Final Project

Siya Bajaj (2160312)
EE/CSE 371
June 6, 2024
Lab Report #6

Design Procedure

Our final project for the class involves using ROM, a VGA controller, and user input to create a custom project with the above requirements, which we chose to be an analog stopwatch and a digital stopwatch, running together. The stopwatch will display each second on the VGA terminal as an analog clock and we utilized the HEX display from the FPGA board to show the same time in digital format. Our stopwatch features a timer where the user can choose a 15-second (SW[1] and SW[5]), 30-second(SW[3] and SW[0]), or 45-second(SW[4] and SW[5]) interval that counts backward until it reaches 0, which is when it stops. This timer is displayed on both the analog and digital stopwatches. Once the timer finishes, LEDR[9] will turn on to notify the user that the timer has completed.

Since color was not a significant part of our program, we decided to use the VGA modules (VGA_framebuffer) that only displays in black and white, and created the analog stopwatch in black and white. We created a ROM as our memory storage for each x and y coordinate for the clock's arm, and a counter processes the logic for each second that needs to be displayed, which parses through the ROM and reads the values for each point. We assigned SW[9] as our reset, SW[8] as start/stop, and used SW[0], SW[1], SW[3], SW[4], and SW[5] for our 15-second, 30-second, and 45-second timers.

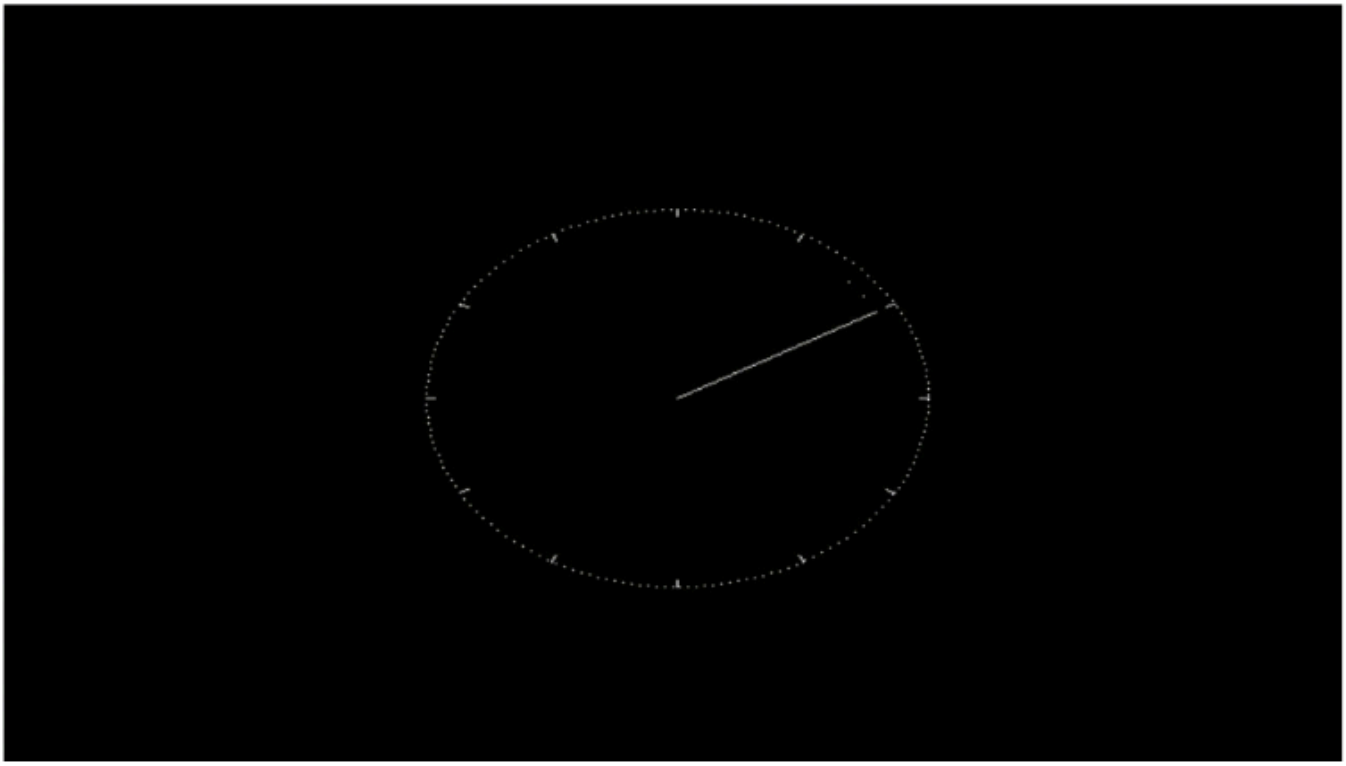

Figure 1: Image of our analog stopwatch on VGA display (stopped at 10 seconds)

Clock_body and clock_arm

To create the outer part of the analog stopwatch, we used Bresenham's Circle Algorithm to create a separate module that takes in the dimensions (centre points and radius) to create a circle which we used to represent the body of our analog clock. It is just a circle that stays still throughout the program in the background giving a impression of a clock. Initially, we chose the center point for the circle and tried to use mathematical calculations to find the 12 coordinates of the circle. This method proved to be time-consuming, so we switched to using Desmos to find the coordinates more efficiently. For every 5 seconds, we created small "line marks" on the circle. To add those marks onto the clock body, we used four 1-port RAMs to store the x and y coordinates of each of the marks. The RAM can hold 12 values (for each 5-sec mark), and 4 RAMs give us the starting and ending coordinates for each of the mark.

These points are then used in a line_drawer module which follows the Breseham's Line Drawing Algorithm that draws a line from one given point to another**.** Once we had our coordinates, we manually input them instead of using RAM. Instead we used a state machine for each point as one state to draw. This approach did not work, possibly due to the lack of a .mif file to ensure that the coordinates were accurate and consistent. There was no way to organize and access the data systematically, which could have caused our points to be read incorrectly. To address this, we created a RAM and placed our coordinates into the .mif file. Despite using the same coordinates, creating the .mif file and inputting those coordinates seemed to be much more effective, systematic, and convenient, with more readable code.

One issue we encountered was that the VGA controller from Lab 5 has limitations, as it can only hold one x and one y value at a time. This posed a problem because we needed both the clock body and the clock arms to appear. To overcome this, we created a multiplexer (mux) that would generate the arms of the clock after the body was drawn.

```
// creating a mux to choose the values from clock arm once the body is done
always_ff @(posedge CLOCK_50) begin
   if(clk_done) begin
      x <= x_arm;
      y <= y_arm;
   end else begin
      x <= x_body;
      y <= y_body;
   end
end // always_ff
```

Figure 2: MUX to choose from the given points to be displays on VGA based on completion of the clock body.

