7076CEM Coursework Submission

Jerome Samuels-Clarke - 10473050

Module Leader – **Dr Amjad Saeed Khan**

April 2022

Contents

1.1 Introduction

This report is about designing an FPGA-based reaction timer using the Nexys-4 DDR board. By using digital systems concepts, paired with the modular nature of FPGA/VHDL, the overall system can be broken down into several modules, that exchange information with one another. The System Design chapter goes over the system specification, as well as shows the flow charts and diagrams designed to complete each module. The Implementation chapter covers the VHDL design of each module including additional information such as the overall static and dynamic power consumption of the system. The Simulation chapter shows how the use of testbenches is used to simulate the hardware design to check for any errors in the modules. The Discussion and Summary chapter concludes the report reviewing the overall design. Lastly, the full code for each module and testbench is included in the Appendix.

1.2 System Design

The objective of the system is to test the reaction time of the user. The reaction time for most people is within 150ms to 300ms. The specification the given to design this is summarised as follows:

- Push a clear button the restart the system, display “HI” on the seven-segment display and turn the LED off.
- Push the start button to start the system. Turn off the seven-segment display.
- After a random interval (2 to 15 seconds), turn the LED on. Increment a millisecond counter on the seven-segment display.
- If the user pushes the stop button, stop the timer.
- If the user does not push the stop button, display “1000” on the seven-segment display.
- If the user pushes the stop button before the LED turns on, display “9999”.

This can be converted into a flowchart (Figure 1), to visually represent each step.

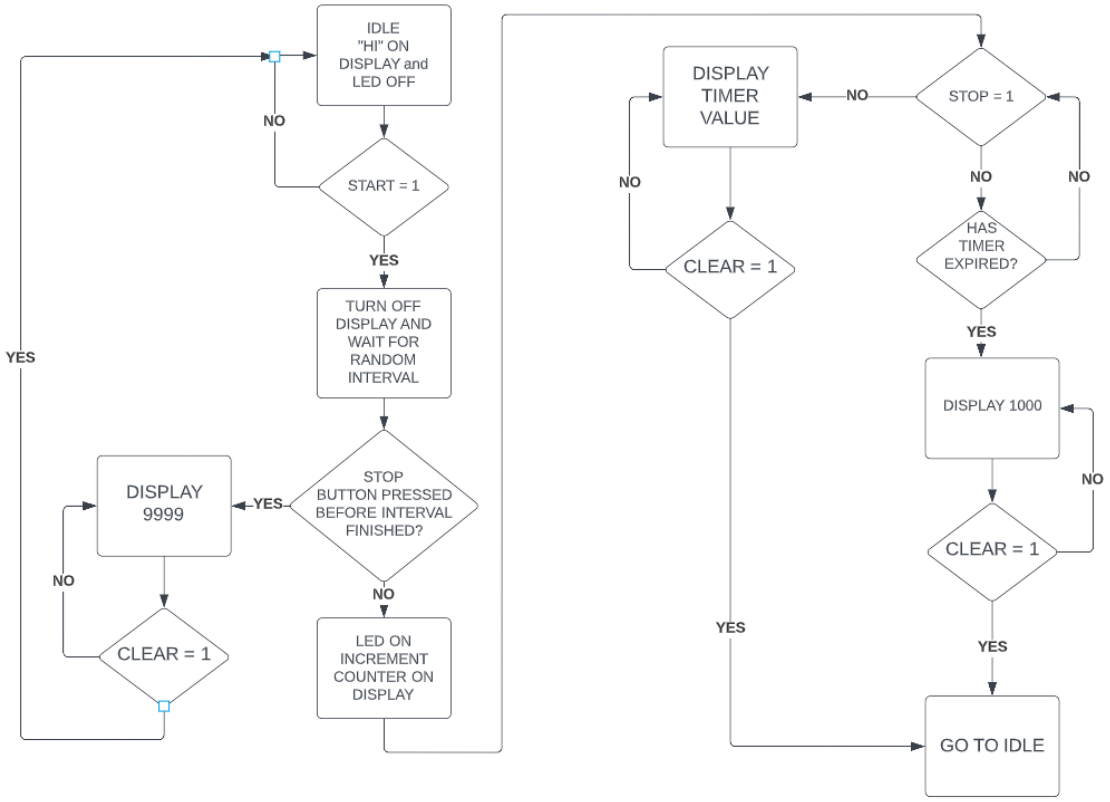


Figure 1: Flowchart of specification

When designing FPGA-based systems, using black-box approach can help identify the inputs and outputs of the design. More importantly, this can then be translated to the entity of the module. This can be created using the specification and flowchart given above. The black-box in Figure 2 would be for the top-level file used when designing the system using structural modelling. There would be three user buttons (clear, start, stop), an LED, a seven-segment display (made up of the segments and anode pins), and lastly a clock signal to synchronise everything.

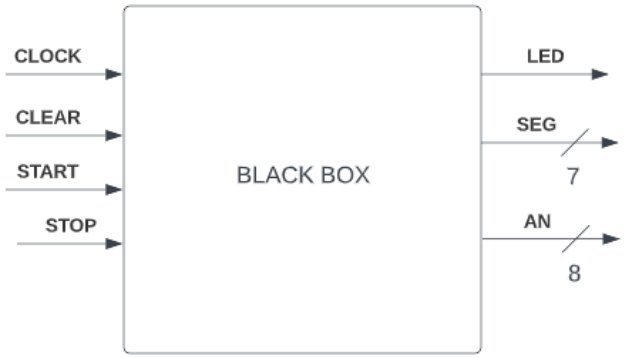


Figure 2: Black box of VHDL design

Going with the structural approach, there should be several modules that go inside the black-box that make up the overall system. Some of these modules can be grouped up into one file or separated if needed. The modules created are:

- Clock (frequency) divider – Reduces the frequency of the clock to generate a slower one.
- Button debounce – Simple button debounce code, which waits for the button to be stable.
- Random interval counter – Used to wait for 2 to 15 seconds, before starting the millisecond counter.
- Millisecond counter – Counts up to 1000 milliseconds, which is sent to the binary to BCD
- Binary to BCD – Converts the from the millisecond counter (in binary) into binary coded decimal format, to display on the seven-segment display. This uses the double dabble algorithm.
- Reaction – A finite state machine that controls the operation of the system, and also generates a random number.

1.2.1 Clock Divider Calculations

The easiest way to generate a clock is to implement a counter that when reaches its value, change the state of a std_logic signal (will either be zero or one, once the count value is reached). The implementation for this can be seen in the implementation chapter. To work out what value to count to, the formula for working this out is shown in Equation 1. In this system there are three four different clocks. The 100MHz clock, which is what the Nexys-4 DDR runs at, a 1ms (1000Hz) clock used to increment the millisecond counter, the refresh frequency for the seven-segment display (explained below), and lastly the 1 second counter, used for the random interval to start the program. The count values are shown in Table 1.

$$count = \frac{\left(\frac{frequency\ in}{frequency\ out}\right)}{2}$$

Equation 1: Equation for calculating the count value for dividing the clock frequency.

Table 1: Table of the count values using the clock divider equation

Module	Input Frequency	Output Frequency	Count Value
1ms Counter	100MHz	1000Hz	50000
Refresh Frequency (4ms)	100MHz	250Hz	200000
1 Second Counter	100MHz	1Hz	50000000

1.2.2 Refresh Frequency for Seven-segment Display

As the seven-segment display is connected via a common anode, to display a different digit on the seven-segment display, it will need to be multiplexed. This means that the anode pins need to be turned on/off fast enough that it appears as if all digits are on. Figure 3 shows an illustration of this, highlighting that the refresh frequency needs to be between 1ms-16ms. A 4ms refresh frequency was selected.

