CS M152A
Lab 3 Report
Stopwatch

Authors:

Aaron Yoo, 004745379

Chayanis Techalertumpai, 404798097

Christian Rodriguez, 804789345

Date: 11/18/2019

TA: Logan Kuo

Introduction and Requirements

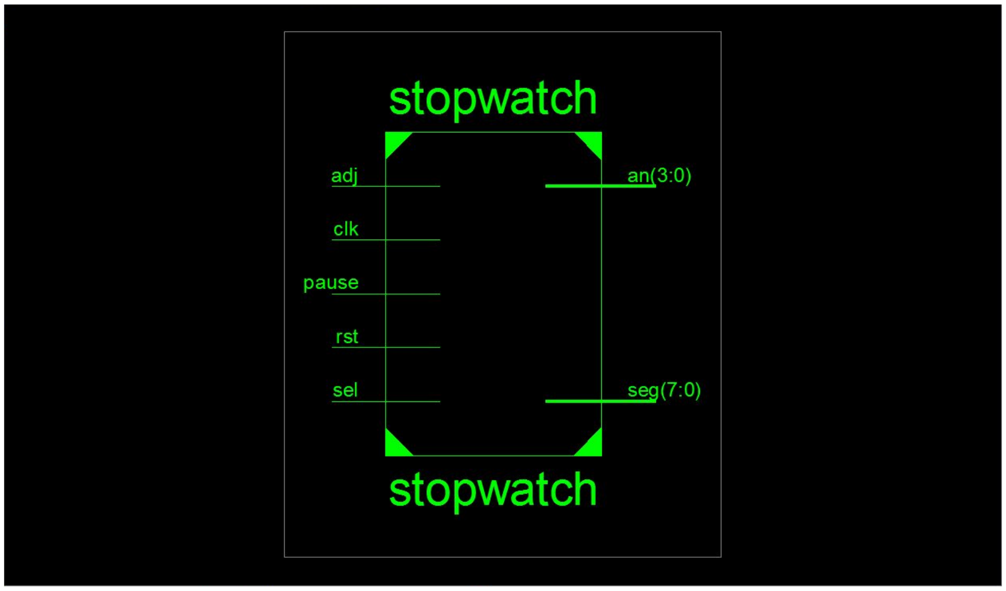
In Lab 3, we were tasked with implementing a stopwatch circuit using the FPGA Board. The stopwatch counter was displayed on the seven segment display of the board. Its inputs were the buttons and the slider switches. Essentially, because we had limited digits, the stopwatch could only go up to 59:59. Once it reaches 59:59, it would go back to 00:00 and recount again. The tens digit of the minutes and seconds can only go up to 5 while the ones digit of the minutes and seconds can go up to 9. The user can also choose to set the stopwatch in adjust mode and select to change either the seconds or minutes of the stopwatch at a rate of 2 ticks per second, 2 numbers per tick. Finally, the user can pause and reset the stopwatch using the buttons on the FPGA Board.

There were 5 main things we had to implement: a counter, a clock, a seven segment display, a debouncer, and the user constraint files. The counter is basically a modulo 10 counter that is used to count the time that has elapsed. For the clocks, there were 4 different clocks - a 2 Hz clock, a 1 Hz clock, a fast clock (50-700 Hz) and a blinking clock (>1 Hz). The fast clock makes sure that the seven segment display is being updated fast enough such that it looks like the digits are not blinking at all. The blink clock is used when the user puts the stopwatch in adjust mode. The 1 Hz clock is used for regular counting while the 2 Hz clock is used to make sure that the stopwatch ticks 2 times per second when in adjust mode. Furthermore, the seven-segment display is used to display the actual stopwatch. The debouncer is implemented with the sole purpose of making sure that our pause and reset buttons are responsive. Finally, the user constraint files maps our code to the actual FPGA Board, enabling our software to mix in with the hardware.