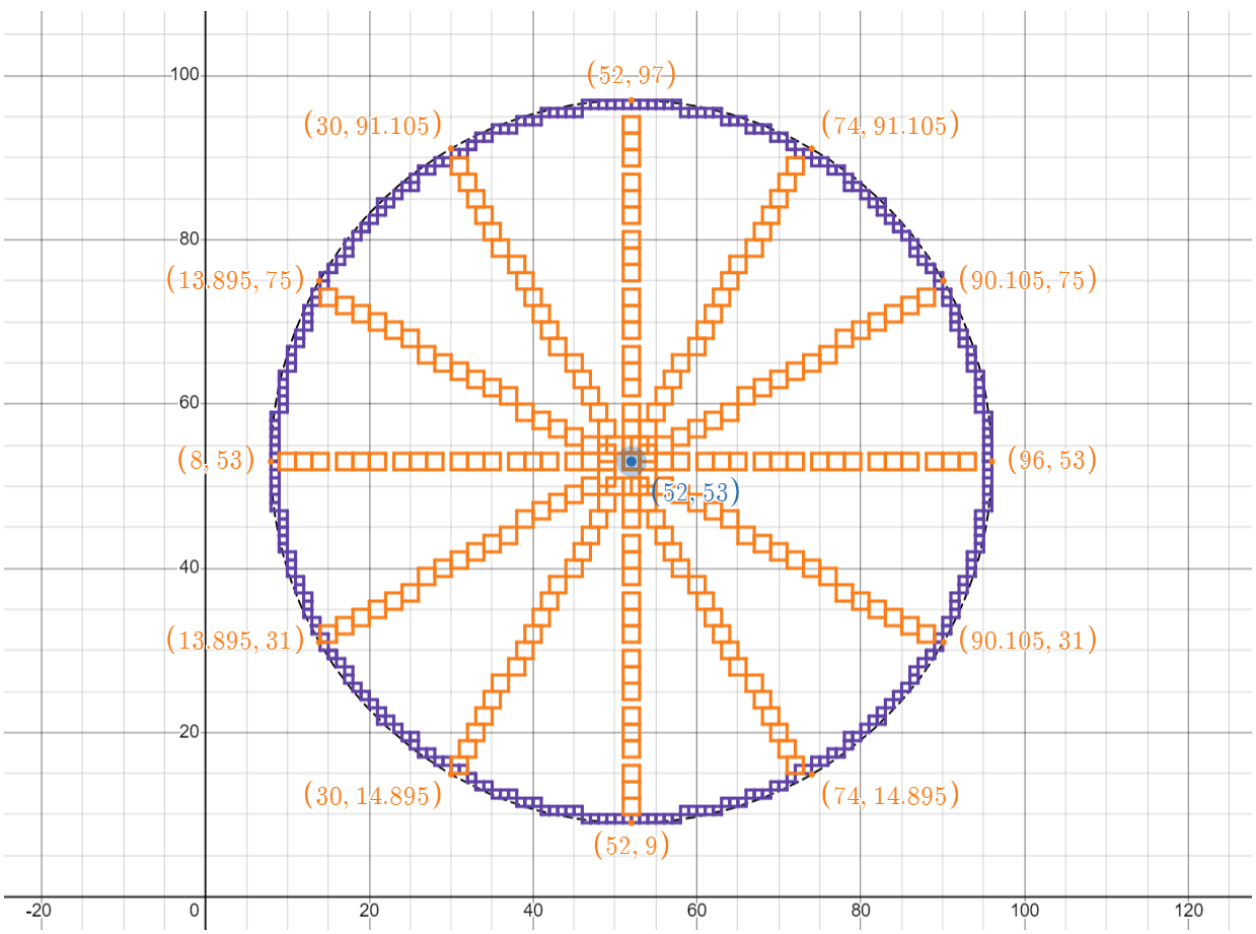

Figure 3: Screenshot of the circle created on Desmos (was enlarged for the code)

The clock_arm module contains two RAM instances for the sixty points of x and y values needed for the analog stopwatch to store where each second would be. It also includes input logic signals: fifteen, thirty, fortyfive, and timer, which provide additional timer functions of 15 seconds, 30 seconds, and 45 seconds for the stopwatch. To implement this timer logic, we used an always_ff block that resets the counter to 0 if reset is active or the counter reaches 60. It then sets the counter to 15, 30, or 45 if timer is active and the respective signals are high. The counter increments or decrements based on the one_second signal and timer mode when start_clk(SW[9]) is active. The start switch (SW[9]) is also the same switch used to stop the stopwatch at a specific time that the user can choose when to stop. This module also includes an instance of the line_drawer, which is responsible for drawing the clock arm from the center to the target coordinates. It then outputs the final x and y coordinates of the clock arm and sets the done signal when the drawing is complete.

clock_divider

As we were implementing the clock body and clock arm, we encountered a significant timing issue. The original clock divider (from our previous labs) provided to us runs at a high frequency of 50 MHz, which translates to roughly 1/50,000,000 seconds per cycle. Initially, we used the clock_divider module to divide the given 50MHz clock into separate clocks. We used Clock_50(50MHz) for our line_drawer/circle_drawer to draw at a faster speed. We used a 1.5 Hz clock for our internal “counter” to run on (which had a parameter of 24 and was roughly a second) for the arm to move with each second. Apart from this we had another clock which was somewhere in between. This clock was introduced since we were having issues with animating the clock arm (to set the color to black/white) since the VGA framebuffer module could only take in one value of color at a time. So we thought of having a clock somewhere in the middle that would control color. However, that method (with three different clock cycles) had multiple timing issues mainly due to none of the clocks would sync with the others. Manipulating the provided clock divider proved to be more difficult than creating our own. In our custom clock divider, we set CLOCK_CYCLES_PER_SECOND to 50,000,000 (since there are 50 million cycles per second) to match the timing requirements for our stopwatch. Additionally, we added an output signal called one_second and change_color, which allowed us to animate the clock arms to move precisely every half-second (change_color goes high every half-second). We also included a 27-bit counter that keeps track of our clock cycles. When the counter reaches the total number of clock cycles in one second (50,000,000), it resets and generates a one-second pulse, which is used throughout our project to have a perfectly synced clock (both analog and digital) and everything runs on one clock (clock 50).

```
module clock_divider (clk, reset, one_second, change_color);

   input logic clk, reset;
   output logic one_second, change_color;

    parameter CLOCK_CYCLES_PER_SECOND = 50000000;

    // Register to keep track of the clock cycles
    logic [26:0] counter;

    always_ff @(posedge clk or posedge reset) begin
        if (reset) begin
            counter <= 0;
            one_second <= 0;
            change_color <= 0;
        end else begin
            if (counter == CLOCK_CYCLES_PER_SECOND - 1) begin
                counter <= 0;
                one_second <= 1; // Generate the pulse
                change_color <= 1;
            end else if (counter == (CLOCK_CYCLES_PER_SECOND - 1) / 2) begin
                counter <= counter + 1'b1;
                one_second <= 0;
                change_color <= 1;
            end else begin
                counter <= counter + 1'b1;
                one_second <= 0;
                change_color <= 0;
            end
        end
    end
endmodule
```

