ECSE 222: Lab 2 Report
Describing Sequential Circuits in VHDL

Group #48
Section 007
Dafne Culha (260785524)
Cheng Lin (260787697)

Table of Contents

I Introduction

This laboratory experiment gave us a better understanding of sequential circuits, Intel's Quartus Prime software, and the ModelSim software through the programming of a counter, clock divider, stopwatch. We used the VHDL language, Quartus Prime's Test Bench Template Writer, ModelSim, and an Altera board to produce the circuits required. The final product was a programmed Altera DE1-SoC board that counts centiseconds, seconds, and minutes and can be stopped, started, and reset using pushbuttons.

II Counter

The first circuit we designed was a 4-bit up-counter that counts values from 0 to 15. This counter accepts three inputs (an active high enable, an asynchronous active low reset, and a clock). This is a sequential circuit because the counter must store its value from the previous clock cycle to increment upwards each period. The output of the clock is a 4-bit vector that stores the count.

II.I Counter design

To begin, we created the entity declaration for the counter, specifying the four inputs/outputs of the circuit.

```
--declare counter entity with three boolean inputs (enalble, reset, clk)
--and one 3-bit vector output (count)

entity g48_counter is
   Port (enable : in std_logic;
         reset : in std_logic;
         clk : in std_logic;
         count : out std_logic_vector(3 downto 0)
         );
end g48_counter;
```

Next, in the architecture of the counter, we declared a temporary 4-bit vector, count_temp, that would store the count value from the previous clock cycle.

```
--declare architecture for counter
architecture behaviour of g48_counter is

   --create signal to hold current count value which will be assigned to the output vector count
   signal count_temp : std_logic_vector(3 downto 0) := "0000";
```

Finally, in the second part of the architecture (below), we created a process that includes the clock and reset inputs as variables on its sensitivity list. This means that the process will execute when there is a change in either of those values. The clock is on the sensitivity list because the counter should increment up every clock cycle; the reset value is on the list because it is an asynchronous signal.

Within the process, we first check if the reset input is active (equals 0) and restart the count_temp accordingly. Otherwise, if clock has a rising edge and our enable input is equal to 1, we increment count_temp by 1.

```
begin
   --declare a process block since this is a sequential circuit
   --define clk and reset in sensitivity list as variables we keep track of
   --all other variables are synchronized with clk so we don't list them
   Process(clk, reset) begin

      --check the value of reset first because it is an asynchronous signal (takes priority)
      if(reset = '0') then
         count_temp <= "0000"; --restart count if reset = 0

      --check for rising edge of clk
      elsif(rising_edge(clk)) then

         --check if enable is on (enable is active high)
         --add one to count if enable = 1
         if(enable = '1') then
            count_temp <= std_logic_vector(unsigned(count_temp) + 1);
         else
            count_temp <= count_temp; --if enable = 0, do nothing
         end if;
      end if;
   end Process;

   --assign count_temp to count
   --(this is outside process block because this wiring is always non-sequential)
   count <= count_temp;

end behaviour;
```

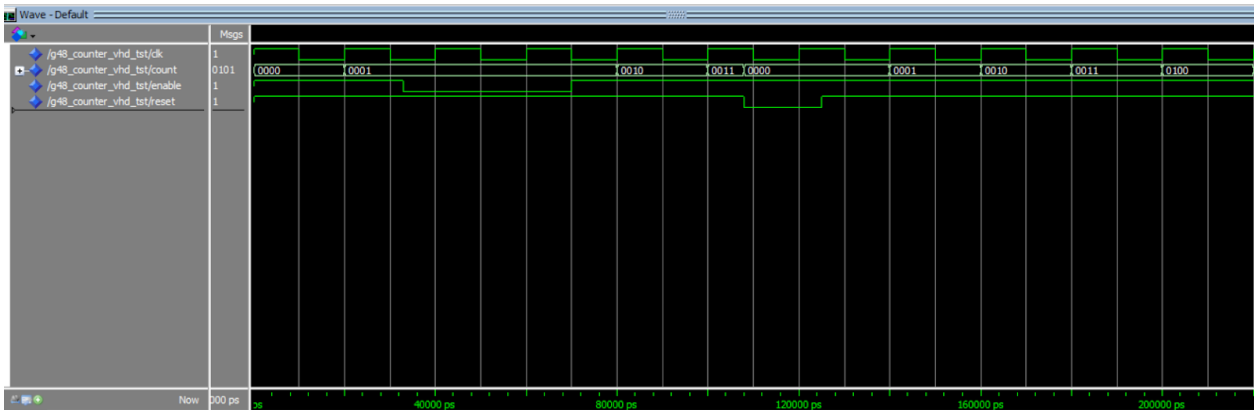
Lastly, outside of the process block, we connect the output with our intermediate count_temp signal.

II.II Counter simulation

We used ModelSim to test our counter and simulate the values of inputs and outputs over time. We compiled our .vhd file in Quartus Prime and added the same file to ModelSim. Then, we used Quartus Prime's Testbench Template Writer to create a testbench file and added simulation processes to run on ModelSim.

As shown below, we used the init process in the .vht file to loop over the values of clk every 20 ns. We programmed clk to flip every 10 ns because a clock with a period of 20 ns adequately mimics the Altera board's 50 MHz clock. The figure below illustrates the signal waveform generated by the .vht file on ModelSim. It can be seen that clk increments each cycle.

```
init : PROCESS
BEGIN
                clk <= '1';
                WAIT FOR 10ns;
                clk <= '0';
                WAIT FOR 10ns;
END PROCESS init;
```



We then used a clock_test process to test count, enable and reset functionalities of the counter. We set the initial values for reset and enable as ‘1’.

