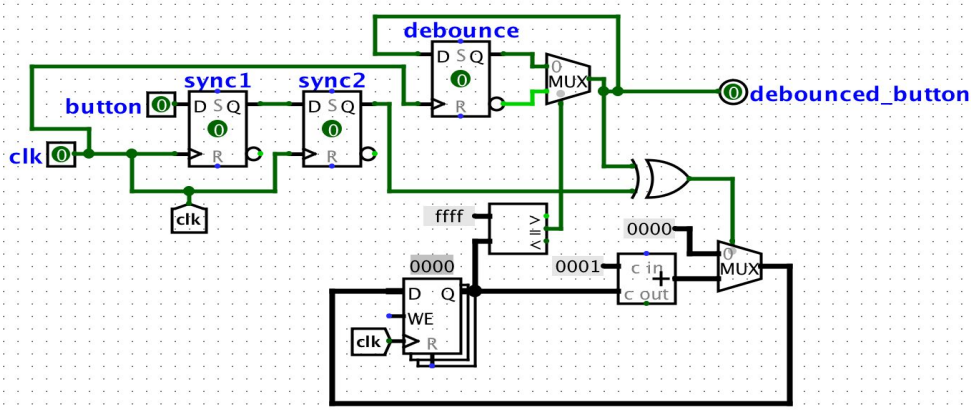

Figure 6: Implementation of the debouncer module: The debouncer module consists of two distinct parts: one which handles metastability and one which deals with bouncing input

The debouncer module actually performed two separate functions to handle input errors due to physical effects. The first of these functions dealt with metastability, a problem which occurs when the input signal is not stable for a long enough period of time before and after a rising clock edge. By passing the input through two registers (labeled sync1 and sync2) before it is passed on to other logic, we ensure that the input signal passed to the rest of the stopwatch will

be stable around clock edges. The rest of the debouncer module deals with the problem of bouncy input buttons. By having a register which counts up on every clock tick when the current value stored in the debounce register is not equal to the input, and only changing after this count reaches 0xffff, we can be sure that the debounced_button will only change when the input button has been stable for 0xffff clock ticks, which we trust is long enough to be an intentional button press or release and not just a bounce in the signal.

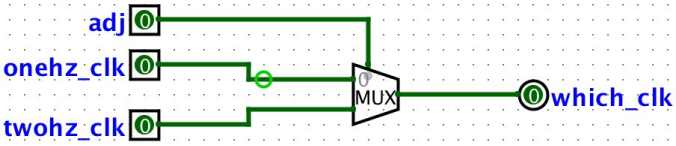

Figure 7: Implementation of sel_adj module: A simple module for determining which clock signal to use

The sel_adj module was a very simple module - we simply passed it the adjust signal, and if it was high, we would use the 2 Hz clock signal instead of the 1 Hz, because the digits of the display were supposed to change at 2 Hz in adjust mode. The output of this module was used in our counter module.

