clShikha Mody (604922996)

Brendon Ng (304925492)

Armaan Singh (404932331)

COM SCI M152A Lab 5

TA: Logan Kuo

**Lab 3 Report**

**Stopwatch**

**Introduction**

In this lab, we were tasked with creating a stopwatch using the seven segment display, buttons, and switches on the FPGA. The overall functionality is of a traditional stopwatch: counting seconds and minutes (ascending and descending), and being able to pause, reset, and adjust the time on the stopwatch. The output was the seven-segment display to show the time. The inputs included switches and buttons for different modes of the stopwatch. See below for all of the inputs:

* Reset button → set the stopwatch to 00:00 and hold state (e.g. if the stopwatch is paused, it should stay paused; if the stopwatch is running, it should keep running after resetting)
* Pause button → pause the stopwatch at the current time
* CNT\_DN switch → switch from counting up to counting down (when this switch is 1, count down; otherwise, count up)
* ADJ switch → adjustment mode indicated with flashing of the selected portion of the stopwatch (either the minutes section or seconds section)
* SEL switch → In adjustment mode, allows either minutes or seconds to be selected for adjustment (switches between which section is flashing)
* ADJ:B switch → selects the type of adjustment being made: if this switch is 0, the selected parts increment up by two (with wraparound capabilities). If this switch is 1, the selected parts can be incremented using increment/decrement buttons
* INCREMENT and DECREMENT buttons → either increment or decrement the selected portion of the stopwatch by 1 when in adjustment mode. These have wraparound capabilities (from 00:00 to 59:59 and vice versa).

The purpose of this lab was to implement a full Verilog project from start to finish, including designing our modules, seeing how they worked together, and implementing it on the FPGA board. Other portions of this lab that were implied were: debouncing all of the buttons to increase sensitivity, creating multiple clocks with different rates for different purposes, and configuring the seven segment display. In terms of clock rates, we had to implement one 1 Hz clock for the actual stopwatch (i.e. counting up every second), a faster clock for the seven-segment display updating, and a clock between 1-2Hz for flashing the screen in adjustment mode. Overall, this lab allowed us to gain experience with creating and implementing a full project from start to finish.

*This information was adapted from the Lab 3 specification.*

**Design Description**

Our overall design of the lab consists of one module with all of the functionalities embedded within it. In hindsight, we could have made separate modules to separate different functionalities to make our code cleaner and more modularized. Nonetheless, we separated the different functions using different always blocks and achieved the desired functionality. The main components of our design are the clocks, seven segment display, flashing capabilities, input handling, and actual stopwatch logic. See below for descriptions of each of these sections.

*Clocks*

We use a clock divider to divide the original hardware clock into three different clocks with different frequencies for different purposes. We maintain the hardware clock clk as the fastest clock in the program. We run our main always block at every positive edge of this clock. We also have a sec\_clock which is a 2Hz clock, cycling every half second. This clock is used for flashing the display at 2Hz. We also keep a "slow clock" which is actually going quite fast but slower than the hardware clock. This clock is used for the 7 segment display. Our last clock, "less slow clock" is faster than slow clock but slower than the hardware clock and is used as a sampling rate for debouncing buttons. We also implemented two "pulses" to indicate when every second and half-second occur, which flip to high then flip back to low for the next clock cycle. These pulses act as flags for certain sections of code to run. The full\_sec pulse is used to update the seconds variable when the timer is running and the half\_sec pulse is used to increase the selected value by 2 at 2Hz in adjustment mode, as seen in Figure 1.



*Figure 1: How pulses were implemented. Counter went up by one at every hardware clk cycle and was reset to 0 every second (10,000,000) cycles. The pulses are high for only one clk cycle.*

*7 Segment Display*



*Figure 2: State Diagram of the display state machine:*

*Inputs: certain digit of the minutes/seconds*

*Outputs: 'an' specifying which digit to display, 'seg' specifying which segments to light up*

Our implementation of the seven segment display is essentially a state machine (Figure 2). To operate, we specify a four-bit 'an' variable which tells the hardware which digits shall be 'on'. We also specify an 8-bit 'seg' variable which tells the hardware which of the seven segments (plus the period) is to be on. In both of these cases, 0 is on and 1 is off. We use our clock, slower than the hardware clock but fast enough that our human eyes cannot detect the difference of switching from one digit to the next, illuminating them one-by-one with the proper combination of segments to appear as if four digits were illuminated simultaneously. We pass in a numerical value 0-9 for each digit, num0 through num3. When the state machine is in the state for that specific digit, we set 'an' to the proper setting so only that digit is illuminated and set the segment combination to the result of our function that takes in the numerical digit and converts it to the 8 bit segment form of the desired digit. This state machine applies for everything that we displayed on the screen (including adjustment and countdown modes).