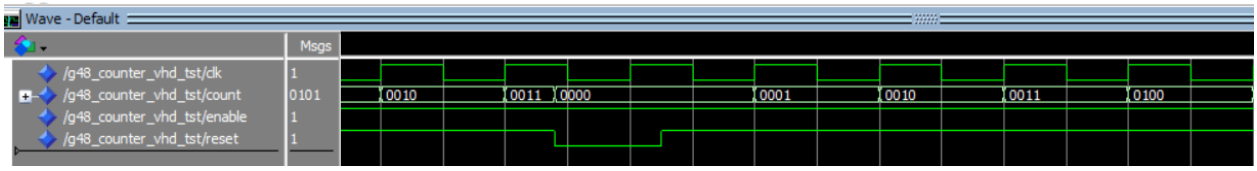
```
--test clock's reset, enable, and counting functionalities
clock_test : PROCESS
BEGIN
                --set initial values
                reset <= '1';
        enable <= '1';
                WAIT FOR 33ns;

                --test enable
                enable <= '0';
                WAIT FOR 37ns;
                enable<= '1';
                WAIT FOR 38ns;

                --test reset
                reset <= '0';
                WAIT FOR 17ns;
                reset <= '1';

                --test counting
                WAIT FOR 100ns;

WAIT;
END PROCESS clock_test;
```

The first part of this process tests the enable input and is shown below. It can be seen from the waveform that the enable input properly controls whether the count values are being incremented or not. When enable is l, the counter increments; when enable is 0, the counter does not change.

```
                --test enable
                enable <= '0';
                WAIT FOR 37ns;
                enable<= '1';
                WAIT FOR 38ns;
```

The second part of the process tests the reset input and up-counter overall. The waveform below illustrates that the reset is working properly because the count value becomes 0000 when the reset is low (equal to 0). When the reset value is high, the count value increments at the rising edge of the clk variable, confirming that the up-counter is functioning overall.

```
--test reset
reset <= '0';
WAIT FOR 17ns;
reset <= '1';

--test counting
WAIT FOR 100ns;
```



III Clock Divider

The second circuit we built was a clock divider: a down-counting circuit that asserts an output of 1 for every 500 000 cycles completed by an external clock. Effectively, given the DE1-SoC board's 50 MHz PLL clock, the clock divider would output a 1 every 10 milliseconds. It also accepts active high enable and active low reset inputs to pause and restart the divider's internal variables. Like the counter, the clock divider is a sequential circuit because it requires values stored from previous cycles to operate. In particular, each clock cycle saves its internal down-counting signal to be used in the following cycle's calculations.

III.I Divider design

To begin, we declared the clock divider's entity, which accepts an enable, reset, and clock input and returns an en_out variable (that will become 1 every 10 milliseconds). We also declare a generic time factor, T = 500 000 to be used when resetting the internal clock divider signal. The value of T was calculated using the FPGA board's clock frequency of 50 MHz and the fact that we want en_out to become 1 every 10 milliseconds.

```
--declare clock divider entity with three boolean inputs (enable, reset, clk)
--and one boolean output (en_out)

entity g48_clock_divider is
   --declare a time factor, T
   --Because 10 miliseconds elapses 500000 PLL events, a change in clk occurs every 500000 PLL events
   Generic (constant T : integer := 500000
         );

   Port (enable : in std_logic;
         reset : in std_logic;
         clk : in std_logic;
         en_out : out std_logic
         );
end g48_clock_divider;
```

For the architecture of the clock divider, we defined two internal signals: a T_temp to store the present value of the divider's down counter, and an en_out_temp to store the present value of the circuit's output. The T_temp variable is initiated at 499 999, the beginning of the counting sequence.

```
--create architecture for clock divider
architecture behaviour of g48_clock_divider is

   --create signal to hold current count value
   --create signal to hold temporary en_out value
   signal T_temp : integer := T-1;
   signal en_out_temp : std_logic := '0';
```

Next, we create a process that is sensitive to the clock and reset inputs. These variables are included on the sensitivity list because the clock divider needs to respond to any changes in those two variables. The reset button appears first in the logic because it is an asynchronous signal. When it equals 0, the T_temp value restarts its counting sequence from 499 999 again.

```
   Process(clk, reset) begin

      --check the value of reset first because it is an asynchronous signal (takes priority)
      --reset is active low
      if(reset = '0') then
         T_temp <= T-1; --if reset = 0, restart T_temp
         en_out_temp <= '0';
```

Alternatively, if the circuit reaches a rising clock edge, we first check if the enable input equals 1, and decrement T_temp if that is the case. We then check if T_temp has reached zero, and assert an output of 1 if it has. When T_temp becomes 0, we also restart the circuit with T_temp = 499 999.

```
      --check for rising edge
      elsif(rising_edge(clk)) then

         --check if enable is on (enable is active high)
         if(enable = '1') then
            T_temp <= T_temp - 1; --if enable = 1, decrease count
         else
            T_temp <= T_temp; --if enable = 0, do nothing
         end if;

         --check value of T_temp
         if(T_temp > 0) then
            en_out_temp <= '0';
         else
            en_out_temp <= '1'; --if T_temp = 0, output 1 and restart count
            T_temp <= T-1;
         end if;
      end if;

   end Process;
```

Lastly, at the end of the architecture, we assign the output variable en_out to the intermediate output signal, en_out_temp.

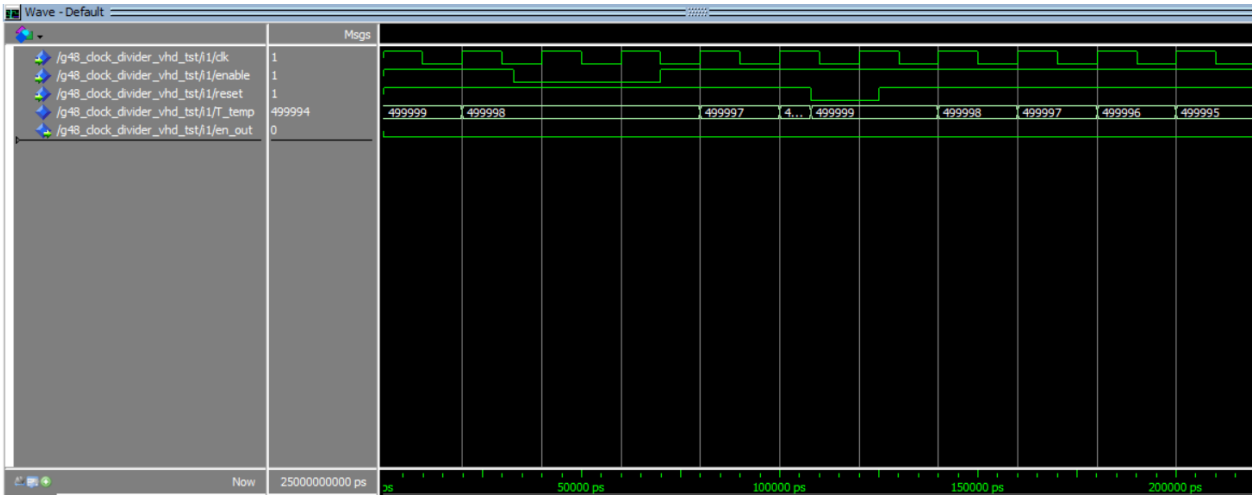
```
   end Process;

   --assign en_out_temp to en_out
   en_out <= en_out_temp;

end behaviour;
```

We designed the clock divider circuit using T_temp as the down-counter variable stored in our register. We chose to *decrement* T_temp rather than increment it since it makes the circuit simpler. We need to check the value of T_temp to decide if we assert an en_out of 1 or 0 at the end of each process loop. When we use a down-counter, we check if T_temp equals 0, whereas when we use an up-counter, we check if T_temp equals 499999. Obviously, a circuit that checks if T_temp equals 0 is simpler than a circuit that checks if T_temp equals 499999.

