Experiment 3
String Recognizer

Amit Kumar — 16D070034

March 18, 2018

Overview

In this experiment, we have implemented a *string recognizer* (A sequential circuit using mealy FSM) that can identify the occurrence of the following words.

- bomb
- gun
- knife
- terror

The VHDL code was compiled on Quartus Prime, and simulated using ModelSim which was then uploaded to the *Krypton v1.1* 5M1270ZT144C5N CPLD-based board via svf file and urjtag.

1 Setup

The english alphabet is represented by 5 bits ($\lceil log_2(26) \rceil$), and a bit each for *clock* and *reset* results in 7 input bits which were encoded by converting alphabet position (1-26) to 5-bit binary, for simplicity. I have implemented recognizer for each string individually using FSM mealy model then combined all entities to detect the presence of all four strings .

1.1 GUN Recognizer

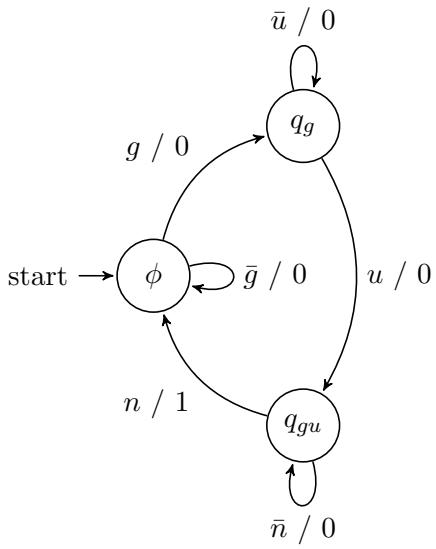


Figure 1: Automata Representation for *GUN*

Using the above state representation, the encoding has been done as shown below.

State	q_1	q_2
ϕ	0	0
g	0	1
gu	1	0

Following the above state encoding and FSM design (Figure), the code for detecting gun is given below.

```
--------------------------------GUNMAN-----------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity GUN is
port (x: in std_logic_vector(4 downto 0); W: out std_logic;
CLK,reset: in std_logic);
end entity GUN;

architecture struct of GUN is

component myDFF is
  port (D, CLK: in std_logic; Q: out std_logic);
end component myDFF;

signal nq1,nq2,q1,q2,U,G,N : std_logic ;

begin
----------------------alphabet encoding-------------------------------------
        G <= ( (not x(4)) and (not x(3)) and x(2) and x(1) and x(0) ) ;
        U <= (x(4) and (not x(3)) and x(2) and (not x(1)) and x(0) ) ;
```

```
        N <= ( (not x(4)) and x(3) and x(2) and x(1) and (not x(0) )) ;

    nq1 <= (( (not q1) and q2 and (not reset) and U )
            or   (q1 and (not q2) and (not reset) and (not N)  ) ) ;

        nq2 <= ( ((not q1) and (not q2) and (not reset) and G )
                or ((not reset) and (not q1) and q2 and (not U) ) ) ;

    W <=  (q1 and (not q2) and N and (not reset)) ;

   dff2 : myDFF port map (nq2,CLK,q2);

   dff1 : myDFF port map (nq1,CLK,q1);

end architecture struct;
```

1.2 Bomb Fecognizer

FSM transition diagram is shown below:

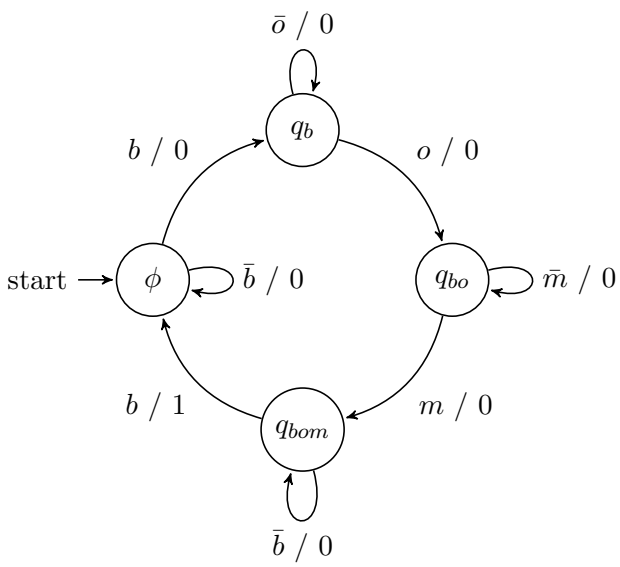


Figure 2: Automata Representation for *BOMB*

As a notation in the above figure, the text on each edge of the graph a/b represents input a to the machine, and output b of the machine.

