Lab 3 Report

UCLA Samueli School of Engineering
CS M152A Spring ‘24
Lab 5 TA Chenda Duan

Group 6
Tae Hwan Kim 506043010
Brad Lowe 905581956

1 Introduction and requirement

Stopwatches are a very useful everyday tool. In this lab, we develop a working stopwatch on an FPGA using Verilog and Vivado, which demonstrates the complete design flow of an FPGA board. The seven-segment display is used to display numbers, including separate minutes and seconds sections. The buttons on the board are configured to pause the stopwatch and to reset it to 0 minutes and 0 seconds. The switches are used for adjustment, which causes the corresponding display (minutes or seconds) to blink and increase. This project used a Basys 3 board on an Artix-7 FPGA, and generated a bitstream to send to it using Verilog HDL and the Xilinx Vivado software. We showed the project to our TA Chenda through a demo of all the basic features of the stopwatch, and tested using a Verilog testbench to catch coding errors.

2 Design Requirements

The stopwatch module takes in a master 100MHz clock, the ADJ and SEL switch signals, and the RESET and PAUSE button signals. It outputs the anode to be updated on the seven-segment display along with a value for each bit of the display to be updated at the current point in time, and loops through the anodes so as to update each number in sequence.

It does this using 4 submodules: the counter, clock, debouncer, and seven-segment display. Along with these, it uses a Basys3 Xilinx Design Constraints file to map the signals to the correct pins on the FPGA.

2.1 Clock Divider Block

The clock divider block takes in the master clock and outputs four new clock signals to use in other modules: a 1 Hz clock for updating the stopwatch value, a 2 Hz clock for updating the numbers when in ADJ mode, a 500 Hz clock for updating the seven-segment display, which is necessary because only one of the anodes can be on at a time, and a 4 Hz clock for blinking while ADJ mode is on. The faster clock is also used for the purposes of debouncing, so we can refine the signal from the buttons.

2.2 Counter Block

The counter block takes as input the 1 Hz clock, the 2 Hz clock, the ADJ and SEL switches, and the RESET and PAUSE button signals. It outputs the seconds and minutes values, and increments them on the 1 Hz clock's positive edge. When the ADJ switch is on, it looks to the SEL switch to decide whether to increase either the minutes or seconds value on the positive edge of the 2 Hz clock. When the RESET button is pushed, it resets all values to 0. When PAUSE is pushed, it stops the values from updating until the PAUSE button is pressed again.

2.3 Debouncer Block

The debouncer block takes the raw signals from the RESET and PAUSE buttons and the 500 Hz clock and outputs a refined signal to be used by other modules. This uses a much lower sampling frequency than the main clock to ensure that one press is not registered as multiple presses, and to create metastability between the buttons and switches. The refined signals are treated as the actual RESET and PAUSE signals for the rest of the modules.

2.4 Seven-Segment Display Block

The seven-segment display block takes the ADJ, SEL, 500 Hz clock and 4 Hz clock signals as input, as well as the current values of the anode, seconds, and minutes. It outputs an 8-bit segment value to display on the section of the display currently lit up by the anode. This value will be used by the top stopwatch module to update the display with whatever values are currently in the seconds and minutes positions. It also must check if the display is supposed to be blinking based on the position of the ADJ switch and the position of the blinking clock, and if so, turn off the display intermittently at the specified rate.

