ECSE 222 Lab 2

Glen Xu 260767363

& Cheng Chen 260775674

Group 56

Counter Circuit

DESIGN

The counter circuit receives 3 inputs and provides 1 output. The circuit is able to count upwards from 0 to 15 with 4-bit binary numbers. The result from the previous counting process will be stored in order to be added upwards.

The circuit requires a `reset`, an `enable`, and a `clk` as inputs, and a 4-bit vector storing `count` as output. The entity is declared with these 4 ports.

```
entity g56_counter is
  Port (enable : in  std_logic;
        reset : in  std_logic;
          clk : in  std_logic;
        count : out std_logic_vector(3 downto 0)
  );
end g56_counter;
```

Then, a 4-bit vector called `dgt` is declared in the architecture to store the result from the previous cycle in the sequential circuit.

```
architecture behaviour of g56_counter is
signal dgt : std_logic_vector(3 downto 0) := "0000";
```

Then, the process is implemented. The reset value is not a synchronous signal, and result of the counter will accumulate for every cycle. In the process, we need to know about changes of both of the signals, so they both have to be in the sensitivity list.

```
begin
Process(clk, reset) begin
```

When `reset` equals 0, i.e. when it is active, set `dgt` to ‘0000’ to restart. Contrariwise, when `clk` is on its rising edge and when `enable` = 1, `dgt` is added by 1.

```
  if(reset = '0') then
    dgt <= "0000";
  elsif(rising_edge(clk)) then
    if(enable = '1') then
      dgt <= std_logic_vector(unsigned(dgt) + 1);
    else
      dgt <= dgt;
    end if;
  end if;
end Process;
```

Finally, assign the temporary `dgt` to `count`.

```
count <= dgt;
end behaviour;
```

SIMULATION

After making sure that the code is without error by compiling it successfully, a testbench of our code is generated by Quartus. We then edit the testbench by changing the inputs, and simulate the circuit in ModelSim. Since the DE1-SoC board works at a frequency of 50 MHz, we set the clock in our circuit with a period of 20 ns. Note that the `WAIT;` in the default testbench has to be deleted, otherwise the clock will not work properly.

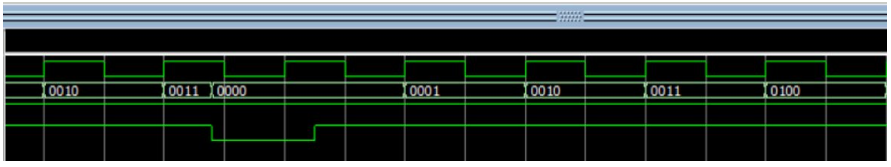
```
BEGIN
  clk <= '1';
  WAIT FOR 10 ns;
  clk <= '0';
  WAIT FOR 10 ns;
```

Then, the other inputs, reset and enable are assigned and changed as well.

```
BEGIN
  reset <= '1';
  enable <= '0';
  WAIT FOR 50 ns;
  reset <= '0';
```

```
  enable <= '0';
  WAIT FOR 50 ns;
  reset <= '0';
  enable <= '1';
  WAIT FOR 50 ns;
  reset <= '1';
  enable <= '1';
WAIT;
```

The wave graph shows that the `reset` and `enable` inputs are functioning normally. The inputs change accordingly to the assignment, and the output stays constant whenever not both of `reset` and `enable` are 1. The `reset` and `enable` are changed independently as well to see if each of them functions as expected, and they did. Only when `reset` and `enable` are equal to 1 simultaneously, the output value starts to accumulate at the rising edge of `clk`.



Clock Divider Circuit

DESIGN

This circuit provides an output when the external clock finishes 500,000 periods. Since the DE1-SoC board works at a frequency of 50 MHz, the output of the clock divider will be given once every 500000 / 50000000 Hz = 0.1 s. The clock divider also requires that the result from the previous counting process to be stored in order to be added upwards.

