Digital Systems VLSI

Nanos Georgios

The purpose of this exercise is to create a small "game" of tennis on the Zybo SoC FPGA, where two players return "hits" (by pressing their personal button) a "ball" (the respective bright LED). Players are only allowed to press the bounce button if the LED lights up in their own "court" (the most left and right ends of the four LEDs in a row), at which point the "ball" changes direction and now approaches the opponent. player. Each movement of the "ball" (turning on the next LED and turning off the current one) lasts 1 second, unless it is repelled at the right time (within the margin of one second).

A player can only lose if he does not manage to repel when the "ball" is on the court, and if this happens, the two LEDs closest to the winner "celebrate".

The following is the relevant code of the exercise and then its explanation (note that the given Frequency Divider was used as it is, with Max Count = [125000000/2 -1 = 62499999]):

```
-- library declaration
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

-- entity
entity game is port(
    p1: in std_logic;
    p2: in std_logic;
    p: in std_logic;
    r: in std_logic;
    s: in std_logic;
    clk125Mhz: in std_logic;
    leds: out std_logic_vector(3 downto 0) := (others => '0')
);
end game;
```

```
-- architecture
architecture Behavioural of game is

-- stc(s clear), sts(s posedge), stm(moving leds), stg(led on goal), ste(end)
    type state_type is (stc, sts, stm1, stm2, stg, ste);
    signal game_clk: std_logic := '0';
    signal ps: state_type := sts;
    signal ns: state_type;
-- ps: previous state, ns: next state

    signal set: std_logic_vector (2 downto 0);
    signal round: unsigned (1 downto 0);
    signal dir: std_logic := '0';
    signal leds_buf: std_logic_vector(3 downto 0);

--    flags apokroushs twn 2 paiktwn
    signal buff_s1: std_logic := '0';
    signal buff_s2: std_logic := '0';
    signal flag_enabler: std_logic := '0';
    signal game_clk_rst: std_logic := '0';

    component freq_divider is
    port(  clk: in std_logic;      -- clock of input frequency (100 MHz)
           rst: in std_logic;      -- negative reset
           sec_mode: in std_logic_vector;
           clk_out: out std_logic  -- clock of output (desired) frequency (25 MHz)
         );
    end component;
begin

    frequency_divider: freq_divider
    port map(
        clk => clk125Mhz,
        clk_out => game_clk,
        rst => '0',
        sec_mode => set
 );
    leds <= leds_buf;

    apokroush1 : process(p1, flag_enabler)
    begin
        if p1 = '1' and ((ps = sts) or (ps = stg and leds_buf = "1000")) then
         buff_s1 <= '1';
        else
         buff_s1 <= '0';
        end if;
    end process;

   apokroush2 : process(p2, flag_enabler)
     begin
         if p2 = '1' and  ps = stg and leds_buf = "0001" then
            buff_s2 <= '1';
         else
            buff_s2 <= '0';
```

```
    end if;
end process;

sync_proc: process(s, r, p, buff_s1, buff_s2, game_clk)
begin
   -- take care of the asynchronous input
   if s = '0' then
      dir <= '0';
      ps <= stc;
   elsif (r = '0' or ps = stc) then
     ps <= sts;
     round <= "00";
     flag_enabler <= '1';
   elsif p = '1' then
      ps <= ps;
   elsif (buff_s1 = '1' and flag_enabler = '1') then
      ps <= stm1;
      round <= round+1;
      dir <= '0';
   elsif (buff_s2 = '1' and flag_enabler = '1') then
      ps <= stm2;
      round <= round+1;
      dir <= '1';
   elsif rising_edge(game_clk) then
      ps <= ns;
      flag_enabler <= '1';
   end if;
end process sync_proc;

comb_proc: process(ps)
begin
   case ps is
      when stc =>
         leds_buf <= "0000";
         ns <= stc;
      when sts =>
         leds_buf <= "1000";
         set <= "001";
         ns <= sts;
      when stm1 =>
         leds_buf <= "0100";
         case dir is
            when '0' =>
               ns <= stm2;
            when '1' =>
               ns <= stg;
            when others =>
               leds_buf <= "1011";
         end case;
      when stm2 =>
            leds_buf <= "0010";
            case dir is
               when '0' =>
                  ns <= stg;
```

```
                when '1' =>
                   ns <= stm1;
                when others =>
                   leds_buf <= "1101";
             end case;
          when stg =>
             case dir is
                when '0' =>
                   leds_buf <= "0001";
                when '1' =>
                   leds_buf <= "1000";
                when others =>
                   leds_buf <= "1001";
                end case;
             ns <= ste;

             if (round = "11") and not(set = "110") then
                set <= set+1;
                round <= "01";
             end if;
          when ste =>
             case dir is
                when '0' =>
                   leds_buf <= "1100";
                when '1' =>
                   leds_buf <= "0011";
                when others =>
                   leds_buf <= "1111";
             end case;
             ns <= ste;
          when others =>
             leds_buf <= "0110";
--              ns <= stc;
       end case;

   end process comb_proc;

end Behavioural;
```