3 Design and Implementation

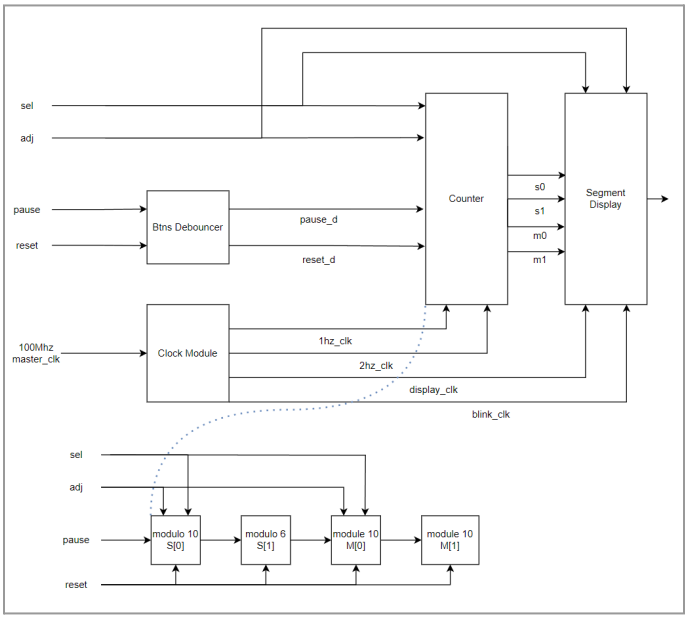

Figure 3.1: Implementation Design Diagram

We drew the diagram to make sure that we understood which inputs are related to which modules. Before coding, the diagram helped us figure out the overall design and approach to coding. Implementing separate modules allows us to test each module individually to ensure they are working correctly.

3.1 Clock Divider Module

```
20  always @(posedge clk) begin
21      if (counter_1hz >= DIVIDE_FACTOR_1HZ - 1) begin
22          clk_1hz <= ~clk_1hz;
23          counter_1hz <= 0;
24      end else begin
25          counter_1hz <= counter_1hz + 1;
26      end
27
28      if (counter_2hz >= DIVIDE_FACTOR_2HZ - 1) begin
29          clk_2hz <= ~clk_2hz;
30          counter_2hz <= 0;
31      end else begin
32          counter_2hz <= counter_2hz + 1;
33      end
```

Figure 3.2: Verilog for clock divider module

The module accepts a master clock input clk and produces four distinct output clocks; clk_1hz, clk_2hz, clk_display, and clk_blink. Each output clock is set with an initial value of zero and toggled to generate different frequencies. The core of this module is built around counters that increment with every positive edge of the input clock and reset upon reaching a predefined threshold.

The module defines several parameters corresponding to the number of cycles needed to achieve the desired frequency based on the 50 MHz assumed frequency of the input clock. Each frequency has an associated counter (counter_1hz, counter_2hz, counter_display, counter_blink) sized appropriately to store the maximum count values without overflow. For each counter, the code checks if the count has reached its respective DIVIDE_FACTOR minus one. If so, the output clock is toggled, and the counter is reset to zero. If not, the counter increments by one.

3.2 Debouncer Module

```
10      always @(posedge clk) begin
11          noisy_signal <= btn;
12          if (noisy_signal == btn_d) begin
13              counter <= 0;
14          end else begin
15              if (counter < DEBOUNCE_TIME) begin
16                  counter <= counter + 1;
17              end else begin
18                  btn_d <= noisy_signal;
19                  counter <= 0;
20              end
21          end
22      end
```

Figure 3.3: Verilog for debouncer module

The debouncer module is designed to clean up noise from a mechanical button input, which is “btn”. It uses a counter to make sure the button state stays the same for a set time before recognizing any changes. The main parts include an input clock signal, which is “clk”, the current state of the button which is “btn”, and the debounced output, which is “btn_d”.

The parameter “DEBOUNCE_TIME” sets how long (in clock cycles) the signal needs to stay stable before a change is recognized. Here, it's set to 1000 cycles. The counter, a 17-bit register, counts each clock cycle when the button state changes, until it reaches the debounce time. The noisy_signal register holds the current state of the button, and it's used to compare with the last stable state (btn_d). As the initial setup, the module starts by setting the btn_d register to 0. This shows the initial debounced state. Each time the clock has a positive edge, the module updates the noisy_signal register with the button's current state, which is “btn”.

