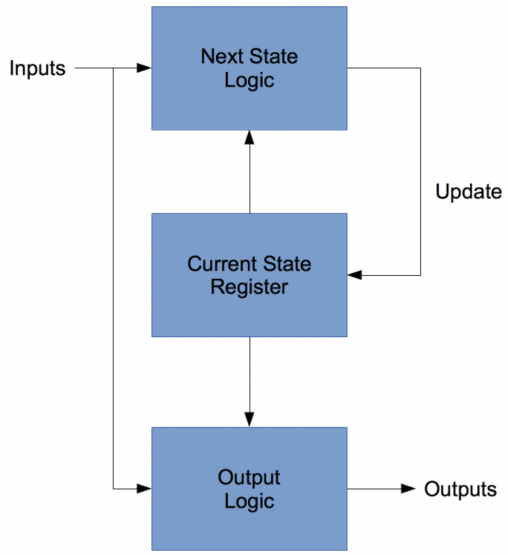
CS M152A Lab 3 Report

Hanna Co

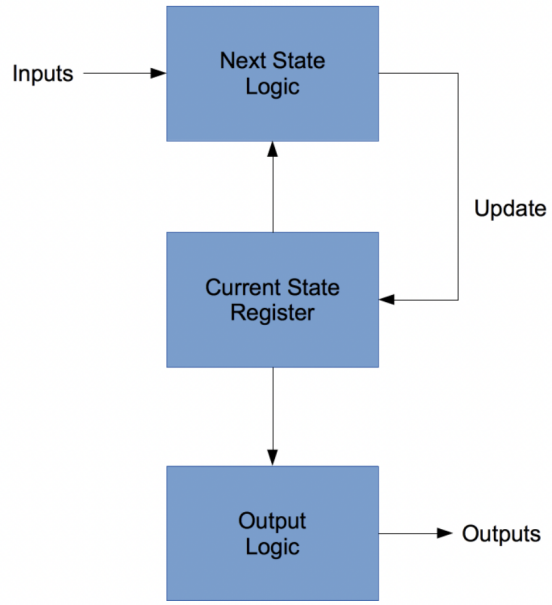
November 10, 2021

1 Introduction and Requirement

A finite state machine is a computational model that can be used to simulate sequential logic, and represent the control flow. In general, a finite state machine has three components: inputs, outputs, and state transition logic that depends only on the input and the current state. There are two types of finite state machines: Mealy machine and Moore machine. A Mealy machine's output depends on both the current state and the inputs. A change in the inputs can be immediately reflected in the output. Below is a diagram illustrating a Mealy machine.



A Moore machine is very similar to a Mealy machine, except that the outputs depend only on the current state. The outputs of a Moore machine always change one clock cycle after the input changes. The diagram below shows a simple Moore machine.



Although Mealy and Moore machines are very similar, they each have their advantages and disadvantages. A Mealy machine has low-latency output and has fewer states than equivalent Moore machines. A Moore machine has a clearer separation between inputs and outputs, can guarantee stable outputs, and is better for static timing.

For this lab, we were asked to design a stopwatch, and implement it on the Nexys3 Spartan-6 FPGA Board. The time would be displayed on the board's seven-segment display, and would take in both switch and button inputs.

The board can display four digits, thus we would be counting the seconds and minutes. For example, without any inputs, the board should display "0143" after one minute and 43 seconds. These numbers are displayed on the board's seven-segment display. We display these numbers by selecting the correct anode, and lighting up certain segments to display the number.

The time can be adjusted with the use of the "SEL" and "ADJ" switches, which we would implement. When the ADJ switch is high, the stopwatch would be in adjust mode. The portion of the clock would blink and increment at twice the normal rate, and the rest of the clock should be frozen. The table below shows the behavior of the stopwatch under ADJ inputs.

