Chapter-7

Clocking

This chapter will be focus on one of the most important component of a sequential process or design i.e. clocking. Clocking is an important issue most of the FPGA implementations. Essentially, using clock in a design means that your system is subject of time. Simply, clock is a square wave so it has a period and frequency. In general, speed of the design depends of its clock frequency besides other factors like complexity and phase delays in between blocks of the overall design. Clock frequency of a microcontroller or a FPGA gives a clue about its performance at first glance. For instance, nowadays computers work with up to 4 GHz clock frequencies while FPGA works with maximum 700 MHz. However, it won't be true to compare a microcontroller (in this case computer CPU) with a FPGA in terms of their clock frequencies. Since, FPGA has a capacity of doing calculations in a parallel manner it can over perform a CPU which has much higher clock frequency.

Nexys 4 DDR programming kit has a crystal that supports 100 MHz clock to FPGA. If anyone wants to increase the clock frequency of any FPGA, a phase lock loop (PLL) based module should be used. Every FPGA chip has this module and it can be activated through calling it to design via using component feature of the VHDL. This topic will be explained in forthcoming chapters.

A FPGA uses clock pulses to accomplish sequential processes. Rising or falling edge of the clock is used to begin a process and continue an ongoing process. Figure 7-1 shows the rising and falling edge of a clock pulse.

Figure 7-1

Example: Design a 3-bit binary up counter. Counter value should be increased by one in every rising edge of the clock pulse. Test your code in ISIM simulator. Use 100 MHz clock during simulation.

Solution: Step-1) As the first step to achive our task, we need to open a new project. After setting project file, ISE Design Suite asks for the FPGA chip family, device, package and speed properties of the chosen FPGA. Set the FPGA chip of Nexys 4 DDR programming kit. As mentioned before, these properties can be achived simply by reading the top side of the FPGA chip.

Step-2) After configuration, click next. An empty project with XC7A100T is created. Open a new VHDL module and name it as "three_bit_up_counter". Copy below program segment PS 7.1.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity three_bit_up_counter is
  port( clk_in: in std_logic;
        counter : out std_logic_vector(2 downto 0));
end entity;

