CRN: 22840

<u>CET3136 – Logic Devices Programming</u>

Spring 2025

Experiment 9

Test Benches, BCD Counters, and Real-Time Clock

Performed By:

Anthony Paul Sevarino

Submitted to:

Prof. Ashley Evans

Department of Electrical & Computer Engineering Technology (ECET)

School of Engineering, Technology, and Advanced Manufacturing (ETAM)

Valencia College

Date Submitted

03/27/2025

Introduction

The intent of this lab is to create three distinct projects which each demonstrate a fundamental understanding of intermediate VHDL topics, those being test benches of clock-based systems, multi-digit BCD counters, and the implementation of various concepts into creating a real-world use-case project, in this case as a real-time clock. The commonality between each of the components to this experiment is the use of the clock, which on the MAX10 development board operates at 50MHz. Understanding of the seven-segment HEX displays is also vital to properly representing the logic derived in all parts of the lab. This study also uses a previously completed lab8B which acts as the system to be tested against in part A. ModelSim Altera will be used to create the functional simulation required for analysis in testbenching for part A.

LIST OF EQUIPMENT/PARTS/COMPONENTS/SOFTWARE

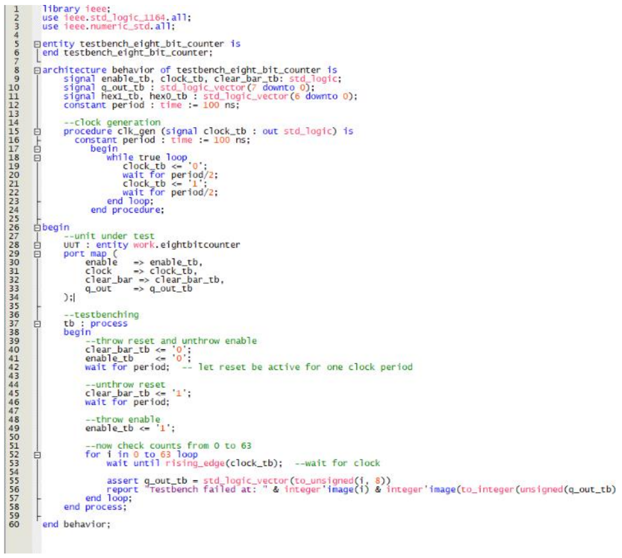
- DE10-Lite MAX-10 Dev Board
- USB 2.0 Type B Cable
- Windows Desktop
- Quartus FPGA Design Software

PROCEDURE / DISCUSSION

Part A – The Test Bench

This part of the study uses Lab8B to create a testbench against a system which utilizes a clock. Thus, an artificial clock must be generated in order to mimic the behavior of the actual clock on the development board.

Code Explanation Part A



```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity testbench_eight_bit_counter is
end testbench_eight_bit_counter;

architecture behavior of testbench_eight_bit_counter is
    signal enable_tb, clock_tb, clear_bar_tb: std_logic;
    signal q_out_tb : std_logic_vector(7 downto 0);
    signal hex1_tb, hex0_tb : std_logic_vector(6 downto 0);
    constant period : time := 100 ns;

    --clock generation
    procedure clk_gen (signal clock_tb : out std_logic) is
      constant period : time := 100 ns;
         begin
            while true loop
               clock_tb <= '0';
               wait for period/2;
               clock_tb <= '1';
               wait for period/2;
            end loop;
         end procedure;

begin
    --unit under test
    UUT : entity work.eightbitcounter
    port map (
        enable    => enable_tb,
        clock     => clock_tb,
        clear_bar => clear_bar_tb,
        q_out     => q_out_tb
    );

    --testbenching
    tb : process
    begin
        --throw reset and unthrow enable
        clear_bar_tb <= '0';
        enable_tb    <= '0';
        wait for period;  -- let reset be active for one clock period

        --unthrow reset
        clear_bar_tb <= '1';
        wait for period;

        --throw enable
        enable_tb <= '1';

        --now check counts from 0 to 63
        for i in 0 to 63 loop
            wait until rising_edge(clock_tb);  --wait for clock

            assert q_out_tb = std_logic_vector(to_unsigned(i, 8))
            report "Testbench failed at: " & integer'image(i) & integer'image(to_integer(unsigned(q_out_tb))
        end loop;
    end process;

end behavior;
```

Figure 1 - Completed Code for Part A

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

entity testbench_eight_bit_counter is
end testbench_eight_bit_counter;

architecture behavior of testbench_eight_bit_counter is
   signal enable_tb, clock_tb, clear_bar_tb: std_logic;
   signal q_out_tb : std_logic_vector(7 downto 0);
            signal hex1_tb, hex0_tb : std_logic_vector(6 downto 0);
   constant period : time := 100 ns;

   --clock generation
   procedure clk_gen (signal clock_tb : out std_logic) is
                        constant period : time := 100 ns;
                                    begin
                                                while true loop
                                                            clock_tb <= '0';
                                                            wait for period/2;
                                                            clock_tb <= '1';
                                                            wait for period/2;
                                                end loop;
                                    end procedure;

begin
   --unit under test
   UUT : entity work.eightbitcounter
            port map (
      enable   => enable_tb,
      clock    => clock_tb,
      clear_bar => clear_bar_tb,
      q_out    => q_out_tb
   );

   --testbenching
   tb : process
   begin
      --throw reset and unthrow enable
      clear_bar_tb <= '0';
      enable_tb    <= '0';
      wait for period;  -- let reset be active for one clock period

      --unthrow reset
      clear_bar_tb <= '1';
      wait for period;

      --throw enable
      enable_tb <= '1';

      --now check counts from 0 to 63
      for i in 0 to 63 loop
         wait until rising_edge(clock_tb);  --wait for clock
```