The circuit requires a `reset`, an `enable`, and a `clk` as inputs, and a 4-bit vector storing `en-out` as output. The entity is declared with these 4 ports. On top of that, a generic time factor `T` = 500000 is declared in order to reset the circuit.

```
entity g56_clock_divider is
  Generic (constant T : integer := 500000);
  Port (enable : in std_logic;
            reset : in std_logic;
              clk : in std_logic;
           en_out : out std_logic);
end g56_clock_divider;
```

Then, two signals are defined in the architecture. The `t1` is used to store the current counter divider value, which is 1 less than `T`, and the `en_out1` is current circuit output value.

```
architecture behaviour of g56_clock_divider is
signal en_out1 : std_logic := '0';
signal t1 : integer := T-1;
```

Then, the process is implemented. The reset value is not a synchronous signal, and result of the counter will accumulate for every cycle. In the process, we need to know about changes of both of the signals, so they both have to be in the sensitivity list.

```
begin
Process(clk, reset) begin
```

When `reset` equals 0, i.e. when it is active, set `t1` to 499999 to restart. Contrariwise, when `clk` is on its rising edge and when `enable` = 1, `t1` is subtracted by 1. When `t1` reaches 0, it automatically restarts from 499999 as well.

```
  if(reset = '0') then
    t1 <= T-1;
    en_out1 <= '0';
  elsif(rising_edge(clk)) then
    --enable: active high
    if(enable = '0') then
      t1 <= t1;
    else
      t1 <= t1 - 1;
    end if;
    if(t1 = 0) then
      t1 <= T-1;
      en_out1 <= '1';
```

```
    else
      en_out1 <= '0';
    end if;
  end if;
end Process;
```

Finally, assign the temporary `en_out1` to `en_out`.

```
  en_out <= en_out1;
end behaviour;
```

In a down-counter, the output does not change when `t1` is equal to any other value, on the contrary, it only changes when `t1` reaches 0. In an up-counter, on the other hand, the output only changes when `t1` equals 499999. The reason why a down-counter is chosen instead of an up-counter is that, it is simpler to determine whether `t1` equals to 0 than to determine whether `t1` equals to 499999. Also, we can simply change the value of T when we want to change the input clock when we are using a down-counter. If we are using an up-counter, then there is a risk of overflow, exceeding the limit. A down-counter is more reusable than an up-counter.

SIMULATION

After making sure that the code is without error by compiling it successfully, a testbench of our code is generated by Quartus. We then edit the testbench by changing the inputs, and simulate the circuit in ModelSim. Since the DE1-SoC board works at a frequency of 50 MHz, we set the clock in our circuit with a period of 20 ns. Note that the `WAIT;` in the default testbench has to be deleted, otherwise the clock will not work properly.

```
BEGIN
  clk <= '1';
  WAIT FOR 10 ns;
  clk <= '0';
  WAIT FOR 10 ns;
```

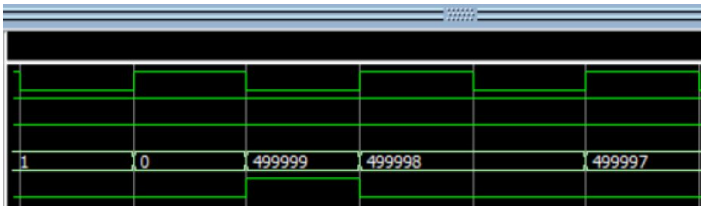
Then, the other inputs, reset and enable are assigned and changed as well.

```
BEGIN
  reset <= '1';
  enable <= '0';
  WAIT FOR 50 ns;
  reset <= '0';
  enable <= '0';
  WAIT FOR 50 ns;
  reset <= '0';
  enable <= '1';
  WAIT FOR 50 ns;
```

```
  reset <= '1';
  enable <= '1';
WAIT;
```

The wave graph shows that the `reset` and `enable` inputs are functioning normally. The inputs change accordingly to the assignment, and the output stays constant whenever `enable` is 1. The `reset` and `enable` are changed independently as well to see if each of them functions as expected, and they did.

Also, when `t1` reaches 0, `en_out` changes into 1, and `t1` is automatically reset to 499999.



Stopwatch Circuit

The stopwatch circuit combines the components of the counter, clock divider and 7 segment decoder. The stopwatch has stop, start, reset and clock as inputs in which stop, start and reset(active low and asynchronous) are controlled by the buttons on the FPGA board whereas clk are generated by the clock divider since we want counters work with real-world time. The outputs are six 7-bit numbers which control the 7 segment displays. The last two displays, the middle 2 displays and the first two displays correspond to centiseconds, seconds and minutes respectively. Due to the rounding convention, counter 0, 1, 2, 4 count from from

0 to 9 while counter 3 and 5 count from 0 to 5. In total, we implemented six counters in order to count up to a six digit number. As discussed above, the least two output digits corresponds to centiseconds. As the least digit(digit 0) count up until it reaches 9, the enable for digit 1 will be active and digit 0 will be reset to 0 and start counting again. For digit 2,3,4 and 5, they work the same as digit 1 and 0. As the counters are counting up, the numbers will also be shown through the 7 segment decoder on the board.