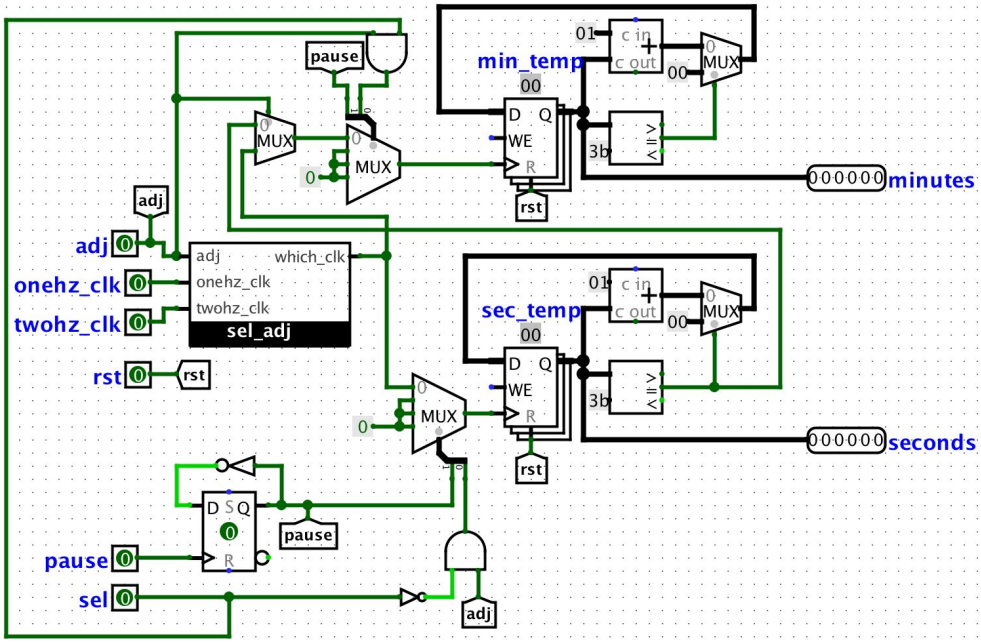

Figure 8: Implementation of counter module: The counter module is the most complex module in our design

As the counter module handles all of the input logic and the core functionality of the stopwatch, it is bound to be the most complicated module in the stopwatch. The core of the counter module is in the sec_temp and min_temp registers, which store the state of the minutes and seconds values. The sec_temp register simply counts up to 59 and then resets to 0 while simultaneously sending a signal that it has reached 60 to the min_temp register, which then increments its value by one (and also resets to 00 after reaching 59). The rest of the module uses the user input to modify when the values for the minutes and seconds are incremented. The min_temp and sec_temp registers were always connected so that when a clock tick came, they would increment. Thus, we used multiplexers to control the clock input to these registers. By using the user input (pause, sel, adj) to control when the clocks were updated, we could control

when the registers incremented their value. The registers in logisim have an asynchronous reset input, so by tying this input to our rst button, we could asynchronously reset the values passed to the display to 0. Importantly, while the adjust and select switches depended directly on their input, the pause button would pause and unpause the stopwatch every time it was pressed (even briefly). Thus, we needed a register to store the previous state of the pause button. Also, the multiplexed value to the clocks was set to zero when the adjust option was high and the select option was the opposite of the specified value for a given register (e.g. when sel was low for sec_temp). This is because in the case where adjust was high and select matched the register, the correct clock value would be given by the output of the sel_adj module, which is passed as the input to the min_temp and sec_temp registers.

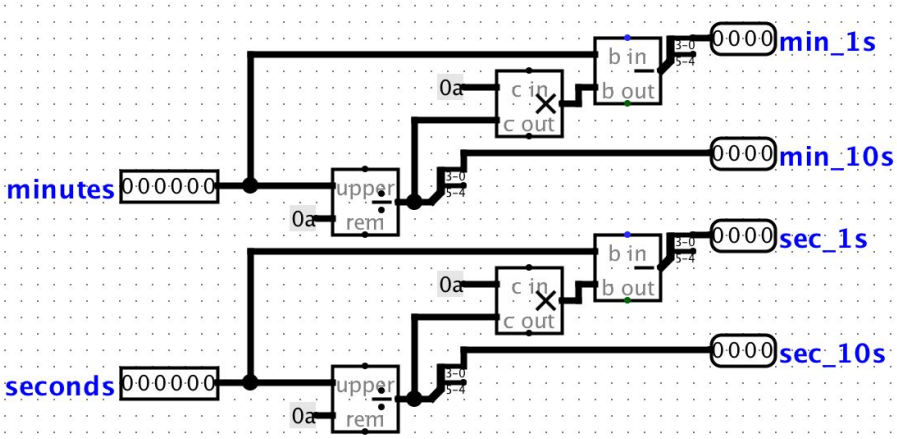

Figure 9: Implementation of separate_digits module: This module performs identical operations for minutes and seconds

The separate_digits performs a simple function. It takes the raw minutes and seconds value from 00 to 59, and separates the minutes and seconds values into their respective 1s and 10s place digits. In order to get the 10s place, we just divide the value by 10. Due to integer truncation in the division module, this gives us the 10s place. In order to get the ones place, we multiply the value of the 10s place by 10 and subtract it from the original number. This effectively gives us the remainder of the division performed to get the 10s place. This method of obtaining the ones place is more complicated than using modulo 10, but the Nexys board only supports modulo operations for powers of two, so we could not use this operation in our Verilog source.

