UNIVERSITY of York

Department of Electronic Engineering

Assessments 2019/20

ELE00067M

Digital Design

This assessment (**Lab Reports 1 & 2**) contributes **10%** of the assessment for this module.

Clearly indicate your **Exam Number** on every separate piece of work submitted.

Unless the assessment specifies a group submission, you should assume all submissions are individual and therefore should be your own work.

All assessment submissions are subject to the Department's policy on plagiarism and, wherever possible, will be checked by the Department using Turnitin software.

Submission is via VLE and is due by **12:00** on **24 October 2019 (Autumn Term, Week 4, Tuesday)**. Please try and submit early as any late submissions will be penalised.

Please remember that if this is your first year of study, you need to complete the mandatory Academic Integrity Tutorial http://www.york.ac.uk/integrity/

ELE00067M Digital Design

Laboratories: Session 2

Sequential elements in VHDL

Report formatting:

There is no formal report structure – you will be marked on the items listed within each lab script. Most reports will include code printout and simulation screenshots – see Appendices for guidelines.

The reports for labs 1 and 2 must be handed in, together in a single pdf file, via the VLE by the deadline indicated on the front page of the script.

Each task should start on a new page, containing all the material required in the order specified. The exam number should be printed on each page.

The PDF file should be named **Yxxxxxxx_DDMSc_Labs1-2.pdf**, where the Yxxxxxxx is the exam number. Exactly one file should be submitted, containing the entire submission.

In all cases, read carefully the instructions on the VLE submission page. Failure to follow the instructions could lead to your assignment not being marked and in any case to a mark penalty.

Submission weight on module mark: 10%

Mark breakdown [80 marks]:

- General issues (e.g. documentation, layout and comments, structural issues): 30 marks
- Task 1A (e.g. VHDL code, testbench, simulation): 5 marks
- Task 1B (e.g. VHDL code, testbench, simulation): 10 marks
- Task 2A (e.g. VHDL code, testbench, simulation): 15 marks
- Task 2B (e.g. VHDL code, testbench, simulation): 20 marks

Task A: Implementation of sequential elements

In this task, you will design a circuit consisting of four 4-bit counters, each using a different combination of reset and enable signals.

Create a new project in the Vivado environment and a new source in the project with three single-line inputs (clk, en, rst) and four 4-bit busses as outputs. Note that the clock input should always be labelled "clock" or "clk" (in lowercase) to allow the tools to identify it easily (if you choose to call it otherwise, it might make testbenches more complex to write).

Using four separate processes and the three inputs (clk, rst, en), define four 4-bit counters with the following behaviours:

1. Synchronous reset and no enable
2. Asynchronous reset (clear) and no enable
3. Synchronous reset and enable
4. Asynchronous reset (clear) and enable

A few additional hints and requirements:

1. No components should be used in the architecture.
2. You will need to add the IEEE packages that define the "+" operator. Do this by un-commenting "`use IEEE.NUMERIC_STD.ALL;`" at the top of the VHDL file that is automatically generated when you create a new source.
3. Note that you will need to define all your vectors as type `UNSIGNED`
4. Because the output of a counter is also its input, and because VHDL does not allow output ports to be read, you will need to define a set of four internal signals (4-bit vectors) for the outputs of the counters. The internal signals should then be assigned to the outputs separately.

Next, define a VHDL testbench to verify the operation of the circuit going through the same steps as above and in the first laboratory. However, this time (and for most of the future labs and indeed the vast majority of digital logic circuits) **your circuit will require a clock signal**.

Luckily, it is quite simple to generate a clock in simulation: after all, it is simply a signal that toggles at regular intervals!

First, you need to define how fast the signal will toggle. This will set the *clock frequency*. In this module, you will only be using behavioural simulation, where timing is not really relevant. However, by convention FPGA testbenches default to 100MHz clocks (which is the standard frequency of FPGA board oscillators).

To set the frequency of the clock, define a clock period as a constant:

```
constant clk_period : time := 10 ns;
```

Note that this definition should be placed in the internal signal area of the code (i.e. before "begin").

Next, right below the UUT instantiation, you want to add an infinite loop that will generate the clock signal. Assuming that your clock signal is called "clk", the loop would be:

```
-- Clock process
clk_process :process
begin
  clk <= '0';
  wait for clk_period/2;
  clk <= '1';
  wait for clk_period/2;
end process;
```

This is sufficient to generate a clock signal that you can use to control your sequential elements. However, the presence of a clock has significant effects on the timing behaviour of your test patterns!

GOOD DESIGN PRACTICE

Important: note that, in this and all future testbenches, the wait periods between sets of inputs should always be defined as **multiples of the clock cycle**. For example:

```
wait for clock_period*2;
```

or, in some cases:

```
wait for clock_period*7.5;
wait for clock_period*5/2;
```

The only exception to this rule is the initial 100ns "wait" period set by Xilinx, which should never be modified or removed. Any signal assignment in your testbench (except the clock) should only occur after the 100ns wait.

