EE 5143 – M4 Lab – Dr Rahman

The goal of this lab assignment is to give you a first introduction to the DE0-Nano board. We'll also learn how to write straightforward VHDL code that makes use of some simple hardware on DE0-Nano. Throughout the various lab assignments you will do, in some cases you will be provided VHDL code and in other cases you will be asked to write some or all of a VHDL code listing. In every case, we'll compile the code and (usually) test it on the board. It is best to first just read through this document with the DE0-Nano board and its user manual handy.

It is assumed that you are in possession of a DE0-Nano board. If not then you will be able to write and compile VHDL code but not download it to the board and observe its behaviour. The board is strictly not needed for submitting this lab assignment but later assignments may need a board to be physically used.

Your DE0-Nano board is a small but very capable Cyclone IV FPGA application development platform. It is made by Terasic in Taiwan. They make a large number of other boards to cater to different Intel FPGAs. Your board is one of the cheapest ones they make. Most other boards are far more expensive because they use high-end FPGAs and/or provide extensive hardware on the boards. A word of warning here: Terasic provide some example code that can be used with their boards but it is generally all written in Verilog. We'll, therefore, write our own software and not use any of their code. If you later learn Verilog then you can benefit from Terasic's code.

Last week you were asked to read the first three chapters of the DE0-Nano user manual. If you have not done that yet then please start by going through those chapters now. Note that wherever the manual mentions Quartus II you can replace it with the new name of the EDA tool: Quartus Prime.

The DE0-Nano board is based on Intel's Cyclone IV FPGA. Specifically, it uses the EP4CE22F17C6N device. Each family (like Cyclone IV) has dozens of devices with slightly different specifications. This particular device has up to 153 I/O pins. Many (but not all) of these pins are available for user access through the pin headers on the DE0-Nano board. There is a so-called USB-blaster programmer on the board which enables the board to be programmed from a computer over a USB interface. Some FPGA boards using Altera FPGAs may not have this feature built in and then an external USB-blaster cable interface is required to program them. This board also has a serial flash memory (EPCS 16) to store configuration bit stream (holds contents of .sof file which is generated every time you compile VHDL code). We'll not use it in the beginning but will make use of it in a later assignment. During most assignments we'll simply download our .sof file directly to the FPGA, bypassing the configuration memory – this is the usual practice during code development. The FPGA will thus get configured and will keep its configuration until you remove power. If you re-apply power then the FPGA will need to be programmed again using the .sof file.

Please note that your FPGA and the board works on 3.3V logic, not 5V logic. Keep this in mind if you connect anything externally to the board. Page 6 in the user manual lists all prominent user-accessible hardware that is present on the board. We'll use a sub-set of these in our lab assignments. Note that there is a very useful 50 MHz clock oscillator present on the board. This clock signal connects to the FPGA through its pin R8. Thus, if you need a clock input in your code

(most circuits do need a clock) then the input clock signal pin (declared in the entity declaration) will have to be explicitly assigned to FPGA pin R8 using the pin planner, after code compilation. At that stage you will assign other pins too and then re-compile the code with your pin assignments in place. Thus, your VHDL code will need to be compiled twice – once before user's pin assignment and then after user's pin assignment.

Check out your board in accordance with component descriptions given at the beginning of chapter 2. Note that in the block diagram of figure 2-3, X8 etc. mean 8 lines i.e. so many wires or tracks connect the FPGA to a given external component.

Next, in order to further familiarize yourself with the board and its hardware components, go through chapter 4 in the user manual. This lets you explore the board using a proprietary user interface developed by Terasic.