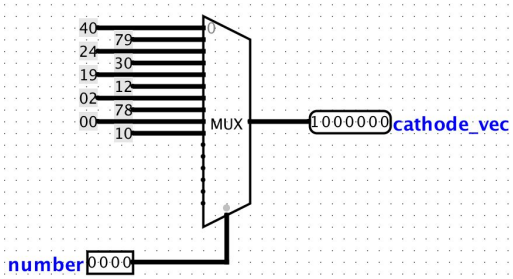

Figure 10: Implementation of display module: This module can be implemented with only one multiplexer

The display module has a slightly misleading name - it does not directly pass values to the display, but simply puts values into a form which can be passed to the display. It takes a

number which is one digit in base 10 as input, and outputs a vector corresponding to which lights should be illuminated on the seven-segment display in order to create that number. By using the number as the select for a multiplexer, we implement this logic very simply.

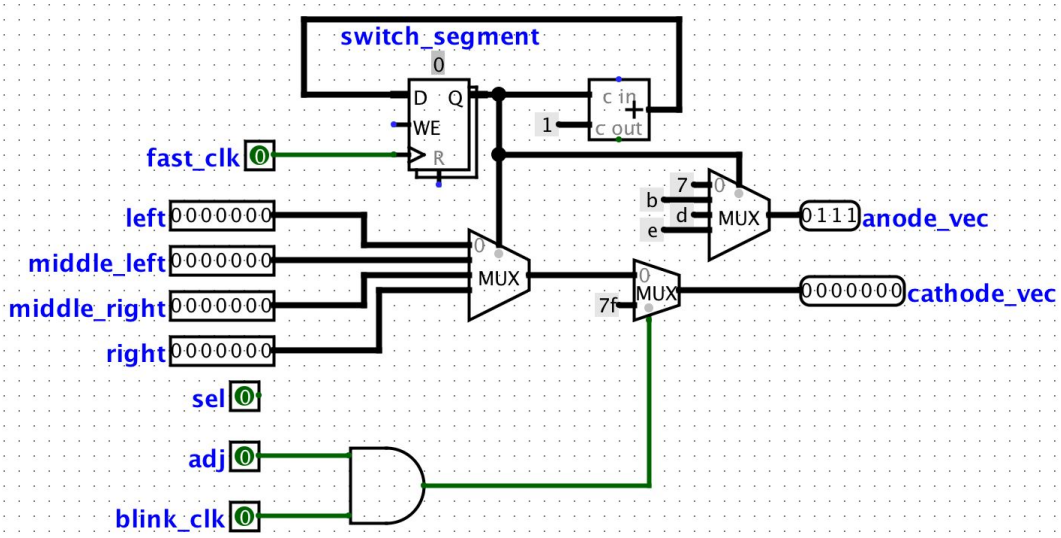

Figure 11: Implementation of final_display module

The output from each display module (one for each digit of the display) was passed to the final_display module, shown here. The switch_segment register is used to control which segment of the display is illuminated during a given tick of fast_clk. By choosing 0, 1, 2, or 3, the anode_vec will output a vector which tells the seven-segment display which anode to make high, and will also choose the correct cathode vector corresponding to that segment using the multiplexer. If the adjust switch is high, then we choose the cathode vector to be dark during whenever blink_clk is high, so that the display will be dark for half of the cycle of blink clk, which will cause the display to blink. This is so that the user can tell when they are in adjust mode.