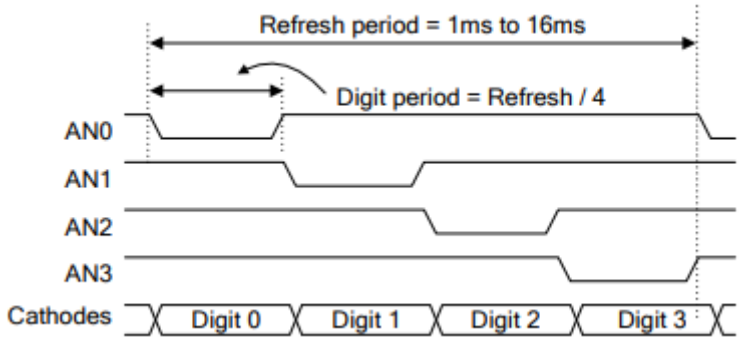


Figure 3: Multiplexing done between four digits. Xilinx. (2014, September 11). Nexys4-DDR Reference Manual [Image]. https://www.xilinx.com/support/documents/university/XUP%20Boards/XUPNexys4DDR/documentation/Nexys4-DDR_rm.pdf

1.2.3 Double Dabble Algorithm

To convert a binary number into a binary coded decimal (BCD), the double dabble algorithm can be used. This is also called the *shift and add three* algorithm, as that is what the algorithm does. After every shift, if the total number of bits in a digit is higher than four, then three needs to be added to the digit. An example of this is shown in Table 2, using a binary input of 1023 and shows the BCD number of each digit in the respective columns.

Table 2: Example of the double dabble algorithm with the binary input as 1023.

Thousands	Hundreds	Tens	Units	Binary Input	Operation
				11 1111 1111	START
			0001	1 1111 1111	SHIFT
			0011	1111 1111	SHIFT
			0111	111 1111	SHIFT
			1010		ADD 3
		0001	0101	11 1111	SHIFT
		0001	1000		ADD 3
		0011	0001	1 1111	SHIFT
		0110	0011	1111	SHIFT
		1001	0011		ADD 3
	0001	0010	0111	111	SHIFT
	0001	0010	1010		ADD 3
	0010	0101	0101	11	SHIFT
	0010	1000	1000		ADD 3
	0101	0001	0001	1	SHIFT
	1000	0001	0001		ADD 3
0001	0000	0010	0011		SHIFT
1	0	2	3		RESULT

1.2.4 Reaction State Machine

State machines are widely used in many systems. To control all the modules of the system a state machine is used in the reaction module, which takes inputs to change states and displays the output. The state machine designed has five states adapted from the original specification, being:

- Idle – The idle state waits for the start button to be pressed, whilst displaying “HI” on the seven-segment display and keeping the LED off.
- Start – The start state turns the seven-segment display off, reads the *random number* and passes it to the random interval module. This will wait in seconds based and returns a signal back to the start state when the interval is up. The millisecond counter will start and will turn the user LED on. If the stop button is pressed before returned signal, the state will change to invalid.
- React – The react state will turn the seven-segment display on, to display the incrementing counter. If the clear button is pressed, the state will go back to idle. If the stop button the state goes to the stop state.
- Stop – When the stop button is pressed the counter stops incrementing and displays the time on the seven-segment display. Pressing the clear button will change the state back to idle.
- Invalid – When in the invalid state the seven-segment display shows “9999”, and changes to idle state once the clear button is pressed.

1.3 Implementation

A top module is created to connect all the previous modules together and allows signals to be passed between each other, including signals in the entity that will connect to the outside world. This is done by instantiating each module, by declaring it's entity and mapping any generics or port maps. The generic values that were used in some of the modules come from this top module, which allows each module instantiated multiple times to operate differently. This can be seen with the clock divider modules in Figure 4, using the count values calculated in Table 1.

```
24  clk_1_ms : entity work.clk_div(rtl)
25      generic map (N => 50000)
26      port map (
27        clk => clk,
28        clk_out => clk_out
29      );
30
31  ssd_refresh : entity work.clk_div(rtl)
32      generic map (N => 200000)
33      port map (
34        clk => clk,
35        clk_out => refresh
36      );
37
38  clk_1_sec : entity work.clk_div(rtl)
39      generic map (N => 50000000)
40      port map (
41        clk => clk,
42        clk_out => rng_clk
43      );
```

Figure 4: Instantiating the clock dividers from the top level VHDL file.

1.3.1 Clock Divider

The Nexys-4 DDR runs at 100MHz, which will need to be reduced using a clock divider. There should be three different clock modules, being:

- 1ms (1000Hz) clock shown on the seven-segment display
- Seven-segment display refresh frequency to multiplex each digit
- One second clock for the random interval wait time

To promote code reusability the same clock divider code can be used for each of the modules as seen in Figure 4. Based on the value passed in from the generic (N in this case), the counter used in the clock divider will change the state of the new clock signal when the limit is reached, shown in Figure 5.

```
18  -- Generate a clock by incrementing a counter
19  process (clk)
20  begin
21      if (rising_edge(clk)) then
22            counter <= counter + 1;
23
24            -- If the value of N is reached, change the state of the temp signal
25            if (counter = N) then
26              temp <= NOT temp;
27              counter <= 0;
28            end if;
29      end if;
30
31      -- Assign the value of temp to clk_out
32      clk_out <= temp;
33  end process;
```

Figure 5: Clock divider code, showing the counter is compared to the value of N.

1.3.2 Display Counter

The display counter is the counter that gets shown on the seven-segment display. The input clock from this comes from the millisecond clock counter, and the output goes into the binary to BCD module. This code uses a two-state state machine (Figure 6). When an enable signal is received, the state changes and the counter starts to increment. When the count reached 1000, or the stop signal is received indicating that the stop button is pressed, the state goes back to idle. A concurrent signal assignment is used to assign the counter to the counter output signal.

```
18  process (clk)
19  begin
20      if (rising_edge(clk)) then
21              case (state) is
22                  -- Wait until the enable signal is '1' to change states
23                  when idle_s =>
24                      if (en = '1') then
25                          counter <= (others => '0');
26                          state <= count_s;
27                      else
28                          state <= idle_s;
29                      end if;
30
31                  -- Increment the counter until count = 1000 or a stop signal is recievied
32                  when count_s =>
33                      if (counter >= 1000) then
34                          state <= idle_s;
35                      elsif (stop_sig = '1') then
36                          state <= idle_s;
37                      else
38                          counter <= counter + 1;
39                      end if;
40              end case;
41      end if;
42  end process;
43
44  -- Assign the value of counter to count_out
45  count_out <= std_logic_vector(counter);
```

Figure 6: Display counter state machine.

1.3.3 Binary to BCD

The double dabble algorithm explain previously is implemented in this module. The entity contains the binary input, and the output is the BCD value split into units, tens, hundreds, and thousands. Using Table 2, it shows that ten shifts are needed. When implementing this, there will always be three initial shifts before the outputs are checked. This means that there should be loop of seven shifts as shown in Figure 7.

```
22      -- Do the first initial shift
23      x(12 downto 3) := unsigned(binary_in);
24      -- Loop seven times
25      for i in 0 to 6 loop
26          -- Check units
27          if x(13 downto 10) > 4 then
28              x(13 downto 10) := x(13 downto 10) + 3;
29          end if;
30          -- Check tens
31          if x(17 downto 14) > 4 then
32              x(17 downto 14) := x(17 downto 14) + 3;
33          end if;
34          -- Check hundreths
35          if x(21 downto 18) > 4 then
36              x(21 downto 18) := x(21 downto 18) + 3;
37          end if;
38           -- Check thousands
39          if x(25 downto 22) > 4 then
40              x(25 downto 22) := x(25 downto 22) + 3;
41          end if;
42
43          x(25 downto 1) := x(24 downto 0);
44      end loop;
45
46      unit <= std_logic_vector(x(13 downto 10));
47      tens <= std_logic_vector(x(17 downto 14));
48      hundreds <= std_logic_vector(x(21 downto 18));
49      thousands <= std_logic_vector(x(25 downto 22));
50
51  end process;
```

Figure 7: VHDL implementation of the double dabble algorithm.

