CS M152A Lab 3

Michael Wu, Haoyu Yun, Jennie Zheng

Introduction and Requirement

We built a stopwatch which displays minutes (two decimal digits) and seconds (two decimal digits).

The stopwatch may be reset or paused using two buttons.

Flipping a switch enters the device into adjustment mode (**ADJ**), which allows users to increment individual digits of the stopwatch. The digit may be specified with two select switches (**SEL**) that represent a binary number, with the lowest order digit indexed as zero and the highest order digit indexed as three.

After flipping the switch high to enter adjustment mode, the digit specified with **SEL** automatically increments at 2 Hz and the other digits are frozen until adjustment mode is exited upon flipping the switch low. While in adjustment mode, the incrementing digit blinks at a rate of 5 Hz.

To deal with debouncing, we utilize a clock with a lower frequency than the master clock to sample inputs from the pause button.

Design Description

To determine the digits to display, we made use of a modulo-10 counter for the ones digit of the minutes and seconds, and a modulo-6 counter for the tens digit of the minutes and seconds.

Clocks

We used 4 clocks to deal with 4 different behaviors of the stopwatch.
A 10 Hz clock controls blinking in the “adjust” mode. Blinking down samples this clock to 5 Hz.
A 2 Hz clock reflects automated incrementation of the specified digit in the “adjust” mode.
A 1 Hz clock is used to increment the stopwatch every second.
A 500 Hz clock is used to cycle through displaying the digits on the 7-segment display.

To generate the signals of the four clocks, we use a **clkModule** with the main **clk** as input and **clk_1**, **clk_2**, **clk_10**, and **clk_500** as outputs. At each rise of the main **clk**, a counter is incremented for **clk_500** through the **clk500** module. Then, at each rise of **clk_500**, counters for the other three clocks are incremented, and their signal is set to high for a single cycle when their counters reach specified targets.

This design decision to use the rising edge of **clk_500** to increment the other counters instead of **clk** was made to reduce the bit size of the counters, since the periods of the other three clocks divides evenly into clk_500's period.

Pausing

When the center button is pressed, the stopwatch enters pause mode, where counters stop incrementation and all digits become frozen. Adjust mode could be active concurrently with pause mode, allowing the specified digit to become unfrozen to be incremented.

A **pause** reg keeps track of whether the device is in pause mode. Before the digit incrementation phase, an if statement checks this reg, and the code skips incrementation if pause is active.

Adjustment

Placing the **ADJ** switch into high toggles adjust mode. In this mode, the digit at the location specified by the two switches is adjusted and the other digits are frozen. The digit to be adjusted blinks at a rate of 5 Hz, downsampling the 10 Hz clock, and is automatically incremented at a rate of 2 Hz.

We programmed this behavior by modifying our incrementation if the adjust switch is on. If it is on, we run incrementation using the 2 Hz clock. Then in our incrementation function, if the adjust switch is on, we only increment the single digit specified by the two switches.

```
case (display_digit)
   0: if (blinkOff && sw[2] && !sw[0] && !sw[1])
         enable_led = 4'b1111;
      else begin
         enable_led = 4'b1110; digit_val=digit0;
      end
   1: if (blinkOff && sw[2] && sw[0] && !sw[1])
         enable_led = 4'b1111;
      else begin
         enable_led = 4'b1101; digit_val=digit1;
      end
   2: if (blinkOff && sw[2] && !sw[0] && sw[1])
         enable_led = 4'b1111;
      else begin
         enable_led = 4'b1011; digit_val=digit2;
      end
   3: if (blinkOff && sw[2] && sw[0] && sw[1])
         enable_led = 4'b1111;
      else begin
         enable_led = 4'b0111; digit_val=digit3;
      end
endcase
```

```
if (clk_1 && !pause && !sw[2])
begin
   incrementDigit0();
end
if(clk_2 && sw[2])
   if(!sw[0] && !sw[1])
      AincrementDigit0();
   else if (sw[0] && !sw[1])
      AincrementDigit1();
   else if (!sw[0] && sw[1])
      AincrementDigit2();
   else if (sw[0] && sw[1])
      AincrementDigit3();
```

The excerpt on the left comes from the **convertToDisplay** task, which displays the **display_digit**-th digit on the display. When adjust switch is on, **blinkOff** alternately turns on and

off to blink the specified digit. Otherwise, we cycle through the digits as usual and specify the value to be shown in **digit_val**.

The excerpt on the right comes from the same module and describes the calling of the digit implementation tasks. When pause and adjust are off, we are in the default mode and the task **incrementDigit0** is executed. Its code also executes other tasks to increment the other digits as appropriate. When adjust mode is on, we use four tasks to increment one of the four possible digits specified by the switches **sw[1:0]**, at the rising edge of **clk_2**.

Debouncing
To deal with debouncing, we use a 500 Hz clock to sample inputs. At the rising edge of **clk_500**, we store the value of our pause button in **btnSDownSample**. This ensures that our signal is not bouncy so we do not register multiple presses of our pause button. We use an always block at the positive edge of **btnSDownSample** to flip the value of **pause**. This correctly implements our pause functionality.

We know that the use of an always block that detects a non-clock signal is bad design, and could have changed our implementation such that we turn a signal high only when **btnSDownSample** changes from 0 to 1 in order to implement this functionality.