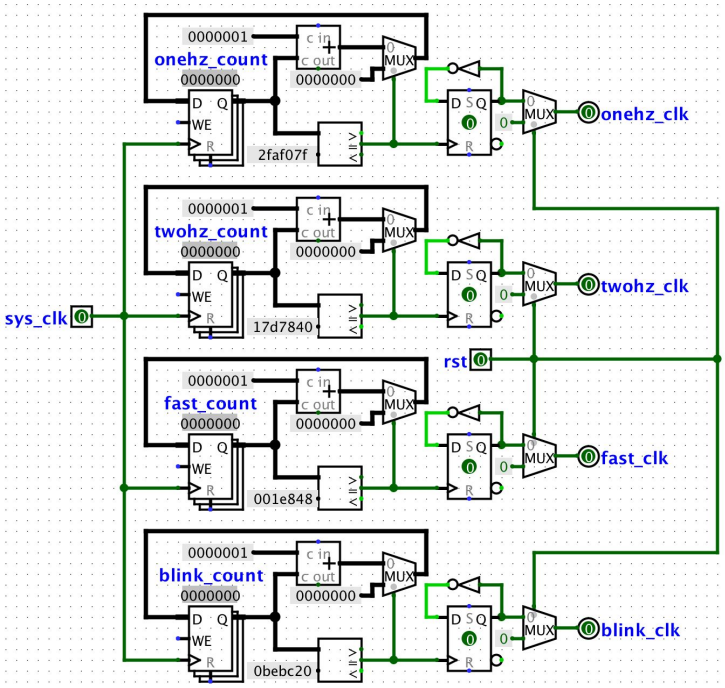

Figure 12: Implementation of clk_div module: Although the module looks complex, the logic for each different signal is nearly identical

The goal of a clock divider is to convert a faster clock to a slower one, and the idea behind it is simple - count up to a certain value and then change a signal, which acts as a slower clock. For each of the signals we needed to use, this is what we did. We used a register to store how far we had counted, adding one to the register every tick of the system clock. When the register reached a certain value (depending on the signal), another register, used to store the state of the clock, would switch, which acted as a clock tick. We wanted each signal to be high for half of its period, and thus would switch the clock twice in every period. Because the system clock was 100 MHz, we counted up to 50 million for the 1 Hz clock, counted up to 25 million for the 2 Hz clock, counted up to 12.5 million for the blink_clk (which blinked at a rate of 4 Hz), and counted up to 250,000 for the fast_clk used to operate the display (which was a 400 Hz clock).

SIMULATION DOCUMENTATION

Unlike with previous labs, this lab required the use of the Nexys 3 Spartan-6 FPGA board, and testing was done both through Xilinx ISE simulation and the physical representation based on the seven-segment display, pressing the buttons, and flipping the switches of the board. Thorough testing was done on each of the sub-modules and the top module to make sure that they each achieved their desired goals. The clock divider and the counter modules were simulated with their respective test benches to make sure that the clocks went high at the correct times, and to test that the counter incremented the time fields (minutes and seconds) and dealt with the edge cases (which will be discussed later on). The other sub-modules were tested by studying the behavior of the FPGA board (e.g. the values were being correctly displayed on the seven-segment display or the reset button being pressed correctly resets both time fields).

First of all, the clock divider module was tested both in simulation and by analyzing the FPGA board's seven-segment display to make sure that the time fields are incrementing at the correct speeds (i.e. 1 Hz, 2 Hz, 4 Hz, and 400 Hz). When in normal mode, the time should be incremented by one value per second (which was seen to be true by watching the board's seven-segment display). Similarly in adjust mode, the time increments two values per second as desired, and the time field that is being adjusted blinks four times per second. Lastly, the frequency for the fast_clk was originally set to 100 Hz. However, this frequency was not high enough as the display could be seen blinking quickly. Therefore, the frequency was increased to 400 Hz, so that it would not appear to be moving to the human eye. As it would take too long to simulate the testing of the clock dividers, below is a waveform depicting what the frequencies of the four clock dividers should like (i.e. the onehz_clk goes high once per second, the twohz_clk goes high twice per second, the blink_clk goes high four times per second, and the fast_clk goes high 400 times per second). However, as it would be hard to fit the fast_clk going high 400 times a second in a single waveform, this is not shown.