ADJ	**Action**
0	Stopwatch behaves normally
1	Stopwatch stops and '*Selected*' increases at 2Hz

The portion of the clock that is selected depends on the SEL switch input. When the SEL switch is low, the 'minutes' portion of the clock is in select mode; when the SEl switch is set to high, the 'seconds' portion of the clock is in select mode. This is represented in the table below.

SEL	Selected
0	Minutes
1	Seconds

In addition to the two switch inputs, there are two button inputs: Pause and Reset. Pause stops the counter when pressed, and restarts when pressed again. Reset sets all counters to 0. There is one issue when checking for the button press, and that is because of how sensitive the buttons are. Thus, we need to debounce the button press. We do this by creating a three-bit register that stores the value of the button on each rising edge of the clock. This way we can check if the button was pressed by checking current and previous values.

2. Design Requirement

The design of our clock required four different timings: a 1Hz clock for the time, a 2Hz clock and a clock for when the stopwatch was in adjust mode, and a very fast clock for displaying numbers. To implement these, we created a module that took a 100HMHz clock, as well as our clock's state as input, and output our desired clock signals. To do this, we created counter variables for each clock. When the counter reached a certain value, we would flip the value of the corresponding clock. For example, for our 1Hz clock, once the corresponding counter variable reached 49999999, we reset the counter to zero and flipped the clock's value. Additionally, since the clock would either move at 1Hz or 2Hz depending on the state, rather than handling which clock signal we would use, we simply had an output called `clk_sel`, which would be either 1Hz or 2Hz, depending on the state. The code segments below illustrate the counting for each of the clocks, as well as the logic for deciding the value of `clk_sel`.

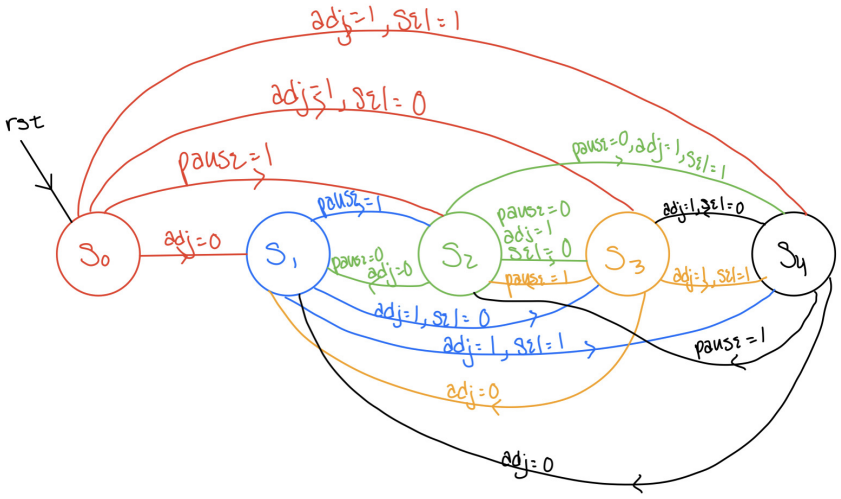
```
if (counter_1 == 49999999) begin
    counter_1 <= 0;
    clk_1 <= ~clk_1;
end
else begin
    counter_1 <= counter_1 + 1;
end

if (counter_2 == 24999999) begin
    counter_2 <= 0;
    clk_2 <= ~clk_2;
end
else begin
    counter_2 <= counter_2 + 1;
end

if (counter_fast == 49999) begin
    counter_fast <= 0;
    clk_fast <= ~clk_fast;
end
else begin
    counter_fast <= counter_fast + 1;
end

if (counter_adj == 9999999) begin
    counter_adj <= 0;
    clk_adj <= ~clk_adj;
end
else begin
    counter_adj <= counter_adj + 1;
end
```

```
if(state == STATE_S3 || state == STATE_S4) begin
    clk_sel <= clk_2;
end
else begin
    clk_sel <= clk_1;
end
```

