In this lab you will combine the techniques you learned in the previous labs and new ones to design an electric wheel of fortune on the Nexys-2 board. New coding techniques introduced in this lab are building vhdl modules using packages and procedures. You have already used process for sequential statements within the main body of architecture. Procedures are similar to processes, but the difference is that they can be placed in a package and can be called from any vhdl module. Recall also that package is a part of library. Typically, functions, procedures, and global declarations are placed in a package for large design projects. To learn more, please read the topics on packages and procedures in the “XST User Guide” which is available from the Help>Software Manuals.

Part 1 is a tutorial on how to handle the Seven Segment Displays (SSD) and on how to use packages and procedures. In Step 1, the procedure is embedded in architecture. In Step 2, you learn how to put the procedure in a package. In Step 3, you will learn how to program the on-board EPROM, so that your program stays on the board even if you turn off the power. Part 2 is the main part of the lab that you will design an electronic wheel of the fortune, utilizing the package and procedure constructs you learned.

Part 1: Seven Segment Display (SSD) programming using procedures and package

You are going to use procedure and package structures in this programming practice. In the first step, a procedure is embedded within an architecture construct. In the second step, the procedure will be moved to a package and it will be called from the architecture.

The example is a simple counter control, which works as follows. The system consists of a switch and SSDs. When the control switch (SW0) is on, counting continues and the number is displayed on the SSD. When the control switch is off, the counting is stopped and the SSD is turned off. The 50MHz clock on the board is used to run the counter.

Step 1: Embedding the procedure within the main vhdl source code.
Create a new project.
Add a new vhdl module source. Use the entity provided below for constructing the source.

```vhdl
entity ssegprocedure is
   Port ( clk : in STD_LOGIC; -- master clock
           sw0 : in STD_LOGIC;
           smodes : out STD_LOGIC_VECTOR (3 downto 0);
           ssseg : out STD_LOGIC_VECTOR (7 downto 0));
end ssegprocedure;
```
Add the signals and a procedure called ssg_decode() given in the next page. This code should be placed between the line “architecture” and “begin” in the ssegprocedures vhdl code. You need to add more SSD bit patterns by referencing Figure 1.

--- Signals
signal clkdiv : std_logic_vector (24 downto 0);
signal cntr : std_logic_vector (3 downto 0);
signal clk : std_logic; -- low frequency clock

--- Procedures
-- It decodes from 4-bit binary to SSD pattern.
procedure ssg_decode(signal hexcode : in std_logic_vector (3 downto 0);
                      signal ssg_out : out std_logic_vector (7 downto 0)) is
    variable temp : std_logic_vector (7 downto 0);
begin
    case hexcode is
      when "0000" => temp (6 downto 0) := "01111111";
      when "0001" => temp (6 downto 0) := "00001101";
      when "0010" => temp (6 downto 0) := "10110111";
      when "0011" => temp (6 downto 0) := "10011111";
      when "0100" => temp (6 downto 0) := "11001110";
      when "0101" => temp (6 downto 0) := "11111001";
      when "0110" => temp (6 downto 0) := "11110111";
      when "0111" => temp (6 downto 0) := "00000111";
      when "1000" => temp (6 downto 0) := "11110111";
      when "1001" => temp (6 downto 0) := "10011111";
      when others => temp (6 downto 0) := "00000000";
    end case;

    -- Your turn: Add the rest of display patterns, A, B, C, D, E, F
    -- bit '1' turns on the corresponding segment.
    ssg_out <= not temp;
    ssg_out(?) <= '1'; --
end ssg_decode;

Note that when you use a variable in the procedure, the assignment becomes “:=” instead of “<=”. Variables in vhdl should be thought of as a temporary storage. Since signals cannot be stored in a temporary storage you will get an error if you try to assign a signal to a variable. For example,

```
temp <= ssg_out;
```

will give an error while

```
ssg_out <= temp;
```

will not.

Next, add the following code (next page) in the main body of the architecture. Three processes work concurrently to make a slow running clock and to display the number on the SSD and to control using a switch.
Note on how the procedure ssg_decode() was called and how signals are mapped in the procedure. Also, note that the procedure is a concurrent statement along with other processes.

Next, assign the pins as follows to test the application.

anodes(0) → F17
anodes(1) → H17
anodes(2) → C18
anodes(3) → F15

mclk → B8 ; it is 50Mhz clock

ssseg(0) → L18
ssseg(1) → F18
ssseg(2) → D17
ssseg(3) → D16
ssseg(4) → G14
ssseg(5) → J17
ssegs(6) $\rightarrow$ H14
ssegs(7) $\rightarrow$ C17

SW0 $\rightarrow$ G18

Please also refer to the I/O pin connections in the following diagram.

Figure 1: I/O connection diagram.

Generate the bit file and download it to the Nexys-2 board. The SSD should display “0000” to “9999” when SW0=’1’.

**Step 2: Procedure in a package**

This time we are going to separate the procedure from the vhdl source code in Step 1 and move it to a package called `mypackage`.