Testing of the Stopwatch

To test the stopwatch, we used the software called ModelSim. At first, We wrote a testbench to set the start active while reset and stop inactive. Theoretically, we should see the stopwatch running as time passes by. Also, each digit will round up when the digit before reaches 9 or 6. Then, we set stop to active and others remain unchanged. The stopwatch stopped working. This situation can be explained by the code below:

```
if(start = '0') then
enable_stopwatch <= '1';
elsif(stop = '0') then
enable_stopwatch <= '0';
```

As we can see above, active start enables the circuit and active stop disables the circuit.

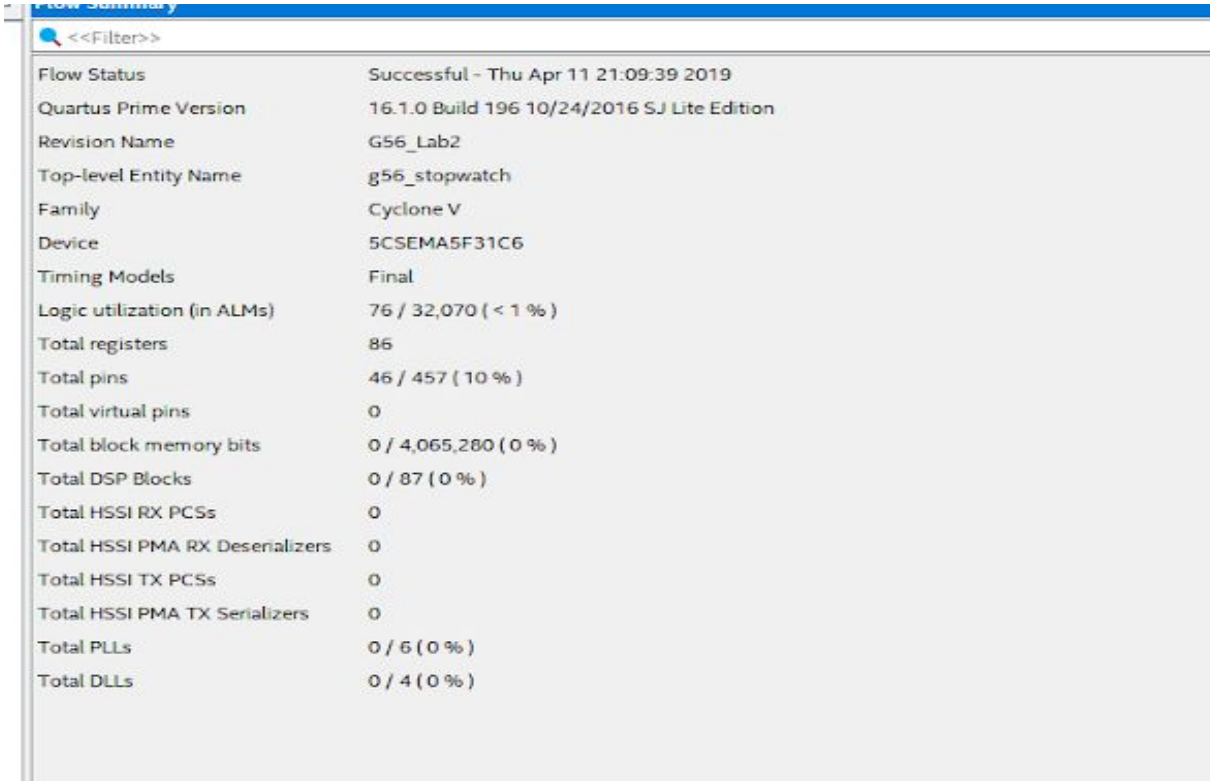
To test the reset function, we set reset to 0 (i.e. active). We found that the numbers shown on the board are always zero no matter what start or stop is. This can be shown by the code below:

```
elsif(reset = '0') then
reset0 <= '0';
reset1 <= '0';
reset2 <= '0';
reset3 <= '0';
reset4 <= '0';
reset5 <= '0';
```

As we can see above, if the reset is 0 (i.e active), all counters will be set to 0.