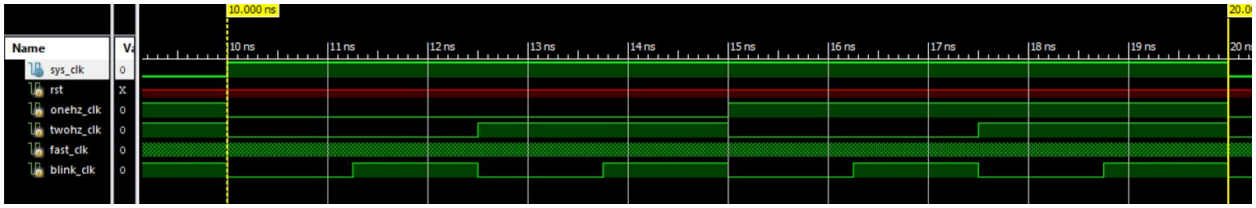

Figure 13: Waveform showing what the four clocks (1 Hz clk, 2 Hz clk, 4 Hz clk, and 400 Hz clk) should like in accordance with the master clock (100 MHz).

The counter.v module was tested initially in simulation. The only job of this module is to increment the time fields of both minutes and seconds at the frequency of the specified clock (i.e. 1 Hz or 2 Hz). The test cases used are listed below:

Test Case	Result	Pass/No Pass
Seconds field incrementing (Normal case)	Correctly increments the one's place of the seconds time field by one	Pass
One's place of seconds (or minutes) field overflow	When the one's place of the time field reaches 9, and is incremented, correctly increments the ten's place, and resets the one's place to 0	Pass
Seconds field overflows to minutes	When the seconds field is at 59, and is incremented, correctly increments minutes field, and resets seconds field.	Pass
Both minutes and seconds field overflows	When both the seconds and minutes are at 59, and is incremented, then reset both fields to 00.	Pass

Figure 14: Table depicting the test cases for the counter module (incrementing of the minutes and seconds time fields).

Below are two waveforms depicting the behavior of two of the counter module edge cases. The edge cases consist of times when overflow of a given time field occurs (i.e. when the seconds time field is at 59, and is incremented, in which the minutes time field should be incremented by one, and the seconds time field should be reset to 00).

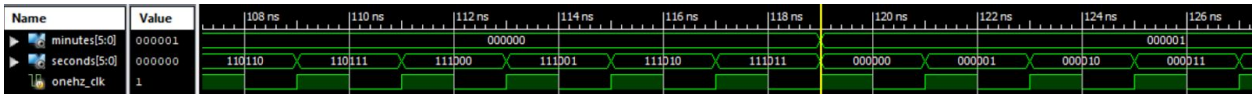

Figure 15: Waveform showing the behavior of the stopwatch when the seconds time field goes from 59 to 00, in which the minutes time field should be incremented from 00 to 01.

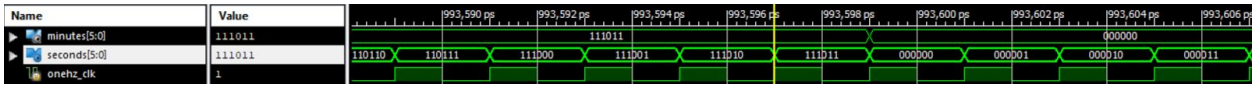

Figure 16: Waveform showing the behavior of the stopwatch when the seconds time field goes from 59 to 00 and the minutes time field goes from 59 to 00, in which both time fields should be reset to 00.

The sel_adj.v module served as a sub-module within the counter module. Before the counter module commenced, the appropriate clock (i.e. 1 Hz or 2 Hz) needed to be selected to handle either the normal mode or the adjust mode. The process of testing whether our program correctly handled the adjust mode, and the proper adjusting of either the minutes or seconds time field was done in simulation. Below are two waveforms: the first depicting the adjusting of the minutes time field, and the second depicting the adjusting of the seconds time field. Note that while one time field is being adjusted, the other is frozen to its current value as desired.

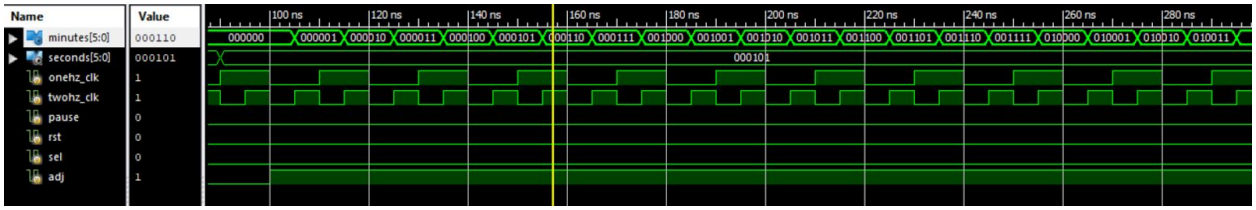

Figure 17: Waveform showing the behavior of the stopwatch when in adjust mode, and when adjusting minutes (i.e. ADJ=1, SEL=0). As desired, the minutes time field is incrementing, while the seconds time field is frozen to the same value.