State	q_1	q_2
ϕ	0	0
b	0	1
bo	1	0
bom	1	1

Following the above state assignment and FSM design (Figure 1), the code for detecting `bomb` is given below.

```
--library ieee;
--use ieee.std_logic_1164.all;
--
--entity myDFF is
--  port (D, CLK: in std_logic; Q: out std_logic);
--end entity myDFF;
--
--architecture WhatDoYouCare of myDFF is
--begin
--
--   process (CLK)
--   begin
--        if CLK'event and (CLK = '1') then
--             Q <= D;
--        end if;
--   end process;
--
--end WhatDoYouCare;
-------------------------------------Bomb----------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity BOMB is
port (x: in std_logic_vector(4 downto 0); W: out std_logic;
CLK,reset: in std_logic);
end entity BOMB;

architecture struct of BOMB is

component myDFF is
  port (D, CLK: in std_logic; Q: out std_logic);
end component myDFF;

signal nq1,nq2,q1,q2,B,O,M : std_logic ;

begin
        B <= ( (not x(4)) and (not x(3)) and (not x(2)) and x(1) and (not x(0)) ) ;
        O <= ((not x(4)) and x(3) and x(2) and x(1) and x(0) ) ;
        M <= ( (not x(4)) and x(3) and x(2) and (not x(1)) and x(0) ) ;

    nq1 <= (
                ((not q1) and q2 and (not reset) and O )
            or (q1 and (not q2) and (not reset) and (not M) )
            or (q1 and (not q2) and (not reset) and M)
            or (q1 and q2 and (not reset) and (not B))
             ) ;

    nq2 <= (   ((not q1) and (not q2) and (not reset) and B )
                or ((not q1) and q2 and (not reset) and (not O) )
                or (q1 and (not q2) and (not reset) and M)
```

```
        or (q1 and q2 and (not reset) and (not B))
                ) ;

    W <=  (q1 and q2 and (not reset) and B) ;

   dff2 : myDFF port map (nq2,CLK,q2);

   dff1 : myDFF port map (nq1,CLK,q1);

end architecture struct;
```

1.3 Knife Recognizer

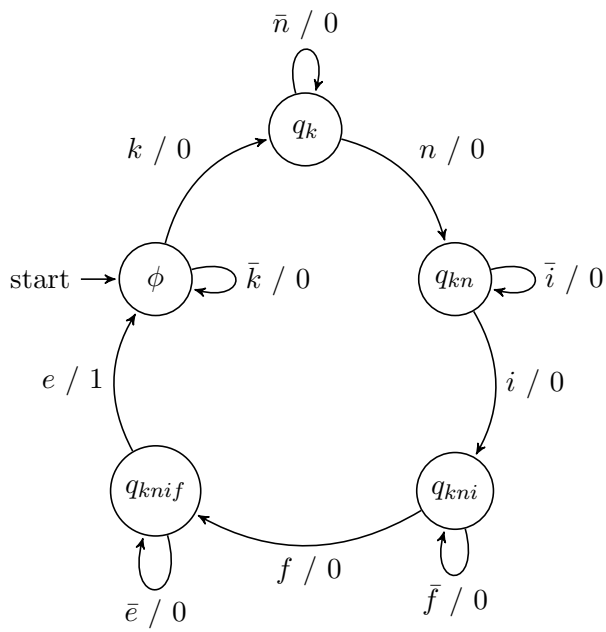


Figure 3: Automata Representation for *KNIFE*

Going with the above state representation, the encoding requires 3 bits (5 states), and hence I have shown encoding below.

State	q_1	q_2	q_3
ϕ	0	0	0
k	0	0	1
kn	0	1	0
kni	0	1	1
$knif$	1	0	0

Following the above state assignment and FSM design (Figure 3), the code for detecting `knife` is given below.

```
---------------------------------KNIFE-----------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