In terms of modularity, our down-counting clock divider circuit is easier to integrate with a clock input that may not be 50 MHz (like the Altera board's is), or a design configuration that requires a clock divider for every 100 milliseconds instead of every 10. In a down-counter, changing the generic constant T only changes what T_temp is reset to. Alternatively, if we were to use an up-counter circuit, changing the constant T would also change the upper limit we check for at the end of each process loop. This requires a change in the entire circuit that completes the check. Requiring such hardware changes makes an up-counting clock divider incompatible with other design needs.

III.II Divider simulation

We used ModelSim to test our clock divider and simulate the values of inputs and outputs over time. We compiled our .vhd file on Quartus, added the same file to ModelSim, and used the Quartus Testbench Template Writer to create an empty testbench file for the clock divider. We then added simulation processes to the .vht file and simulated it using ModelSim. A complete summary of our simulation plot can be found below.



Firstly, we used the init process below to run a clock loop every 20ns. The value of clk was set to flip every half period of 10 ns because the FPGA's clock works at a frequency of 50 MHz, or 20 ns per period.

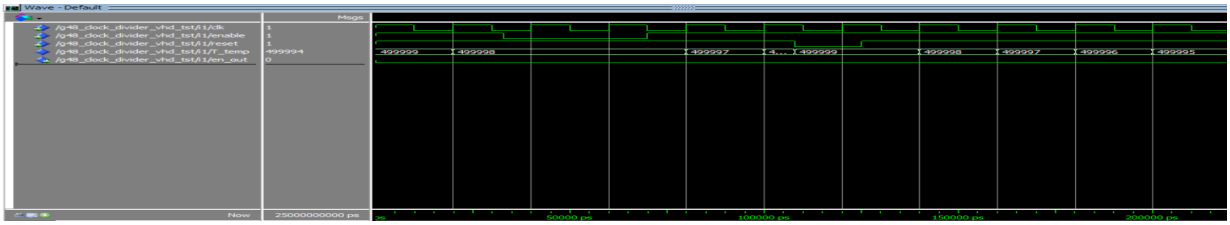
```
--init process to loop clock values
init : PROCESS
BEGIN
                clk <= '1';
                WAIT FOR 10ns;
                clk <= '0';
                WAIT FOR 10ns;
END PROCESS init;
```

Then, we used a clock_divider_test process to test enable, reset and counting functionalities of the clock divider. We set initial values of reset and enable as '1'.

```
clock_divider_test : PROCESS
BEGIN
                --set initial values
                reset <= '1';
        enable <= '1';
                WAIT FOR 33ns;
```

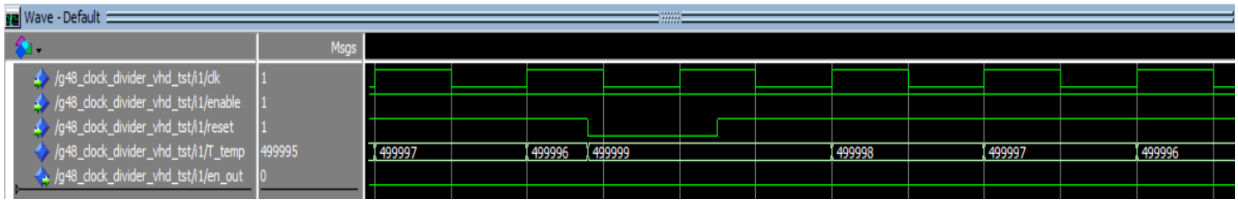
To test enable, we set its value to zero. Enable controls the storage of count values and it can be seen from the waveform that the count value (T_temp) is no longer decremented when enable is 0. Conversely, when we set enable back to '1', it can be seen that T_temp counts down.

```
                --test enable
                enable <= '0';
                WAIT FOR 37ns;
                enable<= '1';
                WAIT FOR 38ns;
```

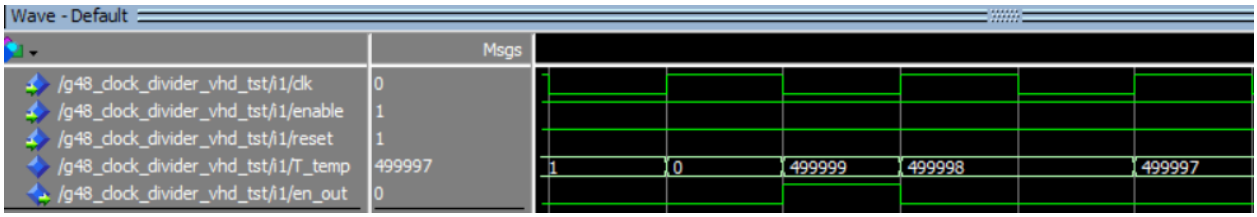


Secondly, to test the reset input and down-counter overall, we begin by setting the reset value as ‘0’. The waveform illustrates that the reset is working properly since count values become 499999 when reset is 0. When reset is 1, the T_temp value decreases at the rising edge of clk, as expected.

```
                --test reset
                reset <= '0';
                WAIT FOR 17ns;
                reset <= '1';

                --test counting
                WAIT FOR 20ms;

WAIT;
```



Finally, our simulation confirms that when T_temp becomes 0, en_out becomes 1 and T_temp is reset to 499999.



IV Stopwatch Circuit

The stopwatch circuit integrates the counter, clock divider, and 7 segment decoder components into a timer with start, stop, and reset inputs. The outputs of the stopwatch circuit are six 7-bit vectors that control one 7 segment LED display each. Two 7 segment displays correspond to centiseconds, two correspond to seconds, and two correspond to minutes (Figure 1). Furthermore, counter 0, 1, 2, and 4 count up to 9, while counter 3 and 5 count up to 5. Lastly, the start, stop, and reset buttons are active low and asynchronous.

