EE 5143 – M5 Lab – Dr Rahman

This week's lab assignment provides you with further practice in using QP in conjunction with your DE0-Nano board. This time you will work on two simple but useful projects that often form parts of larger FPGA projects. The first of these shows you how to build a simple counter in VHDL whereas the other gives you practice in building a pulse width modulation (PWM) circuit. With a PWM circuit you can control the intensity of LEDs and other analog hardware connected to your board. These circuits are somewhat more advanced (especially the second one here) than the simple ones we met in the last lab. Like the previous lab assignment, you will be provided the text of both VHDL code listings. You will be asked to create projects, enter VHDL code, compile code, provide pin assignments using Pin Planner, check RTL, program your board and verify the functionality of your code. I am assuming that all of you now have your DE0-Nano board with you.

Start a new project in QP. Call it CountingLEDs. Follow the usual tasks and enter the following VHDL code in the text editor window:

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

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 CountingLEDs is
        port(
                clk_50MHz : in std_logic;
                reset_btn : in std_logic;
                green_leds : out std_logic_vector (7 downto 0)
        );
end CountingLEDs;

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

architecture behave of CountingLEDs is
```

```
-- signal declaration

signal clk_1Hz  : std_logic ;
signal scaler  : integer range 0 to 25000000  ;
signal pre_count: std_logic_vector(7 downto 0);
signal count: std_logic_vector(7 downto 0);

begin

-- clk_1Hz_process process is used to generate a brief pulse once every second

        clk_1Hz_process : process( clk_50MHz , reset_btn )
        begin
                if (reset_btn = '0') then
                        clk_1Hz <= '0';
                        scaler <= 0;
                elsif(rising_edge(clk_50MHz)) then
                        if (scaler < 25000000)   then
                                scaler <= scaler + 1 ;
                                clk_1Hz <= '0';
                        else
                                scaler <= 0;
                                clk_1Hz <= '1';
                        end if;
                end if;
        end process clk_1Hz_process;

-- 8-bit counter process : counts from 0 to 255  and back

  counter_process : process (clk_1Hz, reset_btn)
  begin
     if reset_btn = '0' then
                  pre_count <= "00000000";
     elsif (clk_1Hz='1' and clk_1Hz'event) then
                  pre_count <= pre_count + "1";
     end if;
                  count <= pre_count;
 end process counter_process;

 -- final part of the program

green_leds <= count;