architecture logic_flow of three_bit_up_counter is

begin
process (clk_in)
variable temp: integer range 0 to 7;
begin
        if (rising_edge(clk_in)) then
            temp := temp+1;
        end if;
        counter <=conv_std_logic_vector(temp,3);
end process;
end architecture;
```

PS 7.1

Step-3) Our implementation is done. Next step is to do simulation of this counter. Follow the steps that are explained in previous chapters. Name your test bench as "three_bit_up_counter_TB". At the end, your project window and test bench code should be look like in Figure 7-2.

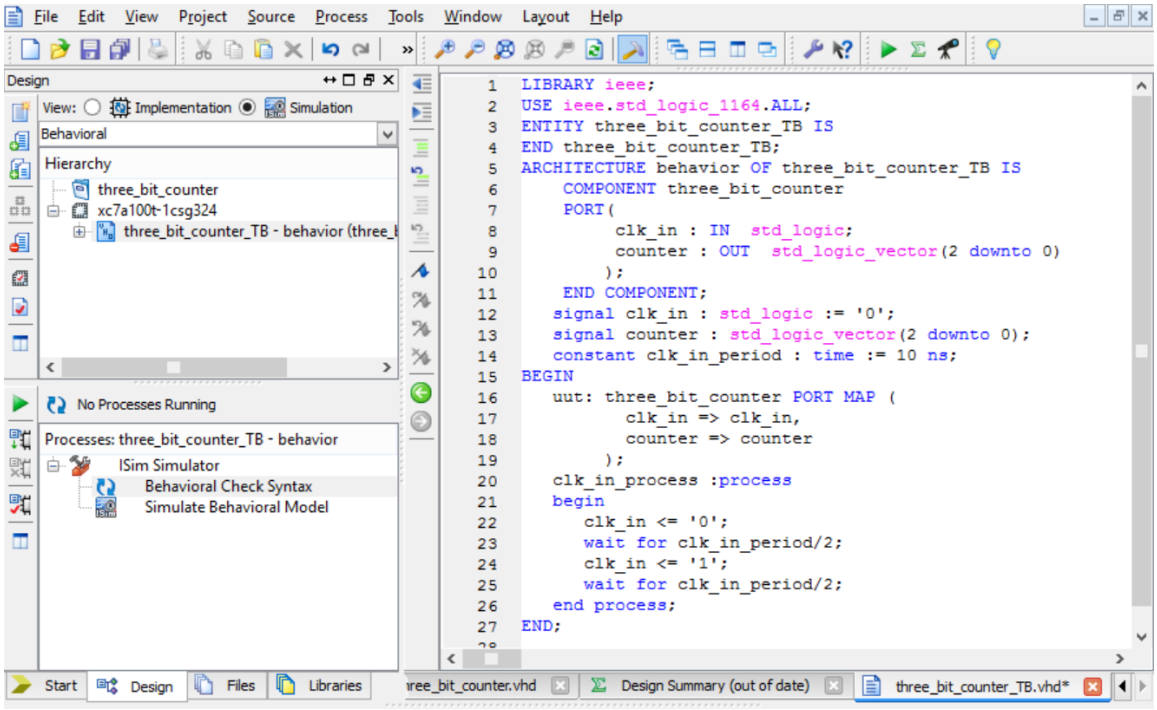

Figure 7-2

Step-4) Simulation result of 3-bit up counter is given below in Figure 7-3. Check your results.

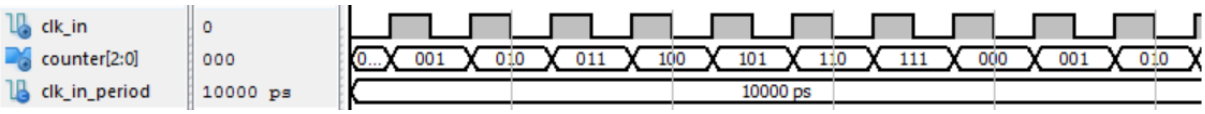

Figure 7-3

As noticed from the Figure 7-4, counter value is increased every rising edge of the clock signal. In PS 7.1, there is one item ("clk_in") in the sensitivity list of the **process** so "rising_edge" keyword understands that command is valid for the input "clk_in". However, what happens there are two or more items in the sensitivity list. PS 7.2 shows a different VHDL coding to get rid of this confusion.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity three_bit_up_counter is
  port( clk_in: in std_logic;
        counter : out std_logic_vector(2 downto 0));
end entity;

architecture logic_flow of three_bit_up_counter is

begin
process (clk_in)
variable temp: integer range 0 to 7;
begin
        if (clk_in'event and clk_in='1') then
            temp := temp+1;
        end if;
        counter <=conv_std_logic_vector(temp,3);
end process;
end architecture;
```

PS 7.2

PS 7.2 also gives same result with Figure 7-2. This issues should be taken into consideration when there more than one item in the sensitivity list of a **process**.

Exercise: Design a 3-bit binary up counter. Counter value should be increased in every falling edge of the clock pulse. Test your code in ISIM simulator. Use 100 MHz clock during simulation.

Outputs of the PS 7.1 and PS 7.2 can be observed from ISIM simulator easily. However, when observation on our programming kit a problem emerges. Assume that we define 3 LEDs for the output "count". If we run our VHDL design, we will see that all the LEDs are on permanently like there is no counter. This is because of human eye can't perceive more than 20 Hz where our clock frequency is 100 MHz. In order to observe counter result we can divide the clock frequency by adding a clock divider process to our VHDL design.

Example: Design a clock divider. Assume that you have 100 MHz clock. Clock divider output should be set to 50 MHz. Test your code in ISIM simulator. Program your Nexys 4 DDR programming kit. Original clock and divided clock should be observed from an oscilloscope.

Solution: Step-1) As the first step to achive our task, we need to open a new project. After setting project file, ISE Design Suite asks for the FPGA chip family, device, package and speed properties of the chosen FPGA. Set the FPGA chip of Nexys 4 DDR programming kit.