The next module we implemented was digit, which would light up certain segments of the seven-segment display. It took in a number, the number we wanted to display, and `blink` as inputs. Based on the value of `blink`, we would either display the number or turn the display off, to make the input "flash" in adjust mode. It outputs values for each segment of the seven-segment display, which would display a number. To determine which segments to light up, we used simple sequential logic based on the number we wanted to display and the value of `blink`. If we wanted the segment turned on, we set its value to 0; otherwise we set it to 1. For example, to display the number 1, we wanted the segments B and C on, so we set their values to 0, and set segments A, D-G to 1. We did this for each digit 0-9, as shown in the code snippet below.

```
assign A = (number == 1) || (number == 4) || blink;
assign B = (number == 5) || (number == 6) || blink;
assign C = (number == 2) || blink;
assign D = (number == 1) || (number == 4) || (number == 7) || blink;
assign E = ((number != 2) && (number != 6) && (number != 8) && (number != 0)) || blink;
assign F = ((number != 4) && (number != 5) && (number != 6) && (number != 8) && (number != 9) && (number != 0)) || blink;
assign G = (number == 0) || (number == 1) || (number == 7) || blink;
```

The final module we implemented was display, which determined the stopwatch's behavior. It took in the clock, buttons, and switches as inputs, and output values for each anode and each segment of the seven-segment display. We first instantiated an instance of `clock`, to get our various clocks. We defined our five states: initial `STATE_S0`, regular `STATE_S1`, pause `STATE_S2`, adjust minute `STATE_S3` and adjust second `STATE_S4`. The transitions between the states are shown in the diagram below.



We initially set both current and next state to `STATE_S0`, the reset state. On every clock tick of the master clock, we append the value of each of the buttons and switches to its previous values, and check to see if any of the switches were flipped or if the buttons were pressed, and set the value of certain variables accordingly, For example, if our pause button was high and is now low, we set flip the value of `pause`, which either pauses or unpauses the clock. The logic for this can be seen in the code snippet below.

```
step_pause[2:0] <= {pauseBtn, step_pause[2:1]};
step_adj[2:0] <= {adjSw, step_adj[2:1]};
step_sel[2:0] <= {selSw, step_sel[2:1]};
if(step_pause[1] && ~step_pause[0]) begin
   pause <= ~pause;
end
if(step_adj[1] && ~step_adj[0]) begin
   adj <= 1;
end
if(~step_adj[1] && step_adj[0]) begin
   adj <= 0;
end
if(step_sel[1] && ~step_sel[0]) begin
   sel <= 1;
end
if(~step_sel[1] && step_sel[0]) begin
   sel <= 0;
end
```

In another `always` block, we are constantly checking the current state, and changing the next state based on the inputs. If we are in `STATE_S0`, the reset state, we check if `pause` is 1; if it is, our next state is the pause state, `STATE_S2`. Otherwise, if the adjust switch is not set to high, we go to the normal counting state, `STATE_S1`. If the adjust switch is set to high, we either go to the adjust minute or adjust second state, based on the value of the select switch. The logic is

essentially the same for each of the states: we check `pause`, `adj`, and `sel` to determine what our next state should be.

```
case(state)
    STATE_S0: begin
        if (pause) begin
            next_state <= STATE_S2;
        end
        else if (adj == 0) begin
            next_state <= STATE_S1;
        end
        else if (sel == 0) begin
            next_state <= STATE_S3;
        end
        else if (sel == 1) begin
            next_state <= STATE_S4;
        end
        else begin
            next_state <= state;
        end
    end
    STATE_S1: begin
        if (pause == 1) begin
            next_state <= STATE_S2;
        end
        else if (adj == 1 && sel == 0) begin
            next_state <= STATE_S3;
        end
        else if(adj == 1 && sel == 1) begin
            next_state <= STATE_S4;
        end
        else begin
            next_state <= state;
        end
    end
    STATE_S2: begin
        if(~pause && adj == 1 && sel == 0) begin
            next_state <= STATE_S3;
        end
        else if (~pause && adj == 1 && sel == 1) begin
            next_state <= STATE_S4;
        end
        else if (~pause && adj == 0) begin
            next_state <= STATE_S1;
        end
        else begin
            next_state <= state;
        end
    end
```

```
    STATE_S3: begin
        if(pause) begin
            next_state <= STATE_S2;
        end
        else if (adj == 1 && sel == 1) begin
            next_state <= STATE_S4;
        end
        else if (adj == 0) begin
            next_state <= STATE_S1;
        end
        else begin
            next_state <= state;
        end
    end
    STATE_S4: begin
        if(pause) begin
            next_state <= STATE_S2;
        end
        else if (adj == 1 && sel == 0) begin
            next_state <= STATE_S3;
        end
        else if (adj == 0) begin
            next_state <= STATE_S1;
        end
        else begin
            next_state <= state;
        end
    end
    default: next_state <= state;
```