Figure 18: Waveform showing the behavior of the stopwatch when in adjust mode, and when adjusting seconds (i.e. ADJ=1, SEL=1). As desired, the seconds time field is incrementing, while the seconds time field is frozen to the same value.

The previous waveforms showed proper handling of the two switches. Next, the testing of the buttons (i.e. PAUSE and RST) were tested in simulation as well. As described in the previous sections, when the pause button is pressed, both the minutes and seconds time fields should be frozen at their current values, and when the reset button is pressed, both time fields should be reset to 00. Below are two waveforms showing the proper handling of the buttons: the

first depicts the behavior of the pause button, and the second depicts the behavior of the reset button.

Figure 19: Waveform showing the behavior of the stopwatch when the pause button is pressed (i.e. PAUSE=1). As shown, both the minutes and seconds time fields are frozen to the same values.

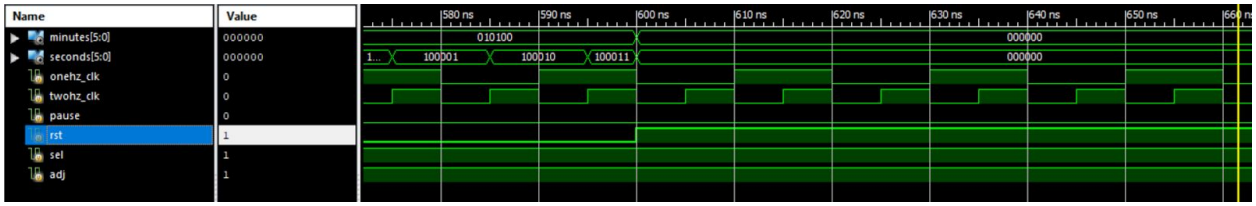

Figure 20: Waveform showing the behavior of the stopwatch when the reset button is pressed (i.e. RST=1). As shown, both the minutes and seconds time fields have their values reset to 00.

Also, these modules were tested to see if they properly had the time fields being incremented in relation to the specified clock. While in normal mode, the stopwatch should be incrementing once per second, and while in adjust mode, the stopwatch should be incrementing twice per second. Below are two waveforms showing that our modules do increment at the correct frequencies: the first waveform shows that in adjust mode, the stopwatch increments twice per second, and the second waveform shows that in normal mode, the stopwatch increments once per second.

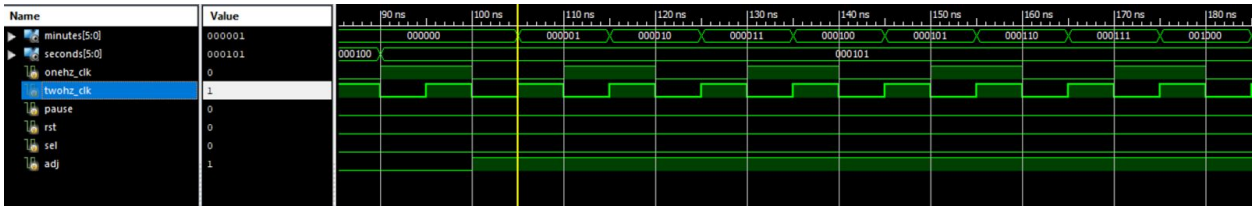

Figure 21: Waveform showing the behavior of the stopwatch in adjust mode. As shown, the minutes time field is incrementing twice per second (synchronized with the 2 Hz clk).