*Flashing*

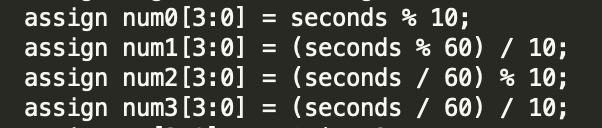
Depending on whether we are in adjustment mode and which section is selected (either minutes or seconds), at every half second cycle, either both minutes digits or both seconds digits are flashed on and off. This always block edited a variable called an2 which serves as a mask to be combined (bitwise or'ed) with the an value coming out of the seven segment display’s state machine to be sent to the actual hardware. By default, not in adjustment mode, an2 is 0000 so when combined (or'ed) with the an from the 7 segment machine, the result retains its value. If we are flashing in adjustment mode, an2 would flip from 0000 to 0011 or from 0000 to 1100 and back. The two 1’s ensure that when an2 is or'ed with the other an code, the two digits that were supposed to be flashing are guaranteed to be off (1's mean off). These bitwise manipulations were helpful to streamline the state machine of the display itself.

*Input Handling*

The primary inputs to this stopwatch are buttons and switches. These affect the state that the stopwatch is in. For instance, the CNT\_DN switch changes the functionality from incrementing the stopwatch by one to decreasing it by one. Similarly, combinations of switches affect if the stopwatch is in adjustment mode or if it is operating normally. To make our stopwatch sensitive to these inputs, we must debounce the buttons. We utilize a similar approach to debouncing the pause, increment, and decrement buttons as we did with Lab 2 (downsampling). We detect a positive edge of each button (when it is pressed) by storing the state of the button (pressed or not) in an array with three slots. If there is a change in state, we set a variable (named ‘is\_[button]\_posedge’ to correspond to each variable). This allows the stopwatch to be more sensitive to single button presses by downsampling with a slower clock signal, as we saw in Lab 2. The switches are more stable than the buttons, so we did not debounce those.

*Stopwatch Logic*

The stopwatch logic is contained in one always block where the variable keeping track of how many seconds are on the timer is maintained and edited. See Figure 3 for how the ‘seconds’ variable is updated in a coordinated manner and shown on the seven segment display. This variable is incremented every time one second elapses (in normal operation), or if extra conditions are satisfied for it to be updated in adjustment mode (i.e. certain switches were active, INCREMENT/DECREMENT buttons were pressed, etc.).



*Figure 3: The method in which the digits of the seven segment display were updated based on an overarching ‘seconds’ variable. This isolated the digits used for ‘seconds’ and the digits used for ‘minutes’*

The large always block constantly samples the hardware clock (fastest) in order to ‘listen’ for button pushes. It is essentially a large if-else statement. In our design, we keep important variables called ‘paused’ and ‘running’. Paused is 1 if we have paused the timer or 0 if the timer is running. The variable called ‘timer’ is assigned to be 1 if we are not paused and seconds is not over the max. It will flip to 0 if we pause the timer or if we reach a maximum value of 59:59. On reset, the variable called ‘seconds’ is set to 0 but ‘paused’ remains the same as it was prior to reset. If reset is pressed while the timer is running, the timer will immediately start counting from 0. We have if statements to catch edge cases like if seconds is the max of 59:59, we pause the timer (only if we are not in adjustment mode). If we are at the maximum 59:59, and we are unpaused, the variable ‘seconds’ is reset to 0 and ‘paused’ is flipped back to 0 and the timer starts counting. If these edge cases are not present, we head into normal operation. If we are not in adjustment mode, we increment or decrement the ‘seconds’ variable by one depending on the status of our CND\_DN switch. We only update on every full\_sec pulse which is high for one clock cycle every second as described earlier in the clocks section, achieving accurate timing. If we are in adjustment/increment at 2Hz mode, we update on every half\_sec pulse and add 2 to the seconds count. We have if statements to handle overflow of both minutes and seconds as well, in order to bring them back to 0 after hitting 59+. In adjustment/button adjustment mode, on the positive edge of our debounced button signals, we either add or subtract one from either minutes or seconds depending on which button is pressed and our SEL switch.