Figure 4: Screenshot of new clock_divider created for Lab 6

Displays module

We created the digital portion of the stopwatch using the displays module, which shows the digital stopwatch on the HEX displays of the FPGA. Since our analog stopwatch includes 15, 30, and 45-second timers, we also added this functionality to the digital stopwatch display. To implement this logic, we used a finite state machine (FSM) with internal 4-bit registers minutes(HEX3) and tens(HEX1), and states named h0, h1, h2, h3, h4, h5, h6, h7, h8, h9 (all on HEX0). We defined how the states change based on the current state and the timer signal, along with the one_second signal. We then created an always_ff block that handles the main counting logic and updates the present state (ps) and counters (minutes, tens). When reset is active and timer is not active, ps is set to h0, minutes to 0, and tens to 0. If timer is active, the tens and ps are set based on the fifteen, thirty, and fortyfive signals. When timer is active and both one_second and start_clk are high, minutes is reset to 0. If ps is h0 and tens is not 0, tens is decremented by 1, and ps is updated to the next state. If ps is h0 and tens is 0, ps remains unchanged. Otherwise, ps is updated to the next state (ns). If timer is not active and both one_second and start_clk are high, ps is updated to the next state (ns). If ps is h9 and tens is 5, minutes is incremented by 1, and tens is reset to 0. If ps is h9, tens is incremented by 1. Otherwise, minutes and tens remain unchanged. In the displays module, we have the output as

our 7-bit HEX displays (H0, H1, H3) and use the provided seg7 module to display the tens on H1 and the minutes on H3.

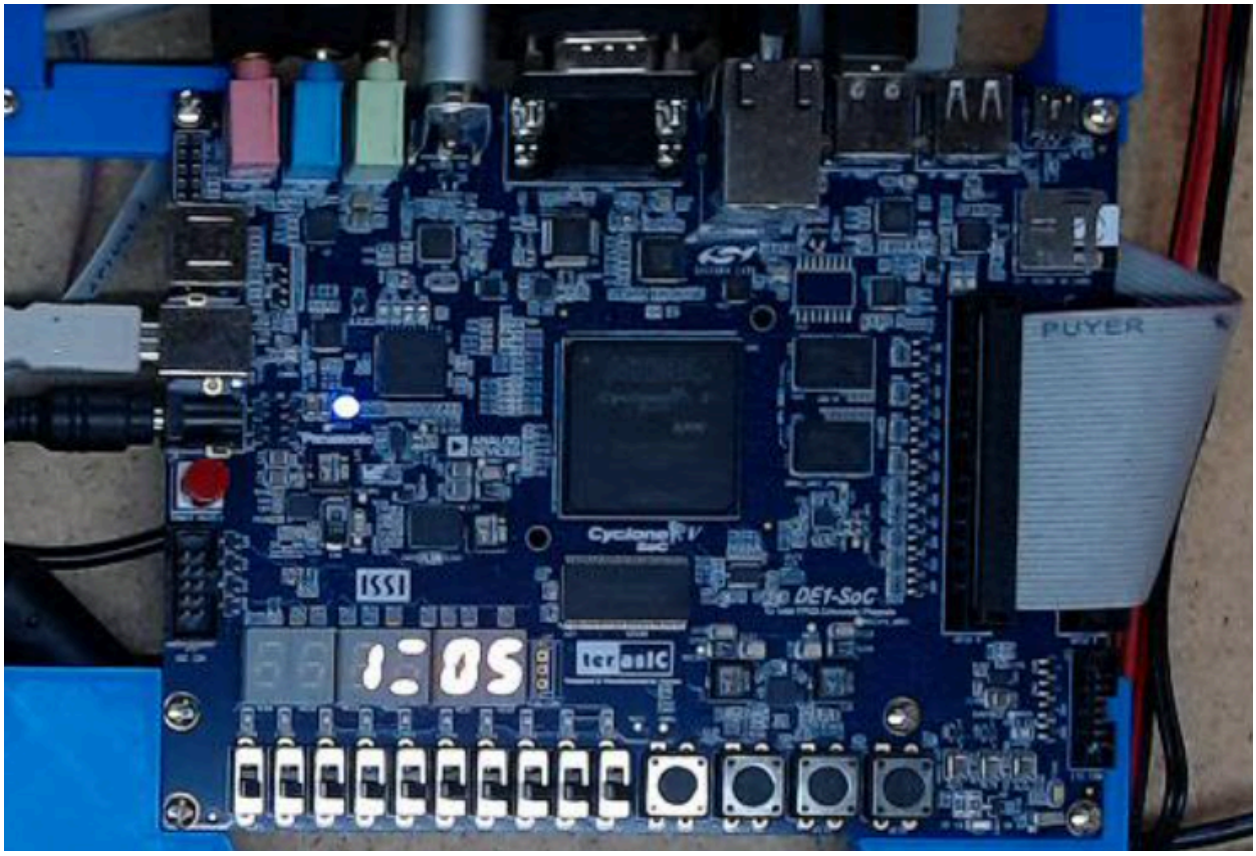

Figure 5: Image of our digital stopwatch on FPGA HEXs

DE1_SoC

Our DE1_SoC module handles user input for the stopwatch functionality. We assigned switches SW[1] and SW[5] to handle the fifteen-second timer, SW[3] and SW[0] for the thirty-second timer, and SW[4] and SW[5] for the forty-five-second timer. The reset is controlled by SW[9], and the start/pause function is controlled by SW[8]. The clock_divider module is instantiated to generate the one_second pulse and the change_color signal. We also included color toggle logic that toggles the color signal when the change_color signal is active. So every half a second, a change in color is seen while looks as if the arm of the clock is being animated since after every half a second, color is changed to black (to draw the same line in black so as to make it disappear) and after another half a second color is switched back to white but now the line drawn is a new line at a new point since one_second is also high after two half-seconds. The clock_body and clock_arm modules are instantiated to handle the coordinates for the stopwatch. The displays module manages the digital display of the stopwatch, updating HEX0,