```
        assert q_out_tb = std_logic_vector(to_unsigned(i, 8))
        report "Testbench failed at: " & integer'image(i) & integer'image(to_integer(unsigned(q_out_tb))) &
std_logic'image(clear_bar_tb) & std_logic'image(enable_tb) severity error;
      end loop;
   end process;

end behavior;
```

The entity portion contains no information, as it is a testbench and will substitute signals for all inputs and outputs of the circuit.

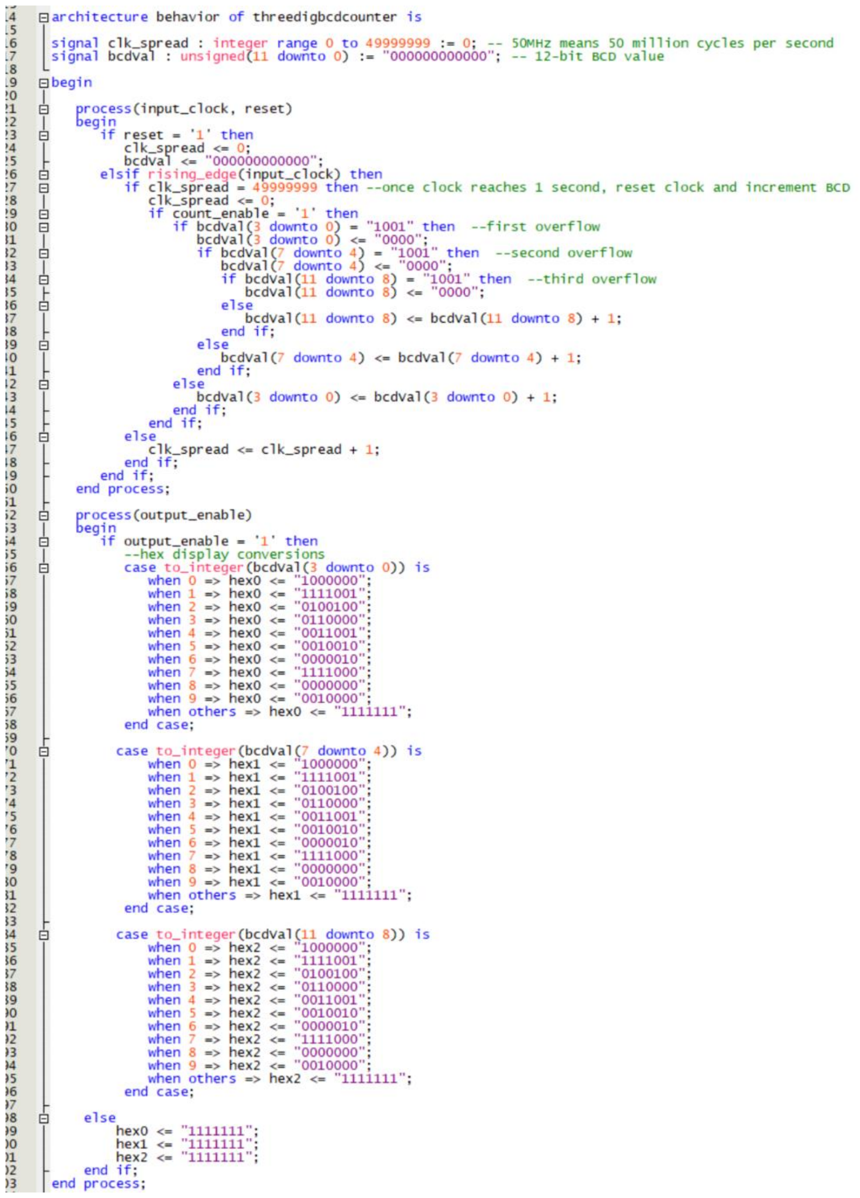
The architecture generates an artificial clock which will run indefinitely, independent of the requirements of the specific system (in this case running to 64). A *unit under test* block is created to attach the tested inputs and outputs to their testbench equivalents. Then, the testbench process tests against numerous cases, starting with clear on and enable off, then waiting for a clock period. Then, clear is disabled while enable is thrown. Finally, the *q_out* result can be checked iteratively through 64 repetitions. Each time, if the count is not matched by the UDT, which is asserted, then a report is given to the console with the respective iteration, value attachment, clear state, and enable state. Note that the part 8B experiment used to demonstrate this testbench was completed unsuccessfully and therefore throws various errors in testbenching.

There is no block diagram for this section of the study, as it is a testbench and not ported to a board nor assigned pins.

Part B – The 3-digit BCD Counter

This part of the lab requires for the construction of a 3-digit BCD counter, meaning each digit must count to *1001* in binary, or 9 in decimal. Thus, a simple binary counter is insufficient for completing this. Note that only a single value is used for the BCD while managing to break it into 3 components to be logically separated in the system.

Code Explanation Part B



```vhdl
architecture behavior of threedigbcdcounter is

signal clk_spread : integer range 0 to 49999999 := 0; -- 50MHz means 50 million cycles per second
signal bcdval : unsigned(11 downto 0) := "000000000000"; -- 12-bit BCD value

begin

   process(input_clock, reset)
   begin
      if reset = '1' then
         clk_spread <= 0;
         bcdval <= "000000000000";
      elsif rising_edge(input_clock) then
         if clk_spread = 49999999 then --once clock reaches 1 second, reset clock and increment BCD
            clk_spread <= 0;
            if count_enable = '1' then
               if bcdval(3 downto 0) = "1001" then  --first overflow
                  bcdval(3 downto 0) <= "0000";
                  if bcdval(7 downto 4) = "1001" then  --second overflow
                     bcdval(7 downto 4) <= "0000";
                     if bcdval(11 downto 8) = "1001" then  --third overflow
                        bcdval(11 downto 8) <= "0000";
                     else
                        bcdval(11 downto 8) <= bcdval(11 downto 8) + 1;
                     end if;
                  else
                     bcdval(7 downto 4) <= bcdval(7 downto 4) + 1;
                  end if;
               else
                  bcdval(3 downto 0) <= bcdval(3 downto 0) + 1;
               end if;
            end if;
         else
            clk_spread <= clk_spread + 1;
         end if;
      end if;
   end process;

   process(output_enable)
   begin
      if output_enable = '1' then
         --hex display conversions
         case to_integer(bcdval(3 downto 0)) is
            when 0 => hex0 <= "1000000";
            when 1 => hex0 <= "1111001";
            when 2 => hex0 <= "0100100";
            when 3 => hex0 <= "0110000";
            when 4 => hex0 <= "0011001";
            when 5 => hex0 <= "0010010";
            when 6 => hex0 <= "0000010";
            when 7 => hex0 <= "1111000";
            when 8 => hex0 <= "0000000";
            when 9 => hex0 <= "0010000";
            when others => hex0 <= "1111111";
         end case;

        case to_integer(bcdval(7 downto 4)) is
            when 0 => hex1 <= "1000000";
            when 1 => hex1 <= "1111001";
            when 2 => hex1 <= "0100100";
            when 3 => hex1 <= "0110000";
            when 4 => hex1 <= "0011001";
            when 5 => hex1 <= "0010010";
            when 6 => hex1 <= "0000010";
            when 7 => hex1 <= "1111000";
            when 8 => hex1 <= "0000000";
            when 9 => hex1 <= "0010000";
            when others => hex1 <= "1111111";
         end case;

        case to_integer(bcdval(11 downto 8)) is
            when 0 => hex2 <= "1000000";
            when 1 => hex2 <= "1111001";
            when 2 => hex2 <= "0100100";
            when 3 => hex2 <= "0110000";
            when 4 => hex2 <= "0011001";
            when 5 => hex2 <= "0010010";
            when 6 => hex2 <= "0000010";
            when 7 => hex2 <= "1111000";
            when 8 => hex2 <= "0000000";
            when 9 => hex2 <= "0010000";
            when others => hex2 <= "1111111";
         end case;

    else
        hex0 <= "1111111";
        hex1 <= "1111111";
        hex2 <= "1111111";
    end if;
end process;
```

Figure 2 - Completed Code for Part B