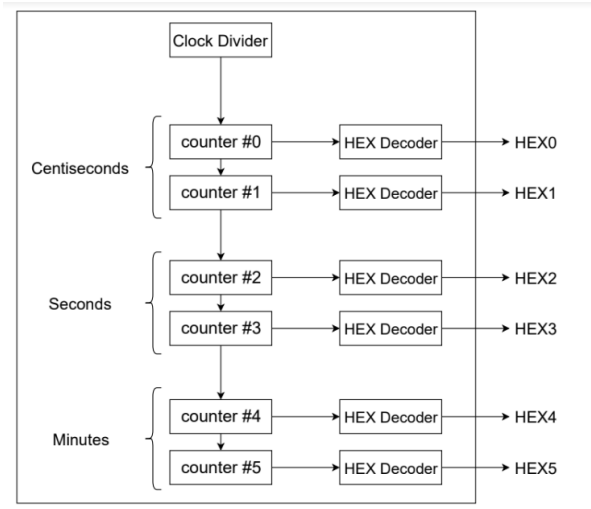


Fig. 1: High-level architecture of the stopwatch circuit.

IV.I Stopwatch design

The stopwatch circuit design begins with the entity declaration, which establishes the four stopwatch inputs (start, stop, reset, clk) and the six 7-bit vector outputs.

```
--entity declaration for stopwatch
--stopwatch takes in four inputs from FPGA board (start, stop, rest, and clk) and outputs to six 7-segment LEDs
entity g48_stopwatch is

    Port (start : in std_logic;
          stop : in std_logic;
          reset : in std_logic;
          clk : in std_logic;
          HEX0 : out std_logic_vector(6 downto 0);
          HEX1 : out std_logic_vector(6 downto 0);
          HEX2 : out std_logic_vector(6 downto 0);
          HEX3 : out std_logic_vector(6 downto 0);
          HEX4 : out std_logic_vector(6 downto 0);
          HEX5 : out std_logic_vector(6 downto 0)
          );
end g48_stopwatch;
```

Next, we import the clock divider, counter, and 7 segment decoder components into our architecture.

```
--create architecture for stopwatch
architecture behaviour of g48_stopwatch is

    --import the clock divider component
    component g48_clock_divider is
        Port(enable : in std_logic;
             reset : in std_logic;
             clk : in std_logic;
             en_out : out std_logic);
    end component g48_clock_divider;

    --import counter component
    component g48_counter is
        Port (enable : in std_logic;
             reset : in std_logic;
             clk : in std_logic;
             count : out std_logic_vector(3 downto 0));
    end component g48_counter;

    --import 7 seg decoder component
    component g48_7_segment_decoder is
        Port( code: in std_logic_vector(3 downto 0);
             segments: out std_logic_vector(6 downto 0));
    end component g48_7_segment_decoder;
```

We then instantiate a variety of signals: six signals that hold the temporary count values for each digit; an enable signal that will be controlled by the start/stop inputs; a signal for the clock divider's output; six signals that control the reset for each of digit; and five signals that control the enable for each digit (excluding digit 0).

```
--create signal to hold current count values for each digit (inputs to HEX0-HEX5)
signal count_temp0 : std_logic_vector(3 downto 0) := "0000";
signal count_temp1 : std_logic_vector(3 downto 0) := "0000";
signal count_temp2 : std_logic_vector(3 downto 0) := "0000";
signal count_temp3 : std_logic_vector(3 downto 0) := "0000";
signal count_temp4 : std_logic_vector(3 downto 0) := "0000";
signal count_temp5 : std_logic_vector(3 downto 0) := "0000";

--create signal to hold enable for stopwatch (controlled by the start/stop inputs)
signal enable_stopwatch : std_logic := '0';

--create signal to hold output of clock divider
signal en_out_divider : std_logic := '0';

--create signals for reset input of each counter (active low)
signal reset0 : std_logic := '1';
signal reset1 : std_logic := '1';
signal reset2 : std_logic := '1';
signal reset3 : std_logic := '1';
signal reset4 : std_logic := '1';
signal reset5 : std_logic := '1';

--create signals for enable input of each counter (active high)
--(not including counter0 because it takes in the stopwatch's enable input)
signal enable1 : std_logic := '0';
signal enable2 : std_logic := '0';
signal enable3 : std_logic := '0';
signal enable4 : std_logic := '0';
signal enable5 : std_logic := '0';
```

Next, we instantiate one clock divider and six instances of our counter circuit. We need six counters because we have six distinct digits, each keeping track of a different values in our counting sequence. In particular, one digit counts centiseconds, one counts deciseconds, one counts seconds, and so on. As a result, we need six components that store and increment count values independently of each other.

It is also key to note that while all counters use the *output* of the clock divider as their clock *input*, each counter has its own reset and enable inputs. The enable inputs of each counter become 1 when the previous digit reaches its maximum capacity and resets. This means that when one digit hits a 9 (or 5), the following digit will increment in the next cycle when the 9 (or 5) resets. The incrementing of the next digit allows us to store the carry out from the increase.

```
begin

   --create clock divider
   --clock divider is controlled by enable_stopwatch, rst, clk
   --it outputs en_out_divider which acts as the clock for all other digits
   clock_divider : g48_clock_divider PORT MAP(enable => enable_stopwatch,
                                              reset => reset, clk => clk, en_out => en_out_divider);

   --create 6 counters
   --each counter outputs its count to count_temp[x] variable
   counter0 : g48_counter PORT MAP(enable => enable_stopwatch, reset => reset0,
                                   clk => en_out_divider, count => count_temp0);

   counter1 : g48_counter PORT MAP(enable => enable1, reset => reset1,
                                   clk => en_out_divider, count => count_temp1);

   counter2 : g48_counter PORT MAP(enable => enable2, reset => reset2,
                                   clk => en_out_divider, count => count_temp2);

   counter3 : g48_counter PORT MAP(enable => enable3, reset => reset3,
                                   clk => en_out_divider, count => count_temp3);

   counter4 : g48_counter PORT MAP(enable => enable4, reset => reset4,
                                   clk => en_out_divider, count => count_temp4);

   counter5 : g48_counter PORT MAP(enable => enable5, reset => reset5,
                                   clk => en_out_divider, count => count_temp5);
```

Within the process block, we keep track of the clock, start, stop, and reset variables. Because start, stop, and reset are all asynchronous, we check those values in our control flow first. Start and stop control the enable of the clock divider and digit 0, which in turn control the enables of the other digits. Therefore, our start/stop control flow allows the user to pause and restart the entire stopwatch.

```
   --declare a process block since this is a sequential circuit
   --define clk, start, stop and reset in sensitivity list as variables we keep track of
   --all other variables are synchronized with clk
   Process(clk, start, stop, reset) begin

      --check value of start first because it is asynchronous (takes priority)
      --start is active low
      if(start = '0') then
         enable_stopwatch <= '1'; --if start = 0, enable stopwatch

      --check value of stop (active low)
      elsif(stop = '0') then
         enable_stopwatch <= '0'; --if stop = 0, turn enable off

      --check value of reset (active low)
      elsif(reset = '0') then
         reset0 <= '0'; --if reset =0, set all resets for each counter to 0
         reset1 <= '0';
         reset2 <= '0';
         reset3 <= '0';
         reset4 <= '0';
         reset5 <= '0';

