# ECSE 222 Digital Logic
# Lab Assignment II

Group 62
Antoine Wang 260766084
Yicheng Song 260763294

- **Description of the counter and clock divider circuits. Explain why these two circuits are considered as sequential designs.**

In our implementation, the counter circuit functions basically as a modulo-n up-counter described in the lecture slides. Although the *CAD* tool does not organize the circuit and the components in the same way, the behavior of the counter is defined following the exact same logic. The key property is defined in the "process" block. It takes in the clock signal "clk" and the reset signal as conditions. Then, there are 3 cases which defines all functions required for the counter. The first case is when reset is '0' (reset is active-low), the current value of the counter will be set to zero. Then, the following "elsif" states that when the counter does not reach the boundary condition, the value should simply keep adding up if the enable is '1' or stays the same when enable is '0'. The "inner_En" signal, which is the rippled enable signal between counters, should also be set to non-active state since there is no need to boost up the higher bit yet. Lastly, the second "elsif" will be enter when the value of the counter is already the largest and "carry out" will be produced. At the positive edge of the clock, if the enable is '1', the counter will reset into zero and produce an active enable-out signal. If enable is '0', the counter will hold the state and nothing active is generated to the next counter.

```
-------------------------------------library declaration of the unCounter entity---------------------------------
library ieee;
use ieee.std_logic_1164.ALL;
use ieee.std_logic_unsigned.ALL;
use ieee.numeric_std.ALL;

-------------------------------------Declaration of the up counter variables-------------------------------------
entity upNCounter is
   Port(enable : in  std_logic;
        reset  : in  std_logic;
        clk    : in  std_logic;
        radix  : in  std_logic_vector(3 downto 0);    --Radix: An input defines the upper bound (modulo) of the upcounter
        en_out : out std_logic;                       --out enable, will be used as ripples to the next counter
        count  : out std_logic_vector(3 downto 0));   --output indicating the current state of the up counter
end upNCounter;

-------------------------------------Architecture of the up counter-----------------------------------------------
   architecture counter of upNCounter is
      signal temp : std_logic_vector(3 downto 0);
      signal inner_En : std_logic;
   begin
      process(clk, reset)
      begin

         if    (reset = '0')        then temp <= "0000"; --current state set to zero


         elsif (rising_edge(clk) and temp /= radix)then

                  if (enable='1')   then temp <= temp + 1;
                                    else temp <= temp;       --when state is under the boundry
                                    end if;                  --keep adding the state and keep the output to be zero
                  inner_En <= '1'; -- will be zero since the output is NOTed

         elsif(rising_edge(clk) and temp = radix) then

                   if(enable='1')   then temp <= "0000";     --when state reaches boundry
                                    inner_En <= '0';         --reset the state to be 0
                                    else temp <= temp;
                                    inner_En <= '1';         -- output set to 0 (will be NOTed to 1 in the end)
                                    end if;

         end if;



      end process;
      en_out <= not inner_En; -- NOT output
      count <= temp;          -- Get the current state(for decoders and LEDs)
   end counter;
```

