VHDL Assignment #6: Storage Elements, Sequential Circuits, and Finite State Machines in VHDL

1 Instructions

- TA in charge: Arish Yaseen (arish.yaseen@mail.mcgill.ca) and Muhammad Bilal Babar (muhammad.babar@mail.mcgill.ca) - please utilize discussion boards on myCourses for questions as much as possible.
- Due date: Friday, December 2, 2022 by 11:59 pm EDT.
- 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 assignment 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 using ModelSim
- Design FSMs in VHDL
- Design sequence detector circuits
- Test the circuit on the Altera board

3 Introduction

In this VHDL assignment, you will describe a sequence detector in VHDL and test it on the Altera DE1-SoC board using a sequence from a memory input (ROM).

If you need any help regarding the lab materials, you can

- Ask the TA for help during lab sessions and office hours.
- Refer to the text book. In case you are not aware, Appendix A “VHDL Reference" provides detailed information on VHDL.
- You can also refer to the tutorial on Quartus and ModelSim provided by Intel (click here for Quartus and here for ModelSim).
- Refer to the DE1-SoC User Manual (in the Content tab on myCourses).

It is highly recommended that you first try to resolve any issue by yourself (refer to the textbook and/or the multitude of VHDL resources on the Internet). Syntax errors, especially, can be quickly resolved by reading the error message to see exactly where the error occurred and checking the VHDL Reference or examples in the textbook for the correct syntax.

4 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-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-triggered 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.

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-triggered flip-flop in VHDL. Now, let us describe a positive-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-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-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.

5 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 Section 7.6 of 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 firstname_lastname_jkff is
Port (clk       : in  std_logic;
J           : in  std_logic;
K           : in  std_logic;
Q           : out std_logic;
end firstname_lastname_jkff;
```

where firstname_lastname in the name of the entity is the name of one of the students in your group.

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 counter. 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" statement in the "clock_generation" process, you cannot use the *run -all* command in ModelSim; you must instead explicitly determine a simulation duration (*e.g.*, run 100 ns).

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

6 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 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 firstname_lastname_counter is
Port (enable     : in  std_logic;
reset      : in  std_logic;
clk        : in  std_logic;
count      : out std_logic_vector(2 downto 0));
end firstname_lastname_counter;
```

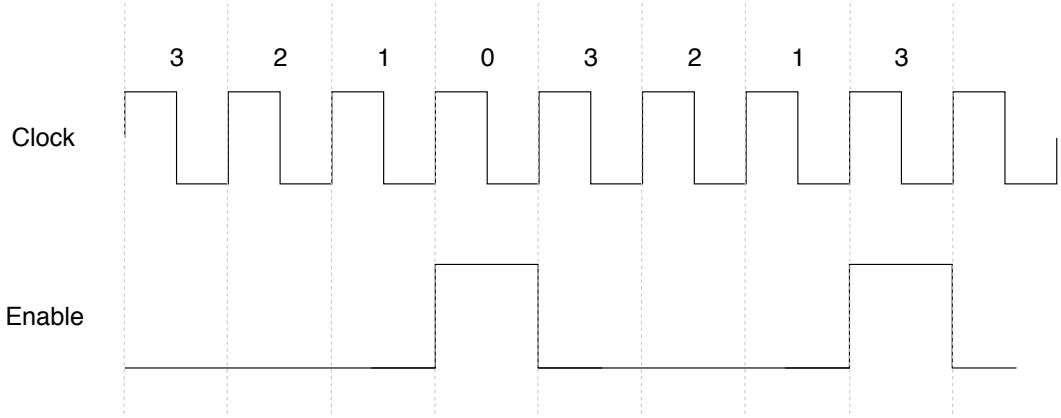
where firstname_lastname in the name of the entity is the name of one of the students in your group.

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.