HEX1, and HEX3 based on the current time and timer settings. Additionally, we set HEX4 and HEX5 to '1' to turn them off and configured HEX2 with two points to represent the "dots" or a “semicolon” of the digital clock.

```
   assign reset = SW[9];
   assign start_clk = SW[8]; // start stop switch
   assign clk = CLOCK_50;

   // adding more ways for the user to interact with the program
   assign fifteen = (SW[1] & SW[5]);
   assign thirty = (SW[3] & SW[0]);
   assign fortyfive = (SW[4] & SW[5]);
   assign timer = (fifteen | thirty | fortyfive);

   clock_divider clock (.clk, .reset, .one_second, .change_color);

   always_ff @(posedge clk or posedge reset) begin
      if(reset)
         color <= 1;
      else if (change_color)
         color <= ~color;
   end

   VGA_framebuffer fb (
      .clk50          (CLOCK_50),
      .reset          (1'b0),
      .x,
      .y,
      .pixel_color    (color),
      .pixel_write    (1'b1),
      .VGA_R,
      .VGA_G,
      .VGA_B,
      .VGA_CLK,
      .VGA_HS,
      .VGA_VS,
      .VGA_BLANK_n    (VGA_BLANK_N),
      .VGA_SYNC_n     (VGA_SYNC_N));

// assign HEX0 = '1;
// assign HEX1 = '1;
   assign HEX2 = 7'b1110110;
// assign HEX3 = '1;
   assign HEX4 = '1;
   assign HEX5 = '1;

   logic clk_done;

   always_ff @(posedge clk) begin
      if(clk_done) begin
         x <= x_arm;
         y <= y_arm;
      end else begin
         x <= x_body;
         y <= y_body;
      end
   end

   clock_body body (.clk, .reset, .x(x_body), .y(y_body), .clk_done);

   clock_arm arm (.clk, .reset, .x(x_arm), .y(y_arm), .one_second, .change_color, .start_clk, .fifteen, .thirty, .fortyfive, .timer);

   displays show (.clk, .reset, .H0(HEX0), .H1(HEX1), .H3(HEX3), .one_second, .start_clk, .fifteen, .thirty, .fortyfive, .timer);
```

Figure 6: Portion of our DE1_SoC code displaying all necessary modules

Overall System

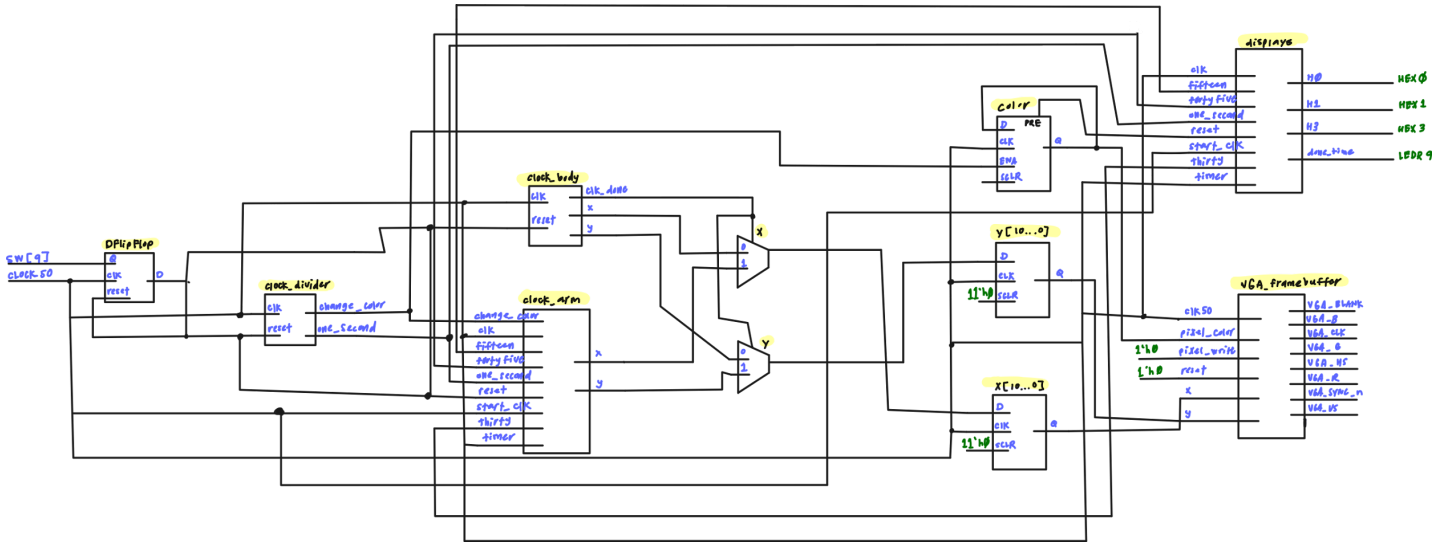

Figure 7: Top-level block diagram of our Lab 6 project on the DE1_SoC

Results

The finished project for Lab 6 does not differ much from the submitted proposal. However, we did make some additions regarding the user input features. We also changed another aspect of our clock from it being blue and red in color. Instead we chose to keep it in black and white. Our graded lab proposal suggested adding another function, as our initial user input was lacking. Which is why our finished project has a lot more user input and user interaction (reset, start, stop, 15sec timer, 30sec timer, 45sec timer) than what was mentioned in our proposal.

We thought that our project was highly practical. Our overall system works just like an online stopwatch with both analog and digital displays. User can **reset** the clock to start from 0, **start** the clock to start incrementing every second by one (as in 1sec). The same button can be used as a **stop** to stop the clock as a user-preferred time. The functionality of our code also includes a timer which allow the user to choose a specific time (in seconds) from 15, 30, 45, and the stopwatch would count down from that timer until it reached 0, after which it stops moving, indicating the timer is done (indicated by LEDR[9]).

Clock_divider

Our clock_divider module does not have a testbench. This is due to clock_divider using CLOCK_50 as the main clock and the outputs being pulse signals at every 50 Million clock cycles. Since we made our clock_divider ourselves and used such a huge number to represent

one second for our implementation, creating a testbench for it proved to be very hard as it was running for quite a long time.