***Figure 1. VHDL code for a modulo-n up counter***

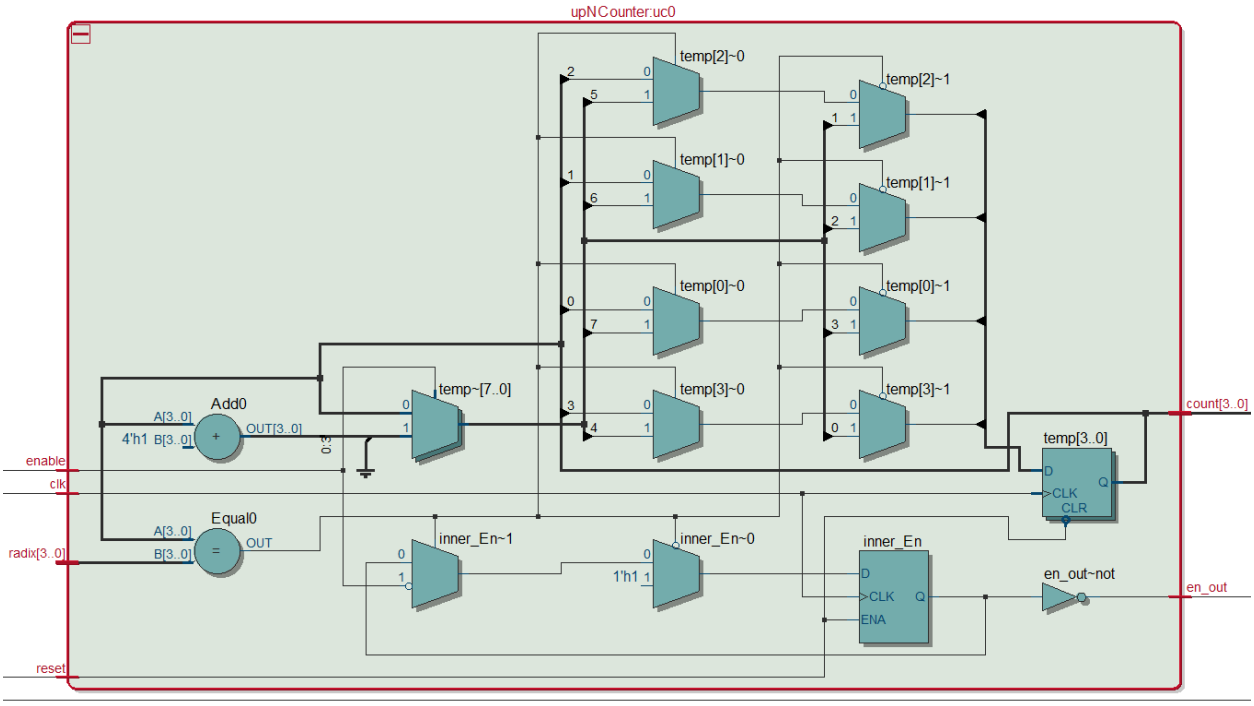


***Figure 2. RTL schematic diagram of the modulo-n up counter***

The clock divider is essentially a down counter that has a specific threshold to scale the frequency of the DEC board input into the frequency favored by us. Thus, the clock divider, in terms of behavior, follows the same logic as up counter. The only changes are first, there is a specialized threshold, $111101000010001111_{(2)}$ which equals to $499999_{(10)}$. From 0 to 499999 there are 500000 periods, which can convert the original period of the DEC board (20ns) into the 0.01s. Secondly, during every positive edge of the clock signal, if the enable is active the value will decrease by one at each time. When the value counts down to zero, it will be scaled back to 499999 which a carry-out enable produced.

```
----------------------------------library declaration of the clocker devider entity---------------------------------
library ieee;
use ieee.std_logic_1164.ALL;
use ieee.std_logic_unsigned.ALL;
use ieee.numeric_std.ALL;

----------------------------------Declaration of the clocker devider variables-------------------------------------
entity clock_divider is
    Port(enable : in  std_logic;
         reset  : in  std_logic;
         clk    : in  std_logic;
         en_out : out std_logic);
end clock_divider;

-----------------------------------Architecture of the clocker devider ---------------------------------------------
    architecture logic of clock_divider is
       signal temp : std_logic_vector(18 downto 0);
       signal innerEnable : std_logic;
    begin
       process(clk, reset)
       begin
          if     (reset = '0')         then temp <= "11110100001000111111"; -- Threshold is 499999
          elsif (rising_edge(clk) and (temp/= 0))   then                    -- 20ns* 500000 = 0.01s which is the prefered period for the stopwatch

                if (enable='1')    then temp <= temp - 1;
                                   else temp <= temp;
                                   end if;                                  -- Implementation information please refer to upNcounter
                                                                            -- the clock divider follows the same idea
                innerEnable <= '1';                                         -- The counting down process is symmetrical
          elsif ((rising_edge(clk)) and (temp = 0)) then
                if (enable='1')    then temp <= "11110100001000111111";
                                   else temp <= temp;
                                   end if;
                innerEnable <= '0';
          end if;
       end process;
       en_out <= not innerEnable;
    end logic;
```