```

```
entity KNIFE is
port (x: in std_logic_vector(4 downto 0); W: out std_logic;
CLK,reset: in std_logic);
end entity KNIFE;

architecture struct of KNIFE is

component myDFF is
  port (D, CLK: in std_logic; Q: out std_logic);
end component myDFF;

signal nq1,nq2,nq3,q1,q2,q3,K,N,I,F,E: std_logic ;

begin
        K <= ( (not x(4)) and  x(3) and (not x(2)) and x(1) and x(0) ) ;
        N <= ( (not x(4)) and x(3) and x(2) and x(1) and (not x(0)) ) ;
        I <= ((not x(4)) and x(3) and (not x(2)) and (not x(1)) and x(0) ) ;
        F <= ((not x(4)) and (not x(3)) and x(2) and x(1) and (not x(0)) ) ;
        E <= ((not x(4)) and (not x(3)) and x(2) and  (not x(1)) and  x(0)) ;

        --not_K <= ( x(4) or (not x(3)) or x(2) or (not x(1)) or (not x(0)) ) ;
        --not_N <= ( x(4) or (not x(3)) or (not x(2)) or (not x(1)) or x(0) ) ;
        --not_I <= ( x(4) or (not x(3)) or (not x(2)) or x(1) or (not x(0)) ) ;
        --not_F <= ( x(4) or  x(3) or (not x(2)) or (not x(1)) or x(0) ) ;
        --not_E <= ( x(4) or  x(3) or (not x(2)) or x(1) or (not x(0)) ) ;
        --not_K,not_N,not_I,not_F,not_E


    nq1 <= (
                ((not q1) and q2 and q3 and (not reset) and F )
            or (q1 and (not q2) and (not q3) and (not reset) and (not E))
             ) ;

        nq2 <= (
                    ((not q1) and (not q2) and q3 and (not reset) and N )
                or ((not q1) and q2 and (not q3) and (not reset) and I )
                or ((not q1) and q2 and (not q3) and (not reset) and (not I))
        or ((not q1) and q2 and q3 and (not reset) and (not F))
                ) ;
        nq3 <= (
                     ((not q1) and (not q2) and (not q3) and (not reset) and K )
                or ((not q1) and (not q2) and q3 and (not reset) and (not N) )
                or ((not q1) and q2 and (not q3) and (not reset) and I)
        or ((not q1) and q2 and q3 and (not reset) and (not F))
                ) ;

    W <=  (q1 and (not q2) and (not q3) and (not reset) and E) ;

    dff1 : myDFF port map (nq1,CLK,q1);
    dff2 : myDFF port map (nq2,CLK,q2);
```

```
    dff3 : myDFF port map (nq3,CLK,q3);


end architecture struct;

```

1.4 Terror Recognizer

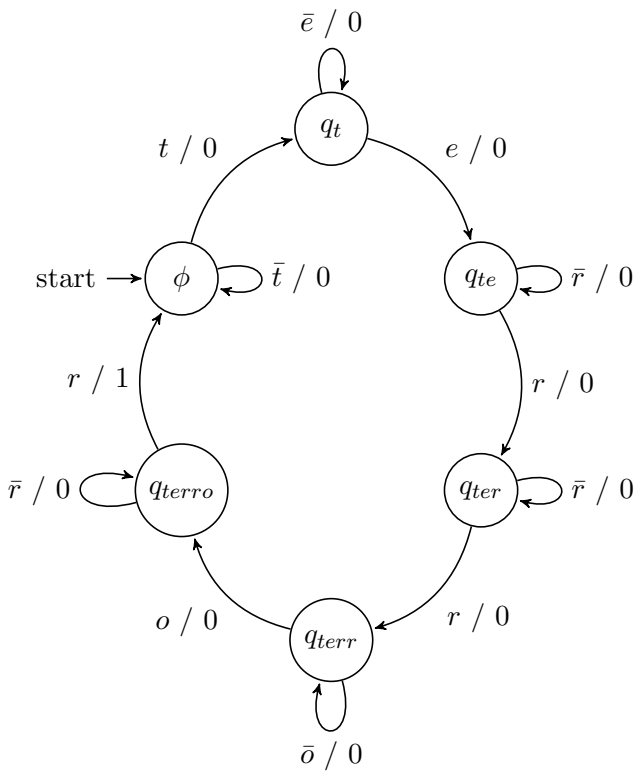


Figure 4: Automata Representation for *TERROR*

Going with the above state representation, the encoding requires 3 bits (6 states), and hence I used following for encoding the states.

