Counter

Task:

Implement a design that would count how many times a pushbutton has been pressed and display this value on LED-s in binary form. The length of displayed value is not strictly defined, it can be e.g. 5, 6, 7 or 8 bits. The counted value should increment by one each time the button is pressed (i.e. it should not increase more then once until the button is released and pressed again). Following is the list of steps for the counter design:

- implement a counter that is enabled for one clock cycle each time a pushbutton is pressed. Note, that at this step the counter may increment more than once after a single button press due to bouncing effect;
- add debouncer for the employed pushbutton. Then the counter should increment only once after a single button press since bouncing effect is eliminated.

Simulate and implement each design step on FPGA development board. For clock input use 100MHz crystal oscillator clock source on the Nexys-4 FPGA board (use pin E3 or uncomment two lines after "## Clock signal" in the Master XDC File).

Edge Detection and Push Button Debounce Circuit

The most straightforward way to detect a button press is to detect a transition when button input goes from LOW (unpressed) to HIGH (pressed). This can be done by sampling the button input and storing the last two values. Whenever these two values are different, it is possible to conclude that there has been a transition in the button input signal. An example in Listing 1 shows a rising edge detector that generates a single one clock period long pulse when the button signal changes from LOW to HIGH. A falling edge detector (that shows when the button is released) can be implemented in similar fashion by changing the *button_pulse* signal equation to detect *button_buf1* = '0' and *button_buf2* = '1' condition.

Listing 1: Rising Edge Single Pulse Generation

```
button_buf1 <= button_input_signal when rising_edge(clock_100MHz);
button_buf2 <= button_buf1 when rising_edge(clock_100MHz);
button_pulse <= button_buf1 and not button_buf2;
```

Another issue may arise due to the bounce effect. The problem is that Push Button contains a metal spring and during press it actually makes contact several times before stabilizing. The high-speed logic circuits may react to the contact bounce as if the Push Button has been pressed several times, because some pulses may have a sufficient voltage and long enough duration. Thus, for the hardware implementation to work correctly, contact bounce must be filtered.

A 4-bit shift register, which is shifted at approximately 100Hz can do the trick. With each shift it saves (samples) values from button's input. Only when all four bits of the shift register are HIGH, the debouncer's output goes also HIGH. This will delay the LOW to HIGH change until the contact bounce stops.

Nexys-4 FPGA board features a 100MHz crystal oscillator clock source. Thus, it is required to slow it down for the debouncer's shift register. Normally, a clock manager device should be used to modify the clock signal. However, since the desired clock for this task is too slow, a possible solution may be to employ a simple delay counter instead. This counter should count the clock cycles of the system clock and signals when it reaches a predefined value that correspond to the required delay (at the same time the counter itself resets and starts counting from the beginning).

Possible VHDL description of such delay counter that generates a single one clock period long pulse per hundred clock cycles is presented in Listing 2. If the frequency of the *clock_100MHz* clock signal is 100MHz, then *clock_1MHz* signal would go HIGH for one *clock_100MHz* clock cycle with 1MHz frequency. It is then possible to use *clock_1MHz* signal as shift enable for the debouncer's shift register to perform button value sampling once per 1000 ns. Note, that in Listing 2 the counter is declared as an integer. This way there is no need to know how wide the counter should be (in bits), as it will be determined during synthesis automatically (as long as the counting range is specified as well).

Listing 2: Delay Counter Example

```
signal clock_1MHz: std_logic;
signal counter: integer range 0 to 99;
begin

process (clock_100MHz)
begin
 if clock_100MHz'event and clock_100MHz = '1' then
   if counter < 99 then
     counter  <= counter + 1;
     clock_1MHz <= ’0’;
   else
     counter <= 0;
     clock_1MHz <= ’1’;
   end if;
 end if;
end process;
```