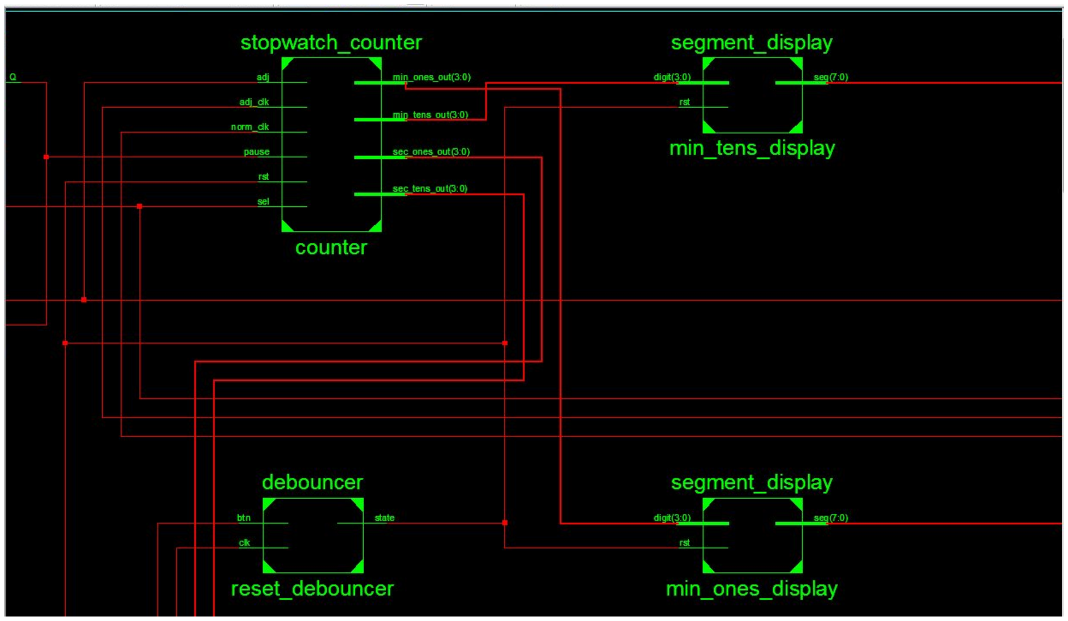
The purpose of this lab was to get us used to building an entire hardware design from the ground up. By breaking down the components of the lab into subcomponents, our task was much easier. A lot of thought was put into why we were doing certain things such as - how many seconds should we make each clock tick, or how should we handle the logic for the counting; in stopwatch or the seven-segment display. This report aims to break down our thought process for each step of the lab.

Design Description

We decided to divide our modules according to the project description in the specs. All modules are connected by the stopwatch module which also contains the logic of constantly updating the seven segment display.



A schematic diagram of the interface of the stopwatch (top module).



A schematic diagram for the core modules of our design. These modules are namely: stopwatch counter, segment_display and debouncer. The only main module missing from the diagram is the clock. The clock has a very simple interface and thus, we chose not to include it.

```
          +
          |
          v
+--------------+      PAUSE          +--------------+
|Clock Running| +-----------------> |Clock Paused |
|Normally     |                      |             |
+--------------+                      +--------------+
        ^  +                               ^   +
        |  |                               |   |
  ADJ   |  |  ADJ                    ADJ   |   |  ADJ
        |  |                               |   |
        +  v                               +   v
+---------------+        PAUSE       +---------------+
|Clock Running | <-----------------+ |Clock Paused  |
|Adjusting     | +-----------------> |Adjusting     |
+---------------+        PAUSE       +---------------+
```

State diagram of the ADJ and PAUSE buttons. The RST button puts the clock back to the "clock running normally" state.

Task 1: Counter

```
module stopwatch_counter(
    input wire rst,
    input wire adj,
    input wire sel,
    input wire norm_clk,
    input wire adj_clk,
    input wire pause,

    output wire[3:0] sec_ones_out,
    output wire[3:0] sec_tens_out,
    output wire[3:0] min_ones_out,
    output wire[3:0] min_tens_out
);
```

The purpose of our counter is to indicate the number that should be displayed on the seven segment display. The seven segment display has 4 digits: the first two indicate minutes and the last two indicate seconds. Our module takes in two clock signals: norm_clk which is a 1 Hz clock and adj_clk which is a 2 Hz clock.

In a normal mode, the seconds counter will increment every second on the posedge of the 1 Hz clock. Every 60 seconds, it reaches the maximum value of 59, the minutes counter is enabled and the seconds counter resets back to 00. When both the minutes and seconds counters reach 59, the clock counter resets back to 0000.

We implement this logic by the following code:

```
if (sec_ones == 4'd9) begin
                sec_ones <= 4'd0;
                if (sec_tens == 4'd5) begin
                    sec_tens <= 4'd0;
                    if (min_ones == 4'd9) begin
                        min_ones <= 4'd0;
                        if (min_tens == 4'd5) begin
                            min_tens <= 4'd0;
                        end
                        else begin
                            min_tens <= min_tens + 4'd1;
                        end
                    end
                    else begin
                        min_ones <= min_ones + 4'd1;
                    end
                end
                else begin
                    sec_tens <= sec_tens + 4'd1;
                end
            end
            else begin
                sec_ones <= sec_ones + 4'd1;
            end
        end
```

In the adjust mode, the stopwatch counter stopped and the ‘Selected’ two digits increase at a rate of 2 Hz, 2 numbers per tick. We implement this by changing the clock signal so that we now increment the counters by 2 at every posedge of the adj_clk.