**Simulation Documentation**

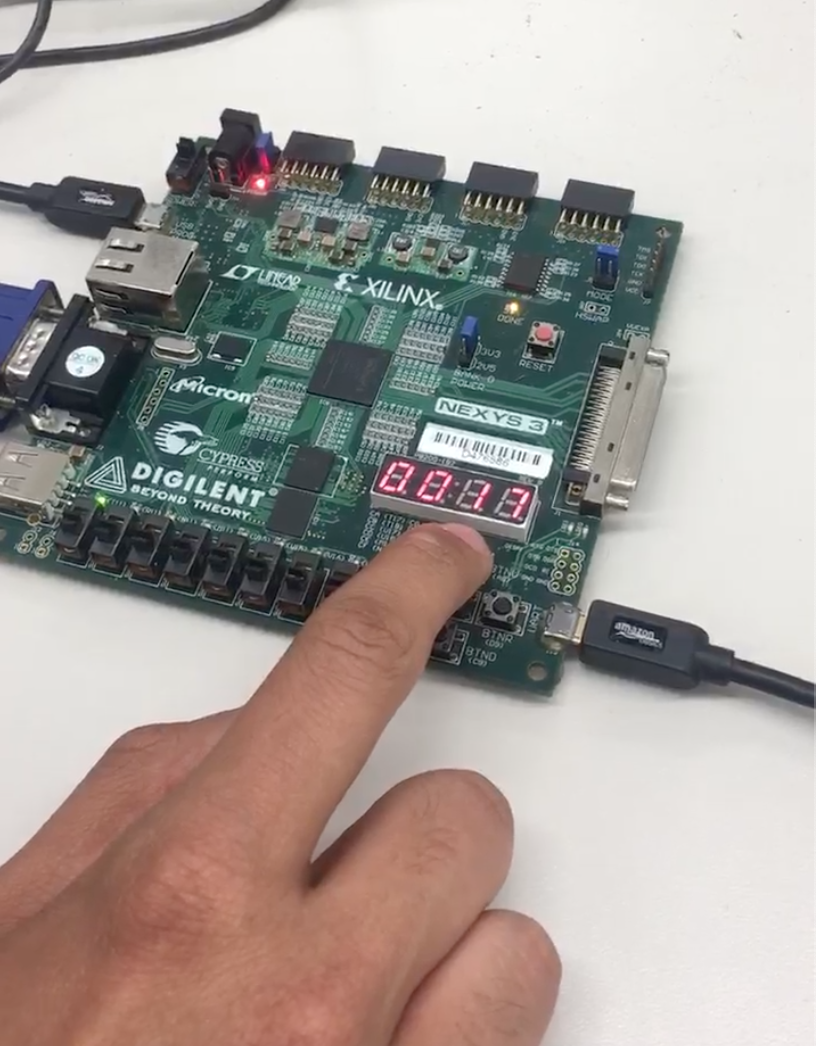
In Lab 3, we did not use much in the way of testbenches and simulation, as the main project was designed around implementation, and many of the clock’s elements did not translate well to simulation, namely the clock speed, display, and the many interactive switches.

Simulations were used in analyzing the transitions of the seven segment display. When implementing the display, we ran into bugs with the same number being displayed on all 4 digits, as well as the segment values being inverted. Using a preliminary testbench we were able to debug these transitions such that the program would, synchronously and quickly, switch which digit it was displaying as well as which segments it should display. This allowed us to show 4 separate numbers such the display would read out “1234”. By analyzing the waveforms, we were able to debug the state machine for the seven-segment display and move on to implementing the other features.