Clock_body

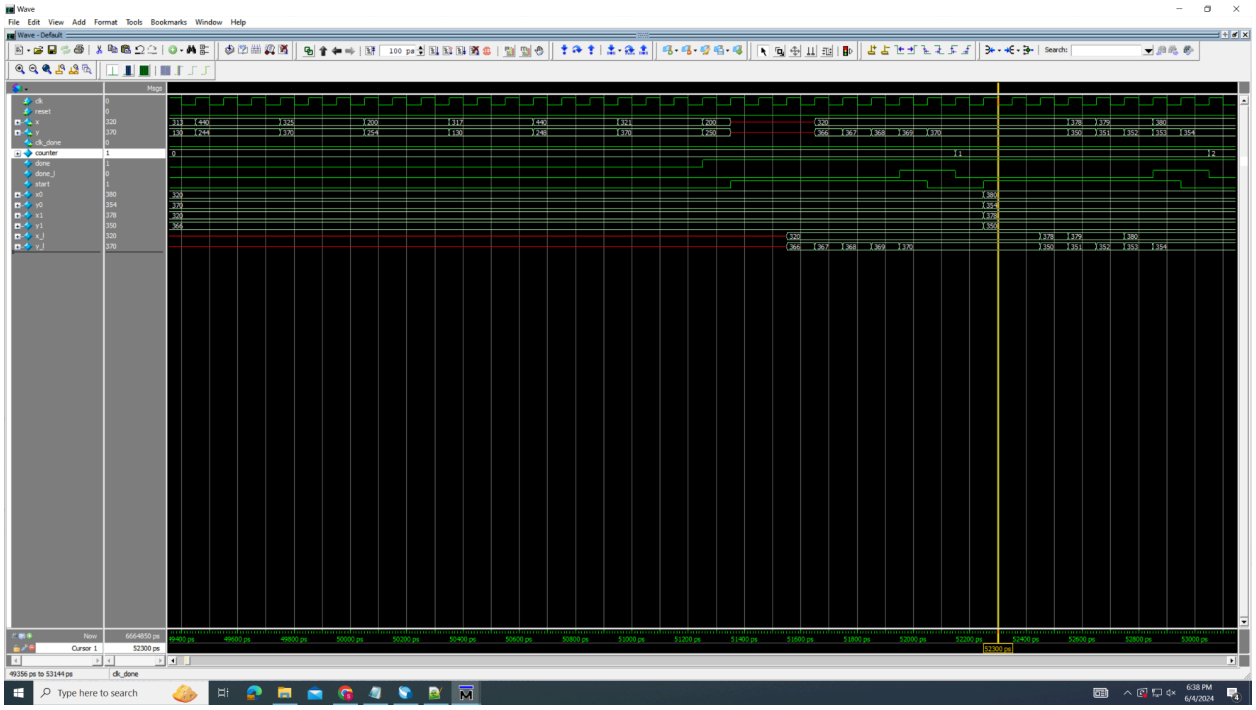

Figure 8: Simulation results for clock_body (until the circle of the clock is done)

Above figure provides a snippet of an important portion of entire simulation results for clock_body. The simulation was created to test whether the function clock_body was working properly. Here in the above testbench, an important signal that we can see is the done going high. The done signal represents the circle of the body of the clock being done. Since the body contains a few components - the circle and the marks for every 5 seconds - we first decided to implement the circle. As seen in the screenshot above, the points x and y that are outputs of our circle_body module are set to the output received from the circle_drawer which follows bresenham's circle algorithm. Once the entire circle is done, the counter (which is an internal variable used as an address to send into the RAMs which reads out the coordinates for each mark of 5sec) increases to 1 after finishing drawing the mark at 0. From the point of the circle being done, the outputs x and y then change to the outputs received from the line_drawer module which draws lines from one given point to another (used for the marks on the clock).

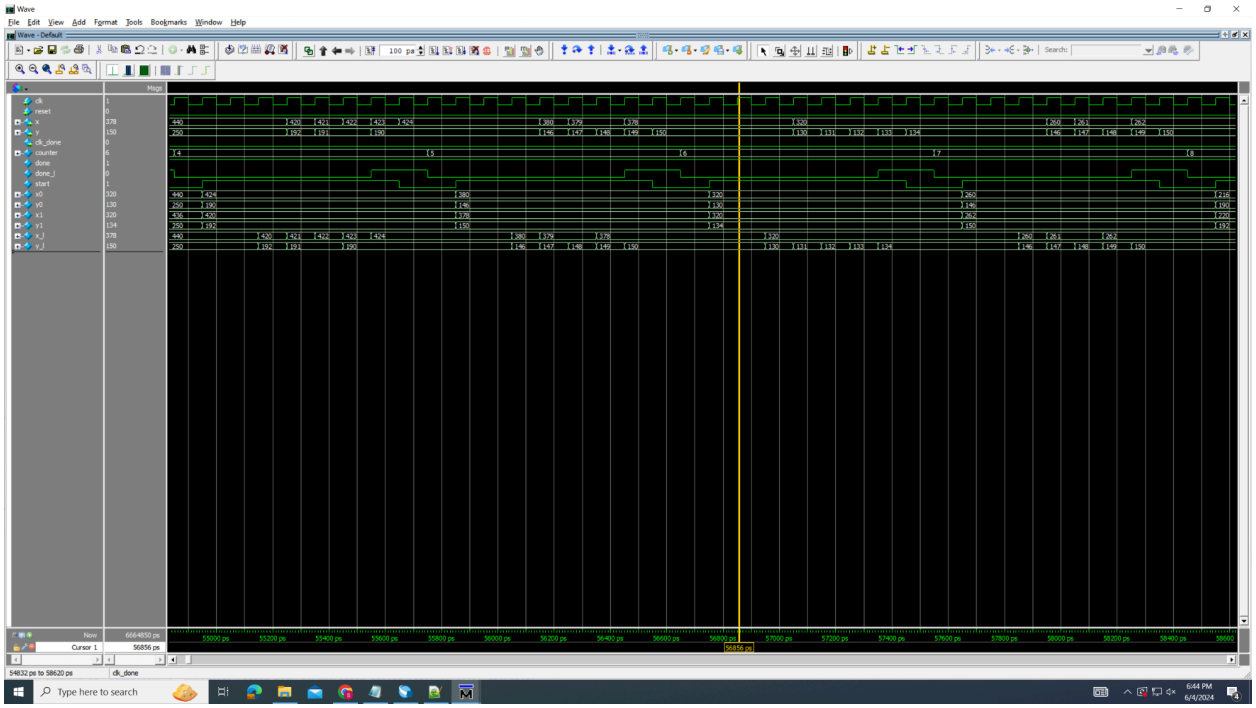

Figure 9: Screenshot of simulation for clock_body (during the process of drawing the 5sec marks)

The above figure is another screenshot of the same testbench created for clock_body to see if each of the 5second marks are being drawn properly. As seen, the counter correctly updates based on the logic used for the counter (which is when the line is done - done_l and when start is deasserted). We can also see that after every line is done (which gives out the signal done_l) start is successfully deasserted since line_drawer follows state logic that requires it to start from the initial state everytime there are new points. With the help of this testbench, we also confirmed that the points that are being read from the RAMs (which we manually set) are correct and the program is correctly reading and drawing from the correct points.