Task 2: Clocks

```
module clock(
      input clk, rst,
      output reg twohz_clk, onehz_clk, fast_clk, blink_clk
);
```

The purpose of our clock is to take an input clock and generate 4 clocks based on that input clock. For our stopwatch, we needed the following clocks (their main purposes are also listed):

- twohz_clk: A two hertz clock to control two hertz blinking

- onehz_clk: A one hertz clock to control counting
- fast_clk: A fast clock to control the refreshing of the numbers on the LED
- blink_clk: Faster than one hertz clock to control the blinking of the segments during adjust

To implement these clocks we used a simple counter like so:

```
initial begin
      onehz_clk = 0;
      onehz_idx = 0;
end

always@ (posedge clk) begin
      if (onehz_idx == 50000000) begin
            onehz_idx = 0;
            onehz_clk = ~onehz_clk;
            onehz_clk = ~onehz_clk;
      end
end
```

Our clocks are used in both the stopwatch_counter module and the stopwatch (top) module.

Task 3: Seven-Segment Display

```
module segment_display(digit, seg, rst);

      input [3:0] digit;
      output [7:0] seg;
    input rst;
// continued
```

The job of our segment display is to translate a number (digits) into the appropriate output for the segment display. Effectively this is translating between the language of numbers and the language of the seven segment display. The main translation is done in this case statement:

```
      case (digit)
            4'b0000: segment_reg = 8'b11000000;
            4'b0001: segment_reg = 8'b11111001;
            4'b0010: segment_reg = 8'b10100100;
            4'b0011: segment_reg = 8'b10110000;
            4'b0100: segment_reg = 8'b10011001;
            4'b0101: segment_reg = 8'b10010010;
            4'b0110: segment_reg = 8'b10000010;
            4'b0111: segment_reg = 8'b11111000;
            4'b1000: segment_reg = 8'b10000000;
            4'b1001: segment_reg = 8'b10010000;
            default: segment_reg = 8'b11111111;
        endcase
```

In the stopwatch module, we update every segment at the frequency of the fast clock so that the output on the seven segment displays appear constantly illuminated on the hardware. In assigning and updating each segment, we need to set the anode to the correct pattern. The pattern is represented by a 4-bit binary numbers which indicates which segments we want to change. For example, if we want to change the ten minute segment, we need to set the anode value to be 4'b0111.

We also handle the logic of the blinking digits when SEL and ADJ are manipulated here by updating the relevant segment if the positive edge of the blink clock is encountered or displaying the blank segment (none of the digits light up) otherwise.

Task 4: Debouncers

```
module debouncer(
        input clk,
        input btn,
        output state
);
```

In order to sample buttons in a reliable way, we need a debouncer (similar to lab 2). The debouncer takes in a clk and the btn input and outputs the debounced "state" of the button. The debouncer is used to debounce our ADJ and RST buttons.

Task 5: User Constraint Files

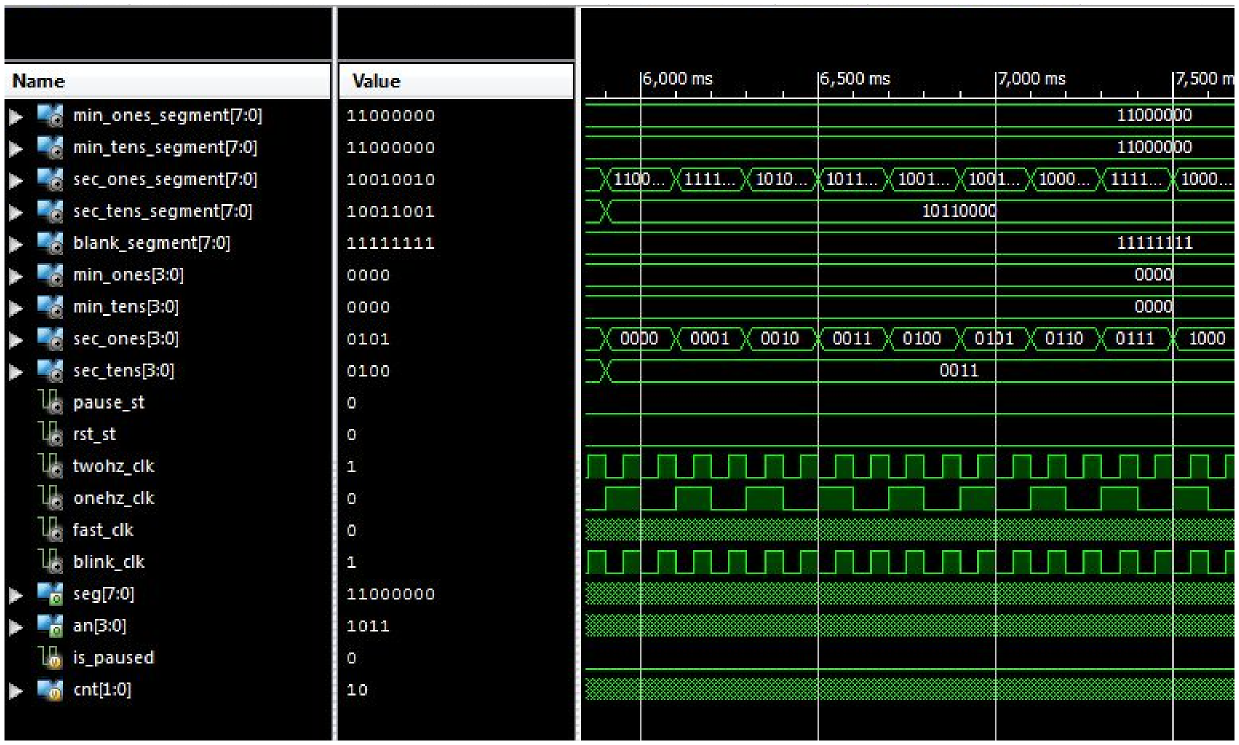
```
NET "seg<0>" ...
NET "seg<1>" ...
NET "seg<2>" ...
NET "seg<3>" ...
NET "seg<4>" ...
NET "seg<5>" ...
NET "seg<6>" ...
NET "seg<7>" ...

NET "an<0>"  ...
NET "an<1>"  ...
NET "an<2>"  ...
NET "an<3>"  ...
```