Debouncing logic works as follows; If noisy_signal is the same as the last debounced state (btn_d), the counter resets to 0, showing there’s no change.
If noisy_signal and btn_d are different, the counter starts counting up.
Once the counter hits the set DEBOUNCE_TIME (1000 clock cycles), the module updates btn_d to match noisy_signal, marking a stable change. The counter is then reset to 0.
This setup ensures that brief, random noise doesn’t mess with the stable output (btn_d), giving a reliable signal for further use.

3.3 Counter Module

```
// Pause control logic
always @(posedge pause) begin
    run <= !run;  // Toggle running state on pause push
end

always @(posedge selected_clk or posedge reset) begin
    if (reset) begin
        // Reset all counters to zero
        s0 <= 0;
        s1 <= 0;
        m0 <= 0;
        m1 <= 0;
    end else if (run) begin
        if (adj) begin
            if (sel) begin  // Increment seconds if sel is 1
                increment_seconds();
            end else begin  // Increment minutes if sel is 0
                increment_minutes();
            end
        end else begin
            increment_seconds();
        end
```

Figure 3.3: Verilog for counter module

The counter module is an implementation designed as a digital clock that can manage time increments with precision. It operates based on two clock inputs, clk_1hz and clk_2hz, which allow for different time incrementation speeds. Users can control the module with several inputs: reset to zero the counters, pause to toggle the clock's operation, “adj” to select between normal and adjustment modes, and “sel” to choose between incrementing seconds or minutes.

The module outputs four digits: two for seconds (s0, s1) and two for minutes (m0, m1), which update based on the selected clock input and mode. It includes logic to manage the running state and clock selection dynamically and uses tasks like increment_seconds and increment_minutes to ensure proper time rollover from 59 seconds or minutes to 00.

It uses a control signal, run, set initially to active, allowing immediate time incrementation unless paused. The module selects between clk_1hz for manual adjustments when adj is high and clk_2hz for regular operation. It toggles the run state with each positive edge of pause, effectively pausing or resuming the clock. The reset input clears all time counters to zero. Depending on the “adj” and “sel” settings, the module increments either seconds or minutes, providing a time adjustment feature.

```
task increment_seconds;
begin
    if (s0 == 9) begin
        s0 <= 0;
        if (s1 == 5) begin
            s1 <= 0;
            increment_minutes();
        end else begin
            s1 <= s1 + 1;
        end
    end else begin
        s0 <= s0 + 1;
    end
end
endtask

task increment_minutes;
begin
    if (m0 == 9) begin
        m0 <= 0;
        if (m1 == 5) begin
            m1 <= 0;
        end else begin
            m1 <= m1 + 1;
        end
    end else begin
        m0 <= m0 + 1;
    end
end

endtask
```

Figure 3.5: Verilog for sub-tasks in counter module

The module contains two tasks, increment_seconds and increment_minutes, to manage the incrementation of time. The increment_seconds task increments the seconds and automatically rolls over to increment minutes after reaching 59 seconds. The increment_minutes task similarly rolls over after 59 minutes. This design allows for accurate, responsive timekeeping that can be manually adjusted or paused, focusing on minute and second increments without extending to hours.

3.4 Segment Display Module

```
always @(posedge clk_blink) begin
    if (adj) begin
        blink <= ~blink;
    end else begin
        blink <= 0;
    end
end

always @ (posedge clk_display) begin
    if (an == 4'b1110)
        digit <= s0;
    else if (an == 4'b1101)
        digit <= s1;
    else if (an == 4'b1011)
        digit <= m0;
    else if (an == 4'b0111)
        digit <= m1;
```

