Charles Zhang
9 November 2021
COM SCI M152A, Lab 5

Lab 3: Stopwatch Report

1 Introduction

In this lab, we developed a simple stopwatch on the Nexys 3 FPGA board, utilizing its buttons, switches, and seven-segment displays. Our stopwatch uses a simple decade counter, displaying the current minutes on the two leftmost seven-segment displays and the current seconds on the two rightmost seven-segment displays. The seconds increment upwards at 1Hz, with the minutes incrementing upwards every 60 seconds. Both of these values will increment up to `59`, at which point they will loop back to their initial value of `00`. In addition to a simple counter, the stopwatch also implements an adjustment mode, detailed by the truth table below:

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

Figure 1: `ADJ` mode truth table

In ADJ mode, the normal counter is stopped and the selected value increments upwards two times faster than normal. The selected value is determined by the following truth table:

SEL	Selected
0	Minutes
1	Seconds

Figure 2: `SEL` truth table

Finally, the stopwatch adds functionality to two buttons on the FPGA: `RESET` and `PAUSE`. When `RESET` is pressed, the stopwatch resets to its initial state of 0000, while maintaining the mode it is in. When `PAUSE` is pressed, the stopwatch stops all counting.

In order to implement these input signals from the FPGA board, we have to deal with the issues of debouncing and metastability. Debouncing is required due to the instability of buttons and switches on the board, which lead to considerable noise when pressed or switched. Without any filtering of this noise, it would cause many unintentional input signals to come through, resulting in a jittery implementation. On a similar note, the asynchronous nature of these input signals forces us to deal with metastability in our implementation. This means that we must ensure that input signals are held constant for a period of time before and after a clock edge at which they will be processed. Otherwise, this may create variance in our outputs due to input signals changing when they are not supposed to.

To handle our display, we had to understand the functionality of the seven-segment display on the FPGA. Each seven-segment display is encoded by a cathode value, which tells the board which of the seven segments should be illuminated. Each segment is encoded as follows:

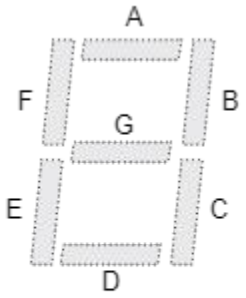

Figure 3: Seven-segment display encoding

Our stopwatch design requires us to encode all four displays on the board, however, the board only provides access to a single display at a time, determined by the anode value it is passed. As a result, our seven-segment displays have to be encoded one at a time, quickly changing the cathode and anode values so that the display looks consistent to the human eye.

2 Design Description

We decided to model the overall design of our stopwatch on the following finite state machine:

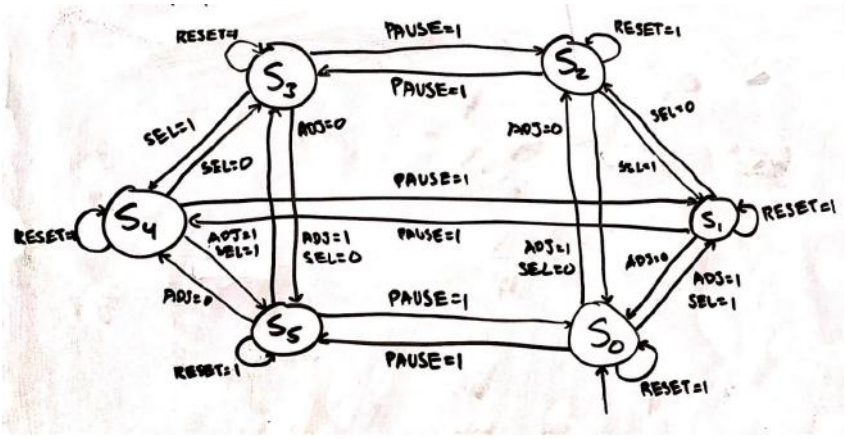

Figure 4: FSM diagram for the stopwatch

Here, state S_0 is the initial/default state, where the counter ticks up at a rate of 1Hz. From this state, we can go to state S_1 is `ADJ = 1` and `SEL = 1`, state S_2 if `ADJ = 1` and `SEL = 0`, or state S_5 if `PAUSE = 1`. State S_1 is the seconds adjustment mode, where the minutes are frozen, the seconds increment at a rate of 2Hz, and the seconds display is blinking at a rate of 1.5Hz. From this state, we can go to state S_0 by setting `ADJ = 0`, state S_2 by setting `SEL = 0`, or state S_4 by setting `PAUSE = 1`. State S_2 is the minutes adjustment mode, where the seconds are frozen, the minutes increment at a rate of 2Hz, and the minutes display is blinking at a rate of 1.5Hz. From this state, we can go to state S_0 by setting `ADJ = 0`, state S_1 by setting `SEL = 0`, or state S_3 by setting `PAUSE = 1`. States S_3, S_4, and S_5 are the paused versions of states S_2, S_1, and S_0, respectively. Each of these states can return to their unpaused versions by setting `PAUSE = 1`. They can also transition between each of the paused states with the same transitions described above for their unpaused counterparts.