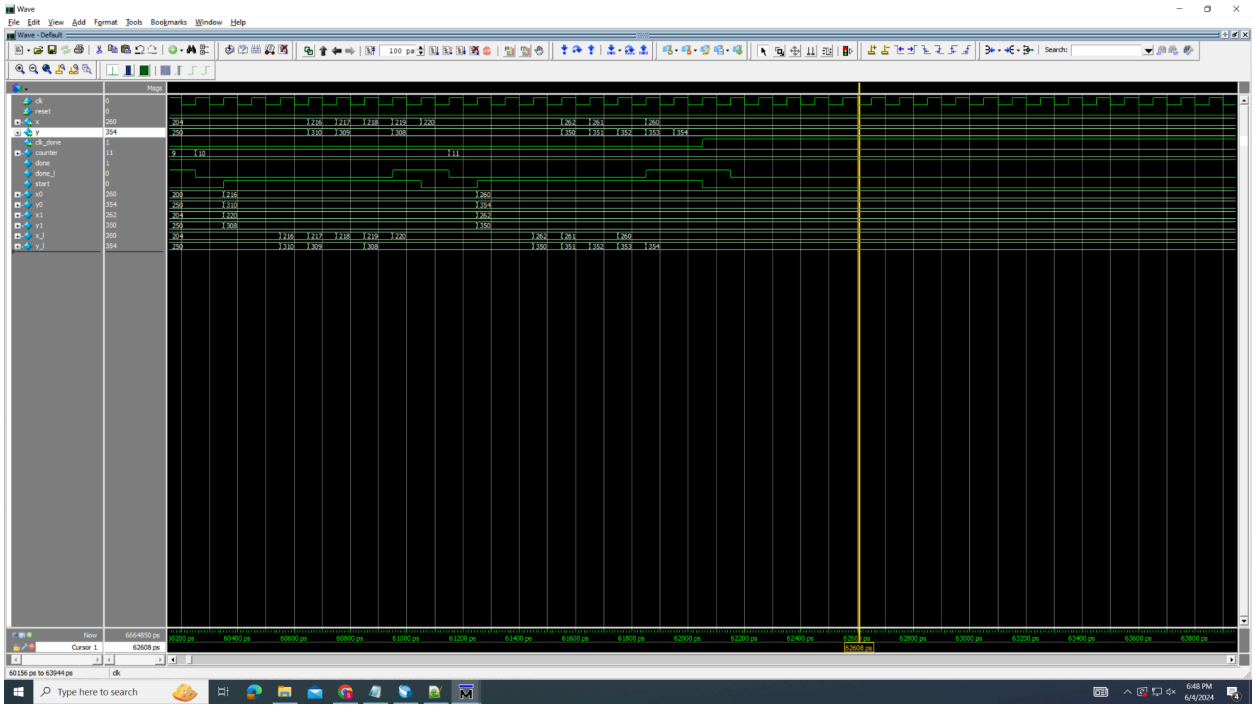

Figure 10: Simulation of testbench created for clock_body (when clk_done signal is received)

Lastly, we have the above figure which shows what happens when the counter reaches 11 (i.e. end of RAM file). Our entire logic in our top-level module is based on when to draw the clock and when to draw the arm. The clk_done signal is a very important signal that is used in the mux created in TLE which decides which points to give into the VGA framebuffer. Above figure shows that when the counter reaches 11, it draws the last 5 second mark, and once done with the clock, it gives out the signal clk_done. Our logic also makes sure that the counter doesn't increment after it reaches 11, which helps in making sure that clk_done signal is forever high once the body of the clock is done.

Clock_arm

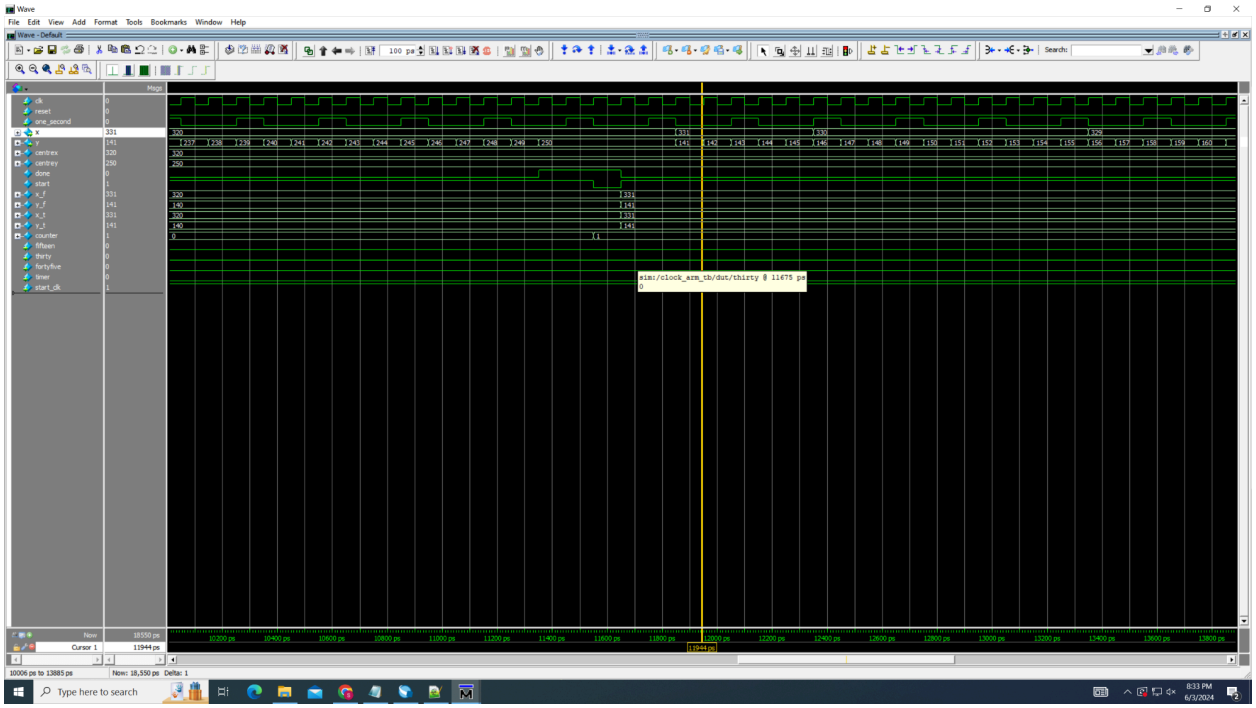

Figure 11: Testbench from clock_arm

The above snippet was taken from the simulation of the testbench made for clock_arm. Some important signals included are counter, x_t, y_t, x, and y, along with one_second and start_clock. The testbench was created to see if our code was working correctly. As seen above, counter moves on and increments when a done signal is received from line_drawer indicating that the previous line is done and when the one_second signal is high at the same time to time it correctly. Since our only purpose was to observe whether that particular module was working, we thought to keep our fifteen, thirty, and fortyfive signals as low. As seen, the code successfully draws a line to/from given point when counter is 0, and then successfully moves on to 1 and starts drawing an arm from those points. A similar trend is seen throughout the simulation.

