Members: Rahul Mallick
Eric Choi
Sahiti Gabrani

CS M152A Lab 3

Introduction and Requirement

The aim of this lab assignment is to design a stopwatch circuit and implement it on the Nexy3 FPGA Board. This lab provides an opportunity to apply the FPGA design flow and review the design techniques learned in previous labs. The stopwatch will count minutes and seconds, and its functionality will be controlled by buttons and slider switches. The digits of the stopwatch will be displayed using the on-board seven-segment display. The left two digits of the seven-segment displays will count minutes, and the two on the right will count seconds.

The time stored in the stopwatch is adjustable by the use of two input switches on the FPGA Board. The first input switch is the adjust switch that toggles between regular stopwatch mode vs. adjustment mode. In the regular stopwatch mode, the stopwatch display goes up by 1 every second thanks to the 1 hertz clock we set up. On the other hand, adjust mode enables the user to modify the stored time by incrementing either the minutes or seconds fields twice per second using a 2 Hz clock. The stopwatch initiates in normal mode, but the mode can be toggled by utilizing an adjust switch (ADJ). However, toggling the adjust switch to “on” mode will allow it to enter into the adjust mode as described earlier. We can return back to the normal stopwatch mode by switching back the adjust switch to “off” mode. The table for the adjustment mode is given below:

Adjust Switch (ADJ)	Function
0	Stopwatch behaves normally
1	Stopwatch is in adjust mode

To determine which field of the stopwatch is adjustable, the second input switch which is the select switch (SEL) is employed. When the select input is set to logic high, the seconds field enters adjust mode, while the minutes field remains frozen at its current value. Conversely, when the select input is set to logic low, the minutes field enters adjust mode, while the seconds field remains unaffected. During adjust mode, the unselected portion of the clock remains static while the selected portion incrementally increases at a rate of 2 ticks per second (2 Hz). Notably, the selected portion of the clock should blink while in adjust mode, providing visual feedback to the user. The table for the Select Mode (when the adjust switch is 1) is given below:

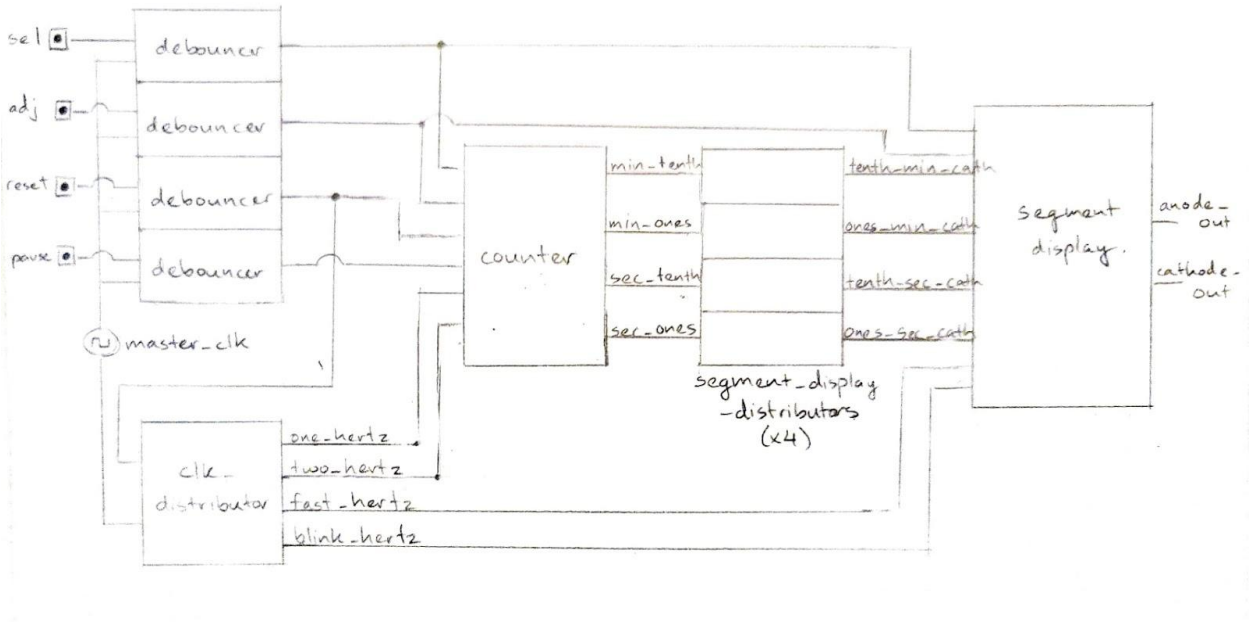
Members: Rahul Mallick
Eric Choi
Sahiti Gabrani

Select Switch (SEL)	Selected
0	Minutes
1	Seconds

Finally, two buttons that we had to implement were the reset and pause modes. For the first feature, reset, the stopwatch had to reset its counter back to 0 (or rather, 00:00 on the display). Immediately afterwards, however, the stopwatch should continue counting. The second feature in the stopwatch is Pause which will pause the current value of the stopwatch to its current time. When the button is pressed again, it will continue the counter where it left off got the Pause feature.