Also note that, if the sequential elements in your design trigger on the rising edge of the clock, **input values in the testbench should change on the falling edge of the clock**. To do this, insert the following line after the 100ns wait period (**and remember to always do this from now on!**):

```
wait until falling_edge(clock);
```

Run the simulation to verify the behaviour of the circuit. What happens when the counters reach the maximum value that can be represented with 4 bits? Test this and keep it in mind for future labs! Finally, synthesize the design and check the Synthesis Report to verify that your code has been interpreted correctly.

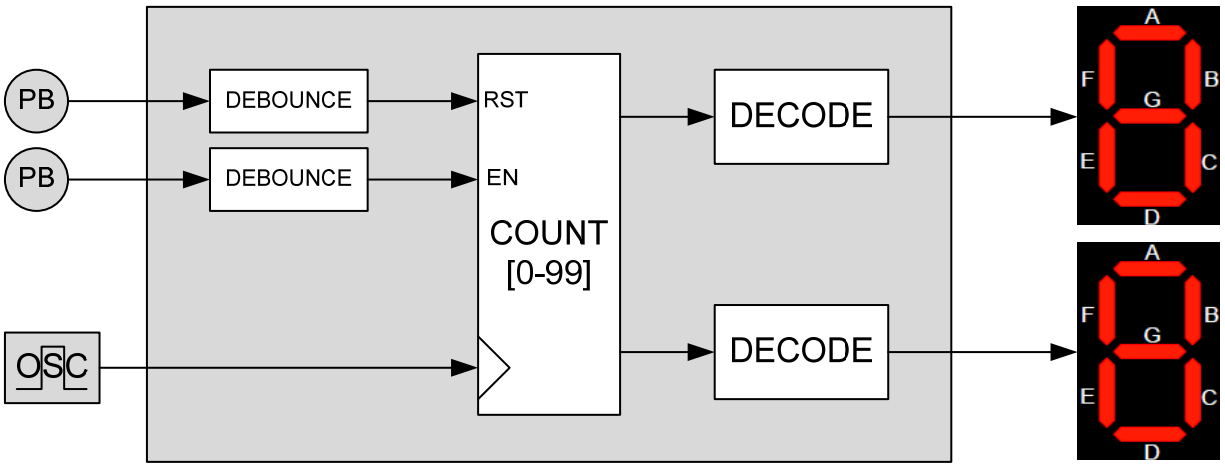
Report:

The report for this task should include, in this order:

- The commented VHDL code for the circuit.
- The commented VHDL code for the testbench (comments should be applied to the stimulus process). Define a sequence of combinations of the three inputs that you feel will adequately test all the differences between the four counter implementations and use comments to explain your choice.
- Screenshot(s) of the simulation, clearly displaying all the differences between the counters, as well as the initial reset, the rollover point of all the counters, and one reset in addition to the initial one (which should clearly illustrate the differences between synchronous and asynchronous resets). You are encouraged to use multiple screenshots if readability is an issue (and it probably will be!). Use hexadecimal notation for the outputs of the counters. Here and in all subsequent reports, remove the "clk_period" default entry from the simulations.
- The "RTL Component Statistics" section of the Synthesis Report.

Task B: Design of a 7-segment display controller

In this task, you will design and simulate a circuit that controls the operation of a 7-segment display.



The circuit will consist of a counter from 0 to 99 that increments by 1 whenever the user presses an *enable* button and goes back to 0 when the user presses a *reset* button (synchronous reset) or when it reaches the limit (99→00). The counter should be implemented as two separate 4-bit counters limited to 9 (and therefore as two separate processes within a single entity), the first (units) enabling the second (tens). Use a single component containing two counter processes.

In addition to the counter, the circuit will consist of two combinational decoders that will transform the value output by the counter into signals to control the seven segments. In practice, the input of each decoder will be the 4-bit vector that encodes the decimal number and its output will be a 7-bit vector where each bit represents one of the segments and will have value '1' if the segment is lit, '0' otherwise. Note that the mapping of the bits is arbitrary but should be specified in the comments to the code (the comments should be self-sufficient and entirely within the code, i.e. do not rely on pictures or descriptions within the report, but only on the text within the code comments). If the input is outside the 0-9 range, the decoders should output the letter E (for error). Hint: the `with-select` construct is ideally suited for this kind of logic structures!

Finally, because the reset and enable inputs are connected to pushbuttons (which are manually operated and therefore both incredibly slow and fully asynchronous), they must be "cleaned up" and shortened. This is the role of special components called *debouncers* (see Appendix A).