```

If no asynchronous signals are active and the clock is at a rising edge, the control flow checks to see if any digits have reached maximum capacity and need to be cleared in the next clock cycle. Digits at 9 (or 5) with all of its previous digits also at capacity will trigger the enable input of the *following* digit to become active. (We turn on the next digit's enable before the current digit is reset because the next digit needs one full cycle with an active enable to increment.) Then, when the current digit hits 10 (or 6), it resets for the next clock cycle. Because the clock cycle follows the FPGA 50 MHz clock, the 20 nanoseconds the digit is at 10 (or 6) will not be observable to users.

Below is a sample of the control flow of the circuit. All digits follow a similar if-statement structure as the one for digit 0.

```
      --check for rising edge
      elsif(rising_edge(clk)) then

         --check if digit 0, 1, 2, 4, are 9 (need to be reset to 0)
         --similarly, check if digit  3 and 5 are 5

         --if digit 0 = 9, enable digit 1
         --(digit 1 will count up during this one cycle, then be shut off)
         if (count_temp0 = "1001") then
            enable1 <= '1';
         --the moment digit 0 hits 10, reset it
         --(the duration of one cycle where count_temp0 = 10 is too small for us to see)
         elsif (count_temp0 = "1010") then
            reset0 <= '0';
            enable1 <= '0';
         --otherwise keep reset = 1 and enable = 0
         else
           reset0  <= '1';
           enable1 <= '0';
         end if;

         --repeat same logic for digit 1
         --digit 2 is only enabled if digit 0 and digit 1 are at their max
         if (count_temp0 = "1001") and (count_temp1 = "1001") then
            enable2 <= '1';
         elsif (count_temp1 = "1010") then
            reset1 <= '0'; --reset when count_temp1 = 10
            enable2 <= '0';
         else
            reset1 <= '1';
            enable2 <= '0';
         end if;
```

Lastly, we instantiate the 7 segment decoders outside of the process block. Each decoder accepts a count_temp[x] vector and converts it to a 7-bit vector. These vectors are assigned to the outputs of HEX0 to HEX5.

```
   --assign inputs/outputs to decoders

   --decoder for counter 0
   decoder0: g48_7_segment_decoder PORT MAP(code => count_temp0,
                        segments => HEX0);
   --decoder for counter 1
   decoder1: g48_7_segment_decoder PORT MAP(code => count_temp1,
                        segments => HEX1);
   --decoder for counter 2
   decoder2: g48_7_segment_decoder PORT MAP(code => count_temp2,
                        segments => HEX2);
   --decoder for counter 3
   decoder3: g48_7_segment_decoder PORT MAP(code => count_temp3,
                        segments => HEX3);
   --decoder for counter 4
   decoder4: g48_7_segment_decoder PORT MAP(code => count_temp4,
                        segments => HEX4);
   --decoder for counter 5
   decoder5: g48_7_segment_decoder PORT MAP(code => count_temp5,
                        segments => HEX5);

end behaviour;
```

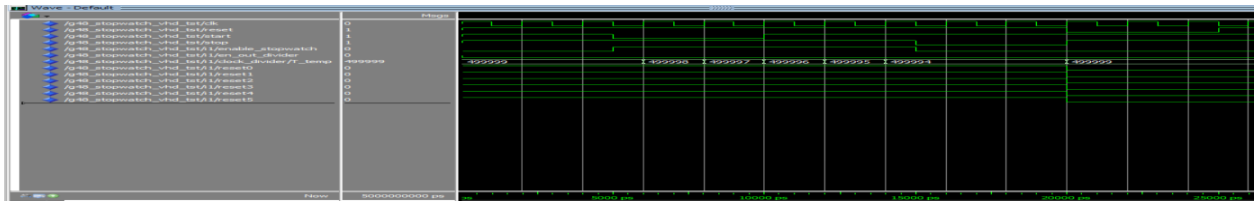
IV.II Stopwatch testing

We tested the stopwatch's features using both ModelSim and the behaviour of the FPGA board.

As with previous simulations, we used the init process of the .vht to run a clock loop every 20 ns. To test the start, stop, and reset buttons of the clock, we observed the T_temp variable during the simulation. Initially, we set the values of reset, start and stop to '1'.

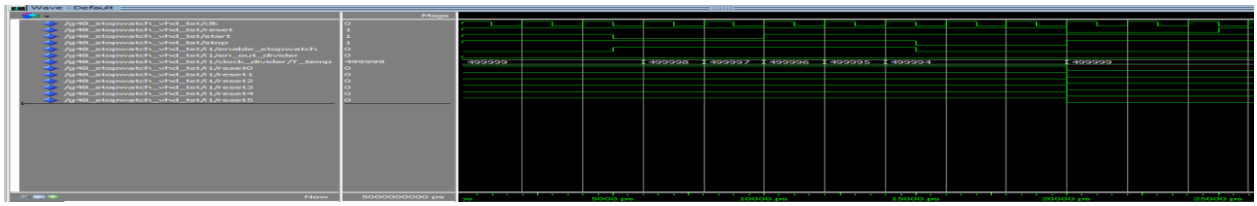
```
stopwatch_test : PROCESS
BEGIN
                --set initial values
                reset <= '1';
        start <= '1';
                stop <='1';

                WAIT FOR 5ns;
                --test start
                start <= '0';
                WAIT FOR 5ns;
                start <='1';
                WAIT FOR 5ns;

                --test stop
                stop <= '0';
                WAIT FOR 5ns;
                stop <='1';

                --test reset
                reset <= '0';
                WAIT FOR 5ns;
                reset <= '1';

                --test counting
WAIT;
```

Then, we observe that T_temp *begins* counting (and enable_stopwatch becomes 1) when start is set to 0.