Design Description

In order to fully implement all the stopwatch's features and functionality, we collectively agreed that modularizing the design will not only allow us to organize our thoughts but also allow us to unit test more effectively. The main modules that we implemented were modules that addressed debouncing, distribution of various clocks, core stopwatch counter, and the seven-segment display. These modules effectively exchange crucial information with one another, enabling users to simply choose one of the four provided inputs and easily influence the stopwatch's behavior.



Members: Rahul Mallick
Eric Choi
Sahiti Gabrani

The "engine" of the whole stopwatch is the clock module, itself. The clock module essentially distributes the various clock cycles that the stopwatch utilizes for various functionalities and aesthetics. On the user-end, the debouncer modules confirm that certain input signals are not false triggers or noise, allowing the stopwatch to be more robust and dependable to the user. These input signals that are filtered to be "true" are then sent to both the count and seven-segment display module. The count module receives the four inputs and two clocks in order to accurately count the time and adjust the timer as necessary depending on the user input. The stopwatch time is outputted from the count module and inputted into the seven-segment display module, where it finds the appropriate cathode and anode values to display the right values.

The specifics of each module are described below.

I. Clock Module ("clk_distributor.v")

As stated previously, the clock module can be considered the "engine" of the whole stopwatch. The stopwatch requires four different clocks - a 1 Hz clock, a 2 Hz clock, a faster clock (50 - 70 Hz), and a blinking clock (>1 Hz); all these clocks derive from the 100 MHz master clock, which is internally connected to the FPGA board's V10 pin. These four clocks address various functionalities and aesthetics that the stopwatch needs. For instance, the faster clock is used for the seven-segment displays while the blinking clock is used when the user is on adjust mode. Thus, the clock module's objective is to distribute the faster master clock and transform the fast master clock to slower clocks. For the faster clock, we decided to use 500 Hz and for the blinking clock, we used 5 Hz; these values were derived by initially making educated estimations of the Hz values and subsequently fine-tuning them through iterative testing rounds as necessary.

In order to implement the clock module, it was simply a matter of keeping track of the master clock cycle and flipping the clocks from low to high or high to low whenever the clocks reached their Hz threshold. For example, for a 2 Hz clock, a separate variable keeps track of its count, gets incremented every master clock cycle, and flips the 2 Hz clock every time the variable reaches 25000000; whenever the reset button is pressed by the user, all the clock counts are set to 0, effectively restarting the clocks all over again.

II. Debouncer ("debouncer.v")

The purpose of the debouncer module is to get rid of any noise due to our project being a physical one. There were two areas of concern that we had to tackle: the stability of the input signal around clock edges and the “bounciness” of the button presses. To handle the first problem, we had two single-bit registers called sync_1 and sync_2 where we passed the input into. We first passed the input signal into sync_1 at every posedge. At the next pos_edge, we set sync_2 equal to sync_1. The logic here is that if the input signal was unstable around clock edges, we don’t register that as an intentional button press and conclude

it is some outside noise. Hence, we wait for at most 2 clock cycles to read the actual input of the button press. After this initial part, we then passed the input signal into the second half of the module, which was responsible for ensuring that the signal was an intentional button press. We had a register that counted up to 0xffff, incrementing after every clock cycle. The logic here is that if after these 0xffff clock cycles, the input button value is still the same, then we can mark this as an intentional button press and output this button press as so. Otherwise, we chalk it up to some other noise in the system.

III. Counter Module ("counter.v")

In essence, the counter module is where all the essential stopwatch logic and functionalities are implemented. The basic goal of the counter module is to increment the second and minute every rising edge of the clock (and not exceed 59 for both the minute or second count); however, the various user inputs must be considered as well, making the counter module the most complex module in the stopwatch in terms of logic and design.

In order to implement the counter module, we initialized four variables - minutes, seconds, clock, and pause_tmp. Everytime the user inputs the pause signal, the pause_tmp variable flips from its saved state and everytime the user inputs the adj signal, the program assigns either the 1 Hz clock or 2 Hz clock depending on whether or not the adj signal is high. For every rising edge of the clock, if no user signals are sent, the counter increments the "seconds" variable until the second hits 60, where the counter then increments the "minutes" variable by 1; when the "minutes" variable hits 60, both the "minutes" and "seconds" values are set back to 0.