The default case is just to ensure we don't run into any issues when setting the next state.

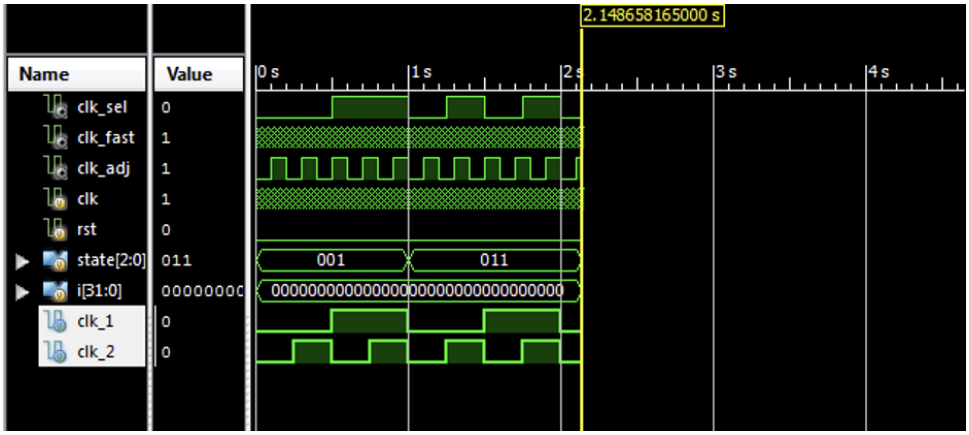
On each clock tick, we also increment the value of the digits that are supposed to change, based on the state. For example, in the regular state `STATE_S1`, all the digits change, but in the adjust minute state `STATE_S3`, we only change the values of `digit2` and `digit3`, the minute digits.

Next, we instantiate an instance of `digit`, and cycle through the anodes to display each digit on the board. We also check the current state, and pass in `blink` if the digit is selected. The final thing we do in this module is to make the digits flash when in adjust mode. To do this, we have a new variable called `shouldBlink`. If we are in one of the adjust states, `shouldBlink` flips on each cycle of `clk_adj`. Otherwise, it remains 0, indicating that we should not blink. When determining which segments should light up, if we are in the adjust state, we set `blink` to

`shouldBlink` and pass that into our `digit` module. In other states, we simply set `blink` to 0 and pass that in.

