Experiment 6-7
String Recognizer using the Krypton CPLD

Dhruv Ilesh Shah — 150070016

March 3, 2017

Overview

Sequential circuits are an important part of digital logic, and hence, in this report, I present the complete implementation of a *string recognizer* that can identify the occurrence of the following words.

- bomb
- gun
- knife
- terror

The code was compiled on Quartus Prime, and simulated using ModelSim. GHDL was also used for simulation purposes, at a low level. This was then uploaded to the *Krypton v1.1* 5M1270ZT144C5N CPLD-based board.

The codes and setup have been covered in section 1. We build the string recognizer piecewise, by implementing each module independently. The VHDL codes have been kept modular and as generic as possible, for reusability and code clarity. Section 2 presents the simulation observations and miscellaneous results. Section 3 presents the observations after running the scan-chain test on the board.

1 Setup

The english alphabet can be represented by 5 bits ($\lceil log_2(26) \rceil$), and a bit each for *clock* and *reset* results in 7 input bits. Managing all these, along with keeping track of the states would actually be a tedious task and we would have to deal with a complex machine with close to **300 states**! That is where the idea of interacting FSMs comes handy. In this report, I implement individual (and independent) FSMs for detecting each of the strings, and an OR of the four outputs gives the required output bit.

The inputs were encoding by converting alphabet position (1-26) to 5-bit binary, for simplicity. The various entities used for the detection include *bomber, gunman, knife_hurler & terrorist*. The output can be represented with one bit, 1 standing for positive (string detected!).

1.1 Bomber

Straight from the description, a description of the problem in terms of a finite-state machine is given below.

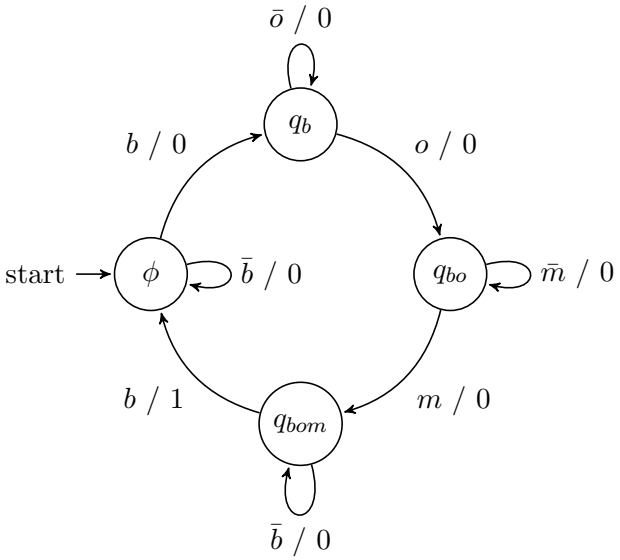

Figure 1: Automata Representation for *Bomber*

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 encoding can be done in a smart way to reduce the combinational overhead for each state bit. For this purpose, I have used *Gray Codes* to encode the states.

State	q_1	q_0
ϕ	0	1
b	1	1
bo	1	0
bom	0	0

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

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  library work;
4  use work.EE224_Components.all;
5
6  entity bomber is
7          port(x: in std_ulogic_vector(4 downto 0);
8                  reset, clk: in std_ulogic;
9                  s: out std_ulogic);
10 end entity;
11
12 architecture bomb_detector of bomber is
13         signal q, nq, qb: std_ulogic_vector(1 downto 0);
14         signal b, b1, b2, b3, n0, n1, n2, n3, n4, bb, q01, q02, q03,
15         phis, bs, bos, boms, m, m1, m2, m3, mb, q11, q12, q13, o, ob: std_ulogic;
16 --State Description:
17 --phi                0 1
18 --b                 1 1
19 --bo                 1 0
```

