VHDL Assignment #5: Description of Storage Elements and Sequential Circuits in VHDL

In this VHDL assignment, you will first learn how to describe storage elements and sequential logic circuits in VHDL. Then, you will design and simulate a 3-bit up-counter in EDAPlayground.

1 Instructions

- Due date: Friday, November 20, 2020 by 5:00pm.
- Submission is in teams using myCourses (only one team member submits). In the report, provide the names and McGill IDs of the team members.
- Late submissions will incur penalties as described in the course syllabus.

2 Learning Outcomes

After completing this lab you should know how to

- Describe sequential elements in VHDL
- Design a clock divider circuit
- Design a 3-bit counter in VHDL
- Perform functional simulation of the counter

3 VHDL Description of Storage Elements

In the previous VHDL assignments, you have learned how to use sequential statements to describe the behavior of combinational circuits inside the *process* block. Sequential statements within the process block can also be used to describe sequential circuits such as storage elements. In digital systems, we have two types of memory elements: latches and flip-flops (FFs). Latches are memory elements that immediately reflect any changes in the input signal to the output signal when the level of the control signal (clk) is high. As such, latches are usually referred to as level-sensitive (or level-triggered) memory elements. Alternatively, FFs change their state when the control signal goes from either high to low or from low to high. Note that FFs working with a control signal that goes from high to low or from low to high are called, respectively, negative-edge-triggered and positive-edge-triggered FFs. In digital systems, edge-triggered flip-flops are superior to level-sensitive latches since FFs are more robust. For example, noise can easily disrupt the output of a latch when the control signal is high. On the other hand, the output of FFs can be disrupted only in presence of noise at the edge transition of the control signal. It is highly recommended, therefore, to use FFs when designing sequential circuits. Nonetheless, some applications benefit from the use of latches.

In VHDL, process blocks are used to describe FFs. Since FFs set their state at the edge of the control signal, we need a statement to detect the edge transition of the control signal. This can be simply done by using a `IF-THEN-ELSE` statement. Assuming that `clk` is the control signal of a FF, a positive edge transition (*i.e.*, a transition from '0' to '1') of the `clk` signal can be detected by the following statement: `RISING_EDGE (clk)`. Similarly, a negative edge transition (*i.e.*, a transition from '1' to '0') can be detected by the following statement `FALLING_EDGE (clk)`. For example, the following process block describes a positive-edge-triggered DFF.

```
PROCESS (clk)
BEGIN
    IF RISING_EDGE(clock) THEN
        Q <= D;
    END IF;
END PROCESS;
```

Since the state Q (output) changes only as a result of a positive clock edge, only the clk signal is listed in the sensitivity list of the process. Note that there are additional ways to detect a clock edge, such as `IF clk'EVENT AND clk = '1'` or `WAIT UNTIL clk'EVENT AND clk = '1'` statements. The syntax `clk'EVENT` uses a VHDL construct called an *attribute*. An attribute refers to a property of an object such as a signal. In this case, the `'EVENT` attribute refers to any change in the clock signal. These two statements (*i.e.*, `clk'EVENT AND clk = '1'` and `WAIT UNTIL clk'EVENT AND clk = '1'`) are described in detail in Examples 7.2 and 7.3 of the textbook. Note that if we use the `WAIT UNTIL` statement, then the sensitivity list of the process block should be empty.

So far, we have described a positive-edge-triggered flip-flop in VHDL. Now, let us describe a positive-edge-triggered flip-flop with asynchronous active-low reset (also known as clear) and asynchronous active-low set signals. When the reset signal is '0', the output of the FF is immediately set to '0', regardless of the value of the control signal (*i.e.*, clk). Similarly, when the set signal is '0', the output of the FF is immediately set to '1', regardless of the value of the control signal (*i.e.*, clk). The sensitivity list of the process contains, therefore, the clk, reset, and set signals, since these three signals trigger a change in the output of the FF (positive clock edge and low-activated set and reset signals. A positive-edge-triggered FF with asynchronous active-low reset and set signals can be described in VHDL as follows:

```
PROCESS (clk, reset, set)
BEGIN
    IF reset = '0' THEN
        Q <= '0';
    ELSIF set = '0' THEN
        Q <= '1';
    ELSIF RISING_EDGE(clk) THEN
        Q <= D;
    END IF;
END PROCESS;
```