```
-------------MAIN BLOCK-------------
------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity threedigbcdcounter is
port    (
                        input_clock, reset, count_enable, output_enable : in std_logic;
                        hex0, hex1, hex2 : out std_logic_vector(6 downto 0)
                );
end threedigbcdcounter;

architecture behavior of threedigbcdcounter is
```

```
signal clk_spread : integer range 0 to 49999999 := 0;           -- 50MHz means 50 million cycles per second
signal bcdVal : unsigned(11 downto 0) := "000000000000"; -- 12-bit BCD value

begin

        process(input_clock, reset)
        begin
                if reset = '1' then
                        clk_spread <= 0;
                        bcdVal <= "000000000000";
                elsif rising_edge(input_clock) then
                        if clk_spread = 49999999 then       --once clock reaches 1 second, reset clock and
increment BCD
                                clk_spread <= 0;
                                if count_enable = '1' then
                                        if bcdVal(3 downto 0) = "1001" then  --first overflow
                                                bcdVal(3 downto 0) <= "0000";
                                                if bcdVal(7 downto 4) = "1001" then  --second overflow
                                                        bcdVal(7 downto 4) <= "0000";
                                                        if bcdVal(11 downto 8) = "1001" then  --third
overflow
                                                                bcdVal(11 downto 8) <= "0000";
                                                        else
                                                                bcdVal(11 downto 8) <= bcdVal(11 downto
8) + 1;
                                                        end if;
                                                else
                                                        bcdVal(7 downto 4) <= bcdVal(7 downto 4) + 1;
                                                end if;
                                        else
                                                bcdVal(3 downto 0) <= bcdVal(3 downto 0) + 1;
                                        end if;
                                end if;
                        else
                                clk_spread <= clk_spread + 1;
                        end if;
                end if;
        end process;

        process(output_enable)
        begin
                if output_enable = '1' then
                        --hex display conversions
                        case to_integer(bcdVal(3 downto 0)) is
                                when 0 => hex0 <= "1000000";
      when 1 => hex0 <= "1111001";
      when 2 => hex0 <= "0100100";
      when 3 => hex0 <= "0110000";
      when 4 => hex0 <= "0011001";
      when 5 => hex0 <= "0010010";
      when 6 => hex0 <= "0000010";
      when 7 => hex0 <= "1111000";
      when 8 => hex0 <= "0000000";
      when 9 => hex0 <= "0010000";
      when others => hex0 <= "1111111";
                        end case;
    case to_integer(bcdVal(7 downto 4)) is
      when 0 => hex1 <= "1000000";
      when 1 => hex1 <= "1111001";
      when 2 => hex1 <= "0100100";
      when 3 => hex1 <= "0110000";
      when 4 => hex1 <= "0011001";
      when 5 => hex1 <= "0010010";
      when 6 => hex1 <= "0000010";
      when 7 => hex1 <= "1111000";
      when 8 => hex1 <= "0000000";
      when 9 => hex1 <= "0010000";
      when others => hex1 <= "1111111";
                        end case;

    case to_integer(bcdVal(11 downto 8)) is
      when 0 => hex2 <= "1000000";
      when 1 => hex2 <= "1111001";
      when 2 => hex2 <= "0100100";
      when 3 => hex2 <= "0110000";
      when 4 => hex2 <= "0011001";
      when 5 => hex2 <= "0010010";
      when 6 => hex2 <= "0000010";
      when 7 => hex2 <= "1111000";
      when 8 => hex2 <= "0000000";
      when 9 => hex2 <= "0010000";
      when others => hex2 <= "1111111";
                        end case;

else
    hex0 <= "1111111";
    hex1 <= "1111111";
    hex2 <= "1111111";
end if;
end process;