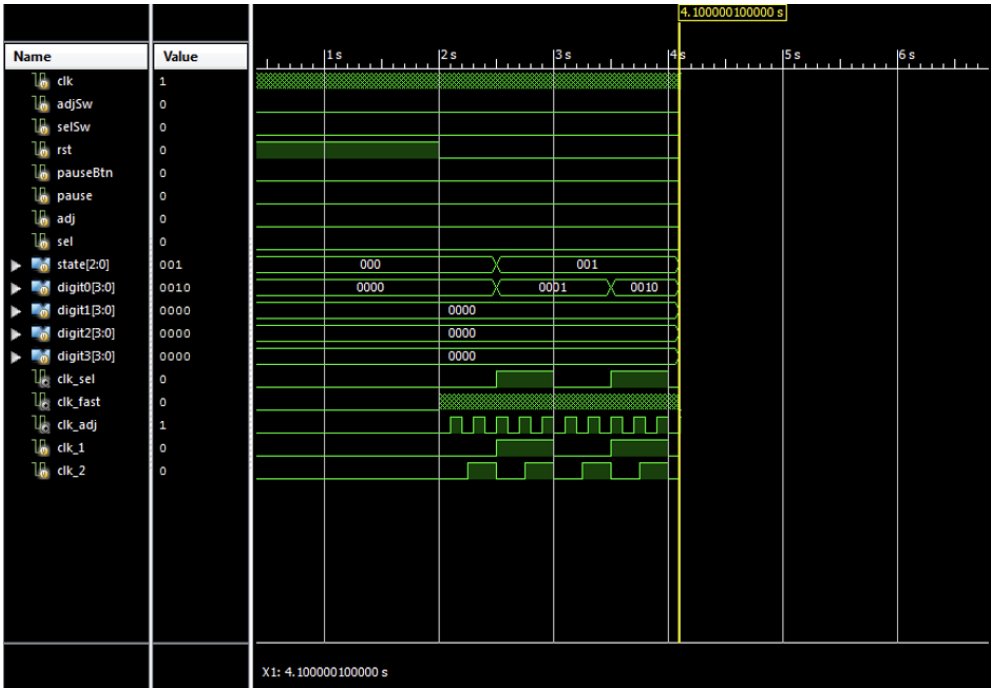
3 Simulation Documentation

Since the stopwatch was implemented on the board, we only created a couple testbench files, to ensure that our clocks, the 1Hz and 2Hz clocks in particular, were accurate. We also wanted to check that the right clock was being selected based on the state passed in. We tested this by setting the state to `STATE_S1`, letting the clock run, then setting the state to `STATE_S3`. Below is a screenshot of the waveform.

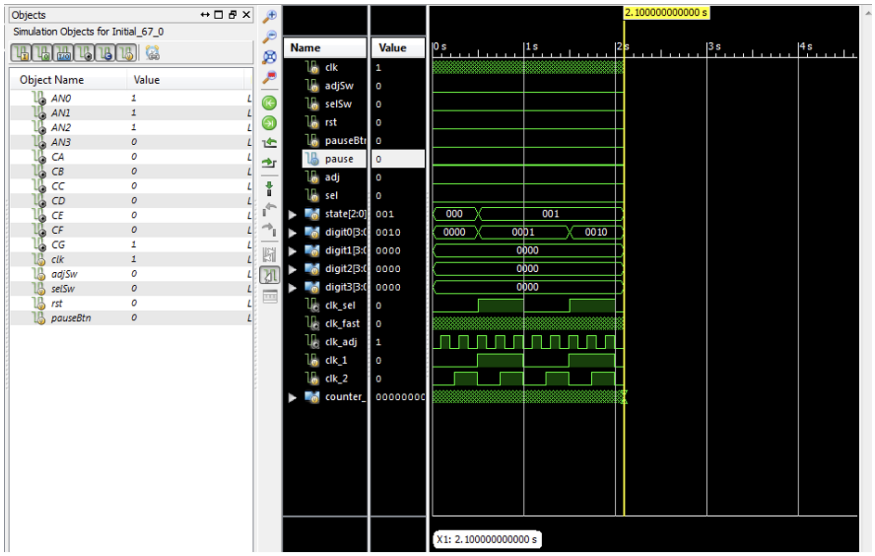


We can see that while `state = 1`, `clk_sel` matches the 1Hz clock. When the state transitions from 1 to 3, `clk_sel` takes on the value of the 2Hz clock. We can also see that `clk_fast` and `clk_adj` move faster than the 1Hz clock. Additionally, by looking at the waveform, we see that the 1Hz clock has a period of 1 second, and the 2Hz clock has a period of 0.5 seconds, as desired.

