Lab 5

A VHDL Reaction Timer
Lab 5 – A VHDL Reaction Timer

This lab will combine many advanced VHDL techniques including timers, seven segment display drivers, packages and functions, and state machines to create a simple game that to test a person’s reaction time.

Task 1: Create a New Project

1. **Start the New Project Wizard**
   Create a new project using the same procedure from the previous labs. Set the **Working Directory**, **Project Name**, and **Top-Level Design Entity** as seen in the table below:

<table>
<thead>
<tr>
<th>Working Directory</th>
<th>H:\Altera_Training\Lab5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Name</td>
<td>Lab5</td>
</tr>
<tr>
<td>Top-Level Design Entity</td>
<td>Lab5</td>
</tr>
</tbody>
</table>

   *Note: After clicking **Next**, a window may pop up stating that the chosen working directory does not exist. Click **Yes** to create it.*

   Continue With the New Project Wizard using the same settings as before.

2. **Create a New File**
   Create a new VHDL file, as discussed previously. Copy and paste the following code into your new vhdl file and save it as **Lab5.vhd**:

   ```vhdl
   Library IEEE;
   use IEEE.std_logic_1164.all;
   USE IEEE.std_logic_unsigned.ALL;
   USE work.bin_to_7seg.ALL;

   ENTITY Lab5 IS
   PORT(
       -- 50Mhz clock signal from the DE2 board
       clk: IN std_logic;

       -- KEY3 and KEY0 on the DE2 board
       push_button, reset: IN std_logic;

       -- Outputs for the LEDS
       LEDR, LEDG: OUT std_logic_vector(7 DOWNTO 0);

       -- 7 segment display outputs
       digit1, digit2, digit3, digit4: OUT std_logic_vector(6 DOWNTO 0));
   END;

   ARCHITECTURE Lab5_beh of Lab5 IS
   -- Type definition for the four states used for the state machine
   Type Timer_State IS (idle, countdown, timing, score);
   -- Signals of type Timer_State used to control the state machine
   SIGNAL current_state, next_state : Timer_State := idle;
   -- 16-bit binary value of the elapsed time in milliseconds
   SIGNAL elapsed_time: std_logic_vector(15 DOWNTO 0):= (OTHERS => '0');
   -- Flag that indicates the status of the countdown
   SIGNAL countdown_done: std_logic := '0';
   ```
-- Flag that indicates a good input signal and the desire to change states
SIGNAL state_change: std_logic := '0';

-- Sample pulse generated by the SampleGen process used by the Debounce
-- process to either increment the counter or signal a state change
SIGNAL sample: std_logic;

-- Maximum number of good input pulses needed to qualify as clean input
CONSTANT PULSE_COUNT_MAX: integer := 20;

-- Counter used to record the number of good input pulses
SIGNAL counter: integer range 0 to PULSE_COUNT_MAX;

-- The KEY0-3 push buttons on the DE2 board are active low, so sync is the
-- inverse of the KEY3 push button's value
SIGNAL sync: std_logic;

BEGIN

------------------- SampleGen Process -------------------
-- A counter that sends a signal to sample the input button
-- when the maximum count is reached.
SampleGen : process(clk)
begin
  -- Sample_counter is a counter used to control the sample frequency
  -- sample frequency = clock frequency / (sample_counter(max)+1)
  variable sample_counter: integer range 0 to 24999;

  if rising_edge(clk) then
    if (reset='0') then
      sample <= '0';
      sample_counter := 0;
    else
      if (sample_counter = 24999) then
        sample_counter := 0;
        sample <= '1';
      else
        sample <= '0';
        sample_counter := sample_counter + 1;
      end if;
    end if;
  end if;
end process;

------------------- Debounce Process -------------------
-- A counter that is incrementally every sample pulse while the
-- input button is pressed, when the switch is not pressed the
-- counter is reset to zero. If the counter is at its maximum,
-- the debouncer output is high, otherwise its low.
Debounce : process(clk)
begin
  if rising_edge(clk) then
    if (reset='0') then
      sync <= '0';
      counter <= 0;
      state_change <= '0';
    else
      sync <= NOT push_button;
      if (sync='0') then
        -- Button not pressed
        counter <= 0;
        state_change <= '0';
      elseif(sample = '1') then
        -- Button pressed
        if (counter=PULSE_COUNT_MAX) then
          state_change <= '1';
        else
          counter <= counter + 1;
        end if;
      end if;
    end if;
  end if;
end process;
end if;
end process;

-- Sets the next_state for the state machine. The next desired state is stored on the next_state signal, which is used in the State_Transitions process to change the current state.