```
20 --bom                0 0
21
22 begin
23         INV0: inverter port map (x(0), n0);
24         N100: inverter port map (x(1), n1);
25         INV2: inverter port map (x(2), n2);
26         INV3: inverter port map (x(3), n3);
27         INV4: inverter port map (x(4), n4);
28
29         A5b: andi5 port map (n4, n3, n2, x(1), n0, b); -- Identifying B
30         A5m: andi5 port map (n4, x(3), x(2), n1, x(0), m); -- Identifying M
31         A5o: andi5 port map (n4, x(3), x(2), x(1), x(0), o); -- Identifying O
32
33
34         INV5: inverter port map (b, bb);
35         INV6: inverter port map (m, mb);
36         INV60: inverter port map (o, ob);
37
38         -- Deciphering states
39         N7: inverter port map (q(0), qb(0));
40         N8: inverter port map (q(1), qb(1));
41
42         A5: andi2 port map (qb(1), q(0), phis);
43         A6: andi2 port map (q(1), q(0), bs);
44         A7: andi2 port map (q(1), qb(0), bos);
45         A8: andi2 port map (qb(1), qb(0), boms);
46
47         -- Assigning state nq(0)
48         A9: andi2 port map (boms, b, q01);
49         O1: ori2 port map (phis, q01, q02);
50         A10: andi2 port map (bs, ob, q03);
51         O2: ori2 port map (q02, q03, nq(0));
52
53         -- Assigning state nq(1)
54         A11: andi2 port map (phis, b, q11);
55         A12: andi2 port map (bos, mb, q12);
56         O3: ori2 port map (q11, q12, q13);
57         O4: ori2 port map (q13, bs, nq(1));
58
59         -- Assigning the Output
60         A13: andi2 port map (boms, b, s);
61
62         -- Adding dffi's
63         d1: dffi port map (d => nq(1), clk => clk, q => q(1), r => reset);
64         d0: dffi1 port map (d => nq(0), clk => clk, q => q(0), r => reset);
65
66 end bomb_detector;
```

1.2 Gunman

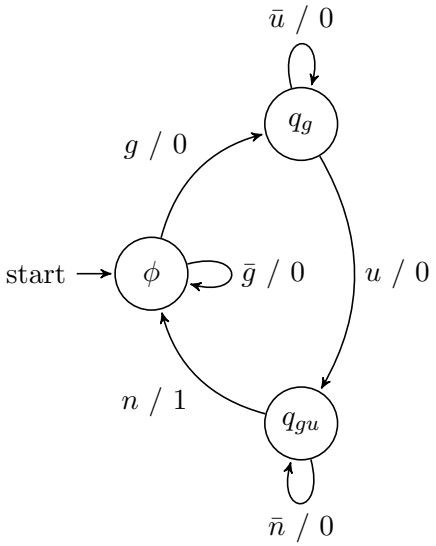

Figure 2: Automata Representation for *Gunman*

Going ahead with the above state representation, the encoding has been done as shown below (since only 2 non-ϕ states, one-hot encoding serves the purpose).

State	q_1	q_0
ϕ	0	0
g	1	0
gu	0	1

Following the above state assignment and FSM design (Figure 2), the code for detecting `gun` is given below.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  library work;
4  use work.EE224_Components.all;
5
6  entity gunman is
7          port(x: in std_ulogic_vector(4 downto 0);
8                  reset, clk: in std_ulogic;
9                  s: out std_ulogic);
10 end entity;
11
12 architecture gun_detector of gunman is
13         signal q, nq, qb: std_ulogic_vector(1 downto 0);
14         signal n0, n1, n2, n3, n4, g, u, n, gb, nb, ub, gs,
15         gus, phis, g11, g12, g01, g02: std_ulogic;
16 --State Description
17 --phi                  0 0
18 --g                  1 0
19 --gu                  0 1
20 begin
```

```
21          INV0: inverter port map (x(0), n0);
22          INV1: inverter port map (x(1), n1);
23          INV2: inverter port map (x(2), n2);
24          INV3: inverter port map (x(3), n3);
25          INV4: inverter port map (x(4), n4);
26
27          A51: andi5 port map (n4, n3, x(2), x(1), x(0), g); -- Identifying G
28          A52: andi5 port map (x(4), n3, x(2), n1, x(0), u); -- Identifying U
29          A53: andi5 port map (n4, x(3), x(2), x(1), n0, n); -- Identifying N
30
31          INV5: inverter port       map (q(0), qb(0));
32          INV6: inverter port map (q(1), qb(1));
33          INV7: inverter port map (g, gb);
34          INV8: inverter port map (u, ub);
35          INV9: inverter port map (n, nb);
36
37
38          -- Declaring the States
39          A1: andi2 port map (qb(0), qb(1), phis);
40          A2: andi2 port map (qb(0), q(1), gus);
41          A3: andi2 port map (qb(1), q(0), gs);
42
43          -- Assigning State nq(0)
44          A4: andi2 port map (phis, g, g11);
45          A5: andi2 port map (gs, ub, g12);
46          O1: ori2 port map (g11, g12, nq(0));
47
48          -- Assigning State nq(1)
49          A6: andi2 port map (gs, u, g01);
50          A7: andi2 port map (gus, nb, g02);
51          O2: ori2 port map (g01, g02, nq(1));
52
53          -- Assigning the Output
54          A8: andi2 port map (gus, n, s);
55
56          -- Adding the dffi's
57          d0: dffi port map (d => nq(0), clk => clk, q => q(0), r => reset);
58          d1: dffi port map (d => nq(1), clk => clk, q => q(1), r => reset);
59
60  end gun_detector;
```