When the user sends the "reset" signal, both the "minutes" and "seconds" values are promptly reset to 0. On the other hand, if the "adj" signal is received, the counter assesses whether the "sel" signal is set to high, enabling it to accurately determine whether adjustments should be made to the "minutes" or "seconds" value.

At the end of the module, the "minutes" and "seconds" values are redistributed by their respective tenths and ones place values; the counter module does so in order to send the digit values to the seven-segment displays.

IV. Seven-Segment Display Module ("segment_display.v" & "segment_display_distributor.v")

We modularized the seven-segment display module with two Verilog modules called "segment_display.v" and "segment_display_distributor.v". The outputs of the clock module are the digits of the minute and second values, which are then directly inputted into the segment_display_distributor module. The purpose of the segment_display_distributor module is to receive the digit value (0-9 in decimal) and output the representative cathode value that corresponds to it. Since this process is done four times each time the

clock increments, we decided that the code would be the neatest if we established this process as a separate module. In terms of translating from digit value to cathode value, we simply utilized the provided documentation and translated the digit values to the correct cathode values.
Lastly, these representative cathode values are then directly sent to the segment_display module, where it displays the digits at the right place and adjusts the display whenever the user presses the select and/or adjust button. In terms of placing the cathode values in the right digit placeholder, the module simply needs to iterate through the four digit spots and specify the digit placeholder with an anode value. Whenever the adj signal is activated, the display has to actively "blink" as well, meaning the module has to flash the cathode value every two clock cycles rather than every clock cycle; depending on the "sel" signal, the module evaluates whether or not the minute placeholders or the second placeholders have to be flashed. These anode and cathode values are then outputted, ultimately providing an output for the whole stopwatch, itself.

V. User Constraint Files ("constraints.ucf")

The stopwatch has to give explicit instructions to the FPGA to direct the mapping, placement, and timing of inputs and outputs. Using the representative LOC values, the various inputs (switches and buttons), the output (display), and the master clock were mapped within the file. This allowed the stopwatch to ultimately implement our design and function as a typical stopwatch.

Simulation Documentation

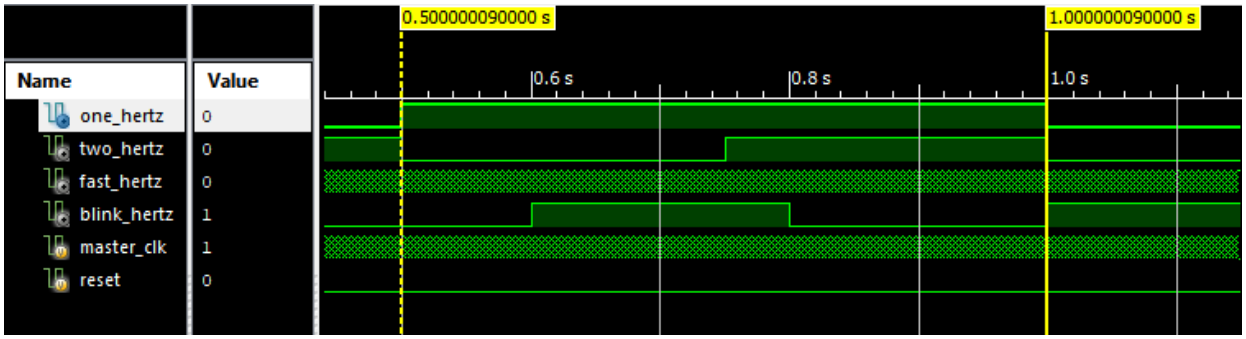
For this lab, we had to use both the Xilinx ISE Simulation as well as the Nexys 3 FPGA board to test the various submodules of our program. In particular, we created test benches for counter.v and clk_distributor.v, whereas we tested the button and display setup on the FPGA board. For the testbenches, we observed the waveforms to ensure that the various clocks' posedges were appearing at the right time as well as ensuring that the correct values of minutes and seconds were being printed. And with regards to the FPGA board, we ran tests involving turning on/off adjustment and selection mode, pressing the pause/reset buttons, and displaying correct numbers/values on the 7-segment display. We made sure to properly test each submodule to ensure that the desired behavior was exhibited. However, in this section, we will focus on the simulation efforts for the clk_distibutor.v files and count.v files.