Figure 3.6: Verilog for segment display module

This module receives several inputs that control its operation. "adj" is a signal that, when active, triggers the blinking of the display to signify an adjustable state and goes into the adjustment mode. "sel" is used to select the minutes or seconds along with "adj" on and determine if the display should engage in a blinking sequence. clk_display drives the update cycle of the display outputs and the visual representation of the digits is synchronized with the system clock. clk_blink manages the timing of the blinking effect, toggling the display visibility on and off to create a noticeable indication. A 4-bit value represents the numeric value to be displayed on the segment. "Segment" is an 8-bit vector that directly controls the individual segments of the 7-segment display, including the optional decimal point or an additional segment.

```
always @(posedge clk_blink) begin
    if (adj) begin
        blink <= ~blink;
    end else begin
        blink <= 0;
    end
end
```

```
always @ (posedge clk_display) begin
    if (an == 4'b1110)
        digit <= s0;
    else if (an == 4'b1101)
        digit <= s1;
    else if (an == 4'b1011)
        digit <= m0;
    else if (an == 4'b0111)
        digit <= m1;

    if (adj && blink) begin
        segment = 8'b11111111;
    end else begin
        case (digit)
            4'h0: segment = 8'b11000000;
            4'h1: segment = 8'b11111001;
            4'h2: segment = 8'b10100100;
            4'h3: segment = 8'b10110000;
            4'h4: segment = 8'b10011001;
            4'h5: segment = 8'b10010010;
            4'h6: segment = 8'b10000010;
            4'h7: segment = 8'b11111000;
            4'h8: segment = 8'b10000000;
            4'h9: segment = 8'b10010000;
        default:
            segment = 8'b11111111; // Blank
        endcase
    end
end
```

Figure 3.7: Verilog for segment display module, ctd.

Activated on the rising edge of clk_blink, this section of code checks if the "adj" mode is active. If so, the blink signal is toggled to alternate the display's state between visible and invisible, creating a blinking effect. If "adj" is inactive, the blinking is disabled by setting blink to 0. Triggered by clk_display, this logic block is responsible for determining the current state of the display based on the input conditions. If the "adj," "blink," and "sel" conditions are all met, the display is turned off to make the blinking effect. Otherwise, the digit input is decoded into a corresponding segment pattern using a switch statement. Each hexadecimal value is mapped to a specific pattern that lights up the appropriate segments on the display. Unmatched values result in a blank display. The use of a binary toggle for the blink signal under the "adj" condition allows for minimal computational overhead, making the blinking effect both simple and reliable. The direct mapping of digital numbers to segment patterns via the case statement provides clear feedback on the display.

3.5 Stopwatch Module

```
21      initial begin
22          an = 4'b1110;
23      end
24
25      always @ (posedge clk_display) begin
26          if (an == 4'b1110)
27              an <= 4'b1101;
28          else if (an == 4'b1101)
29              an <= 4'b1011;
30          else if (an == 4'b1011)
31              an <= 4'b0111;
32          else if (an == 4'b0111)
33              an <= 4'b1110;
34      end
```

Figure 3.8: Verilog for anode update section in stopwatch module

This module for a stopwatch includes all functional components we implemented separately: clock divider, debouncer, counter, and a segment display module. It integrates these components to handle input signals from buttons and switches, generate output signals to count time, manage selected time, and finally display it.

The module takes inputs ‘clk’, ‘adj’, ‘sel’, ‘reset_push’, and ‘pause_push’, and outputs ‘an’ and ‘segment’. ‘clk’ is the master clock input, ‘adj’ and ‘sel’ are switch inputs, and ‘reset_push’ and ‘pause_push’ are the inputs for the left and right buttons, respectively. The ‘an’ is a 4-bit output to control which segment of the display is active for updates. The ‘segment’ is an 8-bit output that controls which segments of the display are turned on.

As the initial setup, it sets the ‘an’ signal to 4'b1110, initializing the display control. Then, the display control logic determines which digit on the display is currently active, rotating through four possible values. The clock_divider takes the master clock and generates slower clocks for various functions like adjusting time, internal timing, and updating the display. There are two debouncer instances for ‘pause_push’ and’ reset_push’. These ensure that the input from the buttons is free of noise and mechanical bouncing, providing stable output signals. The counter handles the time increment logic based on the 1hz and 2hz clocks. It responds to reset and pause signals and adjusts the time depending on the ‘adj’ and ‘sel’ inputs. It outputs the current time as digits. The segment_display module controls how the digits (seconds and minutes) are displayed on a 7-segment display, using the blinking and display clocks to manage visibility and blinking, indicative of active settings or paused states.