1.3 Knife Hurler

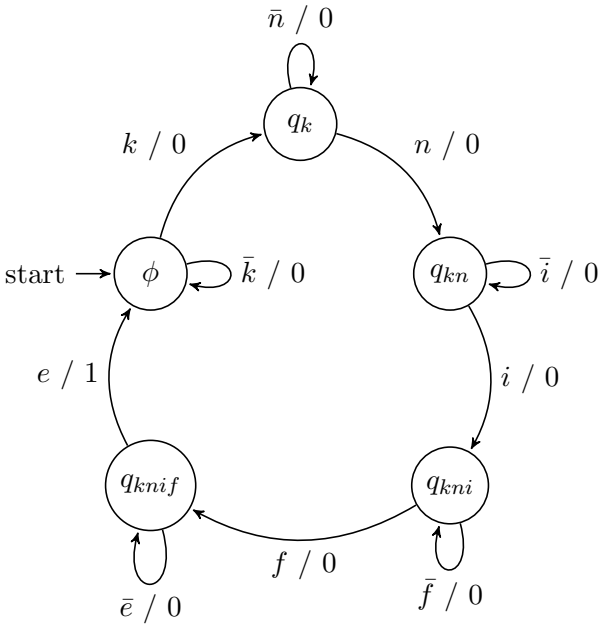

Figure 3: Automata Representation for *Knife Hurler*

Going with the above state representation, the encoding requires 3 bits (5 states), and hence I have again followed *Gray Codes* for the encoding. This is shown below.

State	q_2	q_1	q_0
ϕ	0	0	0
k	0	0	1
kn	0	1	1
kni	0	1	0
$knif$	1	1	0

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

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  library work;
4  use work.EE224_Components.all;
5
6
7  entity knife_hurler is
8          port(x: in std_ulogic_vector(4 downto 0);
9                  reset, clk: in std_ulogic;
10                 s: out std_ulogic);
11 end entity;
12
13 architecture knife_detector of knife_hurler is
14         signal q, nq, qb: std_ulogic_vector(2 downto 0);
15         signal k, n, i, f, e, kb, nb, ib, fb, eb, ks, phis,
```

```
16          kns, knis, knifs, n0, n1, n2, n3, n4,
17          q01, q02, q03, q11, q12, q13, q14, q21, q22: std_ulogic;
18  --State Description
19  --phi                0 0 0
20  --k                 0 0 1
21  --kn                 0 1 1
22  --kni                0 1 0
23  --knif                1 1 0
24
25  begin
26          INV0: inverter port map (x(0), n0);
27          INV1: inverter port map (x(1), n1);
28          INV2: inverter port map (x(2), n2);
29          INV3: inverter port map (x(3), n3);
30          INV4: inverter port map (x(4), n4);
31
32          A5k: andi5 port map (n4, x(3), n2, x(1), x(0), k); -- Identifying K
33          A5n: andi5 port map (n4, x(3), x(2), x(1), n0, n); -- Identifying N
34          A5i: andi5 port map (n4, x(3), n2, n1, x(0), i); -- Identifying I
35          A5f: andi5 port map (n4, n3, x(2), n1, x(0), e); -- Identifying E
36          A5e: andi5 port map (n4, n3, x(2), x(1), n0, f); -- Identifying F
37
38          INV5: inverter port map (q(0), qb(0));
39          INV6: inverter port map (q(1), qb(1));
40          INV7: inverter port map (q(2), qb(2));
41          INV8: inverter port map (k, kb);
42          INV9: inverter port map (n, nb);
43          INV10: inverter port map (i, ib);
44          INV11: inverter port map (f, fb);
45          INV12: inverter port map (e, eb);
46
47          -- Declaring the states
48          A31: andi3 port map (qb(2), qb(1), qb(0), phis);
49          A32: andi3 port map (qb(2), qb(1), q(0), ks);
50          A33: andi3 port map (qb(2), q(1), q(0), kns);
51          A34: andi3 port map (qb(2), q(1), qb(0), knis);
52          A35: andi3 port map (q(2), q(1), qb(0), knifs);
53
54          -- Assigning State nq(2)
55          A1: andi2 port map (knis, f, q21);
56          A2: andi2 port map (knifs, eb, q22);
57          O1: ori2 port map (q21, q22, nq(2));
58
59          -- Assigning State nq(1)
60          O2: ori2 port map (kns, knis, q11);
61          A3: andi2 port map (ks, n, q12);
62          A4: andi2 port map (knifs, eb, q13);
63          O3: ori2 port map (q12, q13, q14);
64          O4: ori2 port map (q11, q14, nq(1));
65
```