State	q_1	q_2	q_3
ϕ	0	0	0
t	0	0	1
te	0	1	0
ter	0	1	1
$terr$	1	0	0
$terro$	1	0	1

Following the above state assignment and FSM design (Figure 4), the code for detecting `terror` is given below.

```
--library ieee;
--use ieee.std_logic_1164.all;
--
--entity myDFF is
--  port (D, CLK: in std_logic; Q: out std_logic);
```

```
--end entity myDFF;
--
--architecture WhatDoYouCare of myDFF is
--begin
--
--   process (CLK)
--   begin
--        if CLK'event and (CLK = '1') then
--             Q <= D;
--        end if;
--   end process;
--
--end WhatDoYouCare;
-------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity TERROR is
port (x: in std_logic_vector(4 downto 0); W: out std_logic;
CLK,reset: in std_logic);
end entity TERROR;

architecture struct of TERROR is

component myDFF is
  port (D, CLK: in std_logic; Q: out std_logic);
end component myDFF;

signal nq1,nq2,nq3,q1,q2,q3,T,E,O,R: std_logic ;

begin
    T <= (x(4) and (not x(3)) and x(2) and  (not x(1)) and (not x(0)) ) ;
        E <= ((not x(4)) and (not x(3)) and x(2) and  (not x(1)) and  x(0) ) ;
        O <= ((not x(4)) and x(3) and x(2) and x(1) and x(0) ) ;
        R <= ( x(4) and (not x(3)) and (not x(2)) and x(1) and (not x(0)) ) ;

    nq1 <= (
                ((not q1) and q2 and q3 and (not reset) and R )
            or (q1 and (not q2) and (not q3) and (not reset) and (not O))
            or (q1 and (not q2) and (not q3) and (not reset) and O)
            or (q1 and (not q2) and q3 and (not reset) and (not R))
             ) ;

        nq2 <= (
                    ((not q1) and (not q2) and q3 and (not reset) and E)
                 or ((not q1) and q2 and (not q3) and (not reset) and R)
                 or ((not q1) and q2 and (not q3) and (not reset) and (not R))
        or ((not q1) and q2 and q3 and (not reset) and (not R))
                ) ;
        nq3 <= (
```

```
                    ((not q1) and (not q2) and (not q3) and (not reset) and T )
                 or ((not q1) and q2 and (not q3) and (not reset) and R)
       or ((not q1) and q2 and q3 and (not reset) and (not R))
                 or (q1 and (not q2) and (not q3) and (not reset) and O )
                 or (q1 and (not q2) and q3 and (not reset) and (not R) )
                 or ( (not q1) and (not q2) and q3 and (not reset) and (not E))
                 );

    W <=  (q1 and (not q2) and q3 and (not reset) and R) ;

    dff1 : myDFF port map (nq1,CLK,q1);
    dff2 : myDFF port map (nq2,CLK,q2);
    dff3 : myDFF port map (nq3,CLK,q3);


end architecture struct;
```

Final String Recognizer

Since I have implemented four individual FSMs , following I have combined all four to output '1' if any of the four strings are found .

```
library ieee;
use ieee.std_logic_1164.all;
-- X4,X3,X2,X1,X0: in std_logic
 entity string_recognizer is
port (X: in std_logic_vector(4 downto 0) ; W: out std_logic;
CLK,reset: in std_logic);
end entity string_recognizer;

architecture struct of string_recognizer is

component GUN is
port (X: in std_logic_vector(4 downto 0) ; W: out std_logic;
CLK,reset: in std_logic);
end component GUN;

component BOMB is
port (X: in std_logic_vector(4 downto 0) ; W: out std_logic;
CLK,reset: in std_logic);
end component  BOMB;

component  KNIFE is
port (X: in std_logic_vector(4 downto 0) ; W: out std_logic;
CLK,reset: in std_logic);
end component  KNIFE;

component  TERROR is
port (X: in std_logic_vector(4 downto 0) ; W: out std_logic;
CLK,reset: in std_logic);
end component  TERROR;
```

```

signal w1,w2,w3,w4: std_logic;
begin

gun1: GUN port map (X => X, CLK => CLK, reset => reset,W => w1);
bomb1: BOMB port map (X => X, CLK => CLK, reset => reset,W => w2);
knife1: KNIFE port map (X => X, CLK => CLK, reset => reset,W => w3);
terror1: TERROR port map (X => X, CLK => CLK, reset => reset,W => w4);

W <= w1 or w2 or w2 or w3 or w4;

end struct;
```

To map our string recognizer to the input string of tracefile , I have made a final DUT to be used for testbench.

```
-- A DUT entity is used to wrap your design.
library std;
use std.standard.all;

library ieee;
use ieee.std_logic_1164.all;

entity DUT is
   port(input_vector: in std_logic_vector(6 downto 0);
                output_vector: out std_logic_vector(0 downto 0));
end entity;

architecture DutWrap of DUT is

component string_recognizer is
port (X: in std_logic_vector(4 downto 0) ; W: out std_logic;
clk,reset: in std_logic);
end component string_recognizer;

begin
dut: string_recognizer port map( X => input_vector(4 downto 0), clk => input_vector(5) , res
end DutWrap;
```

2 Observations

After implementing the design in code, the next major part is to simulate and test the code for a set of inputs. RTL and Gate-Level simulation was performed on the machine, as a whole. Snapshots of the same are given in Figures 5-8. The validity of the code can be ascertained by the fact that all test cases passed successfully.