For the clk_distributor submodule, we have four clocks that we derived from the master_clk: one_hertz, two_hertz, fast_hertz, and blink_hertz. The one_hertz clock increments the stopwatch 1 value per second when it is in normal mode. Similarly, the two_hertz clock increments the stopwatch 2 values per second when in adjustment mode. The blink_hertz (5 hz) is meant to turn the display on and off 5 times per second to indicate that the stopwatch is in adjustment mode. Finally fast_hertz (500 hz) is meant to display the stopwatch in normal mode such that the human eye cannot discern that the screen is blinking. We initially used a value of 250 hz, but we found it to be too slow, in which case we bumped it to 500 hz.

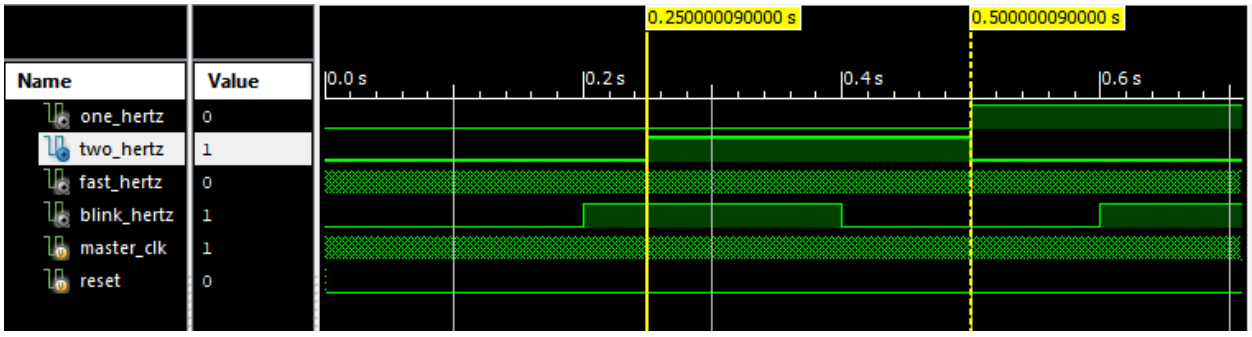
Members: Rahul Mallick
Eric Choi
Sahiti Gabrani

Below, we have screenshots of the various clocks that we have as well as the time-intervals shown to indicate that we have correctly implemented the clock signals.