***Figure 3. VHDL code for a clock divider***

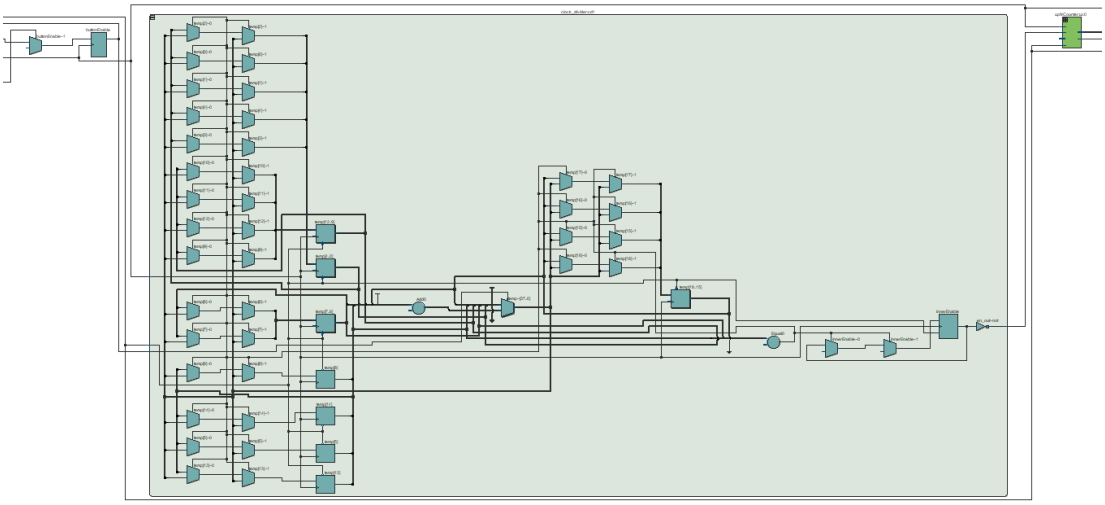

***Figure 4. RTL schematic diagram of the clock divider***

The reason why two circuits are considered as sequential designs is that the outputs of both circuits depend not only on the input, but also on the previous state. Both circuits can produce a carry-out enable signal. The enable is only active when first, the previous value in the counter is at boundary and second, the input enable is active. Another reason is that both circuits can store their current state. If the input enable is non-active, both circuits will simply latch on the current value without making any increments or decrements. From both RTL diagrams, it is evident to find that D flip-flops, which are examples of sequential circuits, are used frequently as memory elements.

**• Explain why even though we could build a clock divider using an up-counter it is easier to build the divider using a down-counter.**

The reason why a down counter is preferred during the actual implementation is because the output logic is simpler and of lower cost. Slide 25 from lecture 25 (figure 2.1 below) gives an example of how to implement a modulo-6 up counter. Note that at the output, an AND gate connects to the most significant bit Q2 and least significant bit Q0. The reason why the AND logic is a valid check for boundary cases is that only the value 5 has Q2 and Q0 both at one. Thus, as long as the two selected bits generate a active signal the counter will be reloaded into $000_{(2)}$ to start another round of counting.