Timer_State_Machine: PROCESS (clk)
BEGIN
next_state <= current_state;
CASE current_state IS

-- If the state_change flag is set and the push_button is pushed then the next desired state is the countdown state
WHEN idle =>
  IF state_change = '1' AND push_button = '1' THEN
    next_state <= countdown;
  END IF;

-- when the countdown_done flag is high the next desired state is the timing state
WHEN countdown =>
  IF countdown_done = '1' THEN
    next_state <= timing;
  END IF;

-- If the state_change flag is high and the push_button is pushed then the next desired state is the score state.
WHEN timing =>
  IF state_change = '1' AND push_button = '1' THEN
    next_state <= score;
  END IF;

-- Stay in score state. Pressing reset will override this
WHEN score =>
  next_state <= score;

-- In case of entering any undefined states, next desired state is idle
WHEN OTHERS =>
  next_state <= idle;
END CASE;
END PROCESS Timer_State_Machine;

-- Loads the next_state onto the current_state, or resets the state to idle if the reset button is pressed.

State_Transitions: PROCESS (reset, clk)
BEGIN
IF reset = '0' THEN
  current_state <= idle;
ELSIF rising_edge (clk) THEN
  current_state <= next_state;
END IF;
END PROCESS state_transitions;

-- This process is what controls the LED the outputs as seen by the user.

Output_Process: PROCESS (current_state, clk)

-- Two 16-bit countdown counters to create a delay for the countdown state
variable countdown_count1, countdown_count2: integer range 0 to 65535;

-- A delay counter used to delay the incrementing of timing_count so that timing_count is in milliseconds
variable timing_counter: integer range 0 to 49999;

-- 16-bit binary value of the time count in milliseconds
variable timing_count: std_logic_vector(15 DOWNTO 0);
BEGIN
   IF rising_edge(clk) THEN
      CASE current_state IS
      -- Reset everything in idle mode
      WHEN idle =>
         LEDG <= (OTHERS => '0');
         LEDR <= (OTHERS => '0');
         elapsed_time <= (OTHERS => '0');
         timing_count := (OTHERS => '0');
         timing_counter := 0;
         countdown_count1 := 0;
         countdown_count2 := 0;

      -- Turn red LEDs on and begin countdown counter, when countdown counter
      -- is done, countdown_done is high.
      WHEN countdown =>
         LEDG <= (OTHERS => '0');
         LEDR <= (OTHERS => '1');
         countdown_count1 := countdown_count1 + 1;
         IF countdown_count1 = 65535 THEN
            countdown_count2 := countdown_count2 + 1;
            countdown_count1 := 0;
         END IF;
         IF (countdown_count2 = 763) THEN
            countdown_done <= '1';
            countdown_count2 := 0;
         END IF;

      -- Turn green LEDs on and begin counting in milliseconds. Set countdown_done
      -- to low.
      WHEN timing =>
         LEDG <= (OTHERS => '1');
         LEDR <= (OTHERS => '0');
         IF (timing_counter = 49999) THEN
            timing_count := timing_count + 1;
            timing_counter := 0;
         ELSE
            timing_counter := timing_counter + 1;
         END IF;
         elapsed_time <= timing_count;
         countdown_done <= '0';

      -- Turn all LEDs on, set timing counters to 0
      WHEN score =>
         LEDG <= (OTHERS => '1');
         LEDR <= (OTHERS => '1');
         timing_count := (OTHERS => '0');
         timing_counter := 0;