Our approach to design utilized an incremental development method. Essentially we would implement a feature, devise test cases, and test the component before moving onto the next. The test cases we used focused on the edge cases for each feature introduced, while also having some sanity check cases for generic situations that may come up. For normal operation, we checked that the counter moved up as expected at a regular, 1 second interval, that seconds overflowed into minutes correctly, and that the pause and reset buttons worked as expected. We also checked that at 59:59, the clock would pause and not continue. In this state, the functionality of the pause button should resume the stopwatch from 00:00 and continue counting up immediately. Reset, on the other hand, should set the stopwatch to 00:00 and remain paused (hold state). When in countdown mode, the stopwatch should similarly stop at 00:00, with reset setting the clock to 00:00 and pausing. We spent a lot of time debugging and restructuring our code to achieve this functionality because our initial design did not account for the reset button holding the stopwatch’s state. Nonetheless, we succeeded in implementing this.

Beyond normal operation, the various options within adjust mode also needed to be tested. The first adjustment mode involved speeding up the clock speed and incrementing either the minutes or seconds separately by 2 seconds at 2 Hz. When the next increment would go past 59, the output instead should wrap around back to 0 or 1 (depending on whether the previous number was 58 or 59). The second adjustment mode required the selected counter to be incremented or decremented by 1 at the push of a button. This was subject to similar wraparound conditions to the automatic adjustment mode, with incrementing at 59 seconds causing the selected field to be set to 00. Both adjustment modes required flashing, however, as this feature was almost purely aesthetic, the only tests needed were that it did not obscure any increment completely. This way, in between flashes the user is able to see what the current value of the counter is, and knows to stop when necessary. When we tested these features manually, we simultaneously tested other stopwatch functionalities because most of the components built atop one another. For example, to ensure that the stopwatch counted up correctly, the seven segment display also had to be functional. Also, the clock timing had to be correct. When we were testing adjustment mode, we could test these features at the same time: the flashing, incrementing, and pausing/unpausing of the actual stopwatch, since they were all components of being in adjustment mode. Similarly, we could isolate test cases by adjusting the stopwatch to certain edge cases, namely 59:59, and seeing what the implementation would do at those moments. This was how we determined if we correctly implemented wraparound functionality, as well as the correct pause/reset button actions. As a result, this lab allowed us to build on top of what we had already implemented, allowing us to test and use the stopwatch features that we already built in order to implement additional ones.

Much of this debugging was done through the use of auxiliary outputs, such as an extra LEDs on the board, to show the current state of a variable being modified. For example, we set an extraneous LED to match the value of a variable we used to keep track of whether the stopwatch was running or not. This type of test was used to ensure our debouncing algorithm was working properly. As prior to fixing, when the pause button was pushed, the LED indicating the paused state would sometimes flash on and off multiple times, indicating that our button was still somewhat bouncy and would act erratically. In a similar fashion, we used another LED to indicate if the button press was being registered or not; when these LEDs were working in conjunction, we could see how the button press correlated to the stopwatch being in the ‘paused’ state, as seen in the image in Figure 4. Many of our problems came from the debouncing of the buttons, so these LED outputs were very helpful in isolating the issues that we needed to fix.



*Figure 4: Manual testing of our stopwatch, specifically the pause button. As indicated by the LED in the far left being on, the stopwatch is currently in the ‘paused’ state. Similar indicators were used to test other features.*

**Conclusion**

Overall, this lab allowed us to understand the inner workings of a full project in Verilog, from setting up the modules and test bench to implementing it and testing it. We now understand the process of development better too, and how iterating and incrementally testing a module is more effective than testing a large amount of features at once. This way, we are able to isolate specific edge cases and what improvements need to be made for the next iteration. We mainly had difficulties with the pause button because our previous implementation did not take the state into account (i.e. for resetting and pause functionality). This made it difficult to achieve the correct transition at 59:59. Nonetheless, by relentlessly debugging and restructuring our code, we were able to successfully complete the lab. Generally, this lab could be improved by providing some structure about design advice or common problems so that our work time could be more efficient. With recommended design feedback, we could focus on implementing features rather than changing an entire logic block to accommodate the feature. All in all, this lab showed us the method and process to develop an entire Verilog project, which will be helpful for our final lab.