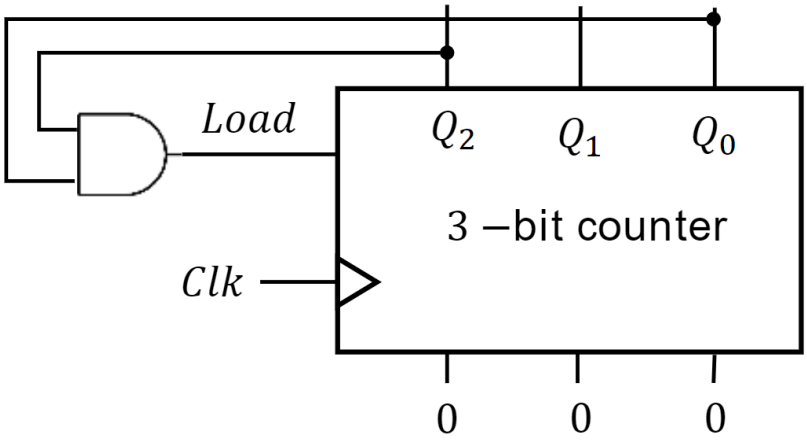


***Figure 1. An example of output logic implementation from Lecture 25 (S25), credit to Prof. I. Psaromiligkos***

Similarly, in the clock divider circuit that we implemented, if a down counter is chosen then the boundary case which is checked is $0000000000000000000_{(2)}$ which can be handled by a fairly simple implementation. However, if an up counter is chosen, the logic circuit will have to check if the binary representation is exactly $11110100001000111111_{(2)}$. Considering the cost and the efficiency, the down counter is definitely a better choice given that it minimized the burden on the circuits without compromising the functionality.

**• A discussion of how the counter and clock divider circuits were tested, showing representative simulation plots. How do you know that these circuits work correctly?**

We test the counter by giving inputs to simulate the real conditions and comparing the simulated results with our expected ones.

```
init : PROCESS
-- variable declarations
BEGIN
   clk <= '0';
wait for 10 ns;
 clk <= '1';
wait for 10 ns;

END PROCESS init;
```

***Figure 1. Simulation of clock signal.***

We set the clock signal with frequency of 50 million to simulate the real one by setting the value of clock value changing in period of 10 nanoseconds. The initial value of temp is set as "0000", reset is set as "0" and enable is set as "1". By doing so we can check whether the reset works. As reset is active low, the output value should be "0000" for all time. If this is not the case then there should be something wrong with the circuit. After 300 nanoseconds, we change the value of reset to "1", which means the counter should start working from that moment.

```
BEGIN
       temp <= "0000";
      reset <= '0';
enable <= '1';
wait for 300 ns;
reset <= '1';
```

***Figure 2. Initial conditions of reset, enable, and temp.***

The simulation plot is just the same as our expectation. The output count doesn't change during the period when reset is "0", and the counter circuit works correctly when reset is set as "1": the increment of output is 1 at every rising edge of the clock signal, and the output is "0000" after "1111".

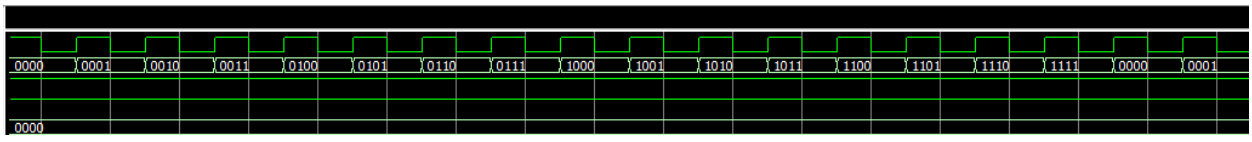


***Figure 3. Simulation plot of the up-counter circuit.***

The simulation of clock divider is similar with that of counter. The clock signal is just the same as the previous one. The reset is also set as "0" at the first 300 nanoseconds and then set to "1". The initial condition of temp is set to "1111010000100011111", which refer to 499999 in digital, and the innerEnable is set to "1".

```
always : PROCESS
BEGIN

temp <= "1111010000100111111";
innerEnable<= '0';
reset <= '0';
enable <= '1';
wait for 300 ns;
reset <= '1';       -- code executes for every event on sensitivity list
WAIT;
END PROCESS always;
```

***Figure 4. Initial conditions for innerEnable, reset and enable.***

If the circuit works as expected, then after the first 300 nanoseconds the output en_out should turn to "1" once every 0.01 second and during the period the en_out should be "0".