      -- Turn all LEDs off
      WHEN OTHERS =>
         LEDG <= (OTHERS => '0');
         LEDR <= (OTHERS => '0');
      END CASE;
   END IF;
END PROCESS Output_Process;

------------------
Time_Display Process ------------------
-- This process displays the time value on four of the seven
-- segment displays on the DE2 board. The functions bin_to_bcd
-- and bcd_to_7seg are called from the included bin_to_7seg.vhd
Time_Display_Process: PROCESS (elapsed_time)
BEGIN
   -- 20-bit binary variable that holds 5 4-bit BCD values, only 4 are used to
   -- display on the 4 7-segment displays
   variable bcd_time: std_logic_vector(19 DOWNTO 0);
First then elapsed time must be converted from binary to BCD using the `bin_to_bcd()` function and stored on to the `bcd_time` variable. Next, the BCD values are fed out to the four 7-segment displays by using the `bcd_to_7seg()` function. The first four bits of `bcd_time` correspond to `digit1`, the next four to `digit2`, and so on.

```vhdl
bcd_time := bin_to_bcd(elapsed_time);
digit1 <= bcd_to_7seg(bcd_time(3 DOWNTO 0));
digit2 <= bcd_to_7seg(bcd_time(7 DOWNTO 4));
digit3 <= bcd_to_7seg(bcd_time(11 DOWNTO 8));
digit4 <= bcd_to_7seg(bcd_time(15 DOWNTO 12));
END PROCESS;
END Lab5_beh;
```

3. Create another New File

Create another new VHDL file. Copy and paste the following code into your new vhdl file and save it in the project directory as `bin_to_7seg.vhd`, make sure the **Add file to current project** box is checked in the **Save As** window:

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
USE IEEE.std_logic_unsigned.ALL;
use ieee.numeric_std.all;

PACKAGE bin_to_7seg IS
  -- Bin_to_BCD takes in a 16 bit binary number and returns a 20-bit BCD
  -- representation of it. This would be the equivalent of a 5-digit integer
  -- with 5 groups of 4-bits, each representing 1 digit.
  FUNCTION Bin_to_BCD (bin :std_logic_vector(15 DOWNTO 0)) return std_logic_vector;

  -- BCD_to_7seg takes a 4-bit BCD value and returns a 7-bit value that
  -- represents the desired segments to turn on for a 7-segment display
  FUNCTION BCD_to_7seg (bcd :std_logic_vector(3 DOWNTO 0)) return std_logic_vector;
END;