end behavior;
------------------------------------
-------------MAIN BLOCK-------------
```

The entity of part B takes four inputs, *input_clock*, *reset*, *count_enable*, and *output_enable*. These represent the clock, the reset switch, the incrementation enable switch, and the HEX display activation enable respectively. Three outputs are present, one for each HEX display required.

The architecture of the system begins with defining two signals, one for the clock cycle amount (50 million for a 50MHz clock) and a 12-bit unsigned value initialized as 0. A sensitivity list process begins with two sensitivity parameters, *input_clock* and *restart*. As both inputs present in this system are included in this sensitivity list, the entire system is therefore asynchronous. The first step in the process is to check whether reset is thrown, and if so, both the clock and the BCD value are reset to 0. Otherwise, on any rising edge of the clock, and after a designated number of risen edges, in this case 49,999,999 edges, the clock is reset, and the individual BCD component checking occurs. Then, after the BCD logic, the clock is incremented, the process is ended, and a new process begins with a single parameter sensitivity list containing the *output_enable* input. This process assigns the correct values to each HEX display if the output is enabled, otherwise it disables all three HEX displays. An issue of note was observed here, that being an attempt to use *with select* statements within a process, which is not possible. Thus, *if case* statements were used to instead drive values to each HEX display conditionally.

Below is the block diagram and pin assignments for this part of the experiment:

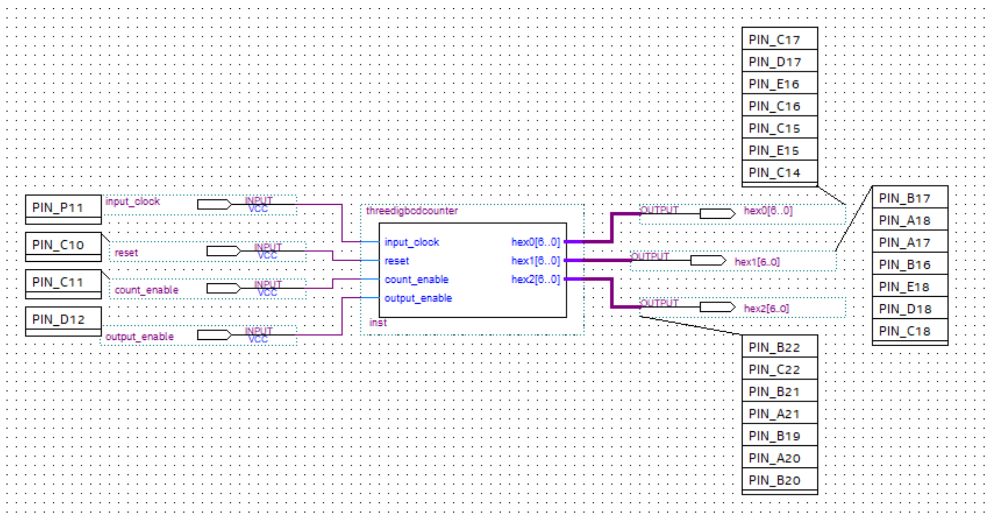
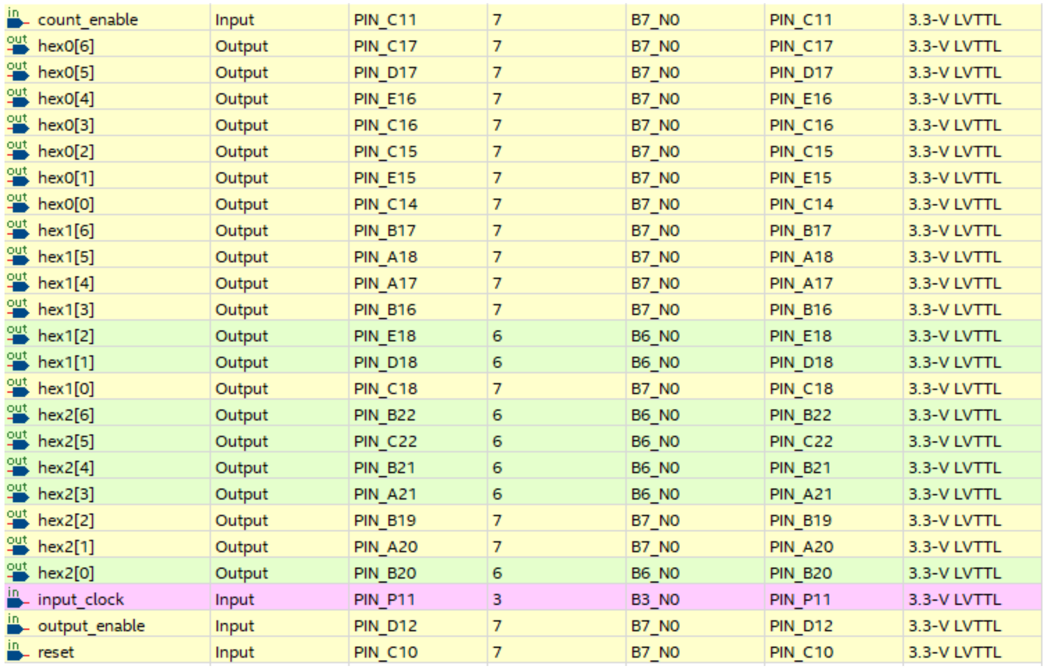


Figure 3 - Part B Block Diagram



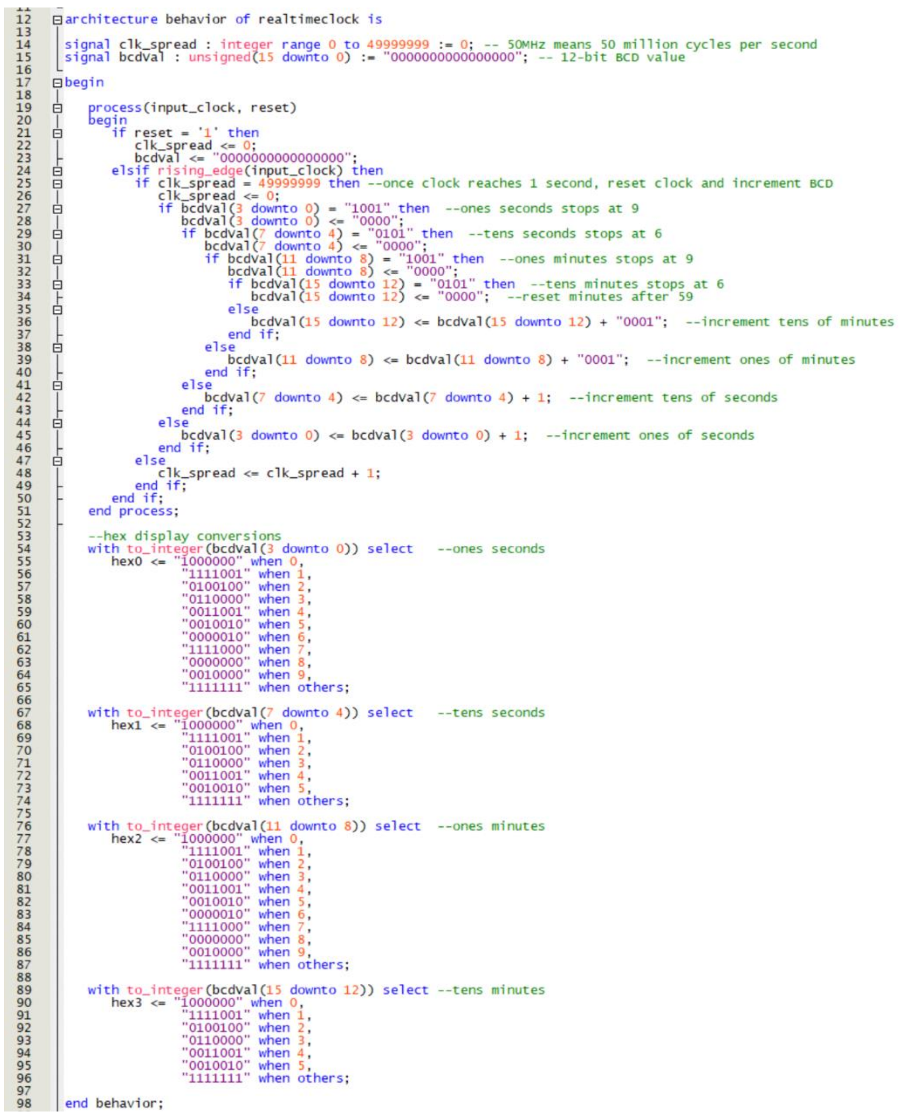
count_enable	Input	PIN_C11	7	B7_N0	PIN_C11	3.3-V LVTTL
hex0[6]	Output	PIN_C17	7	B7_N0	PIN_C17	3.3-V LVTTL
hex0[5]	Output	PIN_D17	7	B7_N0	PIN_D17	3.3-V LVTTL
hex0[4]	Output	PIN_E16	7	B7_N0	PIN_E16	3.3-V LVTTL
hex0[3]	Output	PIN_C16	7	B7_N0	PIN_C16	3.3-V LVTTL
hex0[2]	Output	PIN_C15	7	B7_N0	PIN_C15	3.3-V LVTTL
hex0[1]	Output	PIN_E15	7	B7_N0	PIN_E15	3.3-V LVTTL
hex0[0]	Output	PIN_C14	7	B7_N0	PIN_C14	3.3-V LVTTL
hex1[6]	Output	PIN_B17	7	B7_N0	PIN_B17	3.3-V LVTTL
hex1[5]	Output	PIN_A18	7	B7_N0	PIN_A18	3.3-V LVTTL
hex1[4]	Output	PIN_A17	7	B7_N0	PIN_A17	3.3-V LVTTL
hex1[3]	Output	PIN_B16	7	B7_N0	PIN_B16	3.3-V LVTTL
hex1[2]	Output	PIN_E18	6	B6_N0	PIN_E18	3.3-V LVTTL
hex1[1]	Output	PIN_D18	6	B6_N0	PIN_D18	3.3-V LVTTL
hex1[0]	Output	PIN_C18	7	B7_N0	PIN_C18	3.3-V LVTTL
hex2[6]	Output	PIN_B22	6	B6_N0	PIN_B22	3.3-V LVTTL
hex2[5]	Output	PIN_C22	6	B6_N0	PIN_C22	3.3-V LVTTL
hex2[4]	Output	PIN_B21	6	B6_N0	PIN_B21	3.3-V LVTTL
hex2[3]	Output	PIN_A21	6	B6_N0	PIN_A21	3.3-V LVTTL
hex2[2]	Output	PIN_B19	7	B7_N0	PIN_B19	3.3-V LVTTL
hex2[1]	Output	PIN_A20	7	B7_N0	PIN_A20	3.3-V LVTTL
hex2[0]	Output	PIN_B20	6	B6_N0	PIN_B20	3.3-V LVTTL
input_clock	Input	PIN_P11	3	B3_N0	PIN_P11	3.3-V LVTTL
output_enable	Input	PIN_D12	7	B7_N0	PIN_D12	3.3-V LVTTL
reset	Input	PIN_C10	7	B7_N0	PIN_C10	3.3-V LVTTL

Figure 4 - Part B Pin Assignments

Part C – The Real-time Clock

This step of the experiment marks an exciting milestone in this VHDL embedded systems development journey, as it demonstrates the design and implementation of concepts learned to this point in a relatable and useful project, that being creating a functioning real-time clock. This clock will count from 0 to 59 in seconds, then 0 to 59 in minutes subsequently.

Code Explanation Part C