Step-2) After configuration, click next. An empty project with XC7A100T is created. Open a new VHDL module and name it as "simple_clock_divider". Copy below program segment PS 7.3.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity simple_clock_divider is
  port( clk_in: in std_logic;
          clk_out_original, clk_out_divided : out std_logic);
end entity;

architecture logic_flow of simple_clock_divider is
signal count: natural range 0 to 1;
signal temp_clk_out: std_logic:='0';
begin
clk_out_original <= clk_in;
process(clk_in)
  begin
    if (rising_edge(clk_in)) then
       count <= count + 1;
       if(count=0) then
          temp_clk_out<=not temp_clk_out;
          count<=0;
         end if;
      end if;
 end process;
 clk_out_divided <= temp_clk_out;
end architecture;
```

PS 7.3

Step-3) Clock divider is implemented. Open a VHDL Test Bench and name it as "simple_clock_divider_TB". At the end, your project window and test bench code should be look like in Figure 7-4.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY simple_clock_divider_TB IS
END simple_clock_divider_TB;
ARCHITECTURE behavior OF simple_clock_divider_TB IS
    COMPONENT simple_clock_divider
    PORT(
         clk_in : IN  std_logic;
         clk_out_original : OUT  std_logic;
         clk_out_divided : OUT  std_logic
        );
    END COMPONENT;
   signal clk_in : std_logic := '0';
   signal clk_out_original : std_logic;
   signal clk_out_divided : std_logic;
   constant clk_in_period : time := 10 ns;
BEGIN
   uut: simple_clock_divider PORT MAP (
          clk_in => clk_in,
          clk_out_original => clk_out_original,
          clk_out_divided => clk_out_divided
        );
   clk_in_process :process
   begin
      clk_in <= '0';
      wait for clk_in_period/2;
      clk_in <= '1';
      wait for clk_in_period/2;
   end process;
END;
```

Figure 7-4

Step-4) Simulation result of clock divider is given below in Figure 7-5. Compare your results.

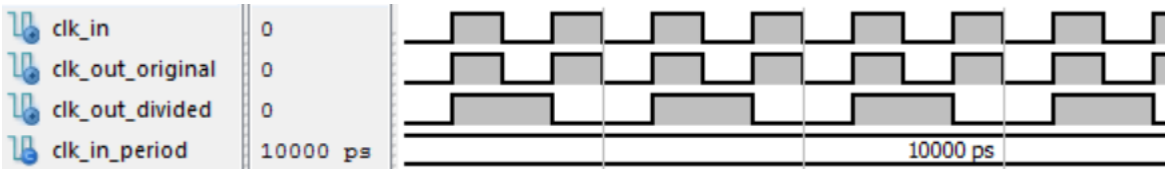

Figure 7-5

Step-5) In this step, we need to choose pins that can be connected to an oscilloscope. Designers of the Nexys 4 DDR programming kit put some connecters to achieve such an operation. These are named **Pmod ports.** Figure 7-6 shows the **Pmod ports**.

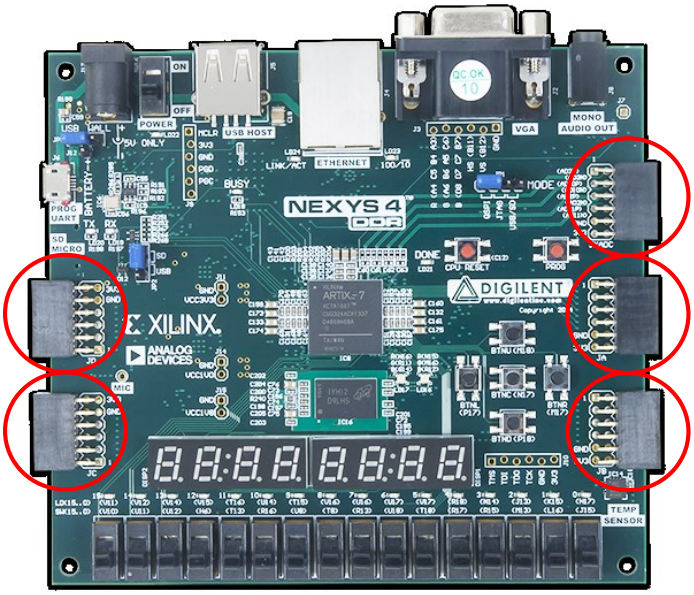

Figure 7-6

Each PMOD port has 12 pins including 2 VCC pins, 2 GND pins and 8 I/O pins. Figure 7-7 shows the orientation of one **Pmod port**.

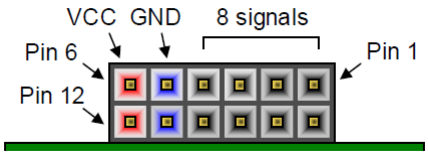

Figure 7-7

Below table 7-1 shows the pin diagram of all **Pmod ports** that are available in Nexys 4 DDR programming kit.