```
66          -- Assigning State nq(0)
67          A5: andi2 port map (kns, ib, q01);
68          A6: andi2 port map (phis, k, q02);
69          O5: ori2 port map (q02, ks, q03);
70          O6: ori2 port map (q03, q01, nq(0));
71
72          -- Assigning the Output
73          A7: andi2 port map (knifs, e, s);
74
75          -- Adding the dffi's
76          d0: dffi port map (d => nq(0), clk => clk, q => q(0), r => reset);
77          d1: dffi port map (d => nq(1), clk => clk, q => q(1), r => reset);
78          d2: dffi port map (d => nq(2), clk => clk, q => q(2), r => reset);
79
80  end knife_detector;
```

1.4 Terrorist

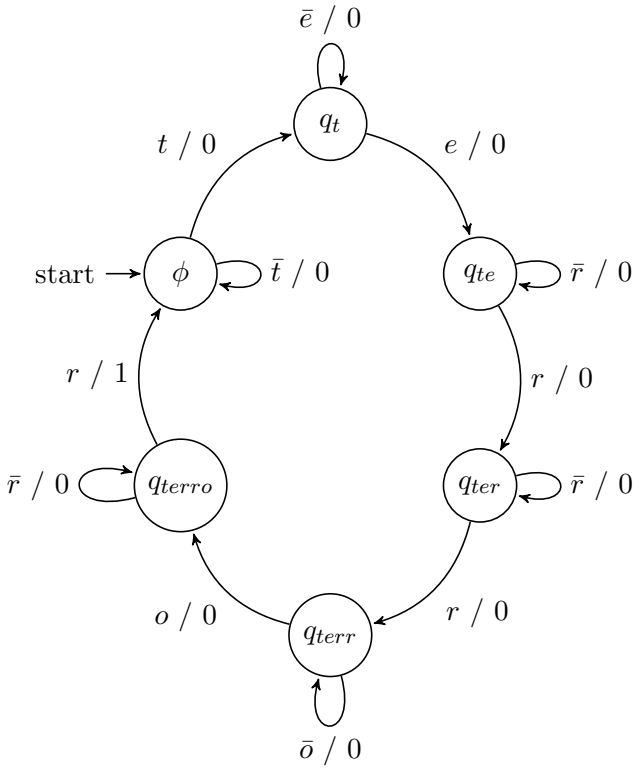

Figure 4: Automata Representation for *Terrorist*

Going with the above state representation, the encoding requires 3 bits (6 states), and hence I have, yet again, followed *Gray-like Codes* for encoding the states.

State	q_2	q_1	q_0
ϕ	0	0	0
t	0	0	1
te	0	1	1
ter	0	1	0
$terr$	1	1	0
$terro$	1	0	0

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;
library work;
use work.EE224_Components.all;


entity terrorist is
        port(x: in std_ulogic_vector(4 downto 0);
                reset, clk: in std_ulogic;
                s: out std_ulogic);
end entity;

architecture terror_detector of terrorist is
        signal q, nq, qb: std_ulogic_vector(2 downto 0);
        signal t, e, r, o, tb, eb, rb, ob, phis, ts, tes, ters,
        terrs, terros, n0, n1, n2, n3, n4, q01, q02, q03, q11,
        q12, q13, q14, q21, q22, q23: std_ulogic;

--State Description
--phi        0 0 0
--t                0 0 1
--te        0 1 1
--ter        0 1 0
--terr        1 1 0
--terro        1 0 0

begin
        INV0: inverter port map (x(0), n0);
        INV1: inverter port map (x(1), n1);
        INV2: inverter port map (x(2), n2);
        INV3: inverter port map (x(3), n3);
        INV4: inverter port map (x(4), n4);

        A5e: andi5 port map (n4, n3, x(2), n1, x(0), e); -- Identifying E
        A5o: andi5 port map (n4, x(3), x(2), x(1), x(0), o); -- Identifying O
        A5t: andi5 port map (x(4), n3, x(2), n1, n0, t); -- Identifying T
        A5r: andi5 port map (x(4), n3, n2, x(1), n0, r);

        INV5: inverter port        map (q(0), qb(0));
        INV6: inverter port        map (q(1), qb(1));
```

```
41          INV7: inverter port map (q(2), qb(2));
42          INV8: inverter port map (t, tb);
43          INV9: inverter port map (e, eb);
44          INV10: inverter port map (r, rb);
45          INV11: inverter port map (o, ob);
46
47          -- Declaring the states
48          A31: andi3 port map (qb(2), qb(1), qb(0), phis);
49          A32: andi3 port map (qb(2), qb(1), q(0), ts);
50          A33: andi3 port map (qb(2), q(1), q(0), tes);
51          A34: andi3 port map (qb(2), q(1), qb(0), ters);
52          A35: andi3 port map (q(2), q(1), qb(0), terrs);
53          A36: andi3 port map (q(2), qb(1), qb(0), terros);
54
55          -- Assigning State nq(2)
56          A1: andi2 port map (ters, r, q21);
57          A2: andi2 port map (terros, rb, q22);
58          O1: ori2 port map (q21, q22, q23);
59          O11: ori2 port map (q23, terrs, nq(2));
60
61
62          -- Assigning State nq(1)
63          O2: ori2 port map (tes, terrs, q11);
64          A3: andi2 port map (ts, e, q12);
65          A4: andi2 port map (terrs, rb, q13);
66          O3: ori2 port map (q12, q13, q14);
67          O4: ori2 port map (q11, q14, nq(1));
68
69          -- Assigning State nq(0)
70          A5: andi2 port map (tes, rb, q01);
71          A6: andi2 port map (phis, t, q02);
72          O5: ori2 port map (q02, ts, q03);
73          O6: ori2 port map (q03, q01, nq(0));
74
75          -- Assigning the Output
76          A7: andi2 port map (terros, r, s);
77
78          -- Adding the dffi's
79          d0: dffi port map (d => nq(0), clk => clk, q => q(0), r => reset);
80          d1: dffi port map (d => nq(1), clk => clk, q => q(1), r => reset);
81          d2: dffi port map (d => nq(2), clk => clk, q => q(2), r => reset);
82
83  end terror_detector;
```