PACKAGE BODY bin_to_7seg IS

  ------------------ Binary to BCD Conversion Function ------------------
  FUNCTION Bin_to_BCD (bin :std_logic_vector(15 DOWNTO 0)) return std_logic_vector
  variable i : integer := 0;
  variable bcd : std_logic_vector(19 downto 0) := (others =>'0');
  variable bint : std_logic_vector(15 DOWNTO 0) := bin;
  BEGIN
    for i in 0 to 15 loop
      bcd(19 downto 1) := bcd(18 downto 0);
      bcd(0) := bint(15);
      bint(15 downto 1) := bint(14 downto 0);
      bint(0) :='0';
      if(i < 15 and bcd(3 downto 0) > "0100") then
        bcd(3 downto 0) := bcd(3 downto 0) + "0011";
      end if;
      if(i < 15 and bcd(7 downto 4) > "0100") then
        bcd(7 downto 4) := bcd(7 downto 4) + "0011";
      end if;
    END FOR;
  END;
END;
```

Lab 5 – A VHDL Reaction Timer – Reaction Timer Code
if(i < 15 and bcd(11 downto 8) > "0100") then
    bcd(11 downto 8) := bcd(11 downto 8) + "0011";
end if;

if(i < 15 and bcd(15 downto 12) > "0100") then
    bcd(15 downto 12) := bcd(15 downto 12) + "0011";
end if;

if(i < 15 and bcd(19 downto 16) > "0100") then
    bcd(19 downto 16) := bcd(19 downto 16) + "0011";
end if;
END LOOP;

RETURN BCD;
END FUNCTION;

------------------
BCD to 7 Seg Display Conversion Function

FUNCTION BCD_to_7Seg (bcd :std_logic_vector(3 DOWNTO 0)) return std_logic_vector IS
    variable seg7 : std_logic_vector(6 downto 0) := (others => '1');
    BEGIN
    CASE bcd IS
        WHEN "0000"=>seg7:="1000000"; -- 0 --
        WHEN "0001"=>seg7:="1111001"; -- 1 --
        WHEN "0010"=>seg7:="0100100"; -- 2 --
        WHEN "0011"=>seg7:="0110000"; -- 3 --
        WHEN "0100"=>seg7:="0011001"; -- 4 --
        WHEN "0101"=>seg7:="0010010"; -- 5 --
        WHEN "0110"=>seg7:="0000010"; -- 6 --
        WHEN "0111"=>seg7:="1111000"; -- 7 --
        WHEN "1000"=>seg7:="0000000"; -- 8 --
        WHEN "1001"=>seg7:="0011000"; -- 9 --
        WHEN OTHERS=>seg7:="1111111"; -- X --
    END CASE;
    RETURN seg7;
END FUNCTION;

END PACKAGE BODY;

4. About the Code
The code in Lab5.vhd is broken up into a variety of processes. SampleGen and Debounce are processes used together to debounce the input signal from the momentary push button (KEY3) on the development board. The hardware samples the push button at a rate of 2 kHz, and after 20 consecutive low readings (in this case the push button switches are active low) the hardware considers the button to have been pressed.

The program uses a finite state machine to represent the various states used. These states are idle, countdown, timing, and score. Idle is when the hardware is waiting for the user to initiate the game. Countdown is a timed countdown sequence which adds a delay before the millisecond timer starts counting. Timing is when the millisecond timer starts counting how long it takes the user to push the button to stop the counting. Score freezes the display to show the user's reaction time.

The process Timer_State_Machine is what defines the requirements to change from one state to the next. This process stores the next desired state onto the next_state signal, which is then set as the current_state by the State_Transitions process.

The Output_Process contains everything that happens within each of the individual states. In the Idle state within the the green and red LEDs are turned off, and all timers and counters are reset to zero. In the Countdown state the red LEDs are turned on and the countdown timer starts. When the
countdown timer is done, \texttt{countdown\_done} is set high. Once \texttt{countdown\_done} is high, the \texttt{current\_state} becomes the \texttt{Timing} state. Here the millisecond timer starts counting until the user presses the button to stop it. Finally in the \texttt{Score} state, the user’s time is displayed in milliseconds.

Pressing the reset button (\texttt{KEY1} on the DE2 development board) will return you to the \texttt{idle} state at any point.

The \texttt{Time\_Display\_Process} takes in the elapsed time in binary format, converts that from binary to BCD, and then to a value suitable to use to display the desired decimal digit on one of four seven segment displays.