Table 7-1

Pmod JA	Pmod JB	Pmod JC	Pmod JD	Pmod XDAC
JA1: C17	JB1: D14	JC1: K1	JD1: H4	JXADC1: A13 (AD3P)
JA2: D18	JB2: F16	JC2: F6	JD2: H1	JXADC2: A15 (AD10P)
JA3: E18	JB3: G16	JC3: J2	JD3: G1	JXADC3: B16 (AD2P)
JA4: G17	JB4: H14	JC4: G6	JD4: G3	JXADC4: B18 (AD11P)
JA7: D17	JB7: E16	JC7: E7	JD7: H2	JXADC7: A14 (AD3N)
JA8: E17	JB8: F13	JC8: J3	JD8: G4	JXADC8: A16 (AD10N)
JA9: F18	JB9: G13	JC9: J4	JD9: G2	JXADC9: B17 (AD2N)
JA10: G18	JB10: H16	JC10: E6	JD10: F3	JXADC10: A18 (AD11N)

In this example JA1 and JA2 are chosen to assign variables of clk_out_original and clk_out_divided, respectively.

Step-6) Pins are chosen. Turn is to open an implementation constraints file and assign our variables to these pins. Open an implementation constraints file and name at as "simple_clock_divider_pins". By doing this an empty "simple_clock_divider_pins.ucf" file is created. Assign pins as seen in Figure 7-8.

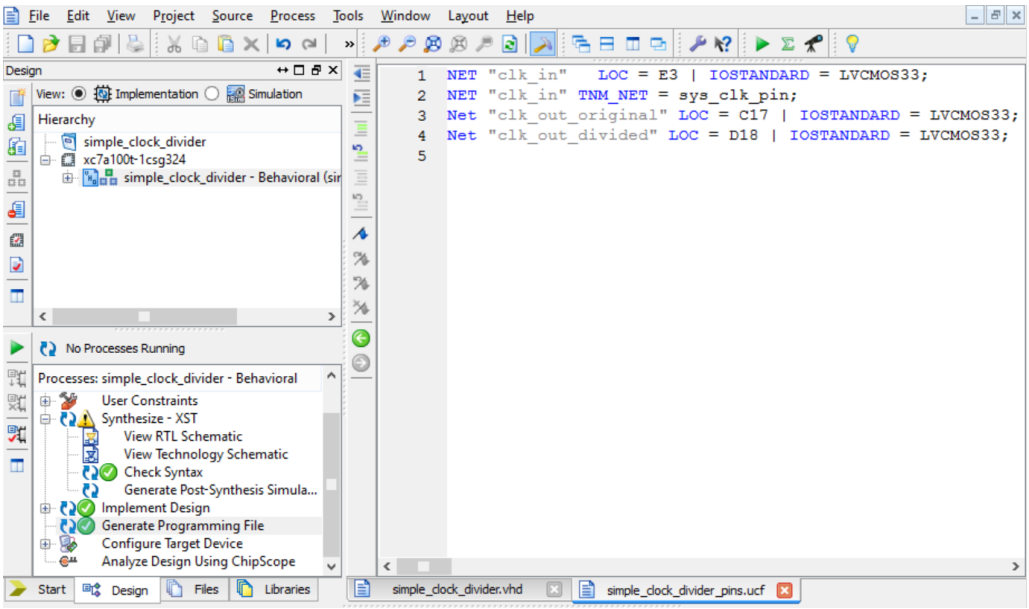

Figure 7-8

Step-7) After completing pin assignment generate programming file. We are ready to generate our "*.bit" file. If editor find no mistakes then green check mark emerges and it means our "*.bit" file is generated inside the project file.

Step-8) Open **iMPACT** and upload your bit file into FPGA. Cable conntections and results can be seen on Figure 7-9.