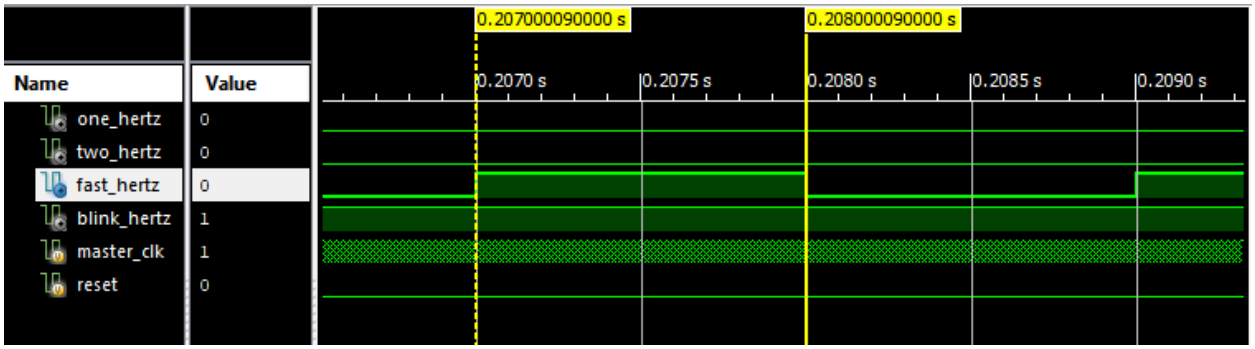
one_hertz -



two_hertz -

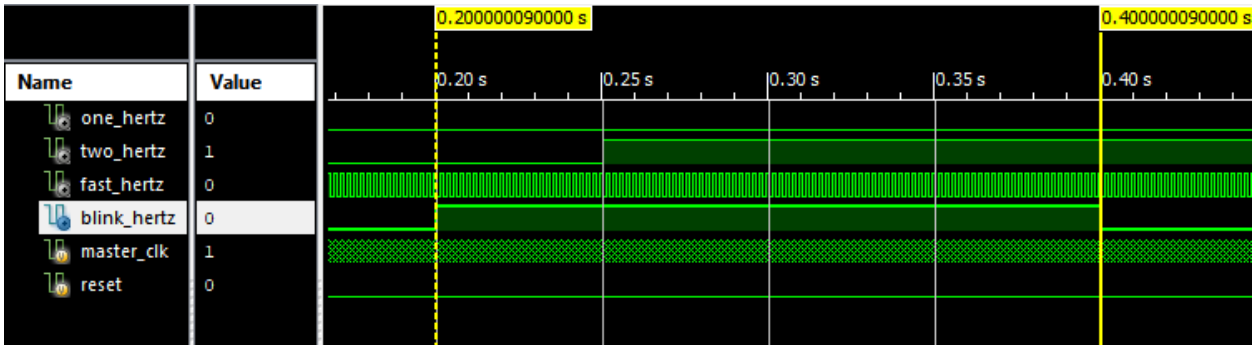


fast_hertz -



blink_hertz -

Members: Rahul Mallick
Eric Choi
Sahiti Gabrani



For the counter.v submodule, whose only job is to increment the seconds/minutes of the stopwatch, we created a testbench so that we could observe the behavior of the clock in the following cases:

Test-Case	Explanation
Counting between 0-9 in **ones-digit of seconds**	Normal incrementing for every second. Once it hits 9, circle back to 0.
Counting between 0-9 in **tens-digit of seconds**	At the moment the ones-digit of seconds hits 9, increment the tens-digit of seconds
Counting between 0-9 in **ones-digit of minutes**	At the moment the value of the seconds section hits 59, increment the ones-digit of minutes.
Counting between 0-9 in **tens-digit of minutes**	At the moment the ones-digit of minutes hits 9, increment the tens-digit of minutes.

We made sure we fulfilled these test cases by examining the clock signals, and checking to make sure our values are expected. For example, we checked to make sure the register that holds sec_tenth increments when the register for sec_ones hits 9. The same applied for the other digits. Also done in our simulation was testing to make sure that the counter gets incremented twice as fast (depending on whether sel is set to 0 - seconds, or 1 - minutes) when the adj (adjustment mode) is set, in which we took a look at the interval of each increment using the Xilinx ISE Simulation software. Finally, we tested the pause and reset buttons in simulation to make sure we get the same behavior as outlined in the spec.

Unfortunately, we were unable to take a screenshot of these simulation efforts as we were preoccupied with a litany of bugs when testing our counter. Namely, we observed that our sec_ones and sec_tenth were incrementing properly, but our min_ones was incrementing at the wrong time. In turn, this led to our min_tenth value to also be incremented incorrectly. After more than an hour of debugging, we realized that our issue stemmed from incorrectly incrementing min_ones using a temporary variable meant for a different digit. However, this was only part of the problem. We were using temporary variables called minutes and seconds to do the actual counting, and then setting min_ones, min_tenth, sec_ones, and

sec_tenth to their respective values. We had calculated the min_tenth and min_ones values incorrectly, in turn resulting in the wrong values being outputted in the waveform display.

After fixing these bugs, we then went on to work on debouncer.v, segment_display.v, and segment_display_distributer.v. These files required the Nexys 3 FPGA board to test, in which we checked to make sure we were getting the right results after creating each submodule was completed (i.e. ensuring we were displaying the correct numbers for the correct digits place). There would be times when we would have the incorrect segments being turned on due to reading the documentation incorrectly, in which we would go back and fix them to the right segments.

Lastly, we wrapped all the submodules' logic together in the master.v file, and tested to make sure that our stopwatch worked as intended. We realized that in our first couple run-throughs, the stopwatch functionality was working, but the pause button would not function properly. It would pause at only a single digit while the pause button was held, but increment normally after the button was released. We had to change our logic around for the counter.v file to ensure that the pause functionality was working properly. After these final fixes, we were complete with the project.

Conclusion

For this lab, we made use of the seven segment display to show the correct numbers, switches for on/off adjustment and selection mode, and buttons for pause/reset. We had several modules to successfully develop the stopwatch namely the Clock Module, Debouncer, Counter Module, Seven Segment Display and user constraint files. A Clock module was used to create the four clocks used, and a Counter module to increment the two time fields of the stopwatch and the logic, the Debouncer module to get rid of noise and the Seven Segment Display module to display the digits of the stopwatch. We took care of mapping the master clock and the inputs (buttons and slider switches) and outputs (display) in the user constraint files. The lab assignment provided valuable hands-on experience with the complete FPGA design flow and reinforced the understanding of digital design techniques. We encountered several bugs while developing and testing our stopwatch, including an issue with our logic: our min_ones was incrementing at the wrong time and it involved a considerable amount of time spent debugging to fix the issue. While testing the stopwatch, we realized we did not get the pause button to work as it was supposed to. However, all difficulties encountered during the design process were overcome through careful analysis and problem-solving. Overall, this lab assignment served as an effective learning opportunity and helped improve skills in FPGA design and implementation.