7 Clock Divider

A clock divider is a circuit that generates a signal that is asserted once every T clock cycles. This signal can be used as a condition to enable the counter that you designed in Section 6. An example of the clock and output (i.e., "enable") waveforms for $T = 4$ is:

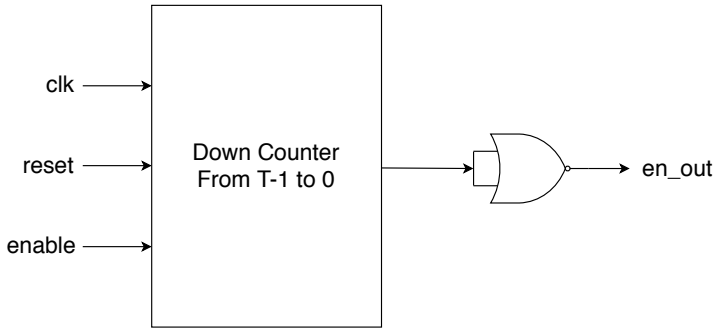


Implementing the clock divider circuit requires a counter counting clock periods. The counter counts down from $T-1$ to 0. Upon reaching the count of 0, the clock divider circuit outputs/asserts 1 and the count is reset to $T-1$. For other values of the counter, the output signal of the clock divider circuit remains 0. In this lab, we want to design a counter counting in increments of 1 second. In other words, we need to assert an enable signal every 1 second. First, find the value of T for the clock divider circuit to generate an enable signal every 1 second. Note that the PLL, the device which supplies the clock for your design on the DE1-SoC board, works at a frequency of 50 MHz. 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 firstname_lastname_clock_divider is
Port (enable     : in  std_logic;
reset      : in  std_logic;
clk        : in  std_logic;
en_out     : out std_logic);
end firstname_lastname_clock_divider;
```

where firstname_lastname in the name of the entity is the name of one of the students in your group.

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 6, and that `en_out` is the NOR of the bits in the counter value.



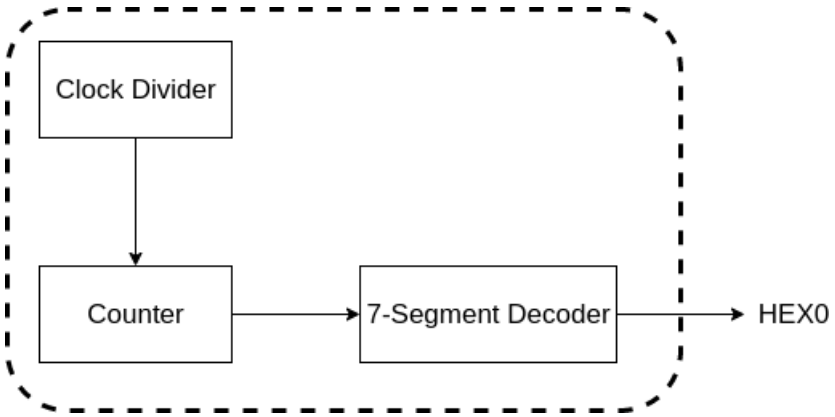
Once you have your circuit described in VHDL, write a testbench code and perform a functional simulation for your VHDL description of the clock divider.

8 3-bit Up-Counter Counting 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 and clock divider 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 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 firstname_lastname_counter and seven_segment_decoder you created in VHDL Assignment #4. Since we measure time in increments of 1 second, the counter that you designed in Section 6 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 counting at every 1 second in VHDL using the following entity declaration:

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

where firstname_lastname in the name of the entity is the name of one of the students in your group.

You will now write a testbench to verify your circuit using the ModelSim 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 Sequence Detector

In this assignment, you will first implement a sequence detector circuit that takes a sequence of bits as its input and detects two different bit patterns in the input sequence. Specifically, you are asked to design a circuit based on Moore-type FSM(s) with an asynchronous reset signal and an enable signal, that takes a sequence of bits as its input "*seq*" and has two outputs "*out_1*" and "*out_2*". The outputs should be "*out_1* = 1" and "*out_2* = 1" at the clock cycle following the input bit patterns 1011 and 0010, respectively; they should be 0, otherwise. Note that state transitions only occur when the enable signal is high. Otherwise, the FSM stays at its current state. An example of the desired behavior is

clock:	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
seq:	0	0	1	1	0	1	1	0	1	1	0	0	1	0	0	1	0	1	1	0	1
out_1:	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	1	0
out_2:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0

Note: It is best to use two FSMs, each detecting one of the two bit patterns (why?). Your VHDL code will be based on the state diagram(s) of your FSMs. It is, therefore, important that you first come up with a simple state diagram. In fact, you should not need more than five states for each of the two FSMs.

Use the following entity declaration for your VHDL description of the sequence detector:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity firstname_lastname_FSM is
    Port (seq        : in  std_logic;
          enable     : in  std_logic;
          reset      : in  std_logic;
          clk        : in  std_logic;
          out_1      : out std_logic; -- generates 1 when the pattern "1011" is detected;
    otherwise 0.
          out_2      : out std_logic); -- generates 1 when the pattern "0010" is detected;
    otherwise 0.
end firstname_lastname_FSM;
```