The stopwatch module serves as an integrated circuit for a digital stopwatch capable of displaying minutes and seconds on a multi-segment display. It supports adjustment and selection modes through switches and reset and pause functions through buttons. It finally

outputs segments based on all properly processed signals, rotating the ‘an’ register to update correctly every clock cycle and displaying all digits properly on the display.

4 Simulation and Testing

The majority of testing for this project was simply done by seeing the numbers that showed up on the FPGA’s display. Near the end of the lab we wrote a very basic testbench for our project, but we only did this to view the simulation waveforms in order to fix a specific problem we had. This problem, where the seconds and minutes values would show up in the wrong places, is explained in more detail in the Difficulties section.

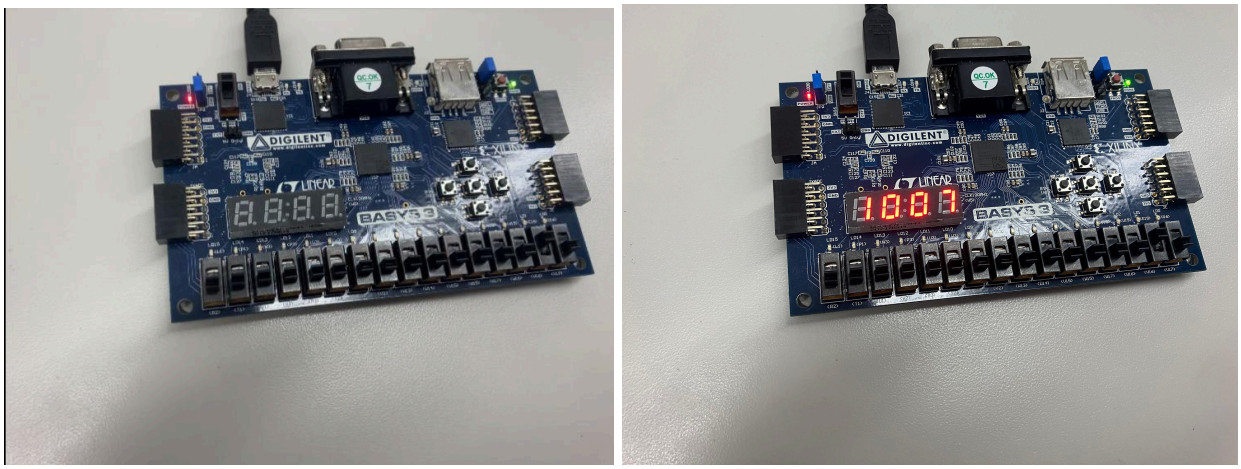

Figure 4.1, 4.2: All numbers blinking rather than only the minutes or seconds blinking

An example of testing is shown in figures 4.1 and 4.2. Through testing on the FPGA itself, we discovered that all 4 digits would blink when ADJ was on rather than only the 2 minutes digits or 2 seconds digits. After discovering this bug, we edited the code where we suspected the issue to be and then waited for the project to build on Vivado. Once the bitstream was generated, we reprogrammed the FPGA and checked to see that the correct behavior was observed. Through testing in this manner, we also discovered an issue where we accidentally swapped the blinking and 1 Hz clocks in the counter module, and were able to quickly fix it.

We ran into another problem near the end where each clock signal was changing half as quickly as we expected. To remedy this, we crafted a basic testbench with each of the modules, based on the testbenches we created in labs 1 and 2. We then used the Run Simulation feature in Vivado to print out our signals over time and see if there was anything wrong with them. We soon realized that we initially set our constants in the clock divider module assuming that one period for the clocks was the time between a positive edge and the next negative edge, rather than a positive edge to the next positive edge. We adjusted our constants accordingly. We did not use any waveforms for testing in this project.