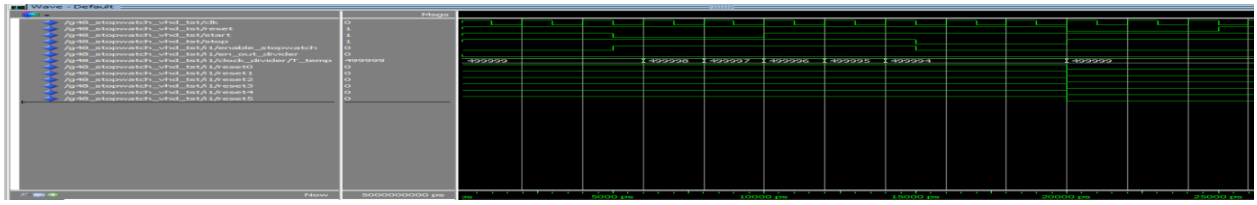
When we set the value of stop to ‘0’, we observe that T_temp *stops* counting and enable_stopwatch becomes 0.

```
                --test stop
                stop <= '0';
                WAIT FOR 5ns;
                stop <='1';
```

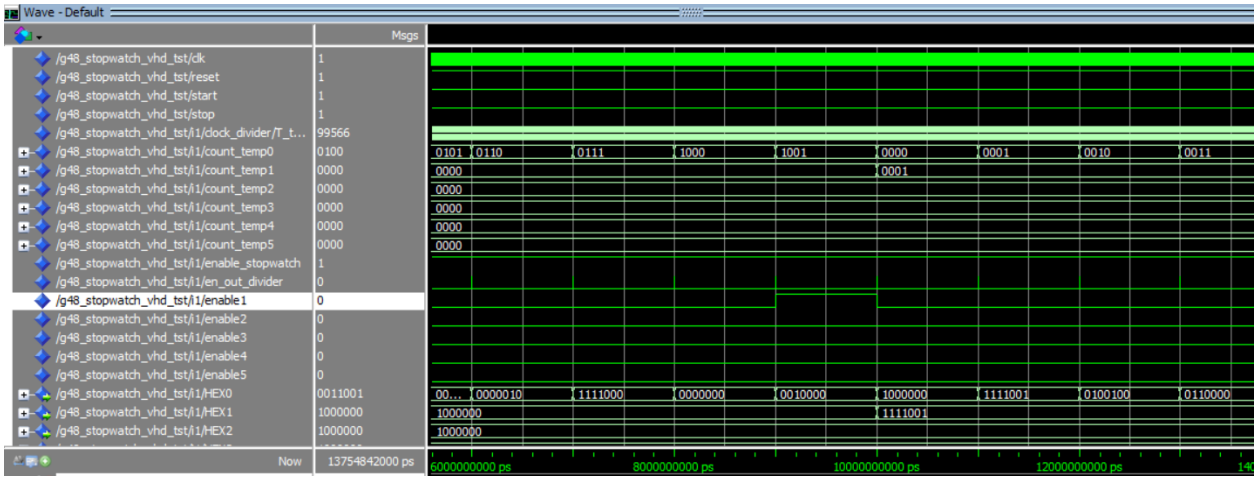


Lastly, we can see the T_temp value reset when the reset becomes low.

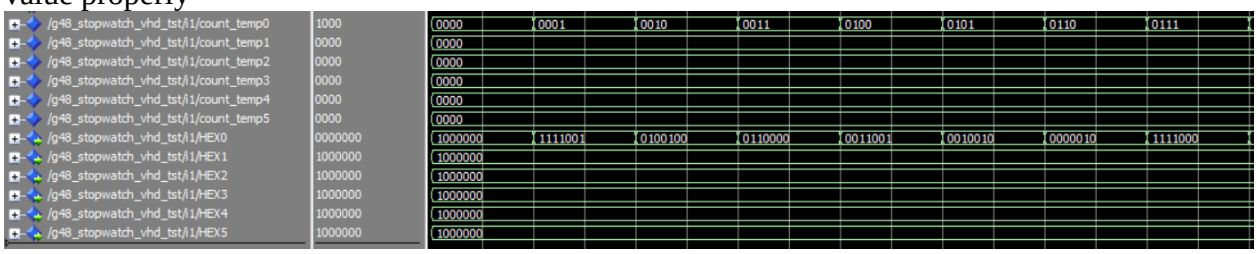
```
--test reset
reset <= '0';
WAIT FOR 5ns;
reset <= '1';
```



We then simulated each of the 6 counters on ModelSim to check that each consecutive digit incremented when the previous one reached capacity.



Finally, we simulated each HEX variable to ensure the decoders are decoding each temp[x] value properly



We also tested the inputs with a programmed Altera FPGA and verified the outputs displayed by the board. We pushed the start button and waited to make sure all the seconds and minutes were displayed correctly. We also confirmed that the reset and stop buttons worked properly.

V FPGA Resource Utilization

Flow Summary

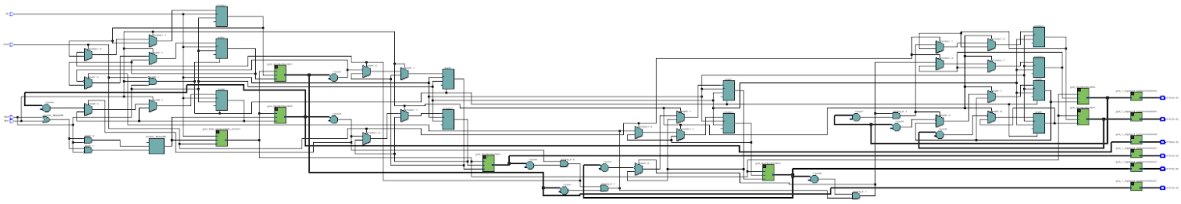
<<Filter>>

Flow Status	Successful - Wed Mar 20 15:56:18 2019
Quartus Prime Version	18.0.0 Build 614 04/24/2018 SJ Lite Edition
Revision Name	g48_lab2
Top-level Entity Name	g48_stopwatch
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	76 / 32,070 (< 1 %)
Total registers	85
Total pins	46 / 457 (10 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total DSP Blocks	0 / 87 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