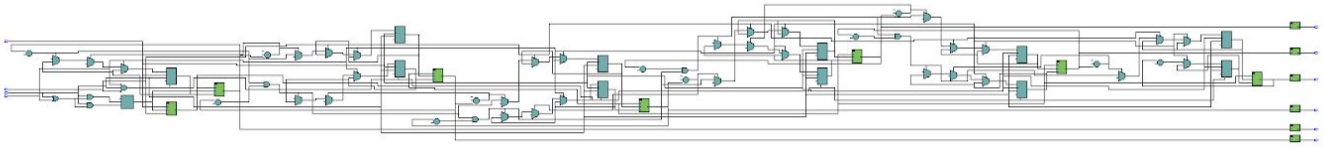
If all the cases discussed work as we expected, the stopwatch is fully tested.

FPGA Resource Utilization

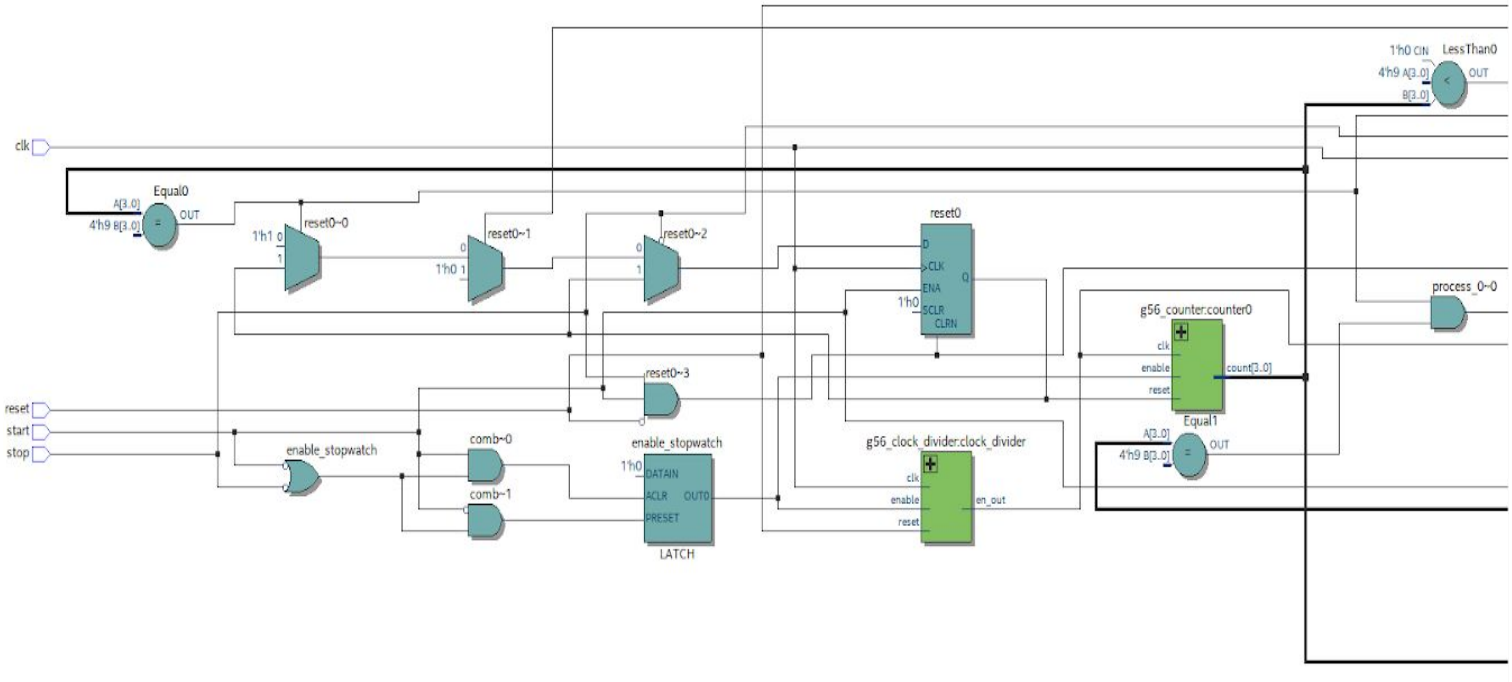


The device suggests what device we are using to run the VHDL code. Logic utilization tells how much of the device used to implement our circuit which is 1%. Total pins shows how many pin assignments we used. The first line tells the user that the compilation is successful.