The \texttt{bin\_to\_7seg.vhd} file contains two functions used by the \texttt{Lab5.vhd} code within the \texttt{Time\_Display\_Process}.

The first function is \texttt{Bin\_to\_BCD}, which uses a double dabble algorithm to convert a 16-bit binary value to five 4-bit BCD values, although only four of these BCD values are used by the \texttt{Lab5.vhd} code. The other function, \texttt{BCD\_to\_7seg}, converts the BCD values to a 7-bit value that represents each digit for the seven segment displays.

As part of this lab exercise, in the \texttt{Lab5.vhd} file, you will need to finish the case statement in the \texttt{Timer\_State\_Machine} process, as well as the function calls in the \texttt{Time\_Display\_Process} used to display the time. Some hints are provided in the comments in the code.

\textbf{Task 2: Pin Assignments}

1. \textbf{Make the Following Pin Assignments}

<table>
<thead>
<tr>
<th></th>
<th>To</th>
<th>Location</th>
<th>DE2 Board Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>push_button</td>
<td>PIN_W26</td>
<td>KEY3</td>
</tr>
<tr>
<td>2</td>
<td>reset</td>
<td>PIN_N23</td>
<td>KEY1</td>
</tr>
<tr>
<td>3</td>
<td>digit1[0]</td>
<td>PIN_AF10</td>
<td>Seven Segment Digit 0[0]</td>
</tr>
<tr>
<td>4</td>
<td>digit1[1]</td>
<td>PIN_AB12</td>
<td>Seven Segment Digit 0[1]</td>
</tr>
<tr>
<td>5</td>
<td>digit1[2]</td>
<td>PIN_AC12</td>
<td>Seven Segment Digit 0[2]</td>
</tr>
<tr>
<td>6</td>
<td>digit1[3]</td>
<td>PIN_AD11</td>
<td>Seven Segment Digit 0[3]</td>
</tr>
<tr>
<td>8</td>
<td>digit1[5]</td>
<td>PIN_V14</td>
<td>Seven Segment Digit 0[5]</td>
</tr>
<tr>
<td>9</td>
<td>digit1[6]</td>
<td>PIN_V13</td>
<td>Seven Segment Digit 0[6]</td>
</tr>
<tr>
<td>10</td>
<td>digit2[0]</td>
<td>PIN_V20</td>
<td>Seven Segment Digit 1[0]</td>
</tr>
<tr>
<td>11</td>
<td>digit2[1]</td>
<td>PIN_V21</td>
<td>Seven Segment Digit 1[1]</td>
</tr>
<tr>
<td>17</td>
<td>digit3[0]</td>
<td>PIN_AB23</td>
<td>Seven Segment Digit 2[0]</td>
</tr>
<tr>
<td>18</td>
<td>digit3[1]</td>
<td>PIN_V22</td>
<td>Seven Segment Digit 2[1]</td>
</tr>
<tr>
<td>20</td>
<td>digit3[3]</td>
<td>PIN_AC26</td>
<td>Seven Segment Digit 2[3]</td>
</tr>
</tbody>
</table>
24  digit4[0]  PIN_Y23  Seven Segment Digit 3[0]
31  clk  PIN_N2  50 MHz on board clock
32  LEDG[0]  PIN_AE22  LED Green [0]
33  LEDG[1]  PIN_AF22  LED Green [1]
40  LEDR[0]  PIN_AE23  LED Red [0]

2. Click **Save** to save the pin assignments.

3. **Compile the Design** by clicking on the **Start Compilation** button.

---

**Task 3: Program the DE2 Development Board**

1. Plug in and power on the DE2 board. Make sure that the **RUN/PROG Switch for JTAG/AS Modes** is in the **RUN** position.

2. In the Quartus II window click the **Programmer** button on the Toolbar to open the Programmer window.

   The **Hardware Setup...** must be **USB-Blaster [USB-0]**. If not, click the **Hardware Setup...** button and select **USB-Blaster [USB-0]** from the drop-down menu for **Currently selected hardware**. **Mode** should be set to **JTAG**.

   Make sure that the **File** is **Lab5.sof**, **Device** is **EP2C35F672**, and the **Program/Configure** box is checked.
Then click the **Start** button to program the DE2 board.  
When the progress bar reaches 100%, programming is complete.

**Task 4: Play the Game**

1. The game is started by pressing **KEY3** on the development board. This causes the hardware to change states from **idle** to **countdown**, turns on the first eight red LEDs, and begins the countdown to when the timer starts. The countdown is not displayed in order to maintain the element of surprise.

2. When the countdown completes the hardware changes to the **timer** state and begins incrementing the millisecond counter, the red LEDs are shut off, and the first eight green LEDs are turned on. The user must press **KEY3** on the development board as soon as possible after the green LEDs light in order to obtain the quickest reaction time.

3. When the user presses **KEY3** the state is changed to the **score** state where the millisecond display is frozen in order to display the user’s reaction time.

4. Pressing the **reset** button (**KEY1**) at this point will return the hardware to the **idle** state where the game can be played again.