Displays

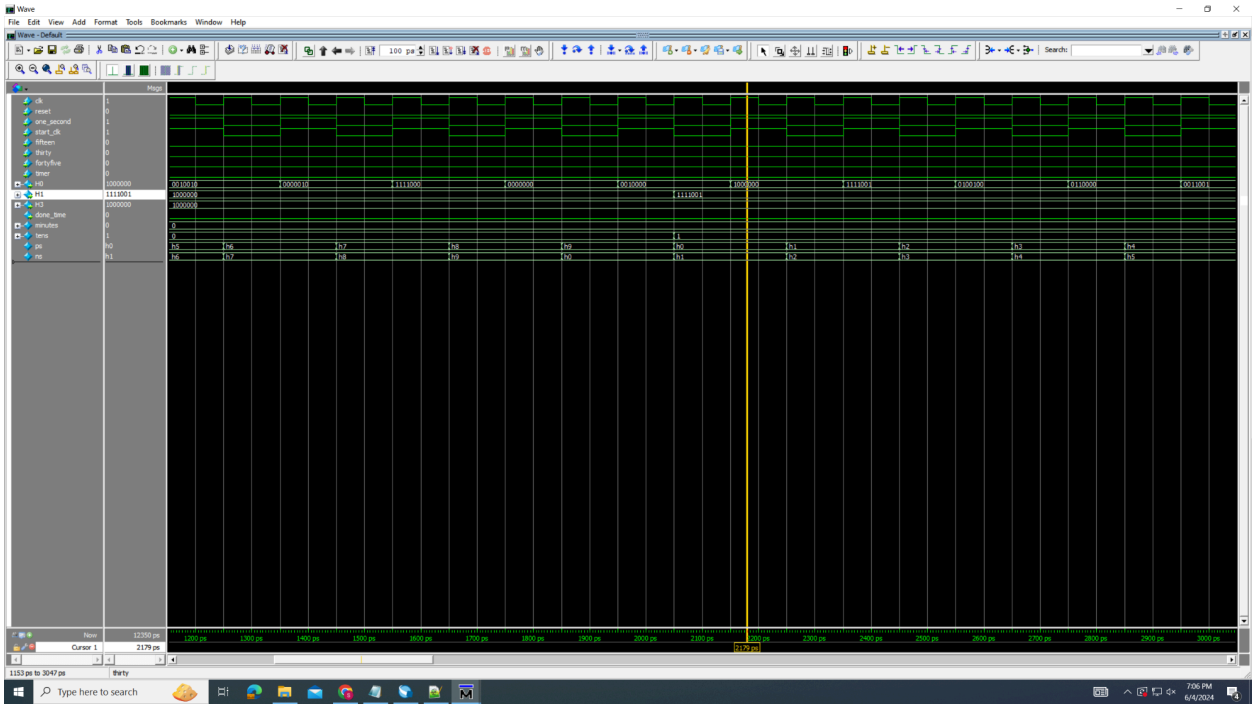

Figure 12: Simulation results for display_tb (for regular stopwatch)

The above screenshot is a snippet taken from the simulation of the testbench created for our display module. The main function of the display is to display correct digits on the HEXs of the FPGA (HEX[3] - HEX[0]) which shows the same stopwatch shown in analog form, in digital. The testbench was mainly created to verify whether the program is working correctly and increasing the time as we want. As seen, H0 (HEX[0]) displays each second, which increases the ‘tens’ variable (displays on HEX[1]) every 10 seconds, and increases the 'minutes’ variable (HEX[3]) every 60 seconds. Towards the end of the simulation H3 (minutes) is seen going high and

displaying a 1.

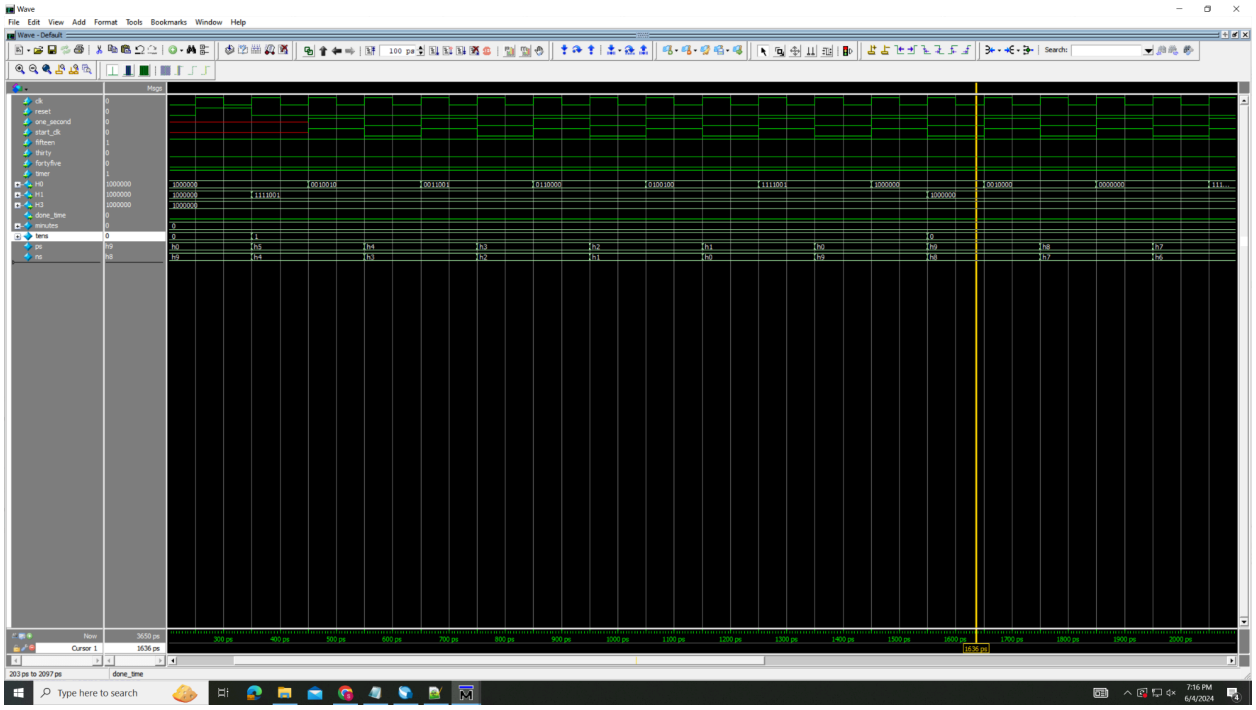

Figure 13: Simulation of results for display_tb (testing 15sec timer)

The above screenshot is another testbench result for displays_tb. With this simulation, it shows that the timer works well for our displays module. In the above screenshot, it is seen that the tens now start from 1 and ps starts from 5 displaying a 15 in HEX0 and HEX1. From then on, at every start_clk, the timer does its job and counts down from 15 to 0. After 5 seconds, it is seen that tens goes down to 0 and after another 10 seconds, it reaches 0 after which the program stops and done_timer signal is high indicating that the timer is done.