Note that since we check the `reset` signal first, it has priority over the other two signals, *i.e.*, `set` and `clk`. Similarly, `set` has priority over the `clk` signal. If we check for the reset and set signals at the positive edge of the clock, we obtain a positive-edge-triggered flip-flop with synchronous active-low reset and set signals as follows:

```
PROCESS (clk)
BEGIN
    IF RISING_EDGE(clk) THEN
        IF reset = '0' THEN
            Q <= '0';
        ELSIF set = '0' THEN
            Q <= '1';
        ELSE
        Q <= D;
        END IF;
    END IF;
END PROCESS;
```

Note that we do not include the `reset` and `set` signals in the sensitivity list of a flip-flop circuit with synchronous reset and set signals.

4 Design of a Storage Element

In this assignment, you are asked to design a JKFF. The inputs to the FF are J, K, and clk. The output of the FF is Q. The operation of a JKFF is described in Lecture 17 and in Section 7.6 in the textbook. Use the following entity declaration for your implementation of the storage element circuit.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity jkff is
    Port (clk          : in  std_logic;
          J            : in  std_logic;
          K            : in  std_logic;
          Q            : out std_logic);
end jkff;
```

To describe your circuit in VHDL, use a single process block. Once you have your circuit described in VHDL, you should simulate it. Write a testbench code and perform a functional simulation for your VHDL description of the JKFF. When writing a testbench for sequential circuits, a clock signal is required within the testbench. One way to generate the clock signal in testbench, is provided below.

```
clock_generation: process
begin
  clk <= '1';
  wait for clock_period/2;
  clk <= '0';
  wait for clock_period/2;
end process clock_generation;
```

Note that the "clock_period" parameter should be replaced with desired clock period value. In this assignment, we will use a clock period of 10 ns. Due to the absence of any indefinite "wait" statements in the "clock_generation" process, you cannot use the *run -all* command which is executed under the hood in EDAPlayground (you can see it in the log); you must instead explicitly determine a simulation duration under the "Run Time" section (*e.g.*, run 100 ns).

Once you have described your circuit in VHDL, you should write a testbench and simulate the circuit. Make sure that all possible inputs to the JKFF are verified in the simulation.

5 Counters

A counter is a special sequential circuit. When counting up (by one), we require a circuit capable of "remembering" the current count and increase it by 1 the next time we request a count. When counting down (by one), we require a circuit capable of "remembering" the current count and subtracting 1 the next time we request a count. Counters use a clock signal to keep track of time. In fact, each increment (or decrement) occurs when one clock period has passed. To design an up-counter counting in increments of 1 second, we will first design a 3-bit up-counter counting at the positive edge of the clock with an asynchronous reset (which should be active low) and an enable signal (which should be active high). The counter counts up when the enable signal is high. Otherwise, the counter holds its previous value. Use the following entity declaration for your VHDL description of the counter:

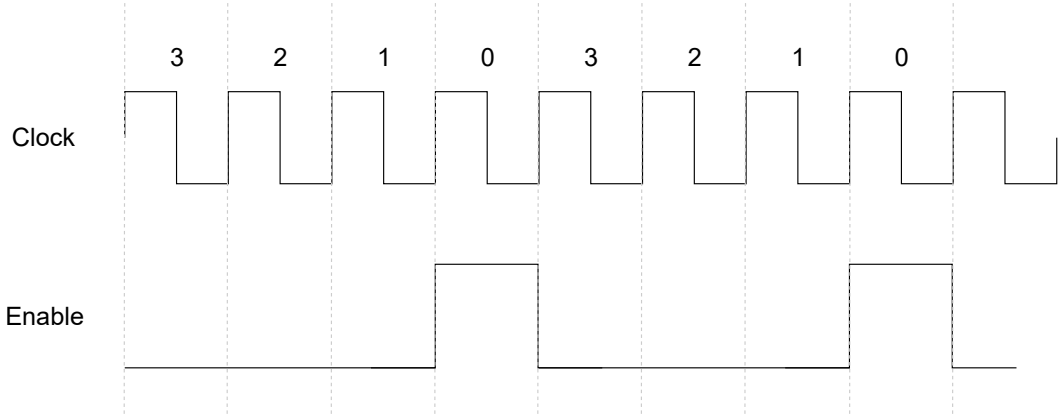
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity three_bit_up_counter is
  Port (enable     : in  std_logic;
        reset      : in  std_logic;
        clk        : in  std_logic;
        count      : out std_logic_vector(2 downto 0));
end three_bit_up_counter;
```