`firstname_lastname` in the name of the entity is the name of one of the students in your group.

10 Sequence Counter

In this part, you are required to implement a sequence counter circuit that counts how many times each of the two patterns occurs in the input bit stream. The sequence counter circuit can be realized using the sequence detector circuit, which detects the patterns, followed by two 3-bit up-counters (one for each output of the FSM-based circuit) that keep track of the number of detected patterns. More specifically, each counter is incremented whenever its corresponding pattern has been detected. Use the sequence detector and the 3-bit up-counter from VHDL Assignment #5 to implement the sequence counter circuit with an asynchronous reset and an enable signals.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity firstname_lastname_sequence_detector is
    Port (seq        : in  std_logic;
          enable     : in  std_logic;
          reset      : in  std_logic;
          clk        : in  std_logic;
          cnt_1      : out std_logic_vector(2 downto 0); -- counts the occurrence of the pattern
    "1011".
          cnt_2      : out std_logic_vector(2 downto 0)); -- counts the occurrence of the pattern
    "0010".
end firstname_lastname_sequence_detector;
```

`firstname_lastname` in the name of the entity is the name of one of the students in your group.

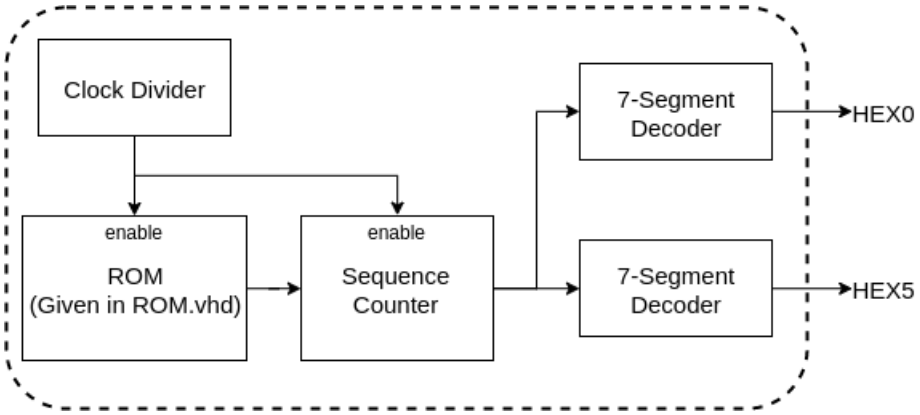
An example of the desired behavior is

clock:	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
seq:	0	0	1	1	0	1	1	0	1	1	0	0	1	0	0	1	0	1	1	0	1
out_1:	0	0	0	0	0	0	0	1	1	1	2	2	2	2	2	2	2	2	2	3	3
out_2:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	2	2	2	2

Note that when the enable signal of the sequence detector circuit is low, its counter and FSM-based instances hold their previous value and state, respectively.

11 Demonstration on the FPGA Board

So far, you have designed a circuit counting the number of occurrences of two different patterns within the input sequence of bits. To demonstrate the functionality of your circuit, you must supply it with a sequence of bits and display the number of occurrences of each pattern on the HEX displays. In contrast to previous VHDL assignments where you inserted the inputs using slider switches and push-buttons, the input sequence cannot be inserted in the same fashion. Instead, we store the input bit sequence into a read-only-memory (ROM) and we supply the circuit the bits of the sequence one after the other every 1 second by reading them from the ROM. The ROM circuit takes clock, asynchronous reset, and enable signals as its inputs and outputs a bit of the sequence under test at the positive edge of the clock signal when the enable signal is high. The VHDL code for the ROM is given in `ROM.vhd`. To read each bit of the sequence under test every 1 second, you will need to create an instance of the clock divider circuit (designed in Section 7) to enable both the ROM and sequence counter circuits at the appropriate rate (once per second). The 7-segment decoder that you created in VHDL Assignment #4 will be then used to display the occurrence number of each pattern. Note that the *clk* port of all the components (*i.e.*, the clock divider, ROM, and sequence counter circuits) is supplied with a clock frequency of 50 MHz. The following figure shows the high-level architecture of the circuit.



Note that Pushbutton PB0 is used to reset the circuit. When the button is pressed, the circuit has to display 0 on the 7-segment displays until it is released. Recall that the output of a pushbutton is high when the button is not being pushed, and is low when the button is being pushed.

Use the following entity declaration:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity firstname_lastname_wrapper is
    Port (reset     : in  std_logic;
          clk       : in  std_logic;
          HEX0      : out std_logic_vector (6 downto 0),
          HEX5      : out std_logic_vector (6 downto 0));
end firstname_lastname_wrapper;
```