Frequency Divider

```
-- Module Name: frequency divider
-- Functionality: produces a clock frequency by reducing the input clock frequency
-- Description:
--   - in_freq     -> input frequency
--   - out_freq    -> output (desired) frequency
--   - scal_factor -> = in_freq/out_freq
--   - count       -> signal that counts until scal_factor/2-1 to generate delay
```

```
--   - new_clk     -> signal that toggles itself when count == scal_factor/2-1
-- Example:
--   - in_freq = 100 MHz
--   - out_freq = 25 MHz
--   - scal_factor = 4
--   - count -> counts until 1
--   - new_clk -> toggles itself when count == 1

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
 entity freq_divider is
port(  clk: in std_logic;      -- clock of input frequency (100 MHz)
     rst: in std_logic;      -- negative reset
      sec_mode: in std_logic_vector;
      clk_out: out std_logic  -- clock of output (desired) frequency (25 MHz)
   );
end freq_divider;
 architecture behavioural of freq_divider is
  signal count : integer range 0 to 62499999 := 0;
 signal new_clk : std_logic := '0';
--  signal reset_enable : std_logic := '1';
 signal max_count : integer range 0 to 62499999 := 62499999;
begin
  process(clk, rst)
 begin
   case sec_mode is
   when "001" =>
      max_count <= 62499999;
   when "010" =>
      max_count <= 499999999;
   when "011" =>
         max_count <= 37499999;
   when "100" =>
         max_count <= 24999999;
   when "101" =>
         max_count <= 124999999;
   when others =>
         max_count <= 124999999;
   end case;
```

```
   if (rst = '1' ) then
--      reset_enable <= '0';
     count <= 0;
     new_clk <= '0';
   elsif clk'event and clk = '1' then
     if count = max_count  then
       new_clk <= not new_clk;  -- toggle
       count <= 0;              -- count reset
--        reset_enable <= '1';
     else
       count <= count + 1;
     end if;
   end if;
 end process;
  clk_out <= new_clk;

end behavioural;

-- count      0    0     1     0     1
--           __    __    __    __
-- clk         |__|  |__|  |__|  |__|
--           ____________
-- clk_out             |____________|
```

Implementation: The focus of the code is the use of 6 different states (states), which indicate which phase of the game we are in at any given time. Specifically, these situations are the following:

- StC (state: Clear): The S switch is off, so the game is not running. All LEDs are off and no button or other switch causes any change.
- StS (state: Set): Switch S is on. The game has not started, as the "ball" is at the end of P1 (lit LD3) and is waiting for the opening blow.
- StM1 and StM2 (state: Moving Leds): Switch S is on. The game is in a state of transferring the "ball" from one end to the other, and at the moment an intermediate LED is lit (LD2 and LD3 respectively). Repulsions are not allowed.

- StG (state: Clear Goalpost): Switch S is on. The ball is in the court and the respective player (and he alone) must repel, otherwise he loses.
- StE (state: Clear End): Switch S is on. One player lost and there are two LEDs on to indicate the winner.

Main process is sync_proc: process (s, r, p, buff_s1, buff_s2, game_clk). In it all the checks are made for the status of the game and the transitions to the next state and it is sensitive to any change of the important switches and the clock. The course of the controls is as follows: θη:

1) Check if switch S is off. If yes, terminate each process and change to Clear mode.
2) Check if the R switch is active or if we are just going from Clear to New Game. A reset is required in the original state of the Set game.

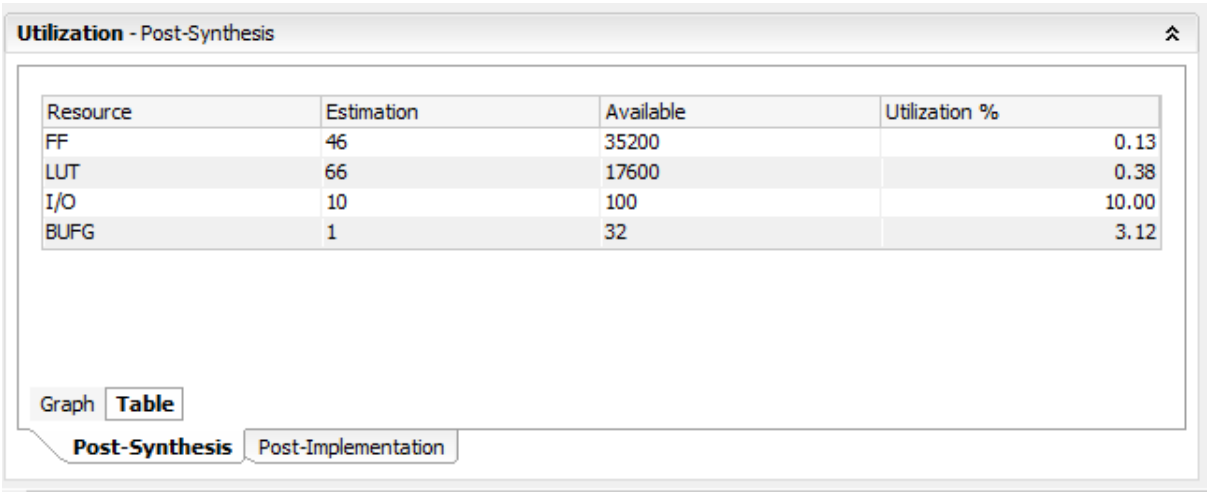
3) Check if switch P is active. Then a pause must be imposed, in which case the current status must be renewed.

4) Check for a valid repulse. Then the number of rounds increases and the direction of transfer of the "ball" changes.
5) Finally, if the clock's positive edge just came, the application goes to the next specified state.

An important process is comb_proc: process (ps). Each time the current state changes, this process handles the output of the LEDs and determines the standard next state. It also calculates the number of sets completed.

Finally, the processes apokroush1: process (p1, flag_enabler), apokroush2: process (p2, flag_enabler) are sensitive to the push of buttons and actually check if these presses are valid. In the event of a bounce, the buff_s1, buff_s2 bounce flags are updated, at which point the sync_proc is checked.

The resources provided in the post-synthesis stage that were used for this application are the following:



while the resources that were finally used in the post-implementation:

Utilization - Post-Implementation

Resource	Utilization	Available	Utilization %
FF	42	35200	0.12
LUT	62	17600	0.35
I/O	10	100	10.00
BUFG	1	32	3.12

Graph **Table**

Post-Synthesis **Post-Implementation**