1.3.4 Interval Counter

The interval counter is used to count in seconds based on the random number generated in the reaction module (Figure 8). The input clock to this is the 1Hz clock, and takes a start signal and a number in the form of a std_logic_vector. Like the display counter, a two-state state machine is used, that changes states when the input signal is active, and starts the counter. When the counter is finished a done signal becomes active, that is sent to the reaction module and the state changes back to idle.

```
process (clk)
begin
if (rising_edge(clk)) then
    case (state) is
         -- Wait until start signal is high to change states
        when idle_s =>
            done <= '0';
            counter <= 0;
            if (start_sig = '1') then
                state <= start_s;
            end if;

        -- Count to the counter value passed in from the reaction module
        when start_s =>
            if (counter = to_integer(unsigned(count_val))) then
                done <= '1';
                state <= idle_s;
            else
                counter <= counter + 1;
            end if;
    end case;
end if;
end process;
```

Figure 8: Interval counter, which counts in seconds until it reaches the value passed in from the reaction module.

1.3.5 Debounce

When interfacing with buttons, they should be debounced as when pressed, the contacts bounce around before settling the state. This settle time is around 10ms, and the VHDL module developed uses a counter to wait for this time before confirming the state of the button (Figure 9). The concurrent signal assignment will update the output signal that will be used in the reaction module. As there are three buttons (clear, start and stop), the debounce module was instantiated three times in the top file.

```
    -- Depending on the button state, when the state changes increment a counter.
    -- If the counter reaches it's limit, then read the button and set that as
    -- the state.
    process (clk) is
    begin
      if rising_edge(clk) then
        if (button_in /= btn_state and count < N) then
          count <= count + 1;
        elsif count = N then
          btn_state <= button_in;
          count <= 0;
        else
          count <= 0;

        end if;
      end if;
    end process;

    -- Assign the debounced state of the button to the output
    button_out <= btn_state;
```

Figure 9: Debounce code, used to debounce the clear, start and stop buttons.

1.3.6 Reaction

The reaction module controls the flow of the other modules, by using the flags/signals to start and event or be flagged when an event is finished. There are three parts to this module. The first is the random number being generated (Figure 10). This is a counter counting from 2 to 15, at a high clock speed (250Hz). This value is then read when in the state machine and sent to the interval counter module (Figure 8).

```
159     -- Increment a counter from 2 to 15. When counter is read the
160     -- current value is sent to the interval counter
161     process (clk, clear)
162     begin
163         if (clear = '1') then
164                 counter <= 2;
165         elsif (rising_edge(clk)) then
166               counter <= counter + 1;
167         end if;
168     end process;
```

Figure 10: Generating a random number from a fast-counting counter.

The next part is to do with the seven-segment display (Figure 11). As digits needs to be multiplex, a two-bit counter is used to switch between which anode is active. In addition, the based-on value of the digit (0-9) the segments will be changed to display that number.

```
127     -- Multiplex the digits by turning them on/off
128     process (clk)
129     begin
130        if (rising_edge(clk)) then
131            s <= s + 1;
132        end if;
133     end process;
134
135     -- Based on the value of digit, write the the segments
136     process (digit)
137     begin
138         case (digit) is
139             when X"0" => seg <= "1000000"; -- Display 0
140             when X"1" => seg <= "1111001"; -- Display 1
141             when X"2" => seg <= "0100100"; -- Display 2
142             when X"3" => seg <= "0110000"; -- Display 3
143             when X"4" => seg <= "0011001"; -- Display 4
144             when X"5" => seg <= "0010010"; -- Display 5
145             when X"6" => seg <= "0000010"; -- Display 6
146             when X"7" => seg <= "1111000"; -- Display 7
147             when X"8" => seg <= "0000000"; -- Display 8
148             when X"9" => seg <= "0010000"; -- Display 9
149             when X"A" => seg <= "0001001"; -- Display H (Changed from A, as it's not being used)
150             when X"B" => seg <= "0000011"; -- Display B
151             when X"C" => seg <= "1000110"; -- Display C
152             when X"D" => seg <= "0100001"; -- Display D
153             when X"E" => seg <= "0000110"; -- Display E
154             when X"F" => seg <= "0001110"; -- Display F
155             when others => seg <= "1111110";     -- Display "-"
156         end case;
157     end process;
```

Figure 11: Multiplexing the seven-segment display, and changing the value shown on the digits.

The last part of the reaction module is the state machine used to control the system. The state machine follows the steps mentions in 1.2.4, with the start and react state shown in Figure 12. The state moves to the start state when the start button is pressed. The value from the counter in Figure 10 , is read and a flag is set, and read by the interval counter. Following the specification, the seven-segment display turns off between 2 to 15 seconds. During this time, if the stop button is pressed state changes and display 9999 on the seven-segment display. Otherwise, the interval counter sets a flag when it's finished, turns the LED on and starts the counter to be seen on the display, and changes the state from start_s to react_s.

In the react_s state depending on the button pressed, either the display will go back to the idle state (clear button pressed), or the stop button is pressed, which will change to the stop_s state stopping the timer from incrementing.

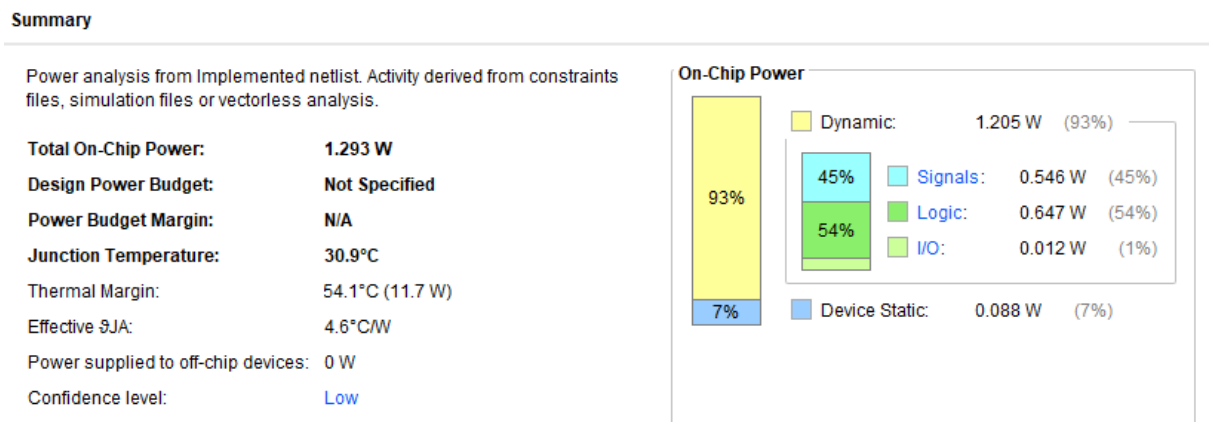
```
45                  -- Read the current counter value for the interval generator
46                  if (start = '1') then
47                      count_val <= std_logic_vector(to_unsigned(counter, count_val'length));
48                      state <= start_s;
49                  end if;
50
51                  when start_s =>
52                      -- Turn the seven segment display off
53                       case (s) is
54                          when "00" => digit <= (others => '0'); an <= "11111111";
55                          when "01" => digit <= (others => '0'); an <= "11111111";
56                          when "10" => digit <= X"1"; an <= "11111111";
57                          when others => digit <= X"A"; an <= "11111111";
58                      end case;
59
60                      -- Send the start signal
61                      start_rng <= '1';
62
63                      -- Wait for signal that random time has finished
64                      if (stop = '1') then
65                          state <= invalid_s;
66                      elsif (rng_done = '1') then
67                          start_rng <= '0';
68                          LED <= '1'; -- Turn LED on
69                          start_counter <= '1';
70
71                          state <= react_s;
72                      else
73                          state <= start_s;
74                      end if;
75                  -- Check if the clear of start buttons have been pressed
76                  when react_s =>
77                      start_counter <= '0';
78
79                      if (clear = '1') then
80                          state <= idle_s;
81                      elsif (stop = '1') then
82                          stop_sig <= '1';
83                          state <= stop_s;
84                      else
85                          state <= react_s;
86                      end if;
87
```