For simulation, I have modified the generic testbench given below:

```
library std;
use std.textio.all;

library std;
```

```
use std.standard.all;

library ieee;
use ieee.std_logic_1164.all;

entity Testbench is
end entity;
architecture Behave of Testbench is

  ----------------------------------------------------------------
  --  edit the following lines to set the number of i/o's of your
  --  DUT.
  ----------------------------------------------------------------
  constant number_of_inputs  : integer := 7;  -- # input bits to your design.
  constant number_of_outputs : integer := 1;  -- # output bits from your design.

  -- component port widths..
  component DUT is
   port(input_vector: in std_logic_vector(number_of_inputs-1  downto 0);
                output_vector: out std_logic_vector(number_of_outputs-1 downto 0));
  end component;

  -- end editing.
  ----------------------------------------------------------------
  ----------------------------------------------------------------

  signal input_vector  : bit_vector(number_of_inputs-1 downto 0);
  signal output_vector : bit_vector(number_of_outputs-1 downto 0);
  signal std_output_vector : std_logic_vector(number_of_outputs-1 downto 0);

  -- create a constrained string outof
  function to_string(x: string) return string is
      variable ret_val: string(1 to x'length);
      alias lx : string (1 to x'length) is x;
  begin
      ret_val := lx;
      return(ret_val);
  end to_string;

begin
  process
    variable err_flag : boolean := false;
    File INFILE: text open read_mode is "/home/amit/Desktop/Digital_lab/string_recognizer/st
    FILE OUTFILE: text  open write_mode is "/home/amit/Desktop/Digital_lab/string_recognizer

    ---------------------------------------------------
    -- edit the next two lines to customize
    ---------------------------------------------------
    variable input_vector_var: bit_vector (number_of_inputs-1 downto 0);
    variable output_vector_var: bit_vector (number_of_outputs-1 downto 0);
```

```
    variable output_mask_var: bit_vector (number_of_outputs-1 downto 0);
    variable output_comp_var: bit_vector (number_of_outputs-1 downto 0);
    constant ZZZZ : bit_vector(number_of_outputs-1 downto 0) := (others => '0');
    ----------------------------------------------------

    variable INPUT_LINE: Line;
    variable OUTPUT_LINE: Line;
    variable LINE_COUNT: integer := 0;


  begin
    while not endfile(INFILE) loop
          -- will read a new line every 5ns, apply input,
          -- wait for 1 ns for circuit to settle.
          -- read output.


          LINE_COUNT := LINE_COUNT + 1;


          -- read input at current time.
          readLine (INFILE, INPUT_LINE);
          read (INPUT_LINE, input_vector_var);
          read (INPUT_LINE, output_vector_var);
          read (INPUT_LINE, output_mask_var);

          -- apply input.
          input_vector <= input_vector_var;

          -- wait for the circuit to settle
          wait for 150 ns;

          -- check output.
          output_comp_var := (output_mask_var and (output_vector xor output_vector_var));
          if (output_comp_var  /= ZZZZ) then
             write(OUTPUT_LINE,to_string("ERROR: line "));
             write(OUTPUT_LINE, LINE_COUNT);
             writeline(OUTFILE, OUTPUT_LINE);
             err_flag := true;
          end if;

          write(OUTPUT_LINE, input_vector);
          write(OUTPUT_LINE, to_string(" "));
          write(OUTPUT_LINE, output_vector);
          writeline(OUTFILE, OUTPUT_LINE);

          -- advance time by 4 ns.
          wait for 4 ns;
    end loop;

```

```
    assert (err_flag) report "SUCCESS, all tests passed." severity note;
    assert (not err_flag) report "FAILURE, some tests failed." severity error;

    wait;
  end process;

        output_vector <= to_bitvector(std_output_vector);
  dut_instance: DUT
            port map(input_vector => to_stdlogicvector(input_vector), output_vector =>std_

end Behave;
```