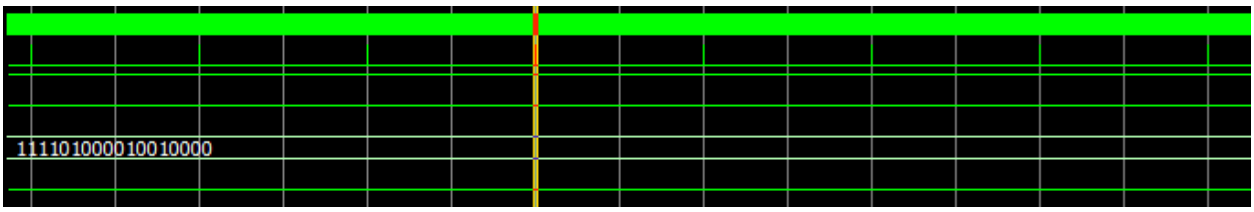


***Figure 5. Simulation plot of clock divider circuit.***

As is shown in the simulation plot, the first row refers to the clock signal and the second row represents the output en_out, which turns to "1" on the integer multiples of 0.01 second and then return to "0" on all other time, which means the circuit works just as expected and meets all the requirements.

**• A description of the stopwatch circuit. Explain why you created six instances of the counter circuit in your design and why?**

There are 3 inputs which refers to the state of the button start, stop and reset, an input for clock signal and 6 instances from HEX0 to HEX5 each represent a 7-segment code for every digit of the stopwatch as outputs.

```
entity g62_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 g62_stopwatch;
```

***Figure 1. Declaration of all inputs and outputs.***

The 6 instances of 4-bit binary numbers which represent the number of every digit on the stopwatch are first declared. At the same time, we declare an instance, buttonEnable, for storing the state of the button, and an instance innerEnable for storing the en_out produced by every stage of the upNcounter, which functions as a up-counter with radix.

```
signal buttonEnable : std_logic;
signal innerEn: std_logic_vector (6 downto 0);
signal LEDbit0: std_logic_vector (3 downto 0);
signal LEDbit1: std_logic_vector (3 downto 0);
signal LEDbit2: std_logic_vector (3 downto 0);
signal LEDbit3: std_logic_vector (3 downto 0);
signal LEDbit4: std_logic_vector (3 downto 0);
signal LEDbit5: std_logic_vector (3 downto 0);
```

***Figure 2. Declarations of all signals exist in the circuit.***

Then the upNcounter, decoder and the clock_divider are imported.

```
component upNCounter  -- Declare upNCounter componenent
   Port(enable : in  std_logic;
      reset  : in    std_logic;
      clk    : in    std_logic;
      radix  : in    std_logic_vector(3 downto 0);
      en_out : out   std_logic;
      count  : out   std_logic_vector(3 downto 0));
end component;

component encoder                    -- Declare decoder componenent
port    (   segmentcode : in  std_logic_vector(3 downto 0);
            segments    : out std_logic_vector(6 downto 0));
end component;


component clock_divider              -- Declare clock divider componenent
      Port(enable : in  std_logic;
         reset : in  std_logic;
         clk      : in  std_logic;
         en_out   : out std_logic);
end component;
```

***Figure 3. Imports of all components.***

A clock divider stage cd1 is first constructed for the basic time dividing. It takes the state of button buttonEnable as enable, inputs reset and clk as reset and clock signal, and produce an innerEn(0) as innerEnable which represent the divider of time.

```
cd1: clock_divider port map(buttonEnable, reset, clk, innerEn(0)); --map to a clock divider
```

***Figure 4. Map to a clock divider.***

Then 6 stages of upNcounter are mapped. Take the first stage, uc0, as example: The innerEn(0) produced on the clock_divider stage is taken as the inputs enable in the upNcounter. As 1 second equals 100 of 0.01 seconds, the radix of the last two digits of the stop watch should both be 10, which means the input radix should be "1001". As a result, a 4-bit binary signal and an innerEn(1) will be produced. The 4-bit binary signal is going to be decoded to a relative 7-segment code by the decoder stage, and the innerEn(1) will work as enable for the next upNcounter stage.

```
uc0: upNCounter port map (innerEn(0),reset , clk, "1001", innerEn(1),LEDbit0 ); --map to a mod-10 (0-9) up counter
uc1: upNCounter port map (innerEn(1),reset , clk, "1001", innerEn(2),LEDbit1 ); --map to a mod-10 (0-9) up counter
uc2: upNCounter port map (innerEn(2),reset, clk, "1001", innerEn(3),LEDbit2 );  --map to a mod-10 (0-9) up counter
uc3: upNCounter port map (innerEn(3), reset, clk, "0101", innerEn(4),LEDbit3 ); --map to a mod-6 (0-5) up counter
uc4: upNCounter port map (innerEn(4), reset , clk, "1001", innerEn(5),LEDbit4 );--map to a mod-10 (0-9) up counter
uc5: upNCounter port map (innerEn(5), reset , clk, "0101", innerEn(6),LEDbit5 );--map to a mod-6 (0-5) up counter
```