Now you are ready to try some VHDL code that can actually work on the board. For the very first exercise, we'll connect the two pushbutton switches on the board to two of the eight LEDs on the board. This shows you how to send switch signals to your FPGA and how to send signals from the FPGA to the LEDs. In this case no processing takes place (except that logic levels are inverted by a NOT operator) and the switch signal is then straightaway sent to the LEDs. Thus, essentially the switches are connected to the LEDs through the FPGA. This is a basic I/O concept – you can do anything else with switch inputs in your program. Note that the reason we invert the switch logic level is because the way Terasic have made the board, the switches give out a logic high signal (3.3V) when not pushed and a logic low signal (0V) when pushed (see circuit diagram in figure 3-1 of the user manual which shows use of two pull-up resistors). We want LEDs to light up when the buttons are pushed down and thus need to convert logic low to logic high, and vice-versa, using the NOT operator. Keep this peculiarity of your switch connection in mind for other code development later on.

The simple VHDL code for you to type in a new project is given on the next page (call it whatever you like but if you use the entity name SWtoLED given here then you will have to name your project SWtoLED).

Note a few things:

1) Library and Use declarations are needed here (these are needed in almost every VHDL program) because of the use of std_logic and std_logic_vector type data objects. These are types that are not built into VHDL and their definitions need to be accessed from an external IEEE-standard library. If you used bit and bit_vector instead then this library is not needed as these types are built into VHDL itself. However, use of bit and bit-vector is quite un-common these days and almost everyone uses std_logic and std_logic_vector in their place.
2) Both sw and led are declared as two-bit buses i.e. two separate signals grouped together. If you need to access their individual bits then those will be referred to as sw[0] and sw[1] (and similarly led[0] and led[1]). But here we can use composite assignments so do not need to refer to the signals individually. This is quite neat. Also notice that in most cases we number the bits in std_logic_vector in descending order as x downto y, as this is more logical in digital electronics due to the way we put MSB at the extreme left and LSB at the extreme right of a string of bits. But you can, of course, also declare a std_logic_vector composite signal object (bus) as x to y, if ascending bit

order is used (is used rarely in practice). Be careful, when comparing two std-logic_vector (or bit_vector) signals both have to be declared in ascending or descending bit order – you cannot mix the two. If you consistently use the descending bit order, which is the usual practice, then there will be no errors. Now copy and paste the entire code below to the Quartus Prime text editor.

```
----------------------------------
--                Library Declaration        --
----------------------------------
-- Like any other programming language, we should declare libraries

library ieee;
use ieee.std_logic_1164.all;

----------------------------------
--                Entity Declaration                --
----------------------------------
-- Here we specify all input/output ports

entity SWtoLED is
        port(
                sw : in std_logic_vector(1 downto 0);
                led : out std_logic_vector(1 downto 0)
        );
end entity;

----------------------------------
--        Architecture Declaration          --
----------------------------------
--        here we put the description code of the design

architecture dataflow of SWtoLED is

begin

led <= not (sw);

end dataflow;
```

---

3) Notice that in mode signals are declared before out mode signals. This can be the other way round too as there is no set rule for this – you can declare interface signals in any order you like but declaring inputs before outputs sounds more reasonable.
4) In the signal assignment statement: led <= not (sw); two bits of sw signal are inverted and assigned to two bits of LED signal (in corresponding numeric order 0 to 0, 1 to 1). This will simply create a straight through connection from the two pushbutton switches to two of the eight LEDs. I read that line as: “led gets not of sw”.

5) You do not need to explicitly write the keyword 'signal' when declaring port signals in an entity because all identifiers declared in an entity are by default assumed to be signals. Most people do not use that keyword in entities. But you can still use the signal keyword, if you want. You can also declare signals inside architectures. These are called internal signals and are not ports so are not accessible from outside of the FPGA. Such signals are only for connections inside the system (used as temporary holders of signal states etc). When declaring signals in architectures, use of the keyword 'signal' is mandatory.

Enter the code in the text editor (follow procedures similar to those described in the previous lab). Then compile it. Fix errors, if any. After first compilation, go into the project directory and locate the folder called output_files. Make sure it contains a file with your project's name and extension .sof. This is the configuration bit stream generated by QP after code compilation.