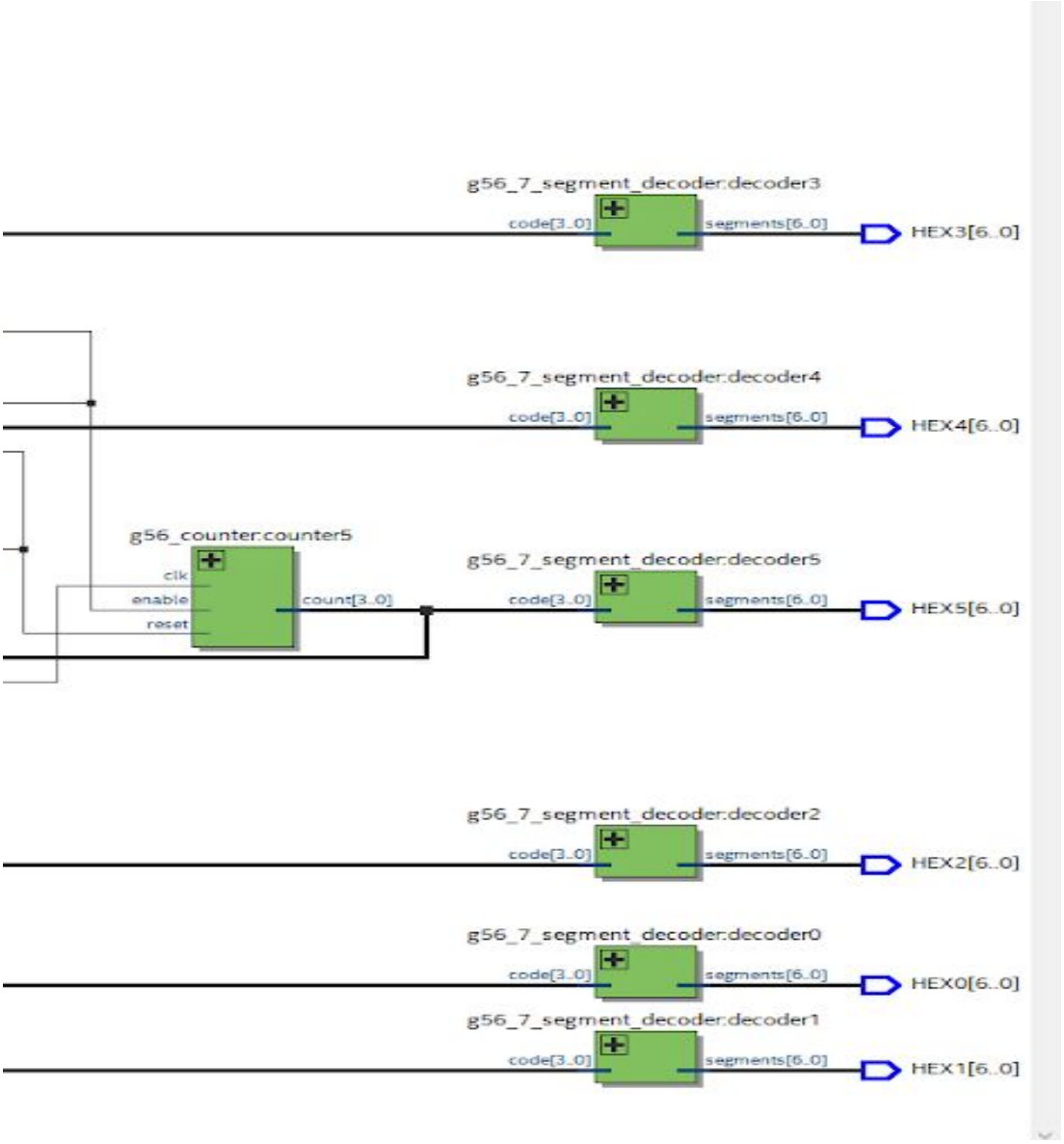
RTL schematic



The figure above is the RTL schematic of the whole stopwatch circuit. The whole circuit includes 1 clock divider, 6 counters and 6 7-segment decoders. Also, we have multiplexers to determine the if the reset and enables are active which corresponds to the if statement in the VHDL code.



The figure above is a sample circuit for the reset, clock divider and counter. Once start is active and stop and reset are inactive, the circuit will use the clock generated by the clock divider to control the counter. Once this counter reaches 9, it will enable its next counter, in this case it enables counter 1. There are six counters in total.



The figure above shows the six 7-segment decoders. The six 7-segment decoders are connected to the counters and they display the numbers in hexadecimal form on the FGPA board while the counters are running.