Modifying the ucf file was the same as any other lab except for the segment display and anode. The segment display controlled output to the 7 segment displays on the board. The anode controlled the changing of these displays.

Simulation documentation

Simulation was difficult in general for this lab because of its reliance on hardware. However, we made an effort to simulate our counter counting up before testing it on the board. We did not simulate the button presses because we figured it would be easier to just test that on the board itself. Also, all of those things required the stopwatch counter to be correct so we decided to tackle that first.



Name	Value
min_ones_segment[7:0]	11000000
min_tens_segment[7:0]	11000000
sec_ones_segment[7:0]	10010010
sec_tens_segment[7:0]	10011001
blank_segment[7:0]	11111111
min_ones[3:0]	0000
min_tens[3:0]	0000
sec_ones[3:0]	0101
sec_tens[3:0]	0100
pause_st	0
rst_st	0
twohz_clk	1
onehz_clk	0
fast_clk	0
blink_clk	1
seg[7:0]	11000000
an[3:0]	1011
is_paused	0
cnt[1:0]	10

Here we can see the behavior of the stopwatch clock in simulation. We mainly used simulation to test the behavior of the counting, as the segment display was nearly impossible for us to simulate. We can see the counters incrementing in the center of the picture.

We did run into a problem with our simulation because rst wasn't being handled correctly. This resulted in us doing a hack:

```
// This line is for simulation
// always@ (posedge clk or rst)
always@ (posedge clk)
...
```

The explanation for this is that the first line allows our simulation to work easily. However, the second line is required for an actual hardware run because the first line causes an error (for some reason you can't have a sensitivity list like the first line.

Another problem we ran into was that we had to drastically adjust our clock speeds to work on the simulator. Otherwise we would be waiting a long time to just see one tick of the stopwatch. These are limitations built into the ISim emulator so we tried to work around them the best we could.

Once we got the stopwatch counter to work, simulation became less and less valuable to use because most of our remaining bugs were from interacting with the hardware, in specific the display.

Conclusion

In conclusion, lab 3 introduced us to a very exciting and comprehensive FPGA project that allowed us to build an entire hardware design from the ground up. This lab helped solidify our understanding of how software and hardware work together as one. We implemented a stopwatch using various clocks, a modulo 10 counter, a seven segment display, debouncers, and user constraint files.

Some of the difficulties we encountered had to deal with debugging both the software and hardware components of our design. In the beginning, we could not get our program to work with the board and we were unsure why. We later figured out that we needed to connect every component we used (rst, pause, adj, and sel components) in Nexys3.ucf to our code in order to make our stopwatch work. We were oblivious to how the switches and buttons linked with our code but once we figured that out, we saw quick progress with our lab.

The lab spec was less thorough this time around but this was done with a purpose. We think the lab was actually pretty enjoyable because we had to think hard about how everything connected together. We learned a lot about how important it is to modularize our designs in order to make it easier for us to build the bigger picture, which in this case was a stopwatch. Although we had a hard time debugging and learning which files to modify so that we can connect the hardware to our code, the lab was still fun! We definitely learned a lot from this lab and will use a lot of our gained knowledge for our final lab.