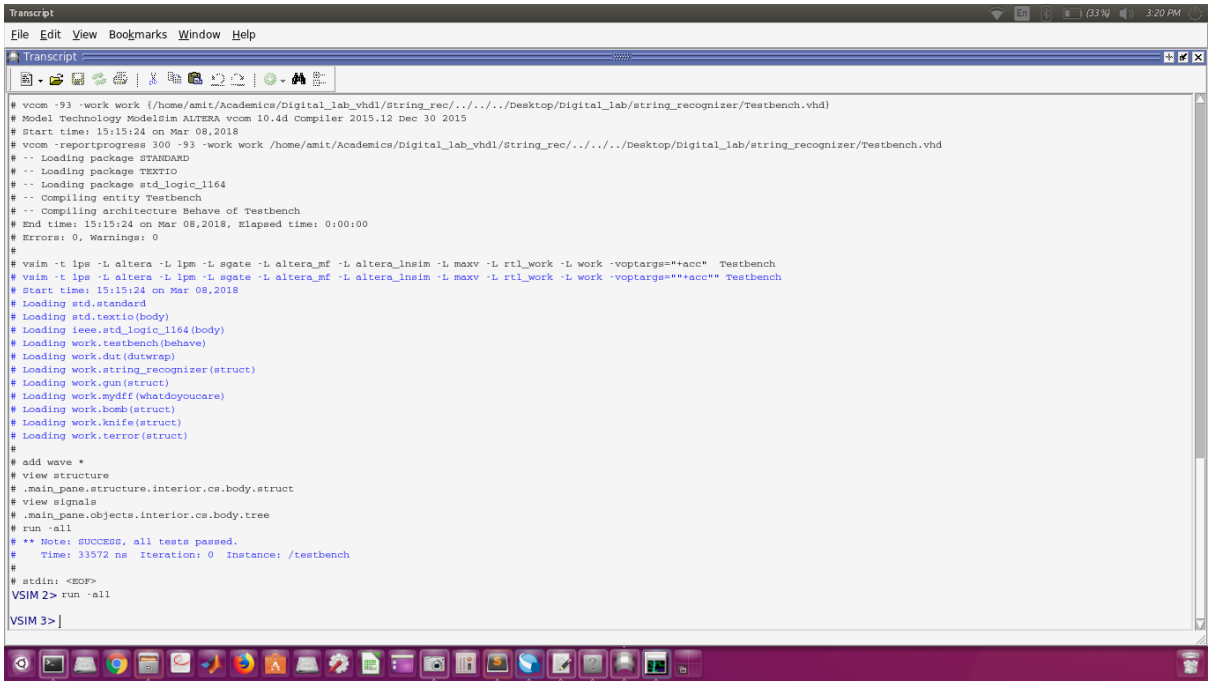


Figure 5: RTL Simulation of the String Detector for tracefile1.txt given

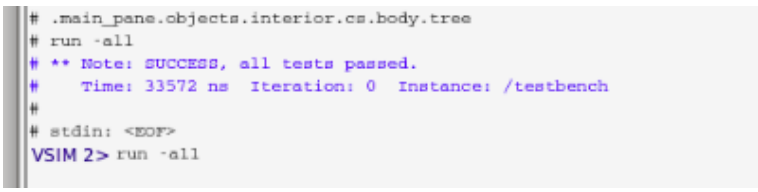


Figure 6: RTL Simulation of the String Detector for tracefile1.txt Zoomed-in

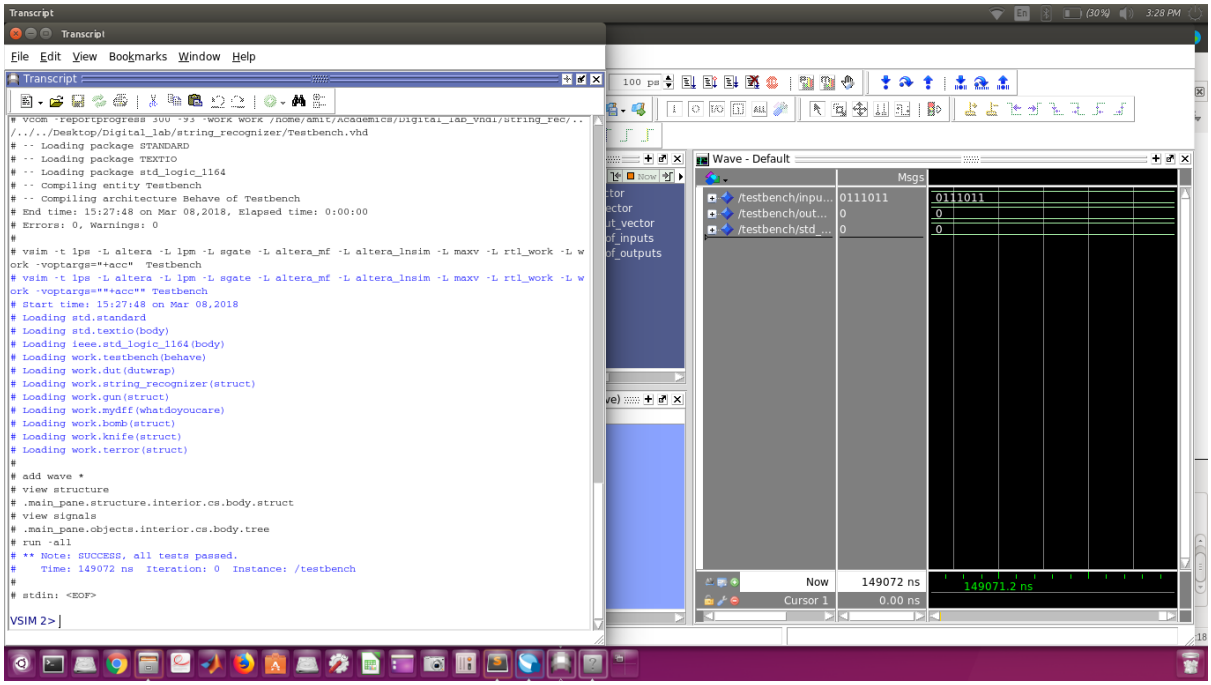