***Figure 5. Map to 6 up-counter with radix.***

```
de0: decoder port map (LEDbit0, HEX0); -- map to a decoder to the LED segment representation
de1: decoder port map (LEDbit1, HEX1);
de2: decoder port map (LEDbit2, HEX2);
de3: decoder port map (LEDbit3, HEX3);
de4: decoder port map (LEDbit4, HEX4);
de5: decoder port map (LEDbit5, HEX5);
```

***Figure 6. Map to decoder for LED segment representation.***

Finally, we built a circuit for storing the state of button by the signal buttonEnable. As we want the stopwatch to work when the start button is pressed, and the button is active low, the buttonEnable should be "1" when the start button is pressed. Similarly, the buttonEnable should be "0" when the stop button is pressed. On all other conditions the buttonEnable should not change so that it could store the state of the buttons.

```
process(clk,reset)
      begin

     if (rising_edge(clk)) then

        if(start ='1' and stop = '0') then
              buttonEnable <= '0'; --stop is pressed, enable set to zero, the entire system froze from clock divider

        elsif(start ='0' and stop = '1') then
              buttonEnable <= '1';  -- start is pressed. Clk_divider starts and activate entire system

        else
              buttonEnable <= buttonEnable;

        end if;

     end if;


end process;
```

***Figure 7. Logic of storing the state of buttons.***

We create 6 counter instances is because all 6 digits on the stop watch have different radix. The radixes for the first and the third digit are 6 and others are 10, which means we couldn't use only one up-counter for stopwatch. As a result, we need to have an up-counter for each digit, which means there should be totally 6 up-counter instances.

- **A discussion of how the stopwatch circuit was tested.**

The stopwatch circuit was tested by using the DE1-SoC board. After the circuits are complied in the Quartus software with them mapped on the board, we can have the stopwatch tested.

First, we need to check whether all three buttons work correctly. As being set, KEY2 on the board refers to start, KEY1 refers to stop and KEY0 refers to reset. The three buttons are tested by pressing buttons from KEY2 to KEY0 in order. The stopwatch started counting after KEY2 is pressed and stopped after KEY1 is pressed. KEY0 was tested twice to check whether the stopwatch could work with the stopwatch stopped or working. Both tests achieve the goal and meet the requirement.

Then we need to check the digits. The digits of HEX4, HEX2, HEX1 and HEX0 should increase from 0 to 9 and come back to 0; digits of HEX5, HEX3 should increase from 0 to 5 and come back to 0. After the start button pressed, the counter works just as our expectation. The last two digits increased from '00' to '99' and got back to '00' during each period; the middle two increased by 1 from '00' every time the last two digits reset to '00' from '59', and they were reset to '00' after '59'; the first two digits increased by 1 every time the middle ones reset to '00' from '59'.

At last we test about the time interval. The last two digits should be reset to '00' every 1 second and the middle two should be reset to '00' every 1 minute and increase by 1 every 1 second. By comparing to the system time on the computer, there seems no obvious error to our stopwatch.