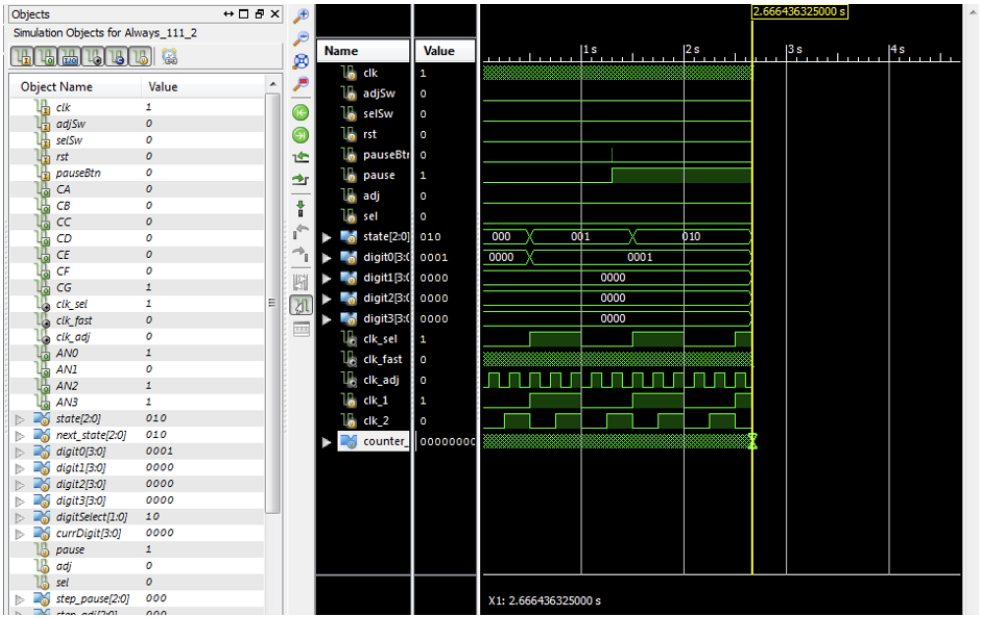
We also wrote testbench files to ensure that the clock and digits behaved correctly during each state. The waveform for `STATE_S0` is shown below.



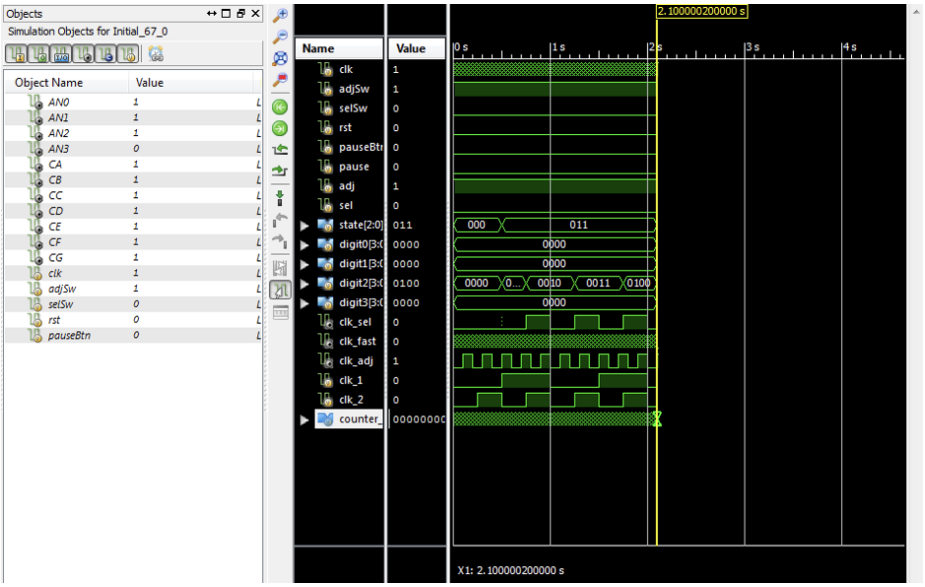
As desired, since `rst` is high, the digits and clock are zero. The next state we wrote a testbench for is `STATE_S1`, where the stopwatch should behave normally.