end behave;
```

Now time for some comments on this code listing.

All needed libraries have been included. This time we are using all eight LEDs so have declared an 8-bit std_logic_vector called green_leds with mode 'out'. We are using one of the two buttons as a reset button to reset the counter value to zero whenever we press that button. The other pushbutton remains unused. Looking at the entity declaration, a total of ten FPGA pins will be used: one to connect to the 50 MHz clock oscillator, one to connect to one of the pushbuttons for generating a reset and eight pins to connect to eight LEDs. It is clear that modes 'in' and 'out' are assigned from the point-of-view of the FPGA: signals that come to the FPGA are assigned mode 'in' while those that go out of the FPGA are assigned mode 'out'. Later, we'll also have occasion to use modes 'inout' and 'buffer'.

The declarative part of the architecture declares four signals for internal use. These are not seen from outside of the FPGA and can only be used for internal computations, book-keeping etc. These signals generate internal nodes inside the FPGA meaning that they exist as real signals inside the chip but are not accessible from the outside. However, there may be occasions when for debugging purposes you may need to see how an internal signal node behaves during run time. QP makes it possible to probe the logic state of internal nodes using a special utility called SignalProbe (available under Tools menu). SignalProbe makes it possible to pull out an internal node to a user-designated external FPGA pin. Once that is done, the logic state of that node can be monitored using a multimeter, oscilloscope or logic analyzer connected to the designated pin. Note that a signal cannot just be of type std_logic and std_logic_vector which we associate with normal digital logic levels. A signal can also be of type integer (or natural, which is a subset of type integer) etc. A signal will always only hold data of the type it is declared to be. Conversely, a variable can be of type std_logic and thus hold logic value. The only major difference in VHDL is that assignments to a variable (:=) are immediate whereas assignments to signals (<=) are always deferred. Instead of declaring scaler, pre_count and count as signals we could also have declared them as variables and in fact that would be the more logical thing to do here. clk_1Hz, on the other hand, has to be declared and used as a signal (why? See below). One final point here: VHDL is what is called a strongly typed language which means that during compile time rigorous type checking ensures that you always make assignments between data objects of the same type i.e. the type of source and destination objects must match in all assignment statements, whether immediate or deferred.

How many concurrent (simultaneously executing) statements are there in the code given above? The correct answer is three. This listing has two processes (you can have any number of processes in VHDL code). Each of those counts as a single concurrent statement although inside the process body there may be any number of statements. All statements present inside any process are sequential. The two processes and the last statement: green_leds <= count; make for three concurrent statements in the code listing above.

The two processes here are such that the second process depends on the first as they communicate using a common signal clk_1Hz. The first process is sensitive to the 50 MHz clock signal clk_50MHz as well as the signal reset_btn. It counts clock pulses and issues just one logic high pulse every second for just one-fifty-millionth part of a second. This pulse is the clk_1Hz signal. The second process monitors the clk_1Hz signal (as well as the reset_btn signal) and performs the actual counting function. The two processes thus perform separate functions: the first scaling down the 50 MHz clock to generate 1 second ticks and the second using 1 second ticks to count (and reset, if needed). This idea of separating functions into dedicated VHDL processes is very useful. You can always use signals for inter-process communications, as seen here. Just remember that no signals can be declared inside any process (although internal constants and variables can be declared). Signals for communicating between processes can only be declared in the declarative region of an architecture body. Then they are visible to all processes inside that architecture body. Now you should understand why clk_1Hz has to be declared as a signal – because its value has to be monitored by the second process and only signals can be present in a process' sensitivity list.

Also notice the use of (clk_1Hz='1' and clk_1Hz'event) in the second process. This could also have been coded as (rising_edge(clk_1Hz). The two constructs are functionally equivalent. The first form was more prevalent in the past but most coders nowadays use the second form. They both appear here just to illustrate this point.

Finally note that in the second process we say, pre_count <= pre_count + "1"; This is as it should be because we are adding 1 not as a number but as a std_logic_vector because pre_count is a std_logic_vector. Again this is because VHDL, as stated above, is a strongly typed language.

The last statement simply assigns the 8-bit value of count to the eight LEDs. I personally read <= symbol as 'gets' and will say green_leds gets count.

Please go over the entire code listing again, in light of the above comments, and make sure that you understand everything. In case of any queries, I am always available.

Compile your code, as usual. This time take a good look at the compilation report that comes up at the conclusion of compilation. Notice that this design made use of 37 out of 22,320 logic elements (LEs) available in a Cyclone IV FPGA. This is much less than 1% of all LEs contained in one Cyclone IV chip. This should give you some idea of the complexity of circuits that can be contained in one rather low-end FPGA, like Cyclone IV. Larger FPGAs, costing hundreds of dollars each, contain millions of LEs. Not only can you place quite complex digital circuits inside an FPGA, you can also place a number of DIFFERENT circuits in the same FPGA. This way the large number of I/O pins and LEs in an FPGA are more effectively utilized. In fact, many FPGAs are sold for this kind of 'logic consolidation' where manufacturers use one chip to replace the functions of several different separate ICs (maybe from an older model of the equipment).

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.

Program the Cyclone IV FPGA on your DE0-Nano board and see the LEDs counting in a binary pattern. They will complete one cycle every 256 seconds but you can rest the count to zero any time by pressing the rest pushbutton.

Next we'll start another project whose VHDL listing appears on the next page. Its purpose is to perform PWM control of the brightness of all eight LEDs. By pressing a button you can brighten all LEDs together. By pressing a second button you can decrease the brightness of all eight LEDs. PWM is a powerful and widely used method to perform analog control using digital hardware such as microcontrollers and FPGAs. It is commonly used for controlling motor speed, brightness of luminaires and even for sound synthesis. In fact, if you connect a small capacitor to an FPGA pin where you take out your PWM output waveform then you have essentially built a digital-to-analog convertor (DAC).

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

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

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

entity PWMLED is
          port(
                    clk_50MHz : in std_logic ;
                    up_btn : in std_logic;
                    dn_btn : in std_logic;
                    pwm_leds : out std_logic_vector(7 downto 0) := "00000000"
          );
end PWMLED;

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

architecture behave of PWMLED is

-- data object declarations are placed in the declarative part of the architecture body
constant max : positive := 2500;
signal clk_1Hz  : std_logic ;
signal scaler  : integer range 0 to max  := 0;
signal t_on : integer range 0 to 100 := 0;
signal top_cnt : integer := 100;
signal count : integer range 0 to 100  := 0;
signal pwm_signal : std_logic;
signal up_btn_state : std_logic := '1';
signal dn_btn_state : std_logic := '1';

begin

-- this process is used to scale down the 50MHz frequency to 50MHz/max i.e. 20KHz which is the
frequency of our PWM waveform (it has to be high enough that the LEDs don't appear to blink)

          clk_1Hz_process : process( clk_50MHz )
          begin
```

```
                    if(rising_edge(clk_50MHz)) then
                              if (scaler < max) then
                                        scaler <= scaler + 1 ;
                                        clk_1Hz <= '0';
                              else
                                        scaler <= 0;
                                        clk_1Hz <= '1';
                              end if;
                    end if;
          end process clk_1Hz_process;
```