5 Conclusion

5.1 Design Summary

Our stopwatch design uses a top-level stopwatch module and 4 inner modules. The first is a clock divider which takes a master clock signal and splits it into 4 separate clocks to be used in other modules. The second is a debouncer which refines the button and switch signals on the FPGA to ensure metastability and remove noise from the input. The third module is a counter which updates the values of the seconds and minutes at a rate that depends on the clock input and the input from the buttons and switches. The last is a seven segment display module which updates the FPGA's display anytime the second or minute value changes. These modules all work together along with the constraint file which specifies which pins to link to each of the top-level inputs and outputs.

5.2 Difficulties

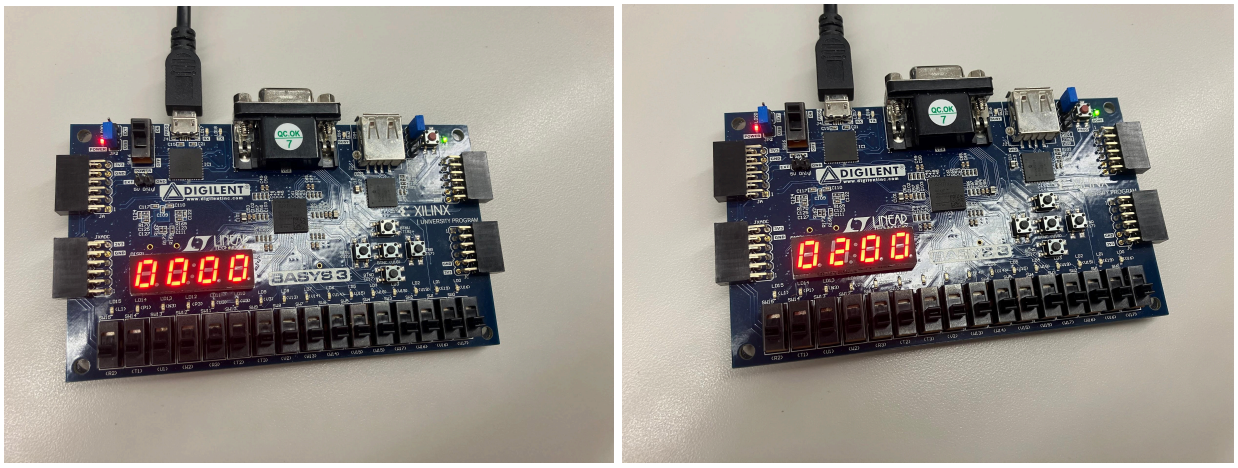

Figure 5.1, 5.2: Images of swapped minutes / seconds values

We ran into an issue where the minutes and seconds values were switched on the display, as shown above. We tried for a while to solve this bug, and we think it has to do with the anode update timing being linked to one of the clocks in a peculiar way. In the end, we asked Chenda and he said he was not able to figure out what was causing the issue in our code. The stopwatch still functions perfectly other than this, just with the seconds on the left and the minutes value on the right side.

Other than this, the only real difficulties we encountered came from learning to work with the entire FPGA design flow, and the somewhat unintuitive nature of only being able to turn on one anode at a time and update the corresponding 7 segments.

5.3 Suggestions

We felt that this lab was much harder than the previous two, but not unreasonably difficult. Some of this difficulty came from a few parts in the spec that were vague and confusing, namely the way to handle the seven-segment display. Luckily, Chenda was very helpful whenever we needed clarification on any of the confusing sections.

Other than that, the logic for each module was relatively easy to implement and, even if it was a step up from the previous labs. The Vivado software was relatively slow, and this meant there was an unnecessary amount of waiting anytime we made changes to the code, but that is probably not something that can be fixed by any changes to the project. Also, writing a testbench for this project was not something that was mentioned in the spec, and we think it may be useful to give a brief explanation of how to do so in the future for simpler testing.