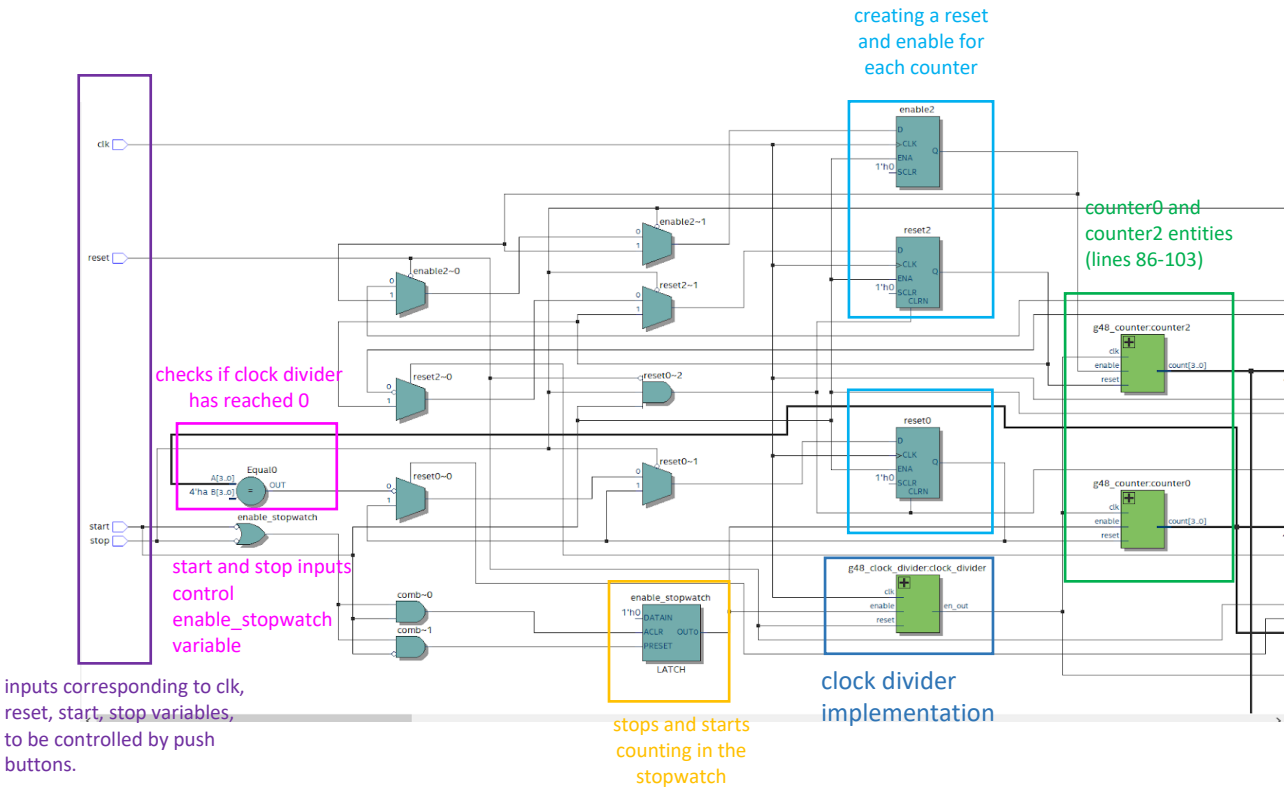
Logic utilization measures how much of the device is used to implement our circuit. As seen above, our compilation flow summary indicates that our logic utilization is less than 1%. This means that we have used very little of the board's total resources to implement our stopwatch.

VI RTL Schematic of stopwatch

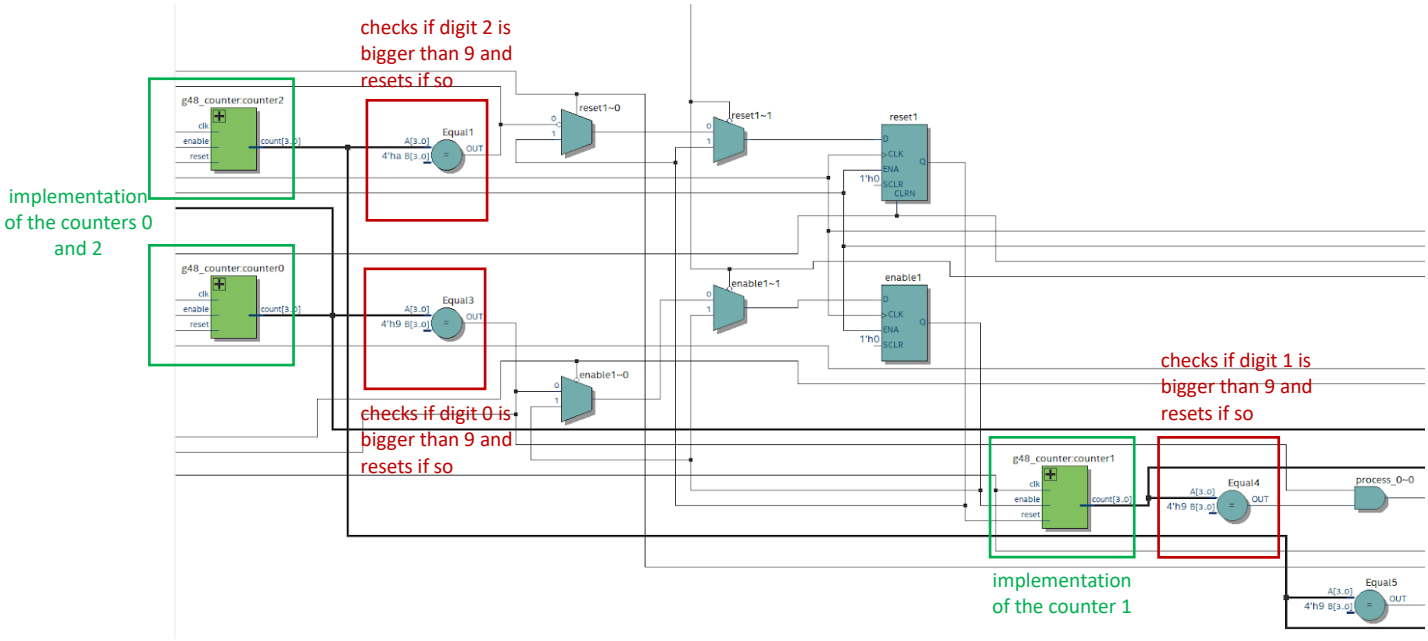
Below are screenshots on the stopwatch's RTL schematic. All the green boxes in the diagram are components. There is 1 clock divider, 6 counters, and 6 decoders. Since a lot of these components are sequential, there are many registers, each represented by blue rectangular boxes. Finally, the multiplexers correspond to the if statements in our code and they are used to determine when reset and enables should be turned on.