Figure 7: RTL Simulation of the String Detector for tracefile2.txt given

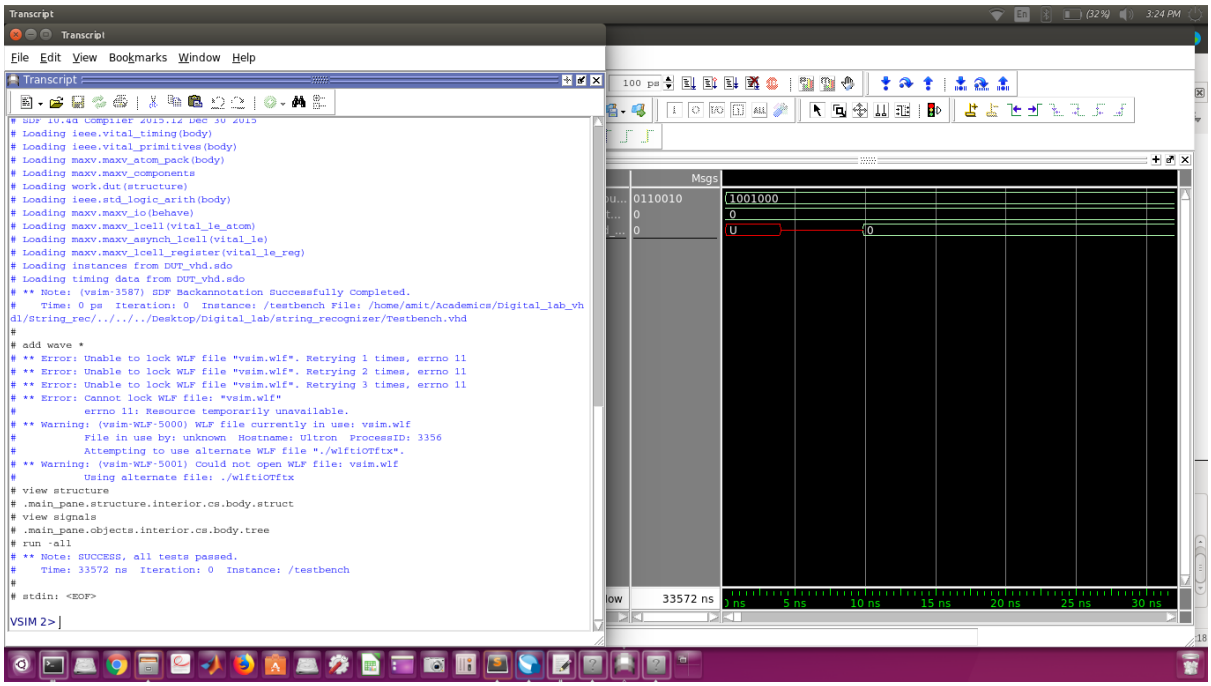


Figure 8: Gate level Simulation of the String Detector for tracefile1.txt given

Figure 9: Gate level Simulation of the String Detector for tracefile1.txt Zoomed-in transcript