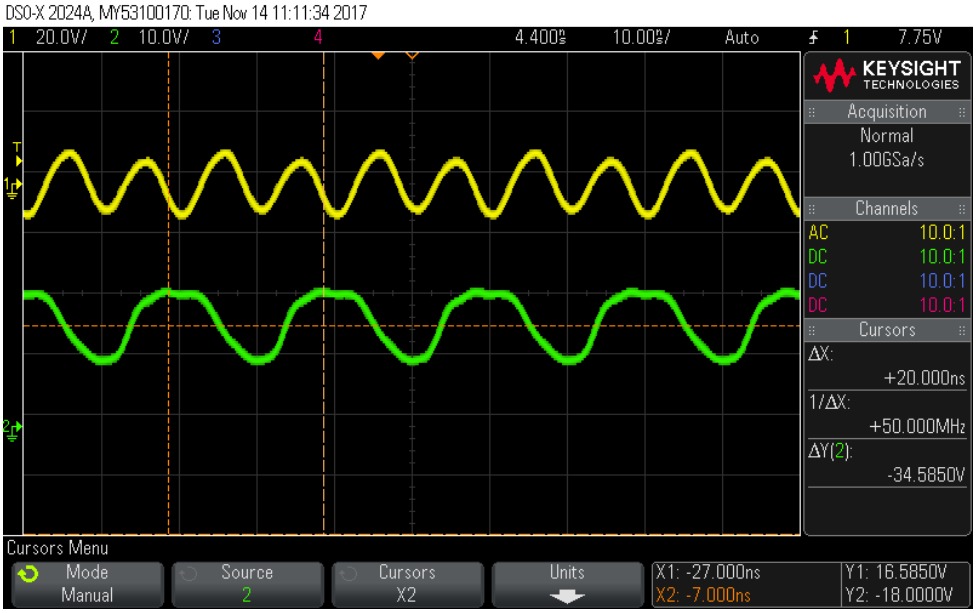

Figure 7-9

As you probably know, a square wave is made up of many sine waves, the fundamental frequency and an infinite number of odd harmonics decreasing in amplitude which give it it's shape. So if you have a square wave at 10MHz, you will have a harmonic at 30MHz at 1/3 the level of the fundamental, one at 50MHz at 1/5 the level of the fundamental and so on. The more harmonics present the more it will look like a square wave. Any square wave can be turned into a sine wave using a low pass filter to remove all harmonics apart from the fundamental, so unless your oscilloscope has the bandwidth to display at least the 3rd and 5th harmonic you will not see something that much resembles a square wave.

Exercise: Design a clock divider that divides original clock to 1000. Test your code in ISIM simulator. Use 100 MHz clock during simulation.

Example: Implement a 3 bit binary up counter. Counter value should be incremented by 1 at each second. Assign LEDs for the counter values. Upload your bit file to your Nexys 4 DDR programming kit.

Solution: Step-1) As the first step to achive our task, we need to open a new project. After setting project file, ISE Design Suite asks for the FPGA chip family, device, package and speed properties of the chosen FPGA. Set the FPGA chip of Nexys 4 DDR programming kit.

Step-2) After configuration, click next. An empty project with XC7A100T is created. Open a new VHDL module and name it as “three_bit_counter”. Copy below program segment PS 7.4.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity three_bit_up_counter is
  port( clk_in: in std_logic;
          counter : out std_logic_vector(2 downto 0));
end entity;

architecture logic_flow of three_bit_up_counter is
signal count: natural range 0 to 100000000;
signal divided_clk: std_logic;
begin

clock_divider_pr:process(clk_in)
  begin
    if (rising_edge(clk_in)) then
      count <= count + 1;
      if(count=100000000) then
         divided_clk <=not divided_clk;
         count<=0;
        end if;
     end if;
 end process;