Your top-level entity should use components for each design element in the block diagram above:

- Two instances of a debouncer component
- Two instances of a decoder component
- One instance of the COUNT component (containing the two 4-bit counters and the connecting logic)

GDP	All simulations, and indeed all your digital designs, **should always begin operation with a reset of all internal registers and sequential components**. As a corollary, **all sequential components should always include a reset**. Unless you have a very good reason to do otherwise (for example, when I tell you to do so as in the previous task), **all resets should be synchronous**.

Define an appropriate VHDL testbench and simulate the circuit. Note that, when designing the testbench, it will be vital to understand how the debouncers operate or your TB might very well fail to work: keep in mind that the debouncers are specifically designed to ignore any input that is too short. Also, remember (here and for future labs) that humans are incredibly slow input devices, and emulating a button press with a very short pulse is not very good practice.

In your simulations, consider what notation is most appropriate for each of the signals.

Finally, synthesize and check the synthesis report.

Report:

The report for this task should include, in this order:

- The commented VHDL code for the complete circuit (i.e. all entities you have written – include the top level and all your components, less the debouncer). Remember that comments should specify the mapping of the bits to the segments. Print the components top-down (i.e. the top level first, followed by the COUNT and decoder components).
- The commented VHDL code for the top-level testbench (comments should be applied to explain the choice of input stimuli). If you choose not to use a FOR loop (appendix A), do not print pages and pages of identical wait statements, but edit your code and include only a sample in the hand-in.
- Screenshot(s) of the top-level simulation, displaying, in addition to the inputs and outputs of the circuit, the output of the debouncers and of the internal counters. Make sure to include all significant events in the circuit (at least: start, 09-10 transition, 99-00 transition, a reset – and the following resume – beyond the initial one). Note that you will probably need to run the simulation for much longer than the default 1us. All simulations that include a debouncer should always display the output of the debouncers, as this is the signal that will actually be used inside your circuit.

The "RTL Component Statistics" and "RTL Hierarchical Component Statistics" sections of the Synthesis Report, with a short explanation of how the components generated by the synthesis relate to your circuit design. Anything odd?

APPENDIX A: Using FOR loops in testbenches

VHDL syntax includes support for loops. Later on in the module, we will discuss the use of loops in synthesizable VHDL (a complex issue): for the moment, you should **not** use loops in your circuits.

However, loops can be extraordinarily useful in non-synthesizable VHDL, i.e. in testbenches. Once again, the topic will be covered more extensively later on, but for the purposes of this lab it might be quite useful to use FOR loops in the testbenches that require repetitive inputs (notably, tasks 2 and 3).

The syntax of the FOR loop in VHDL is the following:

```
[opt_label :] for parameter in range loop
     [sequential statements]
     [next  [opt_label] [when condition];
     [exit  [opt_label] [when condition];
 end loop[ opt_label];
```

Note that the "next" and "exit" clauses can (and should) be ignored for now.

The following is an example of a FOR loop that generates 100 one-clock pulses on input A, with a distance of 5 clock periods between pulses. The loops should be placed within the test pattern process: execution of the process will continue after the loop has finished.

```
pulse_1x100: for i in 0 to 99 loop
     A <= '1';
     wait for clock_period;     -- 1-period pulse
     A <= '0';
     wait for clock_period*5;   -- 5-period distance between pulses
end loop pulse_1x100;
```

Note that the integer variable i is *implicitly declared* and therefore needs no additional definition or declaration in the code.

APPENDIX B: De-bouncing inputs

Whenever an external source is connected to a Xilinx, the quality of the signal is an issue. This is particularly true of user-operated inputs, such as buttons. In this case, the possibility of *bouncing* is quite high.
To address this problem, Xilinx itself proposes a debouncing circuit in its documentation. The circuit is the following:

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

entity Debouncer is
    Port ( clk : in  STD_LOGIC;
           Sig : in  STD_LOGIC;
           Deb_Sig : out  STD_LOGIC);
end Debouncer;

architecture Behavioral of Debouncer is

signal Q0, Q1, Q2 : STD_LOGIC;

begin

process (clk) is
begin
if rising_edge(clk) then
     Q0 <= Sig;
     Q1 <= Q0;
     Q2 <= Q1;
end if;
end process;

Deb_Sig <= Q0 and Q1 and (not Q2);

end Behavioral;
```

In this laboratory (and in most of the following ones), you should use this debouncer whenever you use a pushbutton to control the operation of your circuit. To do so, create a new (blank) VHDL module called "Debouncer", then copy-paste the code above into the file. The entity should then be instantiated as a component in your top level entity and the debounced signals used in your circuit in place of the direct inputs.

P.S. Note that this de-bouncer is, in reality, quite poor. In reality, this is more of a "place-holder" for a real debouncer than a fully designed circuit. There are good reasons to use this circuit in the early stages of design (a "proper" debouncer has vast consequences on simulations) but do not be too surprised if, once your complete circuit is downloaded to the board, you still observe bouncing effects.

P.P.S. Also note that you will need to understand how this circuit work to use it in your design. Feel free to simulate it if you have any doubts as to its behaviour!