Our implementation starts with our top module, `stopwatch`, which takes in the input signals `clk`, `ADJ`, `SEL`, `PAUSE`, and `RESET`, and outputs `cathode` and `anode`. This module implements other modules to filter through these input signals, initialize our counter, and translate our counter to the correct cathode and anode values.

The first module, `clock_divider`, is responsible for splitting the 100MHz master clock signal from the FPGA into four signals: `clk_1Hz`, `clk_2Hz`, `clk_fst`, and `clk_blnk`. It does so by taking in the master clock, `clk`, as well as the reset signal, `rst`. We then simply increment 4 separate counters on every clock tick, each of which is checked against predetermined values. If one of these counters is equal to its corresponding predetermined value, that signal is then flipped, creating a clock signal of a controlled period. Each of these 4 signals is then output from this module for other modules to use.

The next module, `debouncer`, is responsible for debouncing and stabilizing any input signal using the master clock. It takes in a clock signal `clk` and an input signal `signal_i`, and outputs a debounced signal `signal_f`. This module deals with metastability by first storing the input signal into a register, `store`. The rest of the module then reads from this register at the posedge of `clk`, synchronizing any asynchronous input with the master clock. This signal is then compared to the current signal of the stopwatch. If it has changed, a counter is started, incrementing while the signal is constant. If the signal falters, the counter is reset. This way, an incoming signal must be constant for 1ms to be considered valid. Once the counter has counted to 1ms, the new signal is then passed to the rest of the module through `signal_f`. This module is called for each of the input signals, `ADJ`, `SEL`, `RESET`, and `PAUSE`, outputting new signals, `adj`, `sel`, `reset`, and `pause`.

Afterwards is the module `counter`. This module takes in the inputs `clk_1hz`, `clk_2hz`, `sel`, `adj`, `rst`, and `pause`, and outputs `minutes_tens`, `minutes_ones`, `seconds_tens`, and `seconds_ones`. This module starts by checking if the `pause` signal is high, in which case it will flip the bit of `is_paused`, which tells the rest of the module if the stopwatch is paused. Then, at the posedge of either of the clock signals or `rst`, this module will begin counting. If `rst` is high, all counters will be reset to `0`, effectively resetting the stopwatch. If `adj` is high, that means adjustment mode is on. From here, the code checks if `sel` is high or low, and increments the minutes or seconds accordingly at the posedge of `clk_2hz`. If `adj` is low, that means the stopwatch is in normal mode. In this mode, at the posedge of `clk_1hz`, the one's place of the seconds value is incremented by one. If that value has reached its maximum (`9`), it will then loop back to `0` and increment the ten’s place by one. If that value has reached its maximum (`5`), it will then loop back to `0` and increment the one’s place of the minute value by

one. The same pattern is repeated for the ten's place of the minute's value. This process gives each of the four digits of the stopwatch. Each of these digits' is then output from the module to be translated.

The following module, `get_cathode`, is responsible for this translation. It takes in a value `display_state` and outputs a 7-bit value `cathode`. This module simply uses a case statement to map each of the 10 possible digits to a cathode value that corresponds to that digit's display on the seven-segment display. This module is used on each of the digits output by `counter`, translating them for use by the following module.

The final module, `create_display`, is responsible for generating the seven-segment display. This module takes in the clock signals `clk_fst` and `clk_blnk`, the input signals `sel` and `adj`, and the cathode values from `get_cathode`. It outputs a single `cathode` and `anode` value. Using the `clk_fst` signal, this module flips through each cathode value, outputting it along with the anode value that corresponds to its digit placement. By rotating through the 4 digits rapidly, the display on the board appears correct to the human eye. In addition, this module checks if `clk_blnk` is high, in which case it outputs a cathode value corresponding to an empty display, creating the illusion that the display is blinking during adjustment mode.

3 Simulation Documentation

To test our stopwatch, we developed a testbench that was designed to test each of the possible modes. It starts in normal mode, then changes to minute adjustment mode, then to seconds adjustment mode, then back to normal mode, then paused mode, and finally, is reset. In addition, to make testing faster, we doubled the frequency of the master clock, such that half a second in the simulation corresponds to one second in real-time.