```
architecture behavior of realtimeclock is

signal clk_spread : integer range 0 to 49999999 := 0; -- 50MHz means 50 million cycles per second
signal bcdVal : unsigned(15 downto 0) := "0000000000000000"; -- 12-bit BCD value

begin

   process(input_clock, reset)
   begin
      if reset = '1' then
         clk_spread <= 0;
         bcdVal <= "0000000000000000";
      elsif rising_edge(input_clock) then
         if clk_spread = 49999999 then --once clock reaches 1 second, reset clock and increment BCD
            clk_spread <= 0;
            if bcdVal(3 downto 0) = "1001" then  --ones seconds stops at 9
               bcdVal(3 downto 0) <= "0000";
               if bcdVal(7 downto 4) = "0101" then  --tens seconds stops at 6
                  bcdVal(7 downto 4) <= "0000";
                  if bcdVal(11 downto 8) = "1001" then  --ones minutes stops at 9
                     bcdVal(11 downto 8) <= "0000";
                     if bcdVal(15 downto 12) = "0101" then  --tens minutes stops at 6
                        bcdVal(15 downto 12) <= "0000";  --reset minutes after 59
                     else
                        bcdVal(15 downto 12) <= bcdVal(15 downto 12) + "0001";  --increment tens of minutes
                     end if;
                  else
                     bcdVal(11 downto 8) <= bcdVal(11 downto 8) + "0001";  --increment ones of minutes
                  end if;
               else
                  bcdVal(7 downto 4) <= bcdVal(7 downto 4) + 1;  --increment tens of seconds
               end if;
            else
               bcdVal(3 downto 0) <= bcdVal(3 downto 0) + 1;  --increment ones of seconds
            end if;
         else
            clk_spread <= clk_spread + 1;
         end if;
      end if;
   end process;

   --hex display conversions
   with to_integer(bcdVal(3 downto 0)) select   --ones seconds
      hex0 <= "1000000" when 0,
              "1111001" when 1,
              "0100100" when 2,
              "0110000" when 3,
              "0011001" when 4,
              "0010010" when 5,
              "0000010" when 6,
              "1111000" when 7,
              "0000000" when 8,
              "0010000" when 9,
              "1111111" when others;

   with to_integer(bcdVal(7 downto 4)) select   --tens seconds
      hex1 <= "1000000" when 0,
              "1111001" when 1,
              "0100100" when 2,
              "0110000" when 3,
              "0011001" when 4,
              "0010010" when 5,
              "1111111" when others;

   with to_integer(bcdVal(11 downto 8)) select  --ones minutes
      hex2 <= "1000000" when 0,
              "1111001" when 1,
              "0100100" when 2,
              "0110000" when 3,
              "0011001" when 4,
              "0010010" when 5,
              "0000010" when 6,
              "1111000" when 7,
              "0000000" when 8,
              "0010000" when 9,
              "1111111" when others;

   with to_integer(bcdVal(15 downto 12)) select --tens minutes
      hex3 <= "1000000" when 0,
              "1111001" when 1,
              "0100100" when 2,
              "0110000" when 3,
              "0011001" when 4,
              "0010010" when 5,
              "1111111" when others;