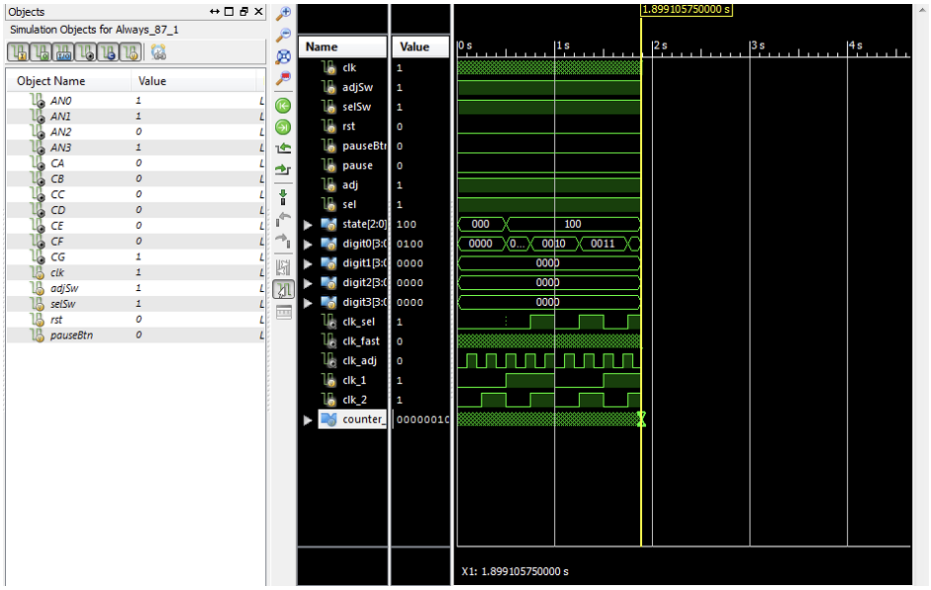


We can see that `clk_sel` takes on the value of `clk_1`, and `digit0` increments by 1 on the clock's rising edge, which is what we want. The next state we generated a waveform for was the paused state, `STATE_S2`.



In this waveform, we first begin the clock in `STATE_S1`, allowing it to increment as normal. After a bit, we "press" pause, and transition to `STATE_S2` on the next rising edge of the clock. The digits are then paused as well. The next two states we tested were our two adjust states, `STATE_S3` and `STATE_S4`. The waveforms are shown below.





As we can see, once we transition into an adjust state, the selected portion of the clock begins incrementing faster, while the unselected portion does not change.

4 Conclusion

In conclusion, we created a clock module that took in one master clock, and used that to create the other clocks we needed for our stopwatch. We also created another module for display numbers on the seven-segment display. Our main module, `display`, implemented all the logic for changing numbers, changing states, and detecting input from the board.

We ran into several problems when implementing the `digit` and `display` modules. At first, no numbers were displayed. However, we realized that this was because we hadn't uncommented the anodes in our .ucf file. Another problem we ran into was that rather than displaying 0000, the board was displaying ----. This turned out to be another small issue, where we had assumed that 1 meant the segment was lit up, when the opposite was true. The last implementation issue we ran into was how to change the clock based on the state. We initially tried having two `always` blocks, but this was not allowed in Verilog. In the end, we went with passing the state into our `clock` module, and deciding there which clock to output.

I found this lab to be the most challenging one we have done, but it was also interesting to implement. My partner and I came up with the implementation together, and wrote the modules together during lab hours. I felt that most of my contribution was during the initial brainstorming, when we were creating the finite state machine and the clock. The only recommendation I have for improving this lab is to give students more time to complete it. Many groups in my section did not finish by the deadline, and from what I heard, most TAs ended up

giving extensions as so few groups were done. Even with my partner and I coming in for extra lab hours and working outside of class, we barely finished by the original deadline.