Figure 12: Start and react state from the state machine in the reaction module

1.3.7 Power Consumption and Resource Utilisation

An estimation of the power consumption and how many resources the design uses can be calculated in Vivado shown in Figure 13. The power is split into dynamic and static. As expected, the static power is much lower as this is when the system is idle. When the system is running the power reaches 1.205W, using 93% of the overall power. Within the dynamic power, the I/O powers use 1% as there is not a lot of interaction with the I/O. Internal logic and internal signals use are half of the remaining dynamic power, due to the modules exchanging information between each other.

The resource utilisation shows how many resources (look-up tables, slices, BRAM) were used throughout the design. Figure 13 shows the complete 132 LUT, which is less than 1% of the overall resources on the Nexys-4 DDR board. As expected, the modules that must do the least amount of processing, uses the least resources than others that require more logic.



Name	Slice LUTs (63400)	Slice Registers (126800)	Slice (15850)	LUT as Logic (63400)	Block RAM Tile (135)	Bonded IPADs (2)	BUFIO (24)
N reaction_timer_top	132	172	73	132	172	20	1
clk_1_ms (clk_div)	6	17	7	6	0	0	0
clk_1_sec (clk_div__parameterized1)	8	27	12	8	0	0	0
db_clear (debounce)	15	21	10	15	0	0	0
db_start (debounce_0)	15	21	10	15	0	0	0
db_stop (debounce_1)	16	21	10	16	0	0	0
disp_counter (display_counter)	33	11	10	33	0	0	0
interval (interval_counter)	8	6	4	8	0	0	0
reaction (reaction)	25	29	16	25	0	0	0
ssd_refresh (clk_div__parameterized0)	6	19	7	6	0	0	0

Figure 13: Power consumption (top) and resources used (bottom) in the design.

1.4 Simulation

When writing code for FPGA, an important concept to remember is that VHDL (or other languages) is not a programming language, but a hardware description language. As with unit tests for high level languages, to test the hardware, testbenches are used. This is where the output signals are tested, from the given input signals. Every module should have a testbench associated with it, to see a simulation of how the hardware will be performed when programmed onto the FPGA. The full testbench codes for each module are found in the Appendix, with the simulation results in the following sections.

1.4.1 Clock Divider Testbench

The clock divider testbench is nothing more than generating a clock signal and measuring the new clock signals period. This will show that the new clock is generating at the required frequency. In Figure 14 shows that the clk_out signal was generated from the onboard 100MHz clock. The clk_out has period of 1ms from the waveform.

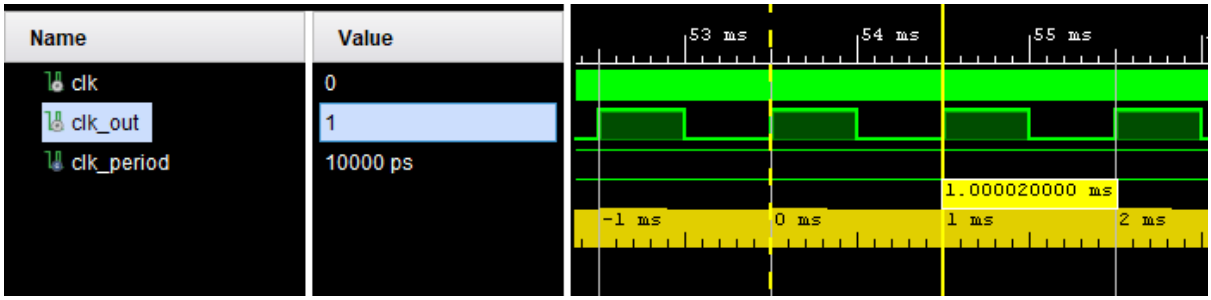


Figure 14: Simulation of the clock divider testbench showing a 1ms clock being generated.

1.4.2 Display Counter Testbench

The display counter increments a counter when an input signal is received. The counter will stop counting when it reaches 1000, or when a stop signal is received. The testbench in Figure 15 shows that the count_out value reaches 1000, then another enable signal is received. This restarts the counter, but a stop signal is received after some time. This stops the counter at ten and returns into the idle state.

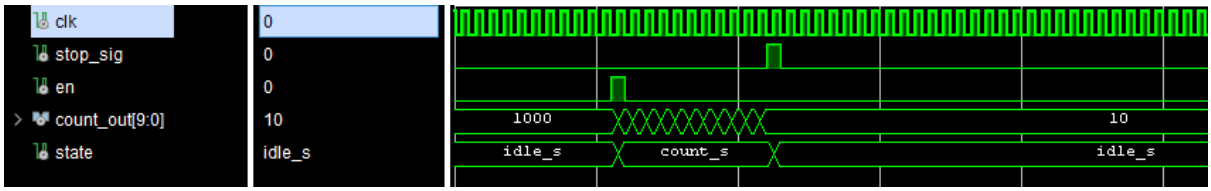


Figure 15: Simulation of the display counter testbench showing the counter counting and being reset.

1.4.3 Interval Counter Testbench

The interval counter is used to count to a random number in seconds. This number comes from the reaction module and passes into this module. When the start signal is received, a counter starts to increment. Once it reaches the required limit, a done signal is sent high. The testbench in Figure 16 shows this, where the first number to count to is four, and the second is nine.

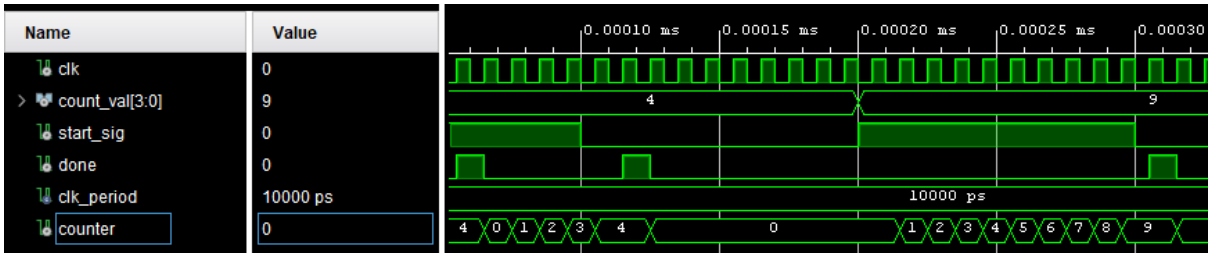


Figure 16: Simulation of the interval counter testbench counting to the number in the count_val signal.