To start the process, right click from the Sources window to get into the New Source Wizard. This time select VHDL Package and name the package as “mypackage” as shown below.
Click **Next** and then **Finish** on the subsequent dialog. When you finish the New Source Wizard, it will create mypackage.vhd with a template that includes the basic structure, example declarations, an example function and an example procedure. Remove most of them except the main constructs of the package. Edit the source, so that its content looks like the code shown below.

```
package mypackage is

  procedure ssg_decode(ssignal hexcode : in std_logic_vector (3 downto 0);
    signal ssg_out : out std_logic_vector (7 downto 0));
end mypackage;

package body mypackage is

  -- Seven segment decoder procedure
  procedure ssg_decode(ssignal hexcode : in std_logic_vector (3 downto 0);
    signal ssg_out : out std_logic_vector (7 downto 0)) is
    variable temp : std_logic_vector (7 downto 0);
    begin
      case hexcode is
        when "0000" => temp (6 downto 0) := "0111111";
        when "0001" => temp (6 downto 0) := "0000110";
        when "0010" => temp (6 downto 0) := "1011011";
        when "0011" => temp (6 downto 0) := "1001111";
        when "0100" => temp (6 downto 0) := "1100110";
        when "0101" => temp (6 downto 0) := "1101101";
        when "0110" => temp (6 downto 0) := "1111101";
        when "0111" => temp (6 downto 0) := "0000111";
        when "1000" => temp (6 downto 0) := "1111111";
        when "1001" => temp (6 downto 0) := "1101111";
        when others => temp (6 downto 0) := "0000000";
      end case;
      ssg_out <= not temp;
      ssg_out (7) <= '1';
    end ssg_decode;
  end ssg_decode;
end mypackage;
```

Save and do **“Check Syntax”** to make sure no syntax error exists in the mypackage.
Next, move back to the main vhdl module and modify the source to use the sseg_decode procedure in the mypackage. First, add phrase “use work.mypackage.all” in the library declaration section, i.e.,

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.mypackage.all;
```

With this declaration, you are now ready to use this procedure in the main vhdl source.

Edit the main source by removing the procedure. The architecture portion of the source code should look like the code shown below.

```vhdl
architecture Behavioral of ssegdemo is

--- Signals
signal clkd: std_logic_vector (24 downto 0);
signal cntr: std_logic_vector (3 downto 0);
signal oclkl: std_logic; -- low frequency clock

begin

-- procedure call from mypackage
sseg_decode(hexcode => cntr, ssg_out => sseg);

-- sseg on/off control by sw0
process (sw0)
begin
  if sw0 = '1' then
    aodes <= "0000";
  else
    aodes <= "1111";
  end if;
end process;

-- divide the master clock to a lower frequency
process (mclk)
begin
  if mclk='1' and mclk'event then
    clkd <= clkd + 1;
  end if;
end process;

oclk <= clkd(24);

-- counter process
process (oclkl)
begin
  if oclkl = '1' and oclkl'event then
    if cntr = "1001" then
      cntr <= "0000";
    else
      cntr <= cntr + 1;
    end if;
  end if;
end process;
end Behavioral;
```
Check the pin assignments and generate the bit file. Download and test the board. It should work the same way as in the previous case.

**Step 3: PROM Download**

Until now you only downloaded your generated bits to SRAM. If the board power is turned off, the bits downloaded are lost. In order for the FPGA to run without downloading the bits when the power is turned off and on, the bits must be stored in an on-board PROM. This step practices that procedure.

Store the above program into the PROM by following two steps below.

1) Important!! Re-generate the bit file by setting
   **Process Properties … > Startup Options > FPGA Start-up Clock** set to CCLK
   (for SRAM download you used the **JTAG Clock** option).

2) Download the bits into the on-board PROM using the browse button in the second row [PROM XCF04S] of the **Digilent Adept** software.

If the bit file is successfully downloaded to the PROM, do the following steps.

- Turn off the board using the on/off power slide switch
- Wait about 3 seconds and then turn back the power on (do not turn back on too fast. It may damage the board)
- Try the reset button.

Notice that your program is running even though you re-circulated the power. This means the bit file is permanently stored in non-volatile memory. This is the way you program your bit file into the EPROM after the development is completed.
Part 2: Implementation of the LED wheel-of-fortune using a package

The LED wheel of fortune you are designing works as follows.

At the startup, the LED lights continuously rotate around using a bit pattern “00000001”, and at the same time the seven segment display (SSD) continuously increments a four digit hex number (16 bit) at a fast speed. The digit starts from 0000 and increments to FFFF and repeats.

When a user presses a button, the LED rotation stops and the SSD displays the number caught at the positive transition of the button press. When the user presses the button again, it starts the whole process again, i.e., the LED lights rotate around, and SSD increments the four digit hex number at a fast speed. The LED movements should be slow enough to be visible. However, the numbers on the SSD should be fast enough, so that it is hard to recognize except when it is stopped.

In order to accomplish this, several things must happen.

1. Add more SSD patterns, so that it can display 0-9 plus A-F.
2. Modify the procedure, so that it can decode a 16-bit number instead of just a four bit number as given.
3. You need to generate another clock that controls the anodes of SSD. This clock should run around 1/60th seconds in order to display different numbers because cathodes are shared.
4. Rotate bits and display the bits on LED. Synchronize the LED pattern to a slow clock, so that the rotation would be visible. Rotating LED can be done using concatenation, for example, rotating a bit to left can be implemented by, `led <= led(6 downto 0) & led(7)`.
5. Write a push button code to detect positive transition and to take a proper action.

Check-off
* In the lab, you just need to show the Part 2 demonstration.

Report
* Objective of this lab and intro.
* Summarize your implementation
* Your conclusion
* Attach your VHDL source code

* Due date for the report is the following Monday of the lab week.