`firstname_lastname` in the name of the entity is the name of one of the students in your group.

Compile the circuit in Quartus. Once you have compiled the circuit, it is time to map it on the Altera DE1-SoC board. Perform the pin assignment for the HEX display, the pushbutton, and the slider switch. Make sure that you connect the clock signal of your design to 50 MHz clock frequency (see the DE1 user's manual for the pin location of 50 MHz clock frequency). Program the board and test the functionality of your circuit.

12 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. Provide waveforms for each of the individual storage elements and sequential circuits (each section) and for the 3-bit up-counter.
2. Perform timing analysis (slow 1,100 mV model) of the 3-bit up counter and find the critical path(s) of the circuit. What is the delay of the critical path(s)?
3. Report the number of pins and logic modules used to fit your 3-bit up counter design on the FPGA board.
4. Why is it better to use two FSMs, rather than one, in the implementation of the sequence detector from Section 9?
5. Briefly explain your VHDL code implementation of all circuits.
6. Provide waveforms for each of the individual circuits (each section) and for the wrapper.
7. Perform timing analysis of the wrapper and find the critical path(s) of the circuit. What is the delay of the critical path(s)?
8. Report the number of pins and logic modules used to fit your design on the FPGA board.

13 Deliverables

You are required to submit the following deliverables on MyCourses. Please note that a single submission is required per group (by one of the group members).

- **NEW** Generate (and include in submitted .zip file) a video of at most 30 seconds of the FPGA board that demonstrates the operation of the sequence counter. Explain in the video what is happening on the board.
- Lab report. The report should include the following parts: (1) Names and McGill IDs of group members, (2) an executive summary (short description of what you have done in this VHDL assignment), (3) answers to all questions in previous section (if applicable), (4) legible figures (screenshots) of schematics and simulation results, where all inputs, outputs, signals, and axes are marked and visible, (5) an explanation of the results obtained in the assignments (mark important points on the simulation plots), and (6) conclusions. Note - students are encouraged to take the reports seriously, points will be deducted for sloppy submissions. Please also note that even if some of the waveforms may look the same, you still need to include them separately in the report.
- Project files. Create a single .zip file named `VHDL#_firstname_lastname` (replace `#` with the number of the current VHDL assignment and `firstname_lastname` with the name of the submitting group member). The .zip file should include the working directory of the project.