1.4.4 Bin2BCD Testbench

The bin2bcd (binary to binary coded decimal) is used to convert the binary input, into its binary coded decimal format, over four digits. The testbench in Figure 17 shows the binary inputs of 0, 8, 10, 799, 252 and 1023, with its bcd format shown over the units, tens, hundreds and thousands vectors.

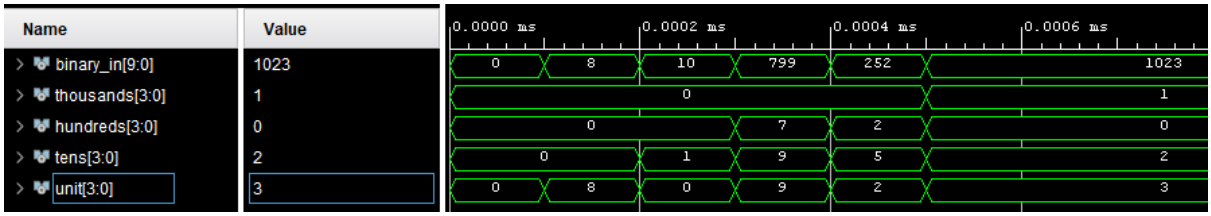


Figure 17: Simulation of Bin2BCD showing the binary input being converted to BCD and shown in correct signal.

1.4.5 Debounce Testbench

The debounce is used when interfacing with buttons. Without this reading from the button can cause false readings. In the testbench shown in Figure 18, a simulation of a debounce is used to test the code. Initially, the button_in signal bounce and the output remains unchanged. Only when the input becomes stable (after 10ms), does the output change states.

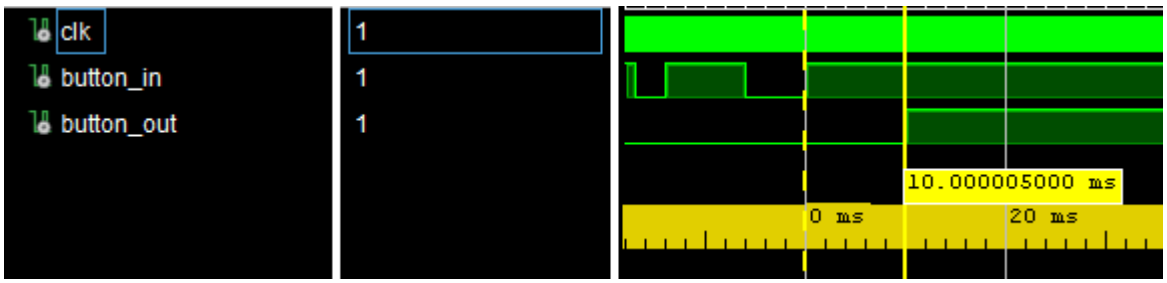


Figure 18: Simulation of the debounce testbench showing the state changing only when the input becomes stable.

1.4.6 Reaction Timer Top Testbench

The last testbench is for the reaction module. To write a full testbench for this is not as straightforward as the other, as there are many signals coming in and out of the module, which change values constantly depending on a condition. For example, the counter shown on the seven-segment display will have to be replicated, as well as the random number generator. For this, it is much easier to write the testbench for the top module, as this will control all other modules. From here, the reaction module simulation can be seen. A lot of signals are being viewed in the simulation shown in Figure 19, but some of the important points are:

- During the start_s state, the seven-segment display is off, which is why the anode signal is FF. When the state changes, the display turns back on to show the counter counting.
- As the stop signal is never simulated, the timer stops at 1000 as expected, which can be seen on the units, tens, hundredths and thousands signals.
- The interval counter counted to 9 (in seconds), before changing to the react_s state.

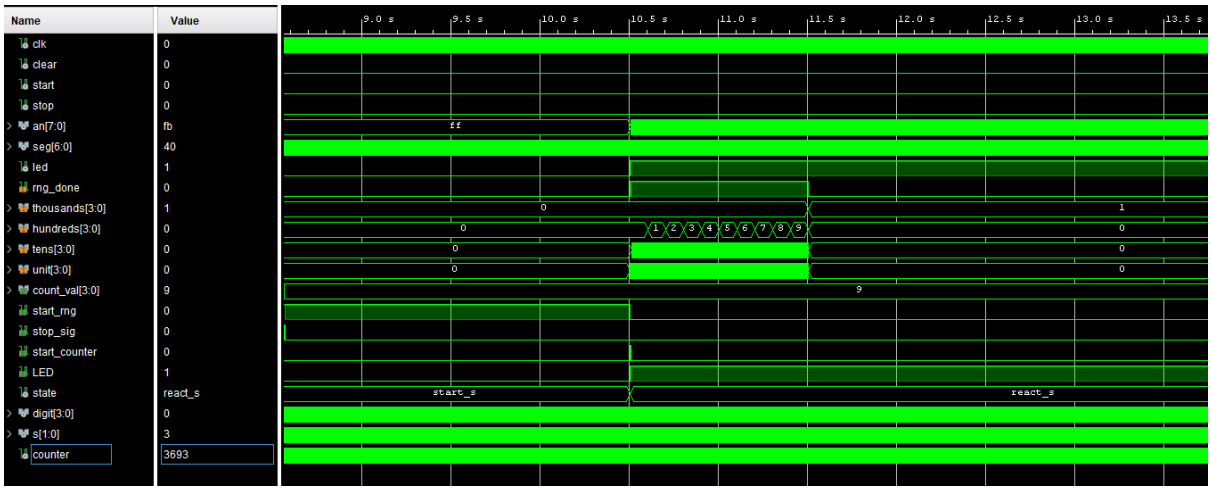


Figure 19: Snippet of the simulation for the reaction timer top module, showing the signals related to the reaction module.

1.5 Discussion and Summary

The way to code was developed promoted code reusability. Instead of hard coding some of the values, using the GENERIC keyword allowed the same module to be instantiated with a different value. Each of the modules was tested with a testbench to see in the simulation how the circuit was performed. Alongside this, the code written removed any metastability issues, which is where the state of an output is metastable. This was done by having all process statements synchronous with the clock, and not having any asynchronous resets.

Although this is an effective way of checking the logic in the code is correct, doing a physical test is equally as important. An example of this is testing the code without the debounce button. The first testing was done without the buttons being debounced. This caused glitches in the performance when pressing a button. For example, pressing the start button very quickly, would turn the seven-segment display off, wait for the random interval, but not start the millisecond counter. This caused the seven-segment display to just display either “0000”, or the last value that the count was at. Once the buttons were debounced, this fixed this glitch and the system worked as expected.

How the random number was generated worked, however an alternative approach would be to use libraries for this or a linear feedback shift register. This would give a more reliable random number and could potentially reduce the overall resources used. In addition, the clock wizard can be used and is the recommended way to design clocks that are at higher frequencies. When doing this, the FPGA will use the dedicated clock pins to generate these clocks to handle the fast state changes.

Sections of the code could be optimised better, to reduce the number of resources the FPGA used. This can become important with larger designs that do not have as many resources, or to increase the performance. In this case, a small fraction of the total resources on the Nexys-4 DDR were used, as well as the system performed reliably.