- **A summary of the FPGA resource utilization (from the Compilation Report's Flow Summary) and the RTL schematic diagram for the stopwatch circuit. Clearly specify which part of your code maps to which part of the schematic diagram.**

Overall, since the algorithm is not complicated with no extensive bit processing. From the flow summary below less than one of the logic utilizations are incorporated in our program. Since we are implementing sequential circuit for this lab, 71 registers are used which is a new feature comparing to lab 1. Lastly, the number of pins used are approximately 10 percent, mainly from the 6 LED output.

| Flow Summary | |
|---|---|
| Flow Status | Successful - Mon Apr 01 15:52:27 2019 |
| Quartus Prime Version | 15.1.0 Build 185 10/21/2015 SJ Standard Edition |
| Revision Name | g62lab2 |
| Top-level Entity Name | g62_stopwatch |
| Family | Cyclone V |
| Device | 5CSEMA5F31C6 |
| Timing Models | Final |
| Logic utilization (in ALMs) | 55 / 32,070 ( < 1 % ) |
| Total registers | 71 |
| 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 % ) |

***Figure 1. A summary of the FPGA resource utilization (from the Compilation Report's Flow Summary)***

Code mapping and RTL diagrams:

Below (figure 2) is the overall RTL diagram for the stop watch. The first few blue components are case switches for button input (start, stop and reset). Then, there are 7 counters described by the green blocks. The first one is the clock divider, corresponding to instance "cd1" in the code (figure 3). The ripped enable is then passed to the first up counter counting every 0.01s (instance "uc0"). The rest of the counter follows the similar idea and every up counter has its current value exported to the decoder for 6 LED representation ("de0" to "de5").

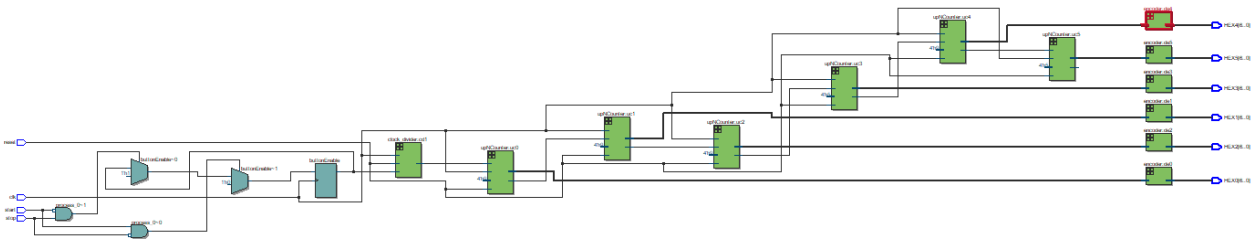

*Figure 2. The overall RTL diagram for the stop watch*

```
cd1: clock_divider port map(buttonEnable, reset, clk, innerEn(0)); --map to a clock divider

uc0: upNCounter port map (innerEn(0),reset , clk, "1001", innerEn(1),LEDbit0 ); --map to a mod-10 (0-9) up counter
uc1: upNCounter port map (innerEn(1),reset , clk, "1001", innerEn(2),LEDbit1 ); --map to a mod-10 (0-9) up counter
uc2: upNCounter port map (innerEn(2),reset, clk, "1001", innerEn(3),LEDbit2 );  --map to a mod-10 (0-9) up counter
uc3: upNCounter port map (innerEn(3), reset, clk, "0101", innerEn(4),LEDbit3 ); --map to a mod-6 (0-5) up counter
uc4: upNCounter port map (innerEn(4), reset , clk, "1001", innerEn(5),LEDbit4 );--map to a mod-10 (0-9) up counter
uc5: upNCounter port map (innerEn(5), reset , clk, "0101", innerEn(6),LEDbit5 );--map to a mod-6 (0-5) up counter

de0: encoder port map (LEDbit0, HEX0); -- map to a decoder to the LED segment representation
de1: encoder port map (LEDbit1, HEX1);
de2: encoder port map (LEDbit2, HEX2);
de3: encoder port map (LEDbit3, HEX3);
de4: encoder port map (LEDbit4, HEX4);
de5: encoder port map (LEDbit5, HEX5);
```

*Figure 3. The code which maps to each component of the stopwatch circuit*

For an entity of clock divider, please refer to figure 1.3 and 1.4. Since the initial case, 499999 is a 19-bit binary representation, the RTL diagram is extensive with multiplexers to deal with each bit.

For an entity of a up counter of radix-n ("upNCounter" in the code), please refer to figure 1 and figure 2 in section 1.