The following waveforms show our stopwatch in normal mode:

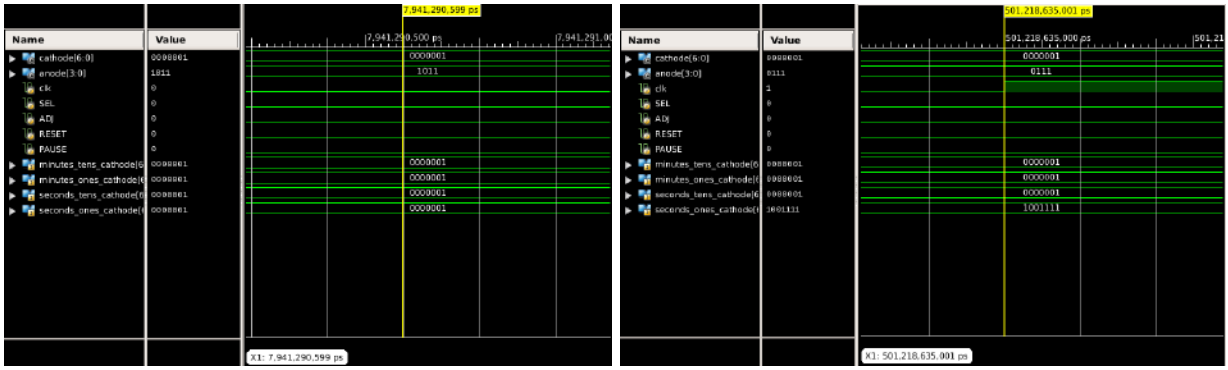

Figure 5: Before and after waveform diagrams in normal mode

Here, we can see that `SEL`, `ADJ`, and `PAUSE` are all `0`, indicating our stopwatch is in normal mode. Initially, we note that all of the cathode values are `0000001`, which is the cathode value corresponding to `0` on the seven-segment display. After the first posedge of `clk_1hz` is encountered, the stopwatch then updates so that the one's place of the seconds value is now `1001111`, which is the cathode value corresponding to `1`. This tells us that our normal mode is working initially, as it has incremented the seconds at a 1Hz frequency.

Next, we can look at the waveforms in minute adjustment mode:

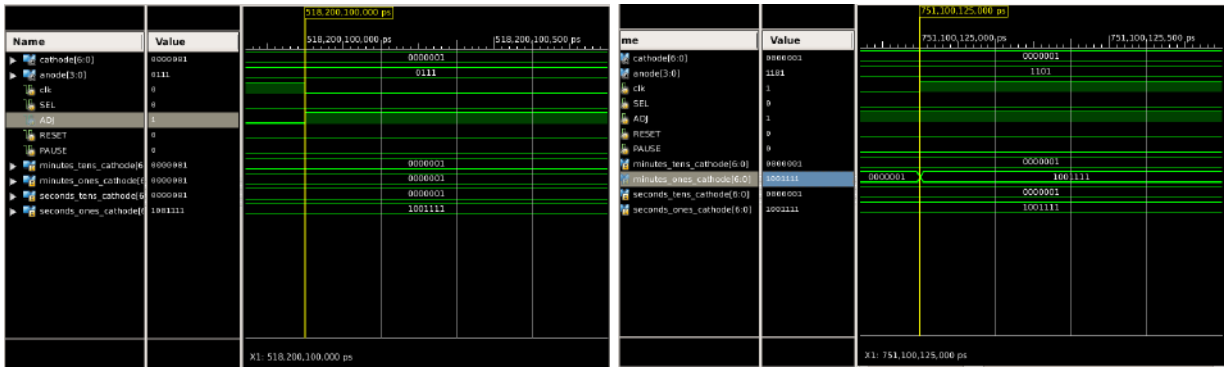

Figure 6: Before and after waveform diagrams in minute adjustment mode

In these diagrams, we can see that `ADJ = 1` and `SEL = 0`, which means our stopwatch should be in minute adjustment mode. Initially, our stopwatch is at the equivalent of 00:01, as we have spent a second in normal mode. After half a second, the one's place of the minute value has now changed to the cathode value corresponding to `1`. This tells us that the minute adjustment mode increments the minutes value at a rate of 2Hz, and is working as intended.