Note that a 3-bit counter counts from 0 to 7. When the current count reaches 7, the next count will automatically wrap around back to 0.

Once you have your circuit described in VHDL, you should simulate it. Write a testbench code and perform a functional simulation for your VHDL description of the counter.

6 Clock Divider

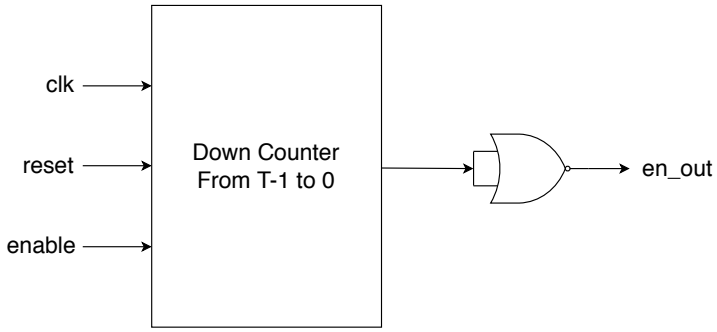
A clock divider is a circuit that generates a signal that is asserted once every N clock cycles. This signal can be used as a condition to enable the counter that you designed in Section 5. An example of the clock and output (*i.e.*, "enable") waveforms for $N = 4$ is:



Implementing the clock divider circuit requires a counter counting clock periods. The counter counts down from $N-1$ to 0. Upon reaching the count of 0, the clock divider circuit outputs/asserts 1 and the count is reset to $N-1$. For other values of the counter, the output signal of the clock divider circuit remains 0. In this assignment, we want to design a counter that counts in increments of 1 second. In other words, we need to assert an enable signal every 1 second. First, find the value of N for the clock divider circuit to generate an enable signal every 1 second. To avoid long simulation delays, we will use 10 Hz as the clock frequency for our simulation. Describe the clock divider circuit in VHDL using the following entity declaration:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
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;
```

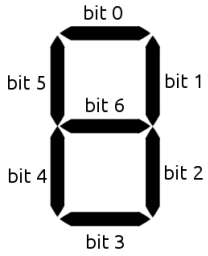
Hint: The following figure shows an example of the clock divider circuit. Also, note that the down-counter inside the clock divider circuit is different from the up-counter that you designed in Section 5, and that `en_out` is the NOR of the bits in the counter value.



Once you have your circuit described in VHDL, write a testbench and perform a functional simulation for your VHDL description of the clock divider.

7 BCD to 7-Segment Decoder

A 7-segment LED display includes 7 individual LED segments, as shown below. By turning on different segments together, we can display characters or numbers. A 7-segment decoder takes as input a 4-bit BCD-formatted code representing the 10 digits between 0 and 9, and generates the appropriate 7-segment display associated with the input code. For many LED displays, segments turn on when their inputs are driven low, that is "0" means on, and "1" means off. This scheme for the inputs is called "active low." Although we are not able to use an actual 7-segment LED display this semester, we will design out counter circuit with a 7-segment display output. The VHDL code for the 7-segment decoder is provided below.



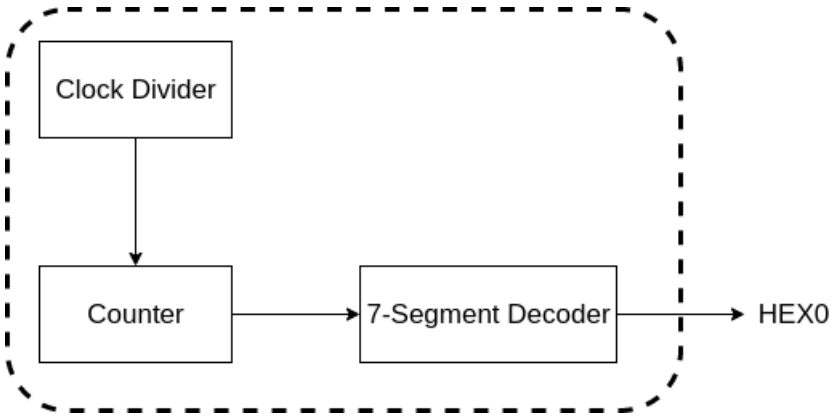

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity seven_segment_decoder is
port (  code            : in std_logic_vector(3 downto 0);
segments_out    : out std_logic_vector(6 downto 0));
end seven_segment_decoder;

architecture decoder of seven_segment_decoder is
begin
  WITH code SELECT
  segments_out <=
    "1000000" WHEN "0000", -- 0
    "1111001" WHEN "0001", -- 1
    "0100100" WHEN "0010", -- 2
    "0110000" WHEN "0011", -- 3
    "0011001" WHEN "0100", -- 4
    "0010010" WHEN "0101", -- 5
    "0000010" WHEN "0110", -- 6
    "1111000" WHEN "0111", -- 7
    "0000000" WHEN "1000", -- 8
    "0010000" WHEN "1001", -- 9
    "1111111" WHEN others;
end decoder;
```

8 3-bit Up-Counter that Counts in Increments of 1 Second

In this part, you will design a simple counter circuit that counts in increments of 1 second using the counter, clock divider, and 7-segment decoder circuits.

Reset is an asynchronous active-low input. The normal condition of the reset is high ('1'). If reset goes low ('0'), the output of the counter should be 0 as long as reset remains low. Once reset goes back to high, the counter should start counting. Enable is a synchronous active-high input. When enable is high ('1') the counter counts every 1 second, the circuit will hold and display the current count otherwise.

You will need to create one instance of your three_bit_up_counter and one instance of the provided seven_segment_decoder. Since we measure time in increments of 1 second, the counter that you designed in Section 5 increments only when the output signal of the clock divider circuit becomes high. The following figure shows the high-level architecture of the circuit.



Describe the circuit that counts every 1 second in VHDL using the following entity declaration:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity wrapper is
Port (enable   : in  std_logic;
      reset    : in  std_logic;
      clk      : in  std_logic;
      HEX0     : out std_logic_vector (6 downto 0));
end wrapper;
```

You will now write a testbench to verify your circuit using simulation. Generate a simulation that shows the functionality of the counter and the reset and enable inputs. Clearly show on the waveform that the counter counts at 1 second intervals.

9 Deliverables

Once completed, you are required to submit a report explaining your work in detail, including answers to the questions related to this assignment. You will find the questions in the next section. You must also submit all of your VHDL, run.do, and .sdc files along with Log content of the synthesized circuits and timing analysis. You must also submit all of your testbench files. If you fail to submit any part of the required deliverables, you will incur grade penalties as described in the syllabus.

10 Questions

Please note that even if some of the waveforms may look the same, you still need to include them separately in the report.

1. Briefly explain your VHDL code implementation of all circuits.
2. Provide waveforms for each of the individual circuits (each section) and for the 3-bit up-counter.
3. Perform timing analysis of the 3-bit up counter and find the critical path(s) of the circuit. What is the delay of the critical path(s)?
4. Report the number of pins and logic modules used to fit your 4-bit comparator design on the FPGA board.