end behavior;
```

Figure 5 - Completed Architecture Block for Section C of the Experiment

---

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

entity realtimeclock is
port (
                input_clock, reset : in std_logic;
                hex0, hex1, hex2, hex3 : out std_logic_vector(6 downto 0)
                );
end realtimeclock;

architecture behavior of realtimeclock is

signal clk_spread : integer range 0 to 49999999 := 0;           -- 50MHz means 50 million cycles per second
signal bcdVal : unsigned(15 downto 0) := "0000000000000000"; -- 12-bit BCD value
```

```
begin

    process(input_clock, reset)
    begin
        if reset = '1' then
            clk_spread <= 0;
            bcdVal <= "0000000000000000";
        elsif rising_edge(input_clock) then
            if clk_spread = 49999999 then    --once clock reaches 1 second, reset clock and
increment BCD
                clk_spread <= 0;
                if bcdVal(3 downto 0) = "1001" then  --ones seconds stops at 9
                    bcdVal(3 downto 0) <= "0000";
                    if bcdVal(7 downto 4) = "0101" then  --tens seconds stops at 6
                        bcdVal(7 downto 4) <= "0000";
                        if bcdVal(11 downto 8) = "1001" then  --ones minutes stops at
9
                            bcdVal(11 downto 8) <= "0000";
                            if bcdVal(15 downto 12) = "0101" then  --tens
minutes stops at 6
                                bcdVal(15 downto 12) <= "0000";  --reset
minutes after 59
                            else
                                bcdVal(15 downto 12) <= bcdVal(15 downto
12) + "0001";  --increment tens of minutes
                            end if;
                        else
                            bcdVal(11 downto 8) <= bcdVal(11 downto 8) +
"0001";  --increment ones of minutes
                        end if;
                    else
                        bcdVal(7 downto 4) <= bcdVal(7 downto 4) + 1;  --increment
tens of seconds
                    end if;
                else
                    bcdVal(3 downto 0) <= bcdVal(3 downto 0) + 1;  --increment ones of
seconds
                end if;
            else
                clk_spread <= clk_spread + 1;
            end if;
        end if;
    end process;

    --hex display conversions
    with to_integer(bcdVal(3 downto 0)) select      --ones seconds
        hex0 <= "1000000" when 0,
                    "1111001" when 1,
                    "0100100" when 2,
                    "0110000" when 3,
                    "0011001" when 4,
                    "0010010" when 5,
                    "0000010" when 6,
                    "1111000" when 7,
                    "0000000" when 8,
                    "0010000" when 9,
                    "1111111" when others;

    with to_integer(bcdVal(7 downto 4)) select          --tens seconds
        hex1 <= "1000000" when 0,
                    "1111001" when 1,
                    "0100100" when 2,
                    "0110000" when 3,
                    "0011001" when 4,
                    "0010010" when 5,
                    "1111111" when others;

    with to_integer(bcdVal(11 downto 8)) select         --ones minutes
        hex2 <= "1000000" when 0,
                    "1111001" when 1,
                    "0100100" when 2,
                    "0110000" when 3,
                    "0011001" when 4,
                    "0010010" when 5,
                    "0000010" when 6,
                    "1111000" when 7,
                    "0000000" when 8,
                    "0010000" when 9,
                    "1111111" when others;

    with to_integer(bcdVal(15 downto 12)) select        --tens minutes
        hex3 <= "1000000" when 0,
                    "1111001" when 1,
                    "0100100" when 2,
                    "0110000" when 3,
                    "0011001" when 4,
                    "0010010" when 5,
                    "1111111" when others;

end behavior;
```

The entity portion of this code takes two inputs as *input_clock* and *reset*, with four outputs acting as the HEX display output vectors (representing the ones and tens places of both seconds and minutes)

The architecture of the system begins with defining two signals, one for the clock cycle amount (50 million for a 50MHz clock) and a 16-bit unsigned value initialized as 0. A sensitivity list process begins with two sensitivity parameters, *input_clock* and *restart*. As both inputs present in this system are included in this sensitivity list, the entire system is therefore asynchronous. The first step in the process is to check whether reset is thrown, and if so, both the clock and the BCD value are reset to 0. Otherwise, on any rising edge of the clock, and after a designated number of risen edges, in this case 49,999,999 edges, the clock is reset, and the individual BCD component checking occurs. Then, after the BCD logic, the clock is incremented, the process is ended, and the HEX displays are assigned accordingly using *with select* statements.

Below is the block diagram for the experiment.

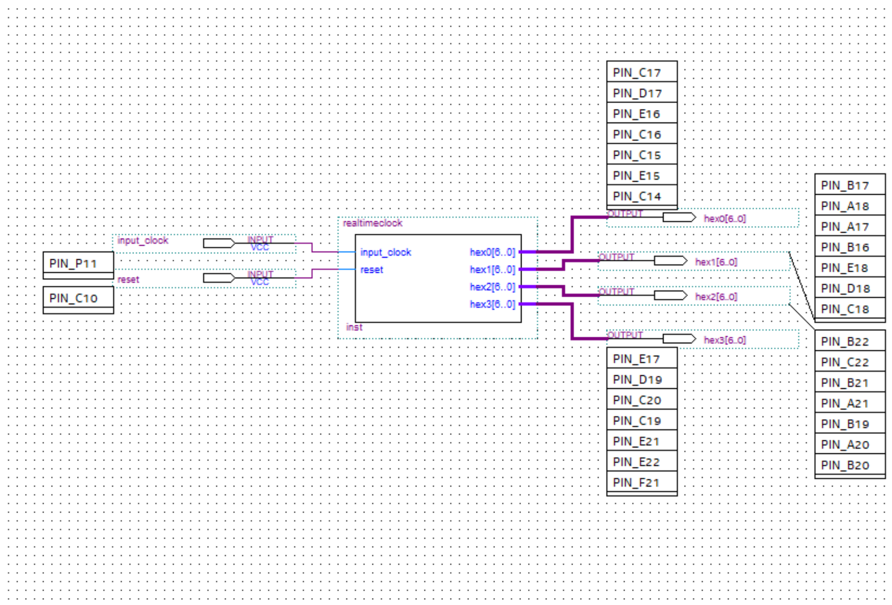
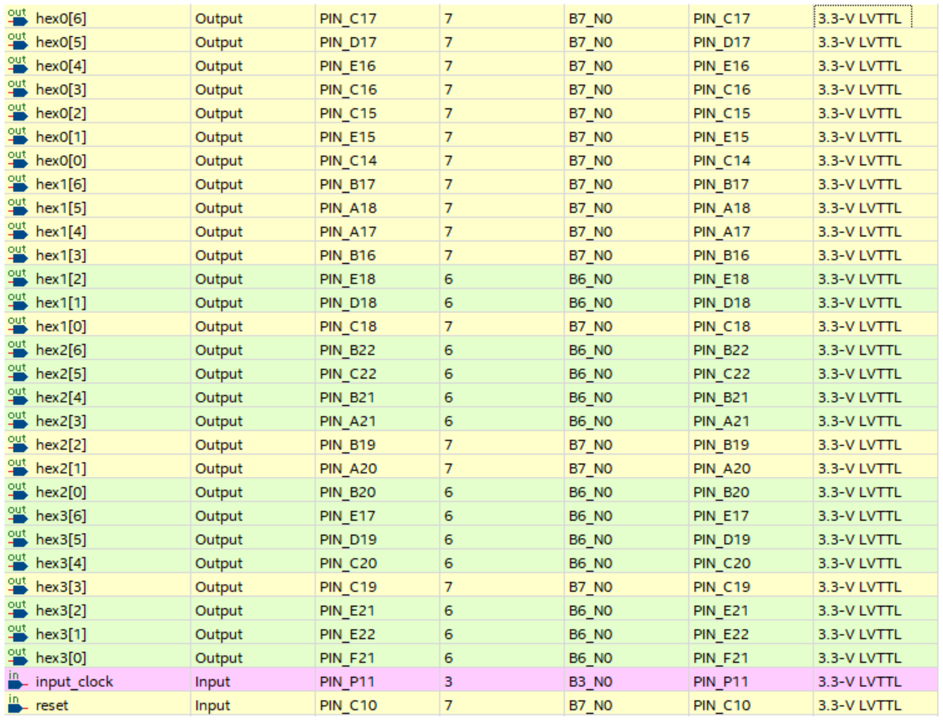