DE1_SoC

We do not have a testbench for our DE1_SoC due to the instance of the new clock_divider (uses CLOCK_50 as the main clock and the outputs being pulse signals at every 50 Million clock cycles). Since our clock divider uses 50 million clock cycles as 1 second, to get to see the simulation results for that 1 second is very difficult, let alone for multiple seconds.

```
input CLOCK_50;
output [7:0] VGA_R;
output [7:0] VGA_G;
output [7:0] VGA_B;
output VGA_BLANK_N;
output VGA_CLK;
output VGA_HS;
output VGA_SYNC_N;
output VGA_VS;

logic [10:0] x, y, x_body, y_body, x_arm, y_arm;
logic one_second, change_color, clk, reset, color, start_clk, fifteen, thirty, fortyfive, timer;

assign reset = SW[9];
assign start_clk = SW[8]; // start stop switch
assign clk = CLOCK_50;

// adding more ways for the user to interact with the program
assign fifteen = (SW[1] & SW[5]);
assign thirty = (SW[3] & SW[0]);
assign fortyfive = (SW[4] & SW[5]);
assign timer = (fifteen | thirty | fortyfive);

clock_divider clock (.clk, .reset, .one_second, .change_color);

always_ff @(posedge clk or posedge reset) begin
   if(reset)
      color <= 1;
   else if (change_color)
      color <= ~color;
end

VGA_framebuffer fb (
   .clk50          (CLOCK_50),
   .reset          (1'b0),
   .x,
   .y,
   .pixel_color    (color),
   .pixel_write    (1'b1),
   .VGA_R,
   .VGA_G,
   .VGA_B,
   .VGA_CLK,
   .VGA_HS,
   .VGA_VS,
   .VGA_BLANK_n    (VGA_BLANK_N),
   .VGA_SYNC_n     (VGA_SYNC_N));
```

Figure 14: First half of our DE1_SoC module

```
// assign HEX0 = '1;
// assign HEX1 = '1;
   assign HEX2 = 7'b1110110;
// assign HEX3 = '1;
   assign HEX4 = '1;
   assign HEX5 = '1;

   logic clk_done;

   always_ff @(posedge clk) begin
      if(clk_done) begin
         x <= x_arm;
         y <= y_arm;
      end else begin
         x <= x_body;
         y <= y_body;
      end
   end

   clock_body body (.clk, .reset, .x(x_body), .y(y_body), .clk_done);

   clock_arm arm (.clk, .reset, .x(x_arm), .y(y_arm), .one_second, .change_color, .start_clk, .fifteen, .thirty, .fortyfive, .timer);

   displays show (.clk, .reset, .H0(HEX0), .H1(HEX1), .H3(HEX3), .one_second, .start_clk, .fifteen, .thirty, .fortyfive, .timer);

endmodule
```

Figure 15: Second half of our DE1_SoC module

Flow Summary

Table of Contents

Flow Summary

<<Filter>>

Flow Status	Successful - Tue Jun 04 19:23:39 2024
Quartus Prime Version	17.0.0 Build 595 04/25/2017 SJ Lite Edition
Revision Name	DE1_SoC
Top-level Entity Name	DE1_SoC
Family	Cyclone V
Device	5CGXFC7C7F23C8
Timing Models	Final
Logic utilization (in ALMs)	658 / 56,480 (1 %)
Total registers	378
Total pins	96 / 268 (36 %)
Total virtual pins	0
Total block memory bits	309,432 / 7,024,640 (4 %)
Total DSP Blocks	0 / 156 (0 %)
Total HSSI RX PCSs	0 / 6 (0 %)
Total HSSI PMA RX Deserializers	0 / 6 (0 %)
Total HSSI TX PCSs	0 / 6 (0 %)
Total HSSI PMA TX Serializers	0 / 6 (0 %)
Total PLLs	0 / 13 (0 %)
Total DLLs	0 / 4 (0 %)

Color

Figure 16: flow summary of our top level module, DE1_SoC

Experience Report

Initially, in this lab, we tried to draw our circle for the analog clock by providing the exact coordinates. However, that approach did not work. We believe this was due to how the coordinates were being read. When we decided to use a .mif file, the circle appeared correctly, even though we used the same coordinates. With the .mif file, we noticed that our first coordinate for the tick mark was showing an extra line and was longer than it was supposed to be. Initially, we thought it might be an issue with the VGA or something beyond our control. However, upon reviewing our testbench, we realized it was an issue within our logic. Our testbench did not display our coordinates correctly, and it took us a couple of hours to debug. We discovered that the warnings about port sizes being truncated were causing the issue.

Since this lab is a project of our own, we learned just how vital it is to create a testbench early. Generally, we do not make our testbench until the very end when everything works and use the testbench as a final measure to confirm our program works. However, since it is more difficult to get guidance for lab 6, we learned that testbenching and programming before uploading it to labsland is easier than trying to write the program on our own.

We also tried to create a Python script to make the clock but ran into trouble. We first tried to find an image that looked like the clock we wanted, but once we ran the Python script, the clock was too small. Luckily, manually inputting the coordinates using RAM for the clock was easier for us to do. We later found out that there was a program we could use to draw our own clock instead of searching for an analog clock to use. However, we found that using Desmos and finding the coordinates that way was easier for us to implement. For next time, if we were making a game, using the Python script would be easier but to create something like a circle, we would most likely stick with using RAM.

The most challenging part of our project was implementing everything at different clocks, which was our first approach. Separating the clock cycle proved to be the most challenging part since we could not calculate a suitable parameter to put into the divider for one second. Also, changing the color of each arm of the clock was very difficult since we were using separate clocks and the input port for color in the VGA buffer could only take one value at a time. As a result, we could not get another parameter that would sync up perfectly with our given clock to represent 1 second as well as our clock_50. After trying to work on this issue for 3 days, we decided to start from scratch and make our own clock_divider which runs at an internal counter, and gives an output signal every second and a color change signal every half a second. These two signals are used throughout the project to move and animate the clock arm, thus receiving what we initially aimed for.

Partner Distribution

For this project, we programmed together in the lab. We met in person to brainstorm ideas on how to create the stopwatch program and then coded it together. One partner created the testbenches for the modules, while the other partner wrote the lab report for the testbenches. One challenge we faced was our schedules not matching with the provided office hours, which made debugging significantly harder. Otherwise, since we had been partners before, there were no significant challenges for this lab.

The amount of time this lab took is as follows:

- Reading & Understanding – 2 hour
- Planning – 7 hours
- Coding – 13 hours
- Testing – 13 hours
- Debugging – 30 hours

It took approximately 65 hours to finish this lab.