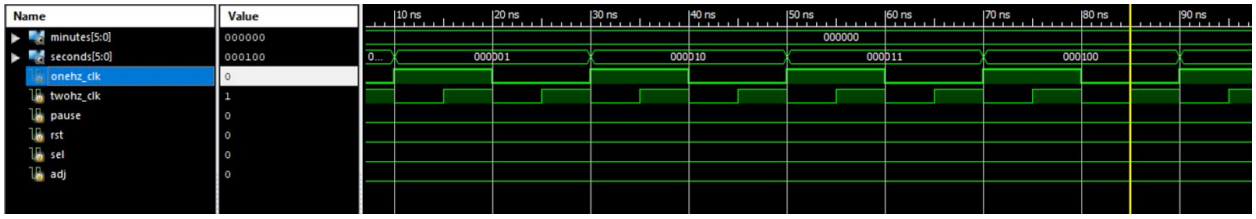

Figure 22: Waveform showing the behavior of the stopwatch in normal mode. As shown, the seconds time field is incrementing once per second (synchronized with the 1 Hz clk).

After making sure that the counter was implemented properly, the correct clock was being selected, and that the switches and buttons were functional, the testing of the

seven-segment display was carried out on the FPGA board. By watching the program in action on the FPGA board, the seven-segment display was adjusted accordingly to make sure that it was displaying the correct values, and that the correct segments were being displayed. For example, at first the segments that were being displayed were inverted, and after seeing this, the segments were corrected by inverting their cathode values.

Lastly, once everything seemed to be fully functional, a simulation of the top module was carried out, and then the board was programmed out to analyze the full run of the stopwatch program. All aspects of the program were tested (i.e. watching time field overflows, pressing the buttons, flipping the switches, and watching the seven-segment display to analyze the four clock frequencies), and it was seen that the stopwatch was fully functional. Below is a waveform of the top module after testing each of the individual sub-modules.

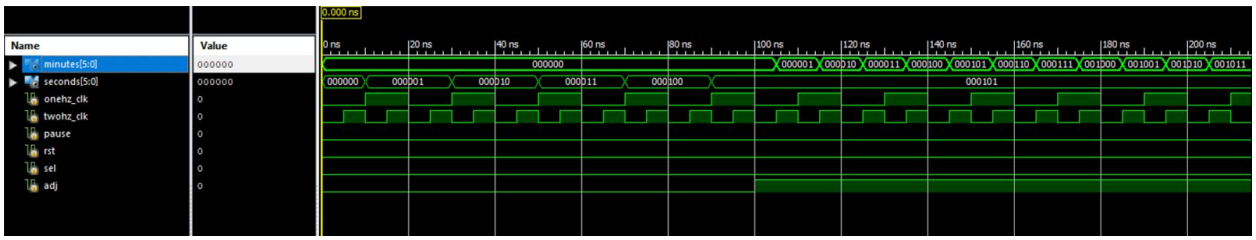

Figure 23: Waveform depicting the final test of our stopwatch program. It was carried out on the top module and shows certain events of the program (i.e. normal mode incrementing and adjusting of the minutes in adjust mode).

CONCLUSION

This lab taught our group how to build a simple stopwatch program using the Nexys 3 Spartan-6 FPGA board. Our program utilized the seven-segment display, as well as two buttons and two switches to implement added functionality to the basic incrementing of the minutes and seconds time fields of the stopwatch. These added features allowed for two basic modes of the stopwatch: normal and adjust mode, and allowed for pause and reset features to take effect. This project consisted of a set of submodules that were used to implement the different functions that went into the overall design of the stopwatch (e.g. a clock divider module to create the four clocks used, and a counter module to increment the two time fields of the stopwatch).

As this was the first lab that involved actual testing on the FPGA board (in contrast to the other labs which were implemented in simulation), this lab came with a certain set of problems that were encountered. First of all, getting the seven-segment display to display the correct segments at the correct times involved some use of trial-and-error (in addition to the simulation testing discussed in previous sections). Moreover, there was a debouncing problem that took place in our design process, as the pause button wasn't fully functional all of the time, and the flipping of the adjust switch led to undefined behavior (i.e. random jump in the stopwatch time value). These debouncing and metastability errors were seen by analyzing the display, and the code was then modified and corrected. Furthermore, a set of testbenches were used to ensure the proper functionality of each of the sub-modules before bringing the entire design together.