Figure 6 - Part C Block Diagram



hex0[6]	Output	PIN_C17	7	B7_N0	PIN_C17	3.3-V LVTTL
hex0[5]	Output	PIN_D17	7	B7_N0	PIN_D17	3.3-V LVTTL
hex0[4]	Output	PIN_E16	7	B7_N0	PIN_E16	3.3-V LVTTL
hex0[3]	Output	PIN_C16	7	B7_N0	PIN_C16	3.3-V LVTTL
hex0[2]	Output	PIN_C15	7	B7_N0	PIN_C15	3.3-V LVTTL
hex0[1]	Output	PIN_E15	7	B7_N0	PIN_E15	3.3-V LVTTL
hex0[0]	Output	PIN_C14	7	B7_N0	PIN_C14	3.3-V LVTTL
hex1[6]	Output	PIN_B17	7	B7_N0	PIN_B17	3.3-V LVTTL
hex1[5]	Output	PIN_A18	7	B7_N0	PIN_A18	3.3-V LVTTL
hex1[4]	Output	PIN_A17	7	B7_N0	PIN_A17	3.3-V LVTTL
hex1[3]	Output	PIN_B16	7	B7_N0	PIN_B16	3.3-V LVTTL
hex1[2]	Output	PIN_E18	6	B6_N0	PIN_E18	3.3-V LVTTL
hex1[1]	Output	PIN_D18	6	B6_N0	PIN_D18	3.3-V LVTTL
hex1[0]	Output	PIN_C18	7	B7_N0	PIN_C18	3.3-V LVTTL
hex2[6]	Output	PIN_B22	6	B6_N0	PIN_B22	3.3-V LVTTL
hex2[5]	Output	PIN_C22	6	B6_N0	PIN_C22	3.3-V LVTTL
hex2[4]	Output	PIN_B21	6	B6_N0	PIN_B21	3.3-V LVTTL
hex2[3]	Output	PIN_A21	6	B6_N0	PIN_A21	3.3-V LVTTL
hex2[2]	Output	PIN_B19	7	B7_N0	PIN_B19	3.3-V LVTTL
hex2[1]	Output	PIN_A20	7	B7_N0	PIN_A20	3.3-V LVTTL
hex2[0]	Output	PIN_B20	6	B6_N0	PIN_B20	3.3-V LVTTL
hex3[6]	Output	PIN_E17	6	B6_N0	PIN_E17	3.3-V LVTTL
hex3[5]	Output	PIN_D19	6	B6_N0	PIN_D19	3.3-V LVTTL
hex3[4]	Output	PIN_C20	6	B6_N0	PIN_C20	3.3-V LVTTL
hex3[3]	Output	PIN_C19	7	B7_N0	PIN_C19	3.3-V LVTTL
hex3[2]	Output	PIN_E21	6	B6_N0	PIN_E21	3.3-V LVTTL
hex3[1]	Output	PIN_E22	6	B6_N0	PIN_E22	3.3-V LVTTL
hex3[0]	Output	PIN_F21	6	B6_N0	PIN_F21	3.3-V LVTTL
input_clock	Input	PIN_P11	3	B3_N0	PIN_P11	3.3-V LVTTL
reset	Input	PIN_C10	7	B7_N0	PIN_C10	3.3-V LVTTL

Figure 7 - Pin Assignments for Part C

VALIDATION OF DATA

Functional Simulation Analysis

Below is the functional simulation analysis results for Part A, labelled accordingly. Note that these simulations were conducted in Modelsim-Altera.