Simulation Documentation

We first verified that the four clocks rise for one cycle and rise synchronously periodically (since they are all factors of the main clock). We had to increase the frequencies of our clocks compared to the master clock in order to generate a simulation in a timely manner.

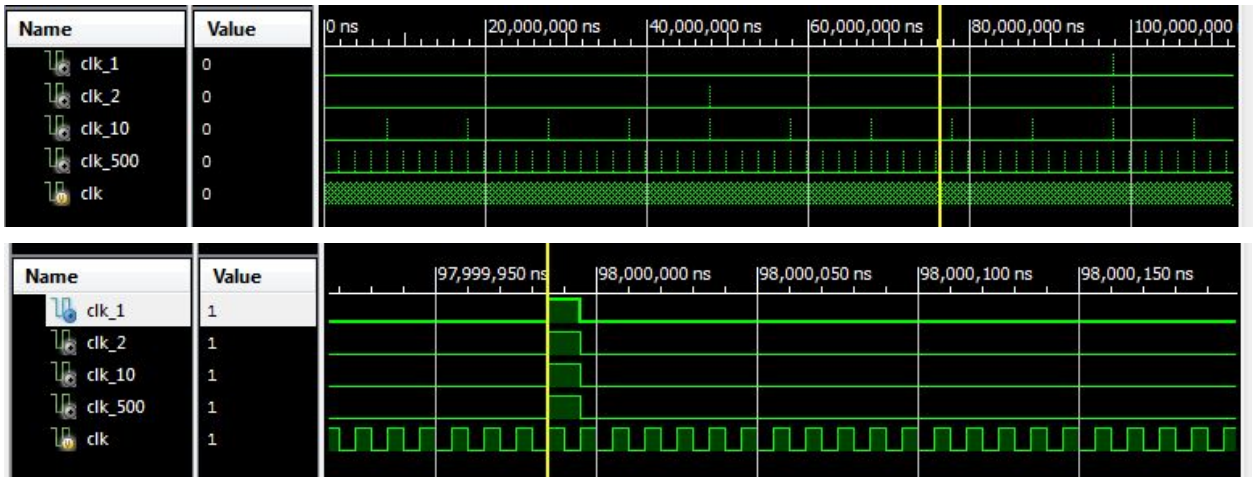

Figure 1: Waveforms from our clock module testbench.

The above screenshots show the relative timings for our clocks, as well as how they are high for one master clock cycle each. We used the testbench to fix an issue where our clocks were not

functioning properly because our count of the master clock cycles would overflow. We fixed it by widening the registers where we stored our count.

Furthermore, we faced the problem of our clocks not ticking at the right frequency. More precisely, after viewing the waveform we realized that the periods of each of our four clocks were all one master clock cycle too long. We adjusted our clock counter to fix this.

Additionally, we fixed an issue where our clocks were not synchronizing correctly. The timings were off, and they were high for only half a master clock cycle.

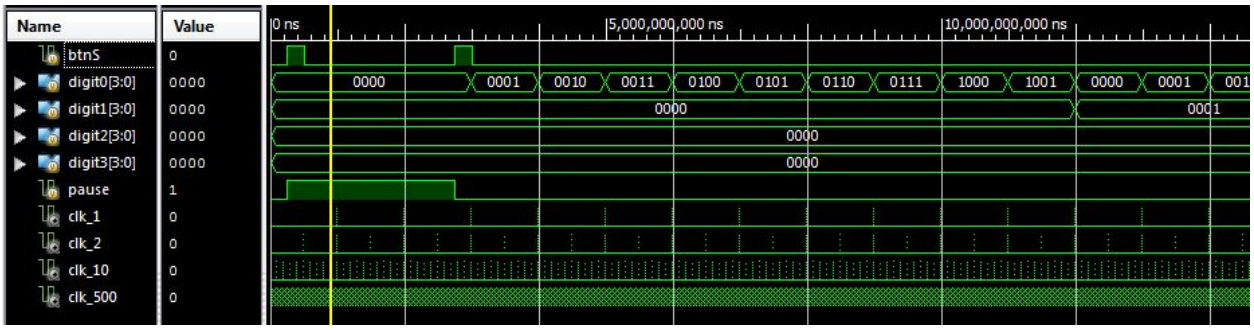

Figure 2: Waveforms from our entire board testbench.

We used a testbench to test digit incrementation. We also tested the pause button's functionality by simulating presses of **btnS**. The above screenshot shows the incrementation happening properly with rollover into higher digits. We had issues with our reset button, but that was fixed by restructuring our code to be inside one always block that detected the positive edge of the master clock.

Finally, we programmed the Nexys 3 board and manually manipulated buttons and switches to confirm that our stopwatch is working as planned.

Conclusion

We built a stopwatch that supports four digits (in the MM:SS format). It also supports pausing and adjustment of the stopwatch through automatically adjusting a specified digit. What's more, the clock was able to blink the correct digit whenever the digit was being adjusted. To achieve the full functionality, our stopwatch employed four clocks in addition to the master clock.

We learned about how to implement multiple clocks through keeping track of counters tied to the main clock's cycles, utilizing tasks to execute routine behaviors, and implementing two distinct modes with overlapping behaviors.