-- This process is used to read pwm rate control buttons (up_btn and dn_btn) and set the duty cycle by setting the value of LED on time : t_on

```
  button_process : process( clk_1Hz )
          begin
          if(rising_edge(clk_1Hz)) then
          if(up_btn = '0' and up_btn_state = '1') then  if (t_on < 100) then t_on <= t_on + 1; else
t_on <= 100; end if; end if;  up_btn_state <= up_btn;
     if(dn_btn = '0' and dn_btn_state = '1') then  if (t_on > 0) then t_on <= t_on - 1; else t_on <=
0; end if; end if;  dn_btn_state <= dn_btn;
          end if;
          end process button_process;
```

-- This process is used to actually generate the PWM waveform

```
  pwm_process : process( clk_1Hz )
  begin
           if (rising_edge(clk_1Hz)) then
                    if (t_on = 0) then pwm_signal <= '0';
                  elsif((count <= t_on) and (count < top_cnt)) then
                       pwm_signal <= '1';
                   count <= count + 1;
                      elsif ((count > t_on) and (count < top_cnt)) then
                       pwm_signal <= '0';
                       count <= count + 1;
                  else pwm_signal <= '0'; count <= 0;
                    end if;
           end if;

          for i in 7 downto 0 loop
            pwm_leds(i) <= pwm_signal;
          end loop;
  end process pwm_process;

end behave;
```

This code listing has three processes in it. All the work gets done in those three processes. Looking at the entity declaration, two buttons are used for turning LED brightness up and down so these will consume two FPGA pins. Another pin will be used to receive the 50 MHz clock signal. Eight other FPGA pins will connect to the eight LEDs. Thus, in total 11 FPGA pins will be used (as you will see when you look at the compilation report after code compilation).

One constant and several internal signals are declared in the declarative part of the architecture body. Notice how the clk_1Hz_process works: it counts max (2500) pulses from the 50 MHz clock while keeping the clk_1Hz signal at logic low. When it has counted exactly 2500 pulses (logic highs) then it raises clk_1Hz to logic high for just one time period of the clk_50MHz signal. Then it goes to logic low again. Thus, the clk_1Hz signal stays at logic low going to logic high only for an extremely short duration before going back to logic low again. Thus, clk_1Hz is not an appropriate name for this signal (but the compiler doesn't care). Replace all instances of clk_1Hz by clk_tick (use the find and replace facility in the text editor). Note that this signal generated by the first process is in the sensitivity list of both the other processes. It is simply used as the clock which times operations performed in those two processes.

The second process reads the up and down buttons and updates the value of t_on which is used in the third process. It contains nested if-then-else loops. These are not correctly indented so insert appropriate new lines and spaces to properly format statements in this process (study the formatting of nested statements in the first and third processes to see how you will accomplish this). Note that the compiler doesn't care about such niceties as proper indented line formatting in nested statements – it is only for us humans and makes it much easier to understand the logical structure of code. Make sure you do this formatting (although the code will compile correctly even without formatting as the syntax and semantics are correct). Your code screen shot in your lab assignment must show correctly formatted code to get full marks.

The third process actually generates the PWM signal. It is divided into two parts. The first part with the nested if statements is where the main action takes place. If you don't know how PWM works then read about it on the web. It is not difficult to understand. Notice something interesting and that is the for loop seen at the end of the third process. It uses a loop variable (an integer) called i. It can be any other valid VHDL identifier name of your choice. This loop control variable does not need to be declared (although you can explicitly declare it if you so wish). It is automatically created by VHDL when the compiler comes across it in a for statement. Now the more interesting part. Whereas in an ordinary software programming language a for loop will execute one or more statements a set number of times, in VHDL a for loop results in the creating of a set number of identical circuits! This shows you clearly that while appearing similar to software programming languages, VHDL is something totally different – it is a description of a circuit to be synthesized. Always keep this in mind and your code will contain fewer bugs. Thus, if you write a comparison statement like 'if (a < b) then ….' the result is not execution of an arithmetic or logical expression but rather the creation of a circuit that could compare two numeric values. Knowing that a for loop replicates circuits, always use this statement with care. If you place an addition statement inside a for loop that loops ten times then the result will be the creation of ten addition circuits! If used carelessly, a for loop can quickly use up a lot of FPGAs resources by replicating large circuits many times. Here, the loop simply creates eight connections – each time connecting the single bit pwm_signal line to one line of the 8-bit pwm_leds bus. Thus, all LEDs will be driven by the same signal and will brighten up or dim down in unison.

If you look at the RTL view and compilation report of this code then you will notice that due to the higher complexity of this code listing the synthesized circuit is larger and consumes more FPGA LEs.

That is it for this week's lab. Prepare both projects in a similar manner to the last week's lab and submit six screen shots (code, pin planner, RTL for each) in a Word or pdf file. Also, download both projects to your DE0-Nano board to verify that the LEDs and buttons operate in the expected manner.