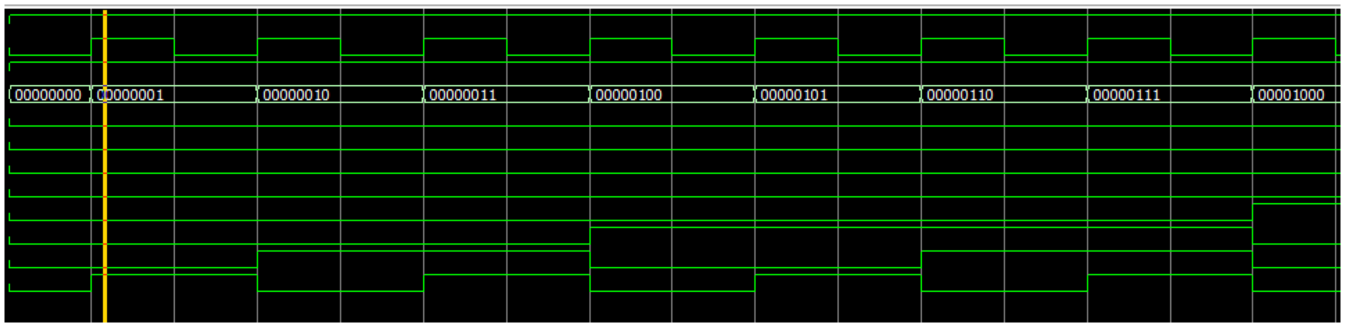


Figure 15 – Correct Incrementation Demonstration

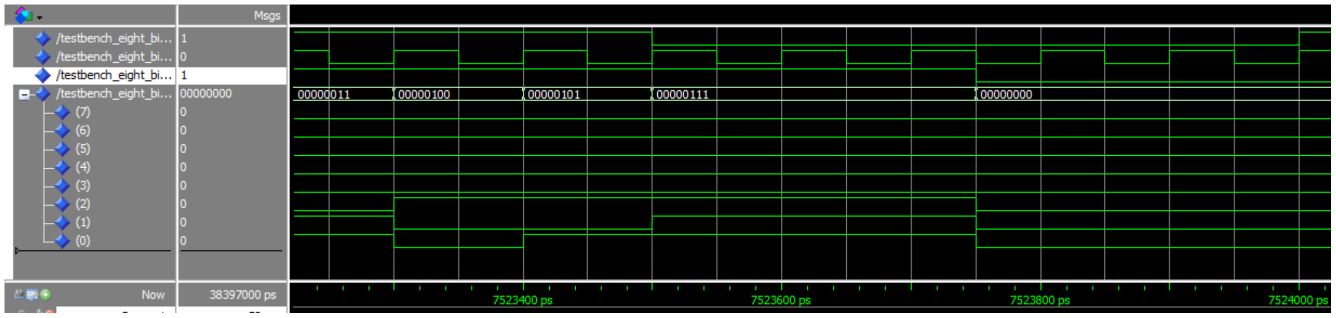
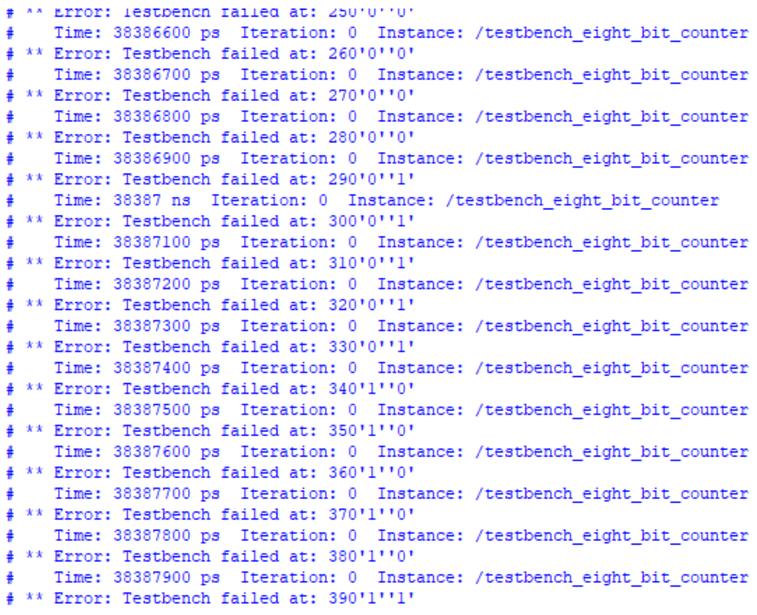


Figure 16 – Clear_Bar Throw for Reset



```
# ** Error: Testbench failed at: 250'0''0'
#    Time: 38386600 ps  Iteration: 0  Instance: /testbench_eight_bit_counter
# ** Error: Testbench failed at: 260'0''0'
#    Time: 38386700 ps  Iteration: 0  Instance: /testbench_eight_bit_counter
# ** Error: Testbench failed at: 270'0''0'
#    Time: 38386800 ps  Iteration: 0  Instance: /testbench_eight_bit_counter
# ** Error: Testbench failed at: 280'0''0'
#    Time: 38386900 ps  Iteration: 0  Instance: /testbench_eight_bit_counter
# ** Error: Testbench failed at: 290'0''1'
#    Time: 38387 ns  Iteration: 0  Instance: /testbench_eight_bit_counter
# ** Error: Testbench failed at: 300'0''1'
#    Time: 38387100 ps  Iteration: 0  Instance: /testbench_eight_bit_counter
# ** Error: Testbench failed at: 310'0''1'
#    Time: 38387200 ps  Iteration: 0  Instance: /testbench_eight_bit_counter
# ** Error: Testbench failed at: 320'0''1'
#    Time: 38387300 ps  Iteration: 0  Instance: /testbench_eight_bit_counter
# ** Error: Testbench failed at: 330'0''1'
#    Time: 38387400 ps  Iteration: 0  Instance: /testbench_eight_bit_counter
# ** Error: Testbench failed at: 340'1''0'
#    Time: 38387500 ps  Iteration: 0  Instance: /testbench_eight_bit_counter
# ** Error: Testbench failed at: 350'1''0'
#    Time: 38387600 ps  Iteration: 0  Instance: /testbench_eight_bit_counter
# ** Error: Testbench failed at: 360'1''0'
#    Time: 38387700 ps  Iteration: 0  Instance: /testbench_eight_bit_counter
# ** Error: Testbench failed at: 370'1''0'
#    Time: 38387800 ps  Iteration: 0  Instance: /testbench_eight_bit_counter
# ** Error: Testbench failed at: 380'1''0'
#    Time: 38387900 ps  Iteration: 0  Instance: /testbench_eight_bit_counter
# ** Error: Testbench failed at: 390'1''1'
```

Figure 17 – Failure cases showing iteration, value, reset state, enable state

CONCLUSION

The study was conducted successfully and efficiently demonstrated three different projects each following their own unique concept, with a shared idea of using a clock-based system. The first section of the study demonstrated the creation of a testbench for a clock-based system created in a previous study. This required the generation of an artificial clock, and testing against numerous types of failure cases using an iterative approach to ensure a broad range of testing scenarios. The second component to the experiment transitioned to designing a 3-digit BCD counter, which would aid in the design of part C of the study. This part B focused on using a single BCD value to increment three different HEX displays, each counting to 9 then overflowing back to 0, incrementing the next display subsequently. Finally, the third and final part of the experiment designed a real-time clock which counted up to an hour in minutes and seconds. A similar approach to part B was used to complete this task, using a single BCD value to increment selectively increment through to values assigned to each HEX display. If this experiment were to be conducted again, testbenches could be created for each section of the lab, an additional digit could be added to the counter in part B or a structural approach used instead of behavioral, and finally an additional milliseconds component could be implemented into part C.

REFERENCES

[1] Professor Ashley Evans, *Logic Devices Programming Lab Manual*, 1st ed. Orlando, FL. Valencia College, 2025.