counter_pr:process (divided_clk)
variable temp: integer range 0 to 7;
begin
          if (divided_clk'event and divided_clk ='1') then
              temp := temp+1;
          end if;
          counter <=conv_std_logic_vector(temp,3);
end process;
end architecture;
```

PS 7.4

Above VHDL code has two processes, one of them is for dividing clock and the other one stands for the counter operation. Counter process, *counter_pp*, takes the *divided_clk* as clock which is set to 1 Hz.

Step-3) In this step, we need to assign pins to our variables. Implementation constraints file which is named as "three_bit_counter_pins.ucf" and project view can be seen in below figure of Figure 7-10.

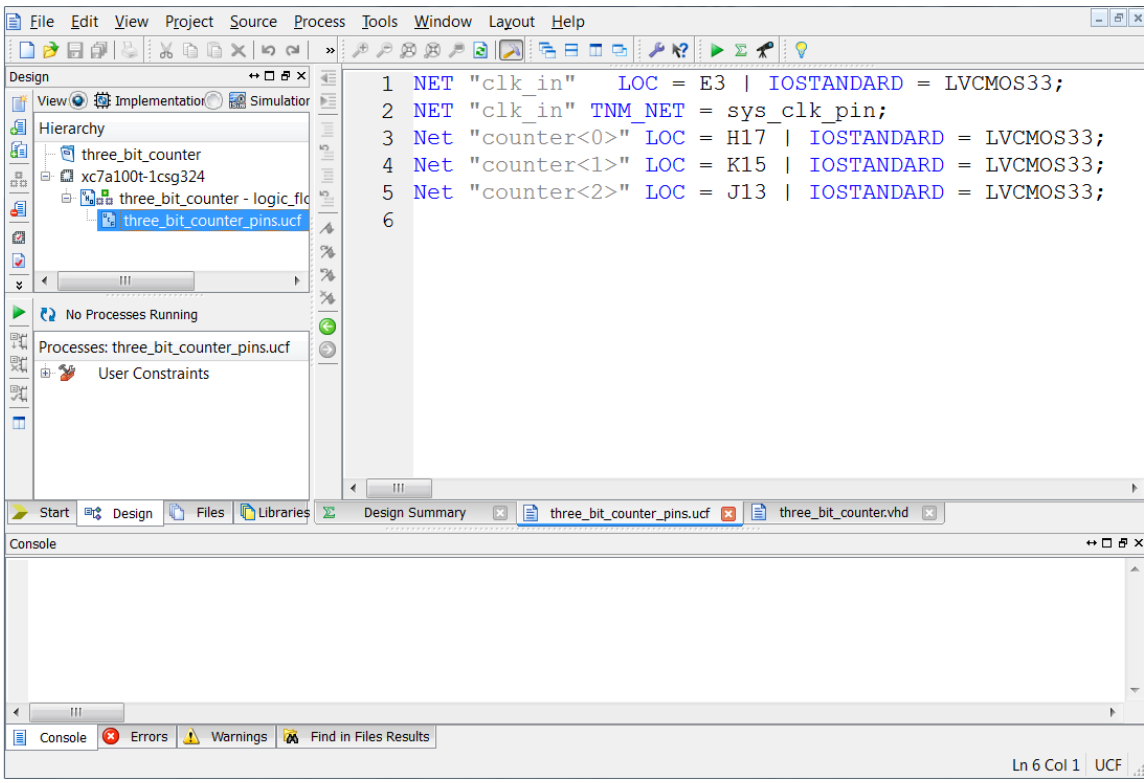

Figure 7-10

Step-4) Generate programming file and upload your bit file in your FPGA programming kit. 3-binary bit up counter starts from 000 to 111, then becomes 000 and counts to 111 again and continues like this in a infinite loop.

Exercise: Design a binary up counter that counts from 0 to 255. Counter frequency should be 5 Hz. Test your code in ISIM simulator. Program your Nexys 4 DDR programming kit and see the results.

Example: Implement a BCD counter from 0 to 9 for a seven segment display. Arrange counter frequency that a human eye can perceive. Assign seven segment pins for the counter values. Upload resulting bit file to Nexys 4 DDR programming kit.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity one_digit_ss_counter is
  port( clk_in: in std_logic;
                                AN : out std_logic;
        SSD: out std_logic_vector(6 downto 0));
end entity;

architecture logic_flow of one_digit_ss_counter is
signal count: natural range 0 to 100000000;
signal clk_1Hz: std_logic;
signal digit: natural range 0 to 9;
begin

clock_divider_pr:process(clk_in)
  begin
    if (rising_edge(clk_in)) then
      count <= count + 1;
      if(count=100000000) then
        clk_1Hz <=not clk_1Hz;
        count<=0;
       end if;
     end if;
 end process;

 AN <= '0';
process(clk_1Hz)
  begin
            if(rising_edge(clk_1Hz)) then
       if(digit=9) then
           digit<=0;
        else
         digit<= digit+1;
        end if;
            end if;
  end process;

     SSD<="1111110" when digit =0 else
              "0110000" when digit =1 else
              "1101101" when digit =2 else
              "1111001" when digit =3 else
              "0110011" when digit =4 else
              "1011011" when digit =5 else
              "1011111" when digit =6 else
              "1110000" when digit =7 else
              "1111111" when digit =8 else
              "1111011";

end architecture;
```

PS 7.5