Next, open pin planner (found under the QP top menu entry 'Assignments') and assign sw[0], sw[1], led[0] and led[1] to FPGA pins J15, E1, A15 and A13, respectively. Simply click on each signal name in the signals list, take it to the pin diagram and drop it on a selected pin. You can see the reason for these particular assignments by looking at pages 13 and 14 of the DE0-Nano user manual. The two pushbutton switches have been permanently connected during manufacture to pins J15 and E1 of the FPGA. The case with LEDs is similar. You thus see the need to keep the user manual handy as it will be frequently used to check connections for pin assignments while developing VHDL code.

The reason for FPGA pins to have names like D3, T9 etc. is that there are some many pins in most FPGAs that these can only be accommodated as a two dimensional array of pins at the bottom of the chip package instead of distributed all around the periphery of the package. Take a look at the Cyclone IV FPGA on your DE0-Nano board and you will see that no pins are visible at the edges. All pins are present underneath the chip and are thus hidden from view. Such packages are called ball grid arrays (BGAs) as the pins are actually tiny solder balls which are soldered to corresponding pads on the board. Because of the 2D pin layout as a grid of balls, the pins are identified in rows and columns using letters and numbers and this explains their labelling scheme.

After carrying out pin assignments in the pin planner run the compiler again and when it has finished your .sof file is bound to your own pin assignments (rather than the default ones automatically generated by QP after the first compilation). Check out the compilation report carefully which gives you a summary of FPGA resource usage. Only thing we have used here is 4 out of 154 or so pins on the FPGA – no logic resources have been used as no logic functions are performed in the code (inversion does not count).

You are now ready to program your FPGA. Follow the procedures given in section 6.9 of the DE0-Nano user manual, beginning on page 77, to program your FPGA. Once successfully programmed, try the two pushbuttons to make sure that they control the two LEDs. Then disconnect power from the board for a few seconds. Apply power again and observe that the switch/LED functionality is lost – the FPGA configuration is volatile. In a later project we'll place the configuration bit stream into the configuration flash memory (requires quite

a few extra steps so we do not do this every time – only when really needed) then the system will not exhibit volatile behaviour.

Place screen shots of your VHDL text, pin planner after your pin assignment, and RTL window in a file as assignment submission. Use either Word or pdf file – no other formats please.

Next we'll start another project whose VHDL listing appears on the next page. Its purpose is to blink an LED on the board. It does this by accepting the 50 MHz clock signal, counting its rising (leading) edges from 0 to 25000000 (takes half a second as this is half of 50 million). Once that happens you toggle an internal variable (you could also have used an internal signal – doesn't matter). This variable is then assigned to an external interface signal called tick_out which is connected to an LED. Pretty simple if you study the code. Note that the VHDL process construct is used here. This is used extremely commonly so get used to it. Code inside a process consists of sequential statements which execute in order of their appearance and only if there is a change (high-to-low or low-to-high) on any of the signals given in the parenthesis after the keyword 'process'. The list, as you probably know by now is called the sensitivity list of the process because the process is sensitive to transitions on those signals (and only on those signals). A process can have multiple signals in its sensitivity list and it will start to execute if there is a logic transition on any one or more of those signals.

Notice that the process has a process label 'clock_1hz_process'. This just identifies the process and serves no other role. Use of a process label is not mandatory but is recommended. Also, observe that two (internal) variables are declared in what is called the declarative region of the process (before the begin keyword). These variables are only visible inside this particular process and keep their values for different invocations of the process. You can also, similarly, declare some constants in this declarative region to use inside the process. However, VERY IMPORTANT, remember you cannot declare signals inside a process. This is never possible. Signals are either interface signals (declared in the entity declaration and visible throughout the VHDL code i.e. visible to all processes) or can be declared in the declarative part (before the 'begin' keyword) of the architecture. Internal signals that are only used inside the code can thus be declared, if needed, at the beginning of the architecture definition in its declarative region.