Bringing it all Together

Now that we have 4 independent FSMs that seem to be doing their job right, we must put them all together, in order to obtain the machine as required in the problem description. Without much effort, this can be done by simply taking an OR operation of the 4 individual outputs. This has been done in the `DUT`.

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

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

architecture WRAP of DUT is
component bomber is
  port(x: in std_ulogic_vector(4 downto 0);
        reset, clk: in std_ulogic;
        s: out std_ulogic);
end component;

component gunman is
        port(x: in std_ulogic_vector(4 downto 0);
                reset, clk: in std_ulogic;
                s: out std_ulogic);
end component;

component knife_hurler is
        port(x: in std_ulogic_vector(4 downto 0);
                reset, clk: in std_ulogic;
                s: out std_ulogic);
end component;

component terrorist is
        port(x: in std_ulogic_vector(4 downto 0);
                reset, clk: in std_ulogic;
                s: out std_ulogic);
end component;

  signal res, bs, gs, ks, ts, clk: std_ulogic;
  signal x: std_ulogic_vector(4 downto 0);
begin
        res <= input_vector(6);
        clk <= input_vector(5);

        output_vector(0) <= bs or gs or ks or ts;
        x <= input_vector(4 downto 0);

        B: bomber port map (reset => res, x => x, s => bs, clk => clk);
        G: gunman port map (reset => res, x => x, s => gs, clk => clk);
        K: knife_hurler port map (reset => res, x => x, s => ks, clk => clk);
        T: terrorist port map (reset => res, x => x, s => ts, clk => clk);
end WRAP;
```

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.