For an entity of a decoder reused from lab 1, the code is shown below (figure 4), which maps to the RTL diagram as figure 5.

```
architecture Display of decoder is
begin
   process(segmentcode)
   begin
      case segmentcode is -- List all the possi
         when "0000" => segments <= "1000000";
         when "0001" => segments <= "1111001";
         when "0010" => segments <= "0100100";
         when "0011" => segments <= "0110000";
         when "0100" => segments <= "0011001";
         when "0101" => segments <= "0010010";
         when "0110" => segments <= "0000010";
         when "0111" => segments <= "1111000";
         when "1000" => segments <= "0000000";
         when "1001" => segments <= "0010000";
         when "1010" => segments <= "0001000";
         when "1011" => segments <= "0000011";
         when "1100" => segments <= "1000110";
         when "1101" => segments <= "0100001";
         when "1110" => segments <= "0000110";
         when "1111" => segments <= "0001110";
         when others => segments <= "1111111";
      end case;
   end process;
```

***Figure 4. VHDL code of a decoder, reused from lab 1***

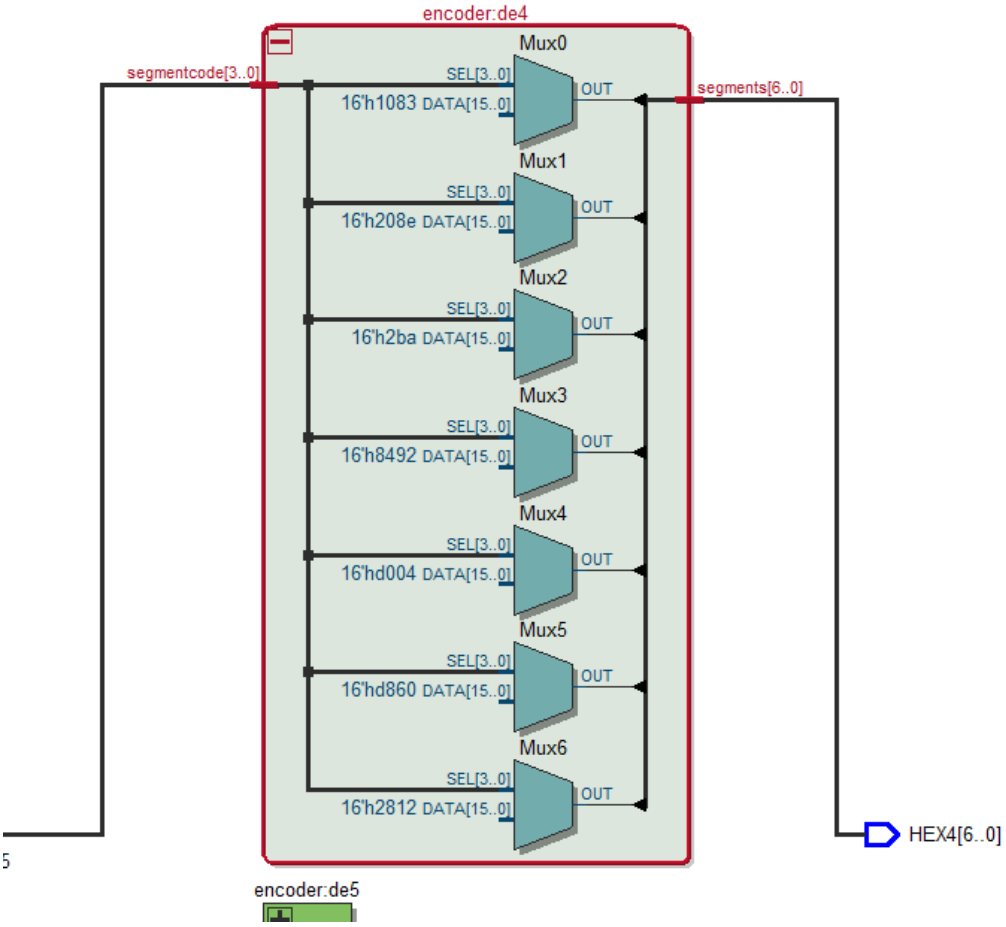


***Figure 5. The RTL diagram for a decoder***