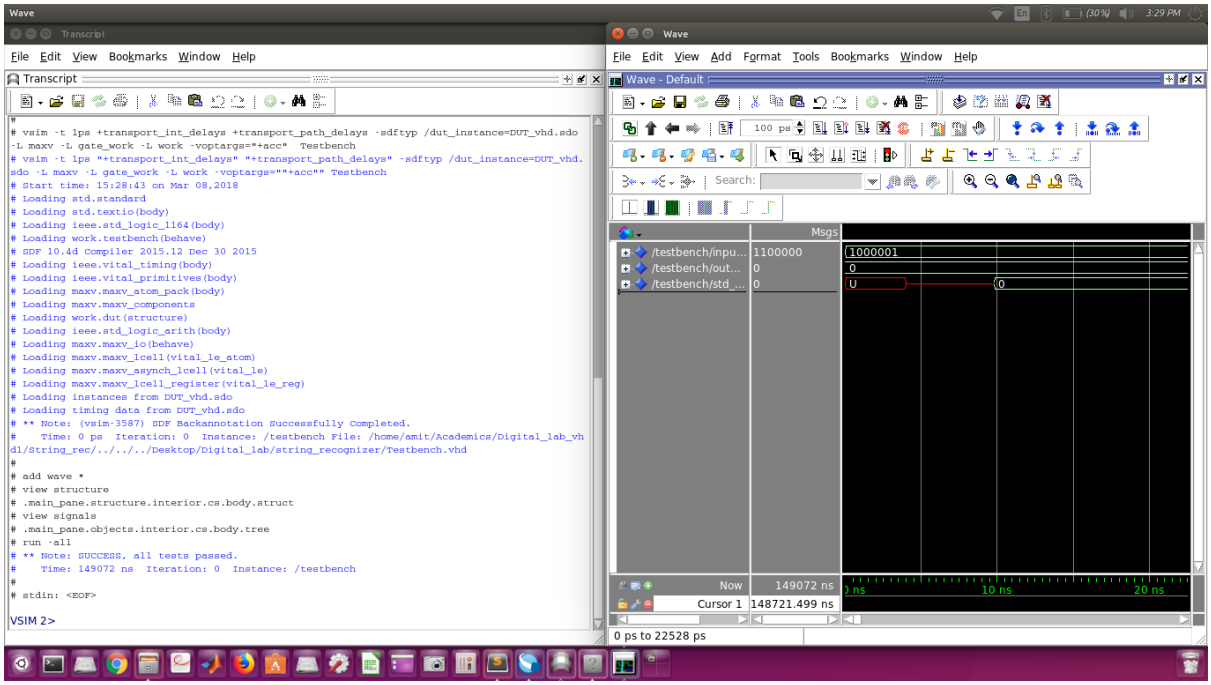


Figure 10: Gate level Simulation of the String Detector for second tracefile2.txt given

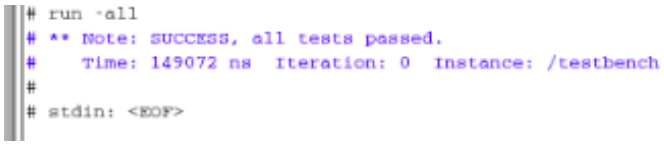


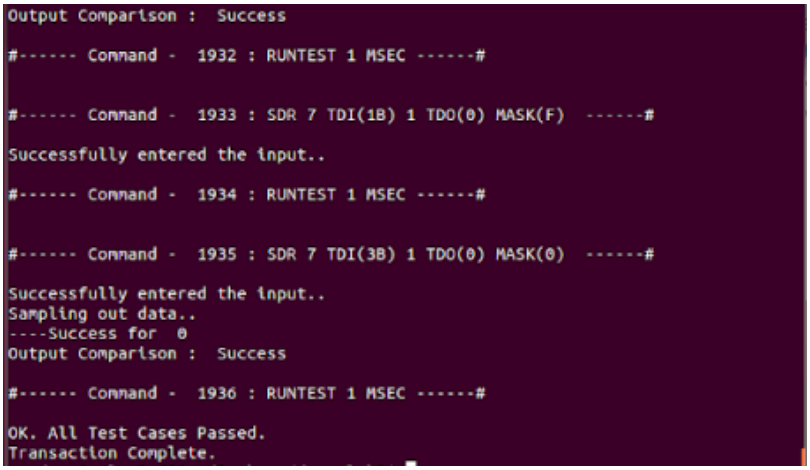
Figure 11: Gate level Simulation of the String Detector for second tracefile2.txt Zoomed-in transcript

Figure 12: Initial error in Gate level Simulation of the String Detector for second tracefile2.txt Zoomed-in transcript

3 Scan-Chain Tests

We have tested the logic using the RTL simulations, emulated the CPLD performance using the gate-level simulation and burned svf file of code on the Krypton board using urjtag . Next, we need to check that the code is actually running as it is expected to, on the board. Hence, we test the uploaded code on the hardware using the scan-chain setup, as suggested in the manual using scanchain files and Tiva-C microcontroller. This setup was run on a set of two collections of text which has occurrences of the concerned string.

Results



```
Output Comparison :  Success

#------ Command -  1932 : RUNTEST 1 MSEC ------#


#------ Command -  1933 : SDR 7 TDI(1B) 1 TDO(0) MASK(F)  ------#

Successfully entered the input..

#------ Command -  1934 : RUNTEST 1 MSEC ------#


#------ Command -  1935 : SDR 7 TDI(3B) 1 TDO(0) MASK(0)  ------#

Successfully entered the input..
Sampling out data..
----Success for  0
Output Comparison :  Success

#------ Command -  1936 : RUNTEST 1 MSEC ------#

OK. All Test Cases Passed.
Transaction Complete.
```

Figure 13: Screenshot of success of the scan-chain test

Since, all the cases passed successfully at all stages and hence the complete string recognizer can be used in hardware, as required.