1.6 Appendix

1.6.1 Reaction Timer Top

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity reaction_timer_top is
  Port (
    clear, start, stop : in STD_LOGIC;
    clk                : in STD_LOGIC;
    an                 : out STD_LOGIC_VECTOR(7 downto 0);
    seg                : out STD_LOGIC_VECTOR(7 downto 0);
    LED                : out STD_LOGIC);
end reaction_timer_top;

architecture rtl of reaction_timer_top is
    signal clk_out, refresh : STD_LOGIC := '0';
    signal en : STD_LOGIC := '0';
    signal rng_clk       : STD_LOGIC := '0';
    signal counter : STD_LOGIC_VECTOR(9 DOWNTO 0) := (others => '0');
    signal unit, tens :  STD_LOGIC_VECTOR(3 DOWNTO 0);
    signal hundreds, thousands : STD_LOGIC_VECTOR(3 DOWNTO 0);
    signal count_val : STD_LOGIC_VECTOR(3 DOWNTO 0) := "0010";
    signal start_sig, stop_sig, rng_start, done : STD_LOGIC := '0';
    signal start_btn, stop_btn, clear_btn : STD_LOGIC := '0';
begin

-- Instantiation for generating a 1ms (1000Hz) clock
clk_1_ms : entity work.clk_div(rtl)
    generic map (N => 50000)
    port map (
      clk => clk,
      clk_out => clk_out
    );

-- Instantiation for generating a 250Hz clock
ssd_refresh : entity work.clk_div(rtl)
    generic map (N => 200000)
    port map (
      clk => clk,
      clk_out => refresh
    );

-- Instantiation for generating a 1Hz clock
clk_1_sec : entity work.clk_div(rtl)
    generic map (N => 50000000)
    port map (
      clk => clk,
      clk_out => rng_clk
    );
```

```
-- Instantiation for the 10bit counter
disp_counter : entity work.display_counter(rtl)
    port map (
      clk => clk_out,
      en => en,
      stop_sig => stop_sig,
      count_out => counter
    );

-- Instantiation for double dabble algorithm
bin2bcd : entity work.bin2bcd(rtl)
    port map (
      binary_in => counter,
      unit => unit,
      tens => tens,
      hundreds => hundreds,
      thousands => thousands
    );

-- Instantiation for interval wait time
interval : entity work.interval_counter(rtl)
    port map (
      clk => rng_clk,
      count_val => count_val,
      start_sig => rng_start,
      done => done
    );

-- The debounce window is around 10ms, and for a 100MHz clock
-- the count value should be 1e6 (1,000,000)
db_start : entity work.debounce(rtl)
    generic map (N => 1e6)
    port map (
      clk => clk,
      button_in => start,
      button_out => start_btn
      );

db_stop : entity work.debounce(rtl)
    generic map (N => 1e6)
    port map (
      clk => clk,
      button_in => stop,
      button_out => stop_btn
      );

db_clear : entity work.debounce(rtl)
```

```
    generic map (N => 1e6)
    port map (
      clk => clk,
      button_in => clear,
      button_out => clear_btn
      );

 -- Instantiation for reaction state machine and random interval generator
 reaction : entity work.reaction(rtl)
    port map (
      clk => refresh,
      clear => clear_btn,
      start => start_btn,
      stop => stop_btn,
      rng_done => done,
      stop_sig => stop_sig,
      unit => unit,
      tens => tens,
      hundreds => hundreds,
      thousands => thousands,
      seg => seg,
      an => an,
      count_val => count_val,
      start_rng => rng_start,
      start_counter => en,
      LED => LED
    );
end rtl;
```

1.6.2 Clock Divider

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity clk_div is
    Generic ( N         : integer);
    Port    ( clk       : in STD_LOGIC;
              clk_out   : out STD_LOGIC);
end clk_div;

architecture rtl of clk_div is

-- Count to the value passed in from the generic
signal counter : integer range 0 to N := 0;
signal temp    : std_logic := '0';

begin

-- Generate a clock by incrementing a counter
process (clk)
begin
    if (rising_edge(clk)) then
          counter <= counter + 1;

          -- If the value of N is reached, change the state of the temp signal
          if (counter = N) then
            temp <= NOT temp;
            counter <= 0;
          end if;
    end if;

    -- Assign the value of temp to clk_out
    clk_out <= temp;
end process;

end rtl;
```

1.6.3 Display Counter

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity display_counter is
    Port ( clk : in STD_LOGIC;
           en : in STD_LOGIC;
           stop_sig : in STD_LOGIC;
           count_out : out STD_LOGIC_VECTOR (9 downto 0));
end display_counter;

architecture rtl of display_counter is
   TYPE STATE_TYPE IS (idle_s, count_s);
   SIGNAL state   : STATE_TYPE := idle_s;
   SIGNAL counter : UNSIGNED(9 downto 0) := (others => '0');
begin

process (clk)
begin
    if (rising_edge(clk)) then
            case (state) is
                -- Wait until the enable signal is '1' to change states
                when idle_s =>
                    if (en = '1') then
                        counter <= (others => '0');
                        state <= count_s;
                    else
                        state <= idle_s;
                    end if;

                -- Increment the counter until count = 1000 or a stop signal
is recievied
                when count_s =>
                    if (counter >= 1000) then
                        state <= idle_s;
                    elsif (stop_sig = '1') then
                        state <= idle_s;
                    else
                        counter <= counter + 1;
                    end if;
            end case;
     end if;
 end process;

-- Assign the value of counter to count_out
count_out <= std_logic_vector(counter);
end rtl;
```

1.6.4 Bin2BCD

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity bin2bcd is
    Port (
           -- Input from the ms counter
           binary_in : in STD_LOGIC_VECTOR(9 DOWNTO 0);

           unit, tens : out STD_LOGIC_VECTOR(3 DOWNTO 0);
           hundreds, thousands : out STD_LOGIC_VECTOR(3 DOWNTO 0));
end bin2bcd;

architecture rtl of bin2bcd is
begin

-- Double dabble algorithm (shift and add three) used to convert the binary
input
-- to BCD, to display on four seven segment digits
process (binary_in)
variable x : UNSIGNED(25 DOWNTO 0);
begin
    -- Initialise the variable to zero
    for i in 0 to 25 loop
        x(i) := '0';
    end loop;

    -- Do the first initial shift
    x(12 downto 3) := unsigned(binary_in);

    -- For a 10-bit input, seven shifts are needed
    for i in 0 to 6 loop

        -- Check units
        if x(13 downto 10) > 4 then
            x(13 downto 10) := x(13 downto 10) + 3;
        end if;

        -- Check tens
        if x(17 downto 14) > 4 then
            x(17 downto 14) := x(17 downto 14) + 3;
        end if;

        -- Check hundreths
        if x(21 downto 18) > 4 then
            x(21 downto 18) := x(21 downto 18) + 3;
        end if;
```

```
         -- Check thousands
        if x(25 downto 22) > 4 then
            x(25 downto 22) := x(25 downto 22) + 3;
        end if;

        -- Shift the bits one to the right
        x(25 downto 1) := x(24 downto 0);
    end loop;

        -- Assign the signals to the corresponding output signals
        unit <= std_logic_vector(x(13 downto 10));
        tens <= std_logic_vector(x(17 downto 14));
        hundreds <= std_logic_vector(x(21 downto 18));
        thousands <= std_logic_vector(x(25 downto 22));

end process;

end rtl;
```

1.6.5 Interval Counter

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity interval_counter is
    Port ( clk       : in STD_LOGIC; -- 1Hz clock
           count_val : in STD_LOGIC_VECTOR(3 DOWNTO 0);
           start_sig : in STD_LOGIC;
           done      : out STD_LOGIC);
end interval_counter;

architecture rtl of interval_counter is
   TYPE STATE_TYPE IS (idle_s, start_s);
   SIGNAL state     : STATE_TYPE := idle_s;
   SIGNAL counter   : integer range 0 to 15 := 0;
begin

process (clk)
begin
if (rising_edge(clk)) then
    case (state) is
         -- Wait until start signal is high to change states
        when idle_s =>
            done <= '0';
            counter <= 0;
            if (start_sig = '1') then
                state <= start_s;
            end if;

        -- Count to the counter value passed in from the reaction module
        when start_s =>
            if (counter = to_integer(unsigned(count_val))) then
                done <= '1';
                state <= idle_s;
            else
                counter <= counter + 1;
            end if;
    end case;
end if;
end process;

end rtl;
```

1.6.6 Debounce

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity debounce is
    Generic (N          : integer);
    Port    (clk        : in std_logic;
             button_in  : in std_logic;
             button_out : out std_logic
        );
end debounce;

architecture rtl of debounce is

  signal count : integer range 0 to N := 0;
  signal btn_state : std_logic := '0';

begin

  -- Depending on the button state, when the state changes increment a
counter.
  -- If the counter reaches its limit, then read the button and set that as
  -- the state.
  process (clk) is
  begin
    if rising_edge(clk) then
      if (button_in /= btn_state and count < N) then
        count <= count + 1;
       elsif count = N then
        btn_state <= button_in;
        count <= 0;
       else
        count <= 0;

      end if;
    end if;
  end process;

  -- Assign the debounced state of the button to the output
  button_out <= btn_state;

end architecture rtl;
```

1.6.7 Reaction

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity reaction is
  Port (
        clk, clear : in STD_LOGIC;
        start, stop : in STD_LOGIC;
        rng_done : in STD_LOGIC;
        unit, tens : in STD_LOGIC_VECTOR(3 DOWNTO 0);
        hundreds, thousands : in STD_LOGIC_VECTOR(3 DOWNTO 0);
        seg : out  STD_LOGIC_VECTOR (7 downto 0);
        an : out  STD_LOGIC_VECTOR (7 downto 0);
        count_val : out STD_LOGIC_VECTOR(3 DOWNTO 0);
        start_rng, stop_sig, start_counter : out STD_LOGIC;
        LED : out STD_LOGIC);
end reaction;

architecture rtl of reaction is
    TYPE STATE_TYPE IS (idle_s, start_s, react_s, stop_s, invalid_s);
    SIGNAL state   : STATE_TYPE := idle_s;
    SIGNAL digit : STD_LOGIC_VECTOR (3 downto 0);
    SIGNAL s     : UNSIGNED(1 downto 0) := "00";
    SIGNAL counter : integer range 2 to 15 := 2;

begin

process (clk)
begin
    if (rising_edge(clk)) then
            case (state) is
                -- Turn LED off and clear flags
                when idle_s =>
                    LED <= '0';
                    start_rng <= '0';
                    stop_sig <= '0';
                    start_counter <= '0';
                -- Display "HI" on the seven-segment display
                 case (s) is
                    when "00" => digit <= (others => '0'); an <= "11111111";
                    when "01" => digit <= (others => '0'); an <= "11111111";
                    when "10" => digit <= X"1"; an <= "11111011";
                    when others => digit <= X"A"; an <= "11110111";
                end case;
                -- Read the current counter value for the interval generator
                if (start = '1') then
```

```
                    count_val <= std_logic_vector(to_unsigned(counter,
count_val'length));
                    state <= start_s;
                end if;

                when start_s =>
                    -- Turn the seven segment display off
                     case (s) is
                        when "00" => digit <= (others => '0'); an <=
"11111111";
                        when "01" => digit <= (others => '0'); an <=
"11111111";
                        when "10" => digit <= X"1"; an <= "11111111";
                        when others => digit <= X"A"; an <= "11111111";
                    end case;

                    -- Send the start signal
                    start_rng <= '1';

                    -- Wait for signal that random time has finished
                    if (stop = '1') then
                        state <= invalid_s;
                    elsif (rng_done = '1') then
                        start_rng <= '0';
                        LED <= '1'; -- Turn LED on
                        start_counter <= '1';

                        state <= react_s;
                    else
                        state <= start_s;
                    end if;
                -- Check if the clear of start buttons have been pressed
                when react_s =>
                    start_counter <= '0';

                    if (clear = '1') then
                        state <= idle_s;
                    elsif (stop = '1') then
                        stop_sig <= '1';
                        state <= stop_s;
                    else
                        state <= react_s;
                    end if;

            -- Display the BCD number from the display counter
                    case (s) is
                        when "00" => digit <= unit; an <= "11111110";
                        when "01" => digit <= tens; an <= "11111101";
```

```
                        when "10" => digit <= hundreds; an <= "11111011";
                        when others => digit <= thousands; an <= "11110111";
                    end case;

  -- If the stop button is pressed stop the counter and display the result
                when stop_s =>
                    stop_sig <= '0';
                    case (s) is
                        when "00" => digit <= unit; an <= "11111110";
                        when "01" => digit <= tens; an <= "11111101";
                        when "10" => digit <= hundreds; an <= "11111011";
                        when others => digit <= thousands; an <= "11110111";
                    end case;
                 -- Go back to idle state if clear button is pressed
                    if (clear = '1') then
                        state <= idle_s;
                    end if;
     -- If the button was pressed before the interval counter done flag is set
     -- display 9999 on the seven-segment display
                when invalid_s =>
                 case (s) is
                        when "00" => digit <= X"9"; an <= "11111110";
                        when "01" => digit <= X"9"; an <= "11111101";
                        when "10" => digit <= X"9"; an <= "11111011";
                        when others => digit <= X"9"; an <= "11110111";
                    end case;
                 -- Go back to idle state if clear button is pressed
                  if (clear = '1') then
                    state <= idle_s;
                end if;
            end case;
        end if;
end process;

    -- Multiplex the digits by turning them on/off
    process (clk)
    begin
       if (rising_edge(clk)) then
           s <= s + 1;
       end if;
    end process;

    -- Based on the value of digit, write the segments
    process (digit)
    begin
        case (digit) is
            when X"0" => seg <= "11000000"; -- Display 0
            when X"1" => seg <= "11111001"; -- Display 1
```

```
            when X"2" => seg <= "10100100"; -- Display 2
            when X"3" => seg <= "10110000"; -- Display 3
            when X"4" => seg <= "10011001"; -- Display 4
            when X"5" => seg <= "10010010"; -- Display 5
            when X"6" => seg <= "10000010"; -- Display 6
            when X"7" => seg <= "11111000"; -- Display 7
            when X"8" => seg <= "10000000"; -- Display 8
            when X"9" => seg <= "10010000"; -- Display 9
            when X"A" => seg <= "10001001"; -- Display H (Changed from A, as
it's not being used)
            when X"B" => seg <= "10000011"; -- Display B
            when X"C" => seg <= "11000110"; -- Display C
            when X"D" => seg <= "10100001"; -- Display D
            when X"E" => seg <= "10000110"; -- Display E
            when X"F" => seg <= "10001110"; -- Display F
            when others => seg <= "11111110";    -- Display "-"
        end case;
    end process;

    -- Increment a counter from 2 to 15. When counter is read the
    -- current value is sent to the interval counter
    process (clk, clear)
    begin
        if (clear = '1') then
                counter <= 2;
        elsif (rising_edge(clk)) then
              counter <= counter + 1;
        end if;
    end process;

end rtl;
```

1.6.8 Reaction Timer Top Testbench

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity reaction_timer_top_tb is
end reaction_timer_top_tb;

architecture sim of reaction_timer_top_tb is
    signal clk          : std_logic := '0';
    signal clear          : std_logic := '0';
    signal start          : std_logic := '0';
    signal stop          : std_logic := '0';
    signal an        :  STD_LOGIC_VECTOR(7 downto 0) := (others => '0');
    signal seg       :  STD_LOGIC_VECTOR(7 downto 0) := (others => '0');
    signal led    : std_logic := '0';

    constant clk_period : time := 10 ns;
begin

test_proc: entity work.reaction_timer_top(rtl)
    port map (
      clear => clear,
      start => start,
      stop => stop,
      clk => clk,
      an => an,
      seg => seg,
      LED => LED
    );

clk_proc: process
    begin
    clk <= '0';
    wait for clk_period / 2;
    clk <= '1';
    wait for clk_period / 2;
end process;

stim_proc: process
begin
    -- Pulse the clear pin
    clear <= '1';
    wait for 10 ms;
    clear <= '0';
    wait for 10 ms;

    -- Pulse the start pin
    start <= '1';
```

```
    wait for 15 ms;
    start <= '0';
    wait;
end process;

end sim;
```

1.6.9 Clock Divider Testbench

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity clk_div_tb is
end clk_div_tb;

architecture sim of clk_div_tb is

signal clk          : std_logic := '0';
signal clk_out      : std_logic := '0';
constant clk_period : time := 10 ns; -- 10ns period is equal to 100MHz

begin

-- Instantiate clk_div module
test_proc: entity work.clk_div(rtl)
generic map(N => 50000)
port map(
    clk    => clk,
    clk_out => clk_out);

-- Generate a clock signal based on the period
clk_proc: process
begin
clk <= '0';
wait for clk_period / 2;
clk <= '1';
wait for clk_period / 2;
end process;

end sim;
```

1.6.10 Display Counter Testbench

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity display_counter_tb is
end display_counter_tb;

architecture sim of display_counter_tb is
    signal clk    : std_logic := '0';
    signal en     : std_logic := '0';
    signal stop_sig    : std_logic := '0';
    signal count_out : STD_LOGIC_VECTOR(9 DOWNTO 0) := (others => '0');

    constant clk_period : time := 10 ns;
begin

test_proc : entity work.display_counter(rtl)
    port map (
      clk => clk,
      en => en,
      stop_sig => stop_sig,
      count_out => count_out
    );

clk_proc: process
    begin
    clk <= '0';
    wait for clk_period / 2;
    clk <= '1';
    wait for clk_period / 2;
end process;

stim_proc: process
begin
    en <= '1';
    wait for 10 ns;
    en <= '0';
    wait for 100000 ns;
    en <= '1';
    wait for 10 ns;
    en <= '0';
    wait for 100 ns;
    stop_sig <= '1';
    wait for 10 ns;
    stop_sig <= '0';
    wait;
end process;
end sim;
```

1.6.11 Bin2BCD Testbench

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity bin2bcd_tb is
end bin2bcd_tb;

architecture sim of bin2bcd_tb is

signal binary_in :  STD_LOGIC_VECTOR(9 DOWNTO 0) := (others => '0');
signal unit, tens :  STD_LOGIC_VECTOR(3 DOWNTO 0) := (others => '0');
signal hundreds, thousands :  STD_LOGIC_VECTOR(3 DOWNTO 0) := (others => '0');

begin

test_proc: entity work.bin2bcd(rtl)
port map(
    binary_in => binary_in,
    unit => unit,
    tens => tens,
    hundreds => hundreds,
    thousands => thousands
    );

stim_proc: process
begin
-- Input is 0
binary_in <= "0000000000";
wait for 100 ns;
-- Input is 8
binary_in <= "0000001000";
wait for 100 ns;
-- Input is 10
binary_in <= "0000001010";
wait for 100 ns;
-- Input is 799
binary_in <= "1100011111";
wait for 100 ns;
-- Input is 252
binary_in <= "0011111100";
wait for 100 ns;
-- Input is 1023
binary_in <= "1111111111";

wait;
end process;

end sim;
```

1.6.12 Interval Counter Testbench

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity interval_counter_tb is
end interval_counter_tb;

architecture sim of interval_counter_tb is
    signal clk          : std_logic := '0';
    signal count_val    : std_logic_vector(3 DOWNTO 0) := "0010";
    signal start_sig    : std_logic := '0';
    signal done         : std_logic := '0';

    constant clk_period : time := 10 ns;
begin

test_proc: entity work.interval_counter(rtl)
    port map (
      clk => clk,
      count_val => count_val,
      start_sig => start_sig,
      done => done
    );

clk_proc: process
    begin
    clk <= '0';
    wait for clk_period / 2;
    clk <= '1';
    wait for clk_period / 2;
end process;

-- Set a count value and pulse the start signal
stim_proc: process
begin
    count_val <= "0100"; -- test for four
    start_sig <= '1';
    wait for 100 ns;
    start_sig <= '0';
    wait for 100 ns;

    count_val <= "1001"; -- test four nine
    start_sig <= '1';
    wait for 100 ns;
    start_sig <= '0';
    wait;
end process;
end sim;
```

1.6.13 Debounce Testbench

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity debounce_tb is
end debounce_tb;

architecture sim of debounce_tb is
    signal clk    : std_logic := '0';
    signal button_in    : std_logic := '0';
    signal button_out    : std_logic := '0';

    constant clk_period : time := 10 ns;
begin

test_proc : entity work.debounce(rtl)
    generic map (N => 1e6)
    port map (
      clk => clk,
      button_in => button_in,
      button_out => button_out
      );

clk_proc: process
    begin
    clk <= '0';
    wait for clk_period / 2;
    clk <= '1';
    wait for clk_period / 2;
end process;

stim_proc: process
begin
    -- Simulate the input being unstable
    button_in <= '1';
    wait for 1 ms;
    button_in <= '0';
    wait for 3 ms;
    button_in <= '1';
    wait for 8 ms;
    button_in <= '0';
    wait for 6 ms;
    -- Keep the input stable
    button_in <= '1';
    wait for 10 ms;
    wait;
end process;
end sim;
```

1.6.14 Constraints.xdc

```
set_property IOSTANDARD LVCMOS33 [get_ports {an[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {an[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {an[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {an[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg[0]}]
set_property PACKAGE_PIN J17 [get_ports {an[0]}]
set_property PACKAGE_PIN J18 [get_ports {an[1]}]
set_property PACKAGE_PIN T9 [get_ports {an[2]}]
set_property PACKAGE_PIN J14 [get_ports {an[3]}]
set_property PACKAGE_PIN T10 [get_ports {seg[0]}]
set_property PACKAGE_PIN R10 [get_ports {seg[1]}]
set_property PACKAGE_PIN K16 [get_ports {seg[2]}]
set_property PACKAGE_PIN K13 [get_ports {seg[3]}]
set_property PACKAGE_PIN P15 [get_ports {seg[4]}]
set_property PACKAGE_PIN T11 [get_ports {seg[5]}]
set_property PACKAGE_PIN L18 [get_ports {seg[6]}]
set_property PACKAGE_PIN H15 [get_ports {seg[7]}]
set_property PACKAGE_PIN H17 [get_ports LED]
set_property PACKAGE_PIN E3 [get_ports clk]
set_property PACKAGE_PIN P17 [get_ports clear]
set_property PACKAGE_PIN M17 [get_ports start]
set_property PACKAGE_PIN N17 [get_ports stop]
set_property IOSTANDARD LVCMOS33 [get_ports clear]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports LED]
set_property IOSTANDARD LVCMOS33 [get_ports start]
set_property IOSTANDARD LVCMOS33 [get_ports stop]

set_property IOSTANDARD LVCMOS33 [get_ports {an[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {an[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {an[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {an[4]}]
set_property PACKAGE_PIN P14 [get_ports {an[4]}]
set_property PACKAGE_PIN T14 [get_ports {an[5]}]
set_property PACKAGE_PIN K2 [get_ports {an[6]}]
set_property PACKAGE_PIN U13 [get_ports {an[7]}]
```