Following that, we can look at the seconds adjustment mode:

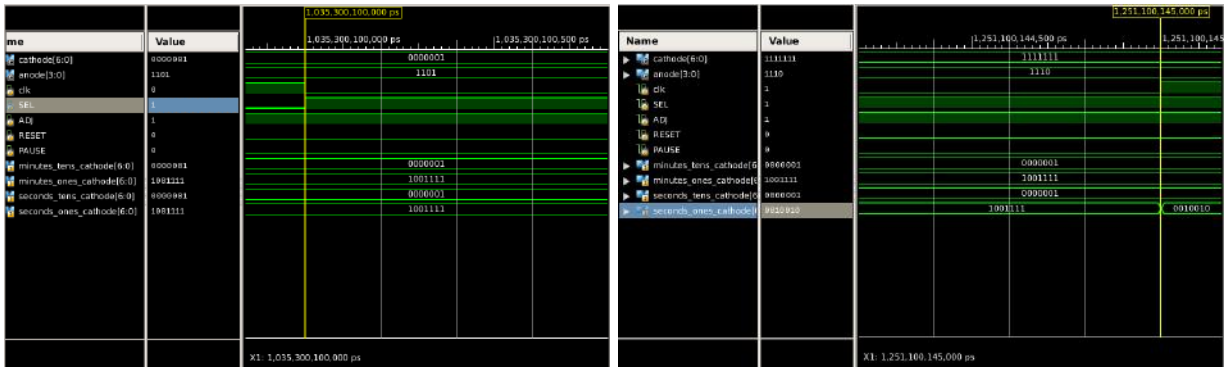

Figure 7: Before and after waveform diagrams in second adjustment mode

Here we can see that when `SEL` was initially toggled to high, the stopwatch would have been displaying 01:01. After another half second, the one's place of the seconds value has now been updated to the cathode value corresponding to `2`. This confirms that adjustment mode works for both seconds and minutes, updating the selected value at a rate of 2Hz, and pausing the unselected value.

After adjustment mode, we then set the stopwatch back to normal mode and find the waveforms shown below:

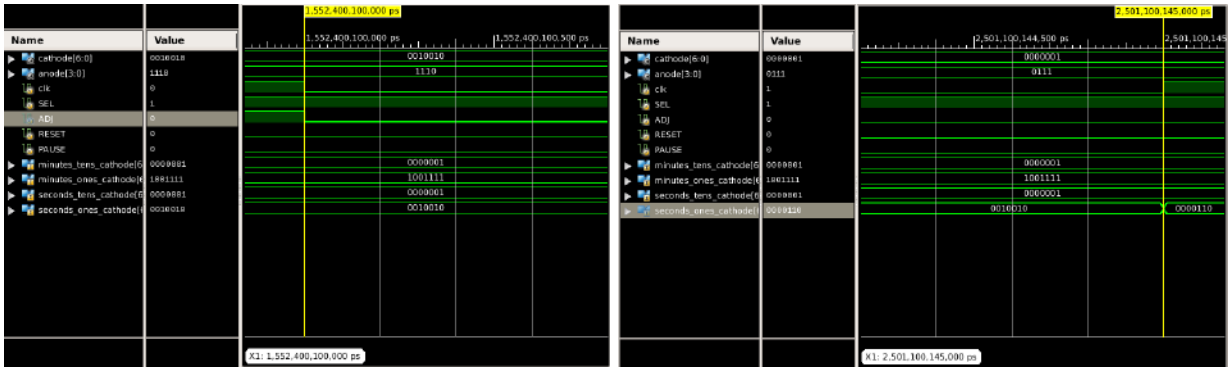

Figure 8: Before and after waveform diagrams in normal mode, after changing back from adjustment mode

Like before, we can confirm the stopwatch is in normal mode, as `ADJ` is once again `0`. Once again, after a full second has passed, the one's digit of the seconds value changes, incrementing to the cathode value corresponding to `3`, confirming normal mode is fully functional.

Next, we test the pausing functionality of the stopwatch, which produces the following results:

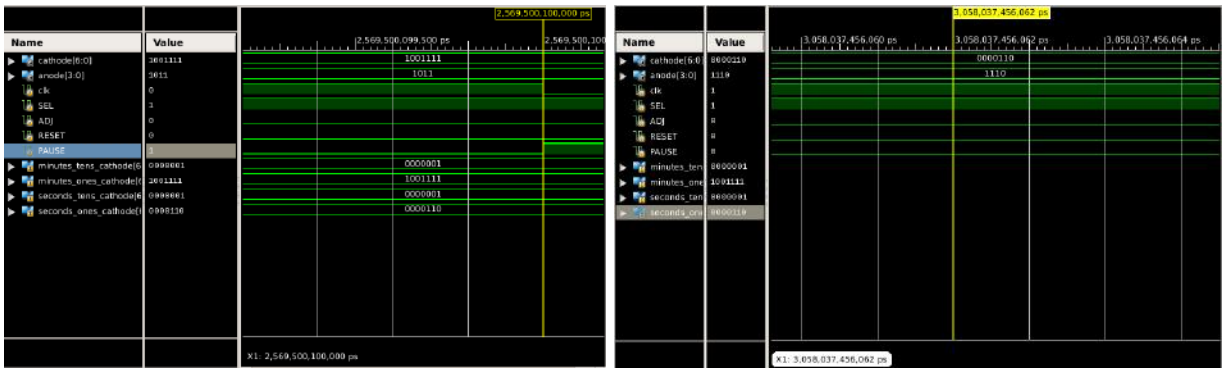

Figure 9: Before and after waveform diagrams in paused mode

We note that the stopwatch is in paused mode, as `PAUSE` was set to `1`. From the waveforms, we can see that, despite a full second passing, the values of the cathodes remain the same, showing us that the stopwatch is indeed paused. Since the pause functionality in our code is implemented in the same way, regardless of state, we can assume from these results that transitions to and from all paused states work correctly as well.

Finally, we test the reset functionality of our stopwatch:

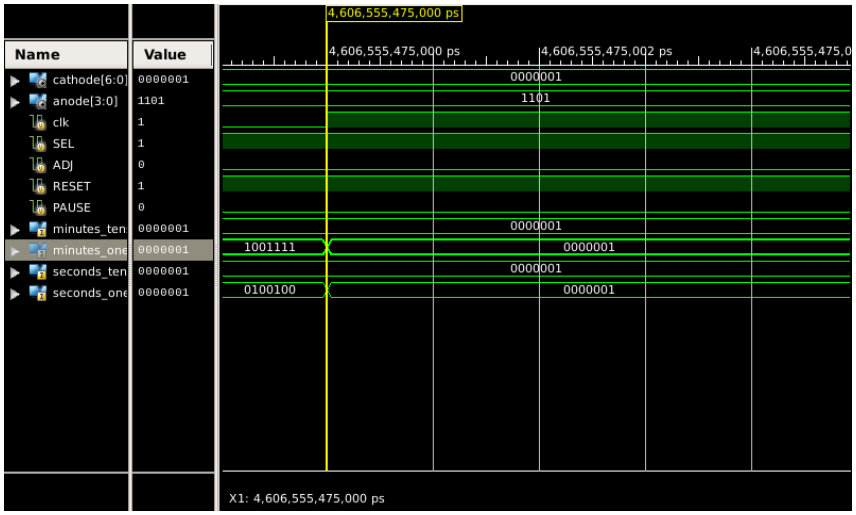

Figure 10: Waveform diagram after reset

We can see that once `RESET` has been set to high, all of the cathode values are reset to their initial values, confirming that reset works properly.

The overall output of our tests can be seen in the following console logs, where the cathode values of each anode are printed at various breakpoints:

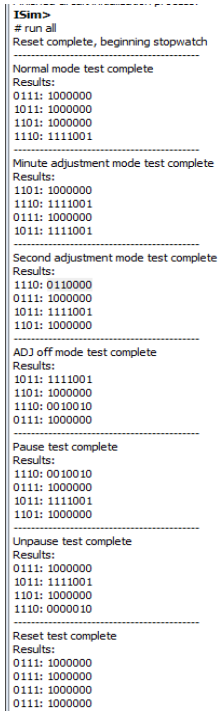

Figure 11: Debugging output for `stopwatch_tb.v`

4 Conclusion

In this lab, we developed a stopwatch that could be controlled with switches and buttons on the Nexys 3 FPGA board, and was displayed on a seven-segment display. Our design incorporated submodules that built up our top module, `stopwatch`. `clock_divider` was used to transform the master clock into a set of clock signals that were needed for the functionality of the counter and display. `debouncer` was used to debounce and ensure metastability on our input signals. `counter` was used to keep track of the current minutes and seconds on the stopwatch, as well as handling adjustment modes and pausing. Finally, `get_cathode` and `create_display` were used to translate our counter into the input to the seven-segment display.

The biggest difficulties we encountered involved the translation of the counter into components that could be recognized by the seven-segment display. We solved this through trial-and-error, programming the FPGA, and seeing the results. Over time, we ended up being able to determine the cathode values we needed for the project. In addition, we had trouble figuring out how to rotate through the anodes to update all four digits, but eventually discovered the brute-force implementation our final design uses.

Overall, I think we both contributed equally to the lab, although I focused on the testing of our program and the programming of the board.