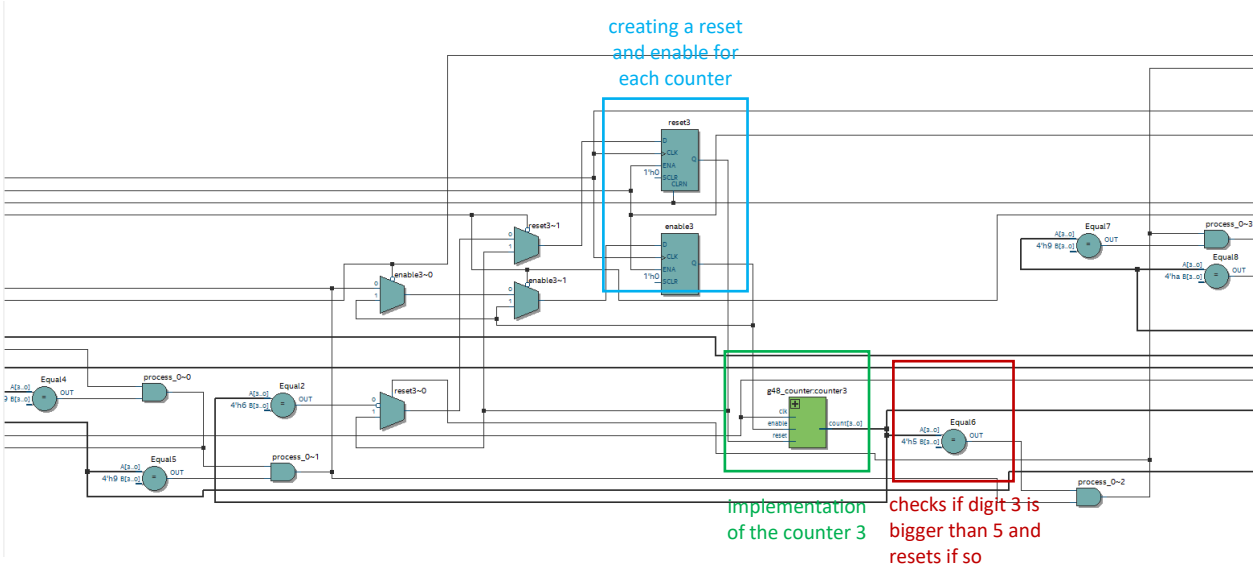


To go into further detail, the RTL schematic has been split up into four sections. In this first section, the schematic illustrates the inputs of the stopwatch, the circuitry through which start/stop control the enable variable (the first part of the stopwatch's process loop), the clock divider, and two counters.

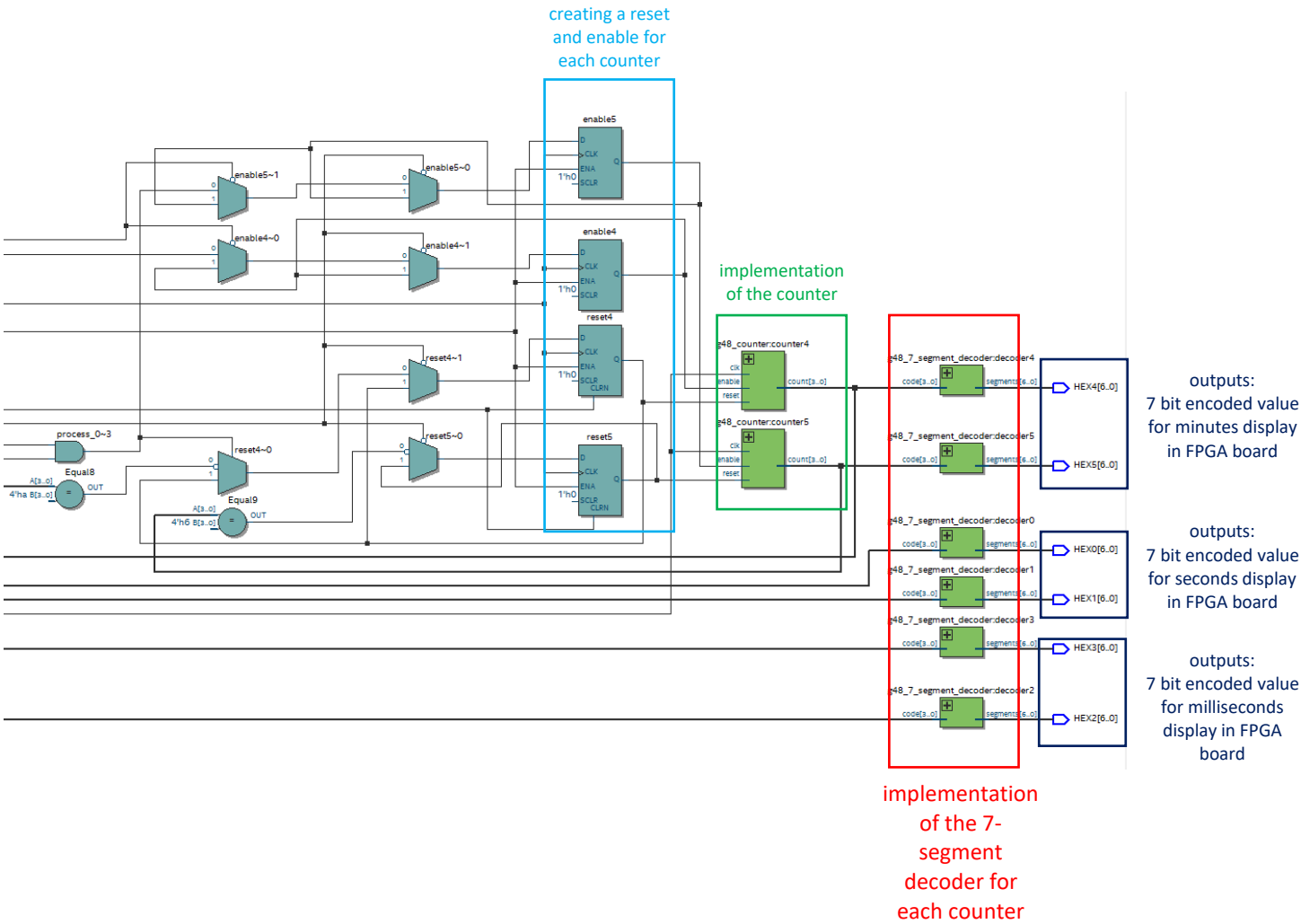


The second and third part of the schematic correspond to more counters and the registers that store their internal values. The components right after the counters labelled with red text check if the counters have reached their capacity of 9 (or 5) and need to be reset (lines 104-169 of the stopwatch .vhd file).





Lastly, the end of the schematic contains two more counters (for a total of 6) as well as the six 7-segment decoders that connect to the 7-segment LED displays. The port map for the 7-segment display inputs and outputs are at lines 177-194 of the stopwatch .vht file.



VII Conclusion

For this laboratory experiment, we designed a counter and a stopwatch in VHDL and simulated its components using ModelSim. We also downloaded our program onto an Altera board to physically test the circuit. Overall, this experiment gave us a better understanding of sequential logic circuits.