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

----------------------------------
--                Entity Declaration                --
----------------------------------
-- Here we specify all input/output ports

entity Blinker is

        port(
                clk_50MHz : in std_logic;
                tick_out : out std_logic
```

```
        );

end entity;

architecture RTL of Blinker is


begin


clock_1hz_process : process(clk_50MHz)  -- generates a 1 Hz clock

variable scaler : natural;
variable pulse : std_logic;

        begin

                if(rising_edge(clk_50MHz)) then
                        if (scaler < (25000000-1)) then
                                scaler := scaler + 1;
                        else
                                scaler := 0;  -- reset scaler variable
                                pulse := not(pulse);  -- toggle the pulse variable
                        end if;
                end if;
                tick_out <= pulse;  - assign the logic state of pulse to tick_out signal
end process clock_1hz_process;

end RTL;
```

Study the code above line by line, very carefully. Note that the attribute rising_edge is used with the clock_50MHz signal to detect the 0 to 1 transition. The process gets activated on both 0 to 1 and 1 to 0 transitions of the 50 MHz clock but only the former leads to any activity because of the if statement. Also, note the use of nested if statements which should be familiar to you from computer programming languages. These statements count leading edges of the 50 MHz clock , resetting the count variable (scaler) after 25 million leading edges and also toggling the variable (pulse) then. Especially notice how indentations are used to bring out the structure of the VHDL code. This is very important because some VHDL code can become extremely complex because of many different nested if-then-else statements.

Lastly, note carefully that variable assignments are made using the := operator whereas signal assignments are made using the <= operator. The difference among them is that the former take effect immediately whereas the latter is deferred until the entire process is over. This difference is subtle but important and understanding it is sometimes the key to mastering VHDL because it lies behind the concept of concurrency which is a hallmark of HDLs. Many VHDL programs develop semantic bugs because of imperfect understanding of deferred and immediate assignments. Notice that the last statement in the process assigns the value of pulse to tick_out.

This is allowed because although pulse (variable) and tick_out (signal) belong to different classes of data objects, their type is the same (std_logic) so that they hold the same type of data. Signal assignment operator is used in this statement because the receiving data object is of class signal. Through this last statement in the process body you are sending out the value of the internal std_logic variable (pulse) to an external interface signal (tick_out). Remember that in VHDL only signal objects (std_logic or std_logic_vector) can communicate between the external world and circuitry inside the FPGA.

Oh, one final thing here. Notice that two extra libraries are included at the top of the listing:
use ieee.numeric_std.all;
use IEEE.std_logic_unsigned.all;

These are needed to enable use of numeric comparison, addition, subtraction etc. i.e. arithmetic operations. These operations are defined in those libraries for the type of data objects that are used here (std_logic). If you do not include these libraries but try to use arithmetic operations then the compiler will complain.

Now you know what to do. Follow all the steps as used in the previous project. Compile the code, assign FPGA pins to clk_50MHz and tick_out signals. For the first, there is only one choice: R8. For the second signal, you have 8 different choices as shown on page 14 of the user manual. Make connection to any LED and it will become bound to your code. Compile again, as usual. Program your FPGA and you should see a blinking LED on your DE0-Nano board.

Again, take screen shots of the code window, pin planner window (after your pin assignment) and RTL window, to submit in the assignment. This assignment should contain screen shots for both projects described in this document. Note that the RTL window shows the way QP has configured your circuit using LEs inside the FPGA. This type of a circuit is called a register transfer level circuit as logic levels are essentially transferred from register to register. You do not need to completely understand this circuit but see how the VHDL code has been mapped to circuit blocks inside an FPGA. I always marvel at the fact that VHDL (or any other HDL) allows you to think of your problem in terms of what you want to accomplish and not worry about designing a digital circuit itself – it just gets done for you. Without this type of expressive power many of today's complex digital systems from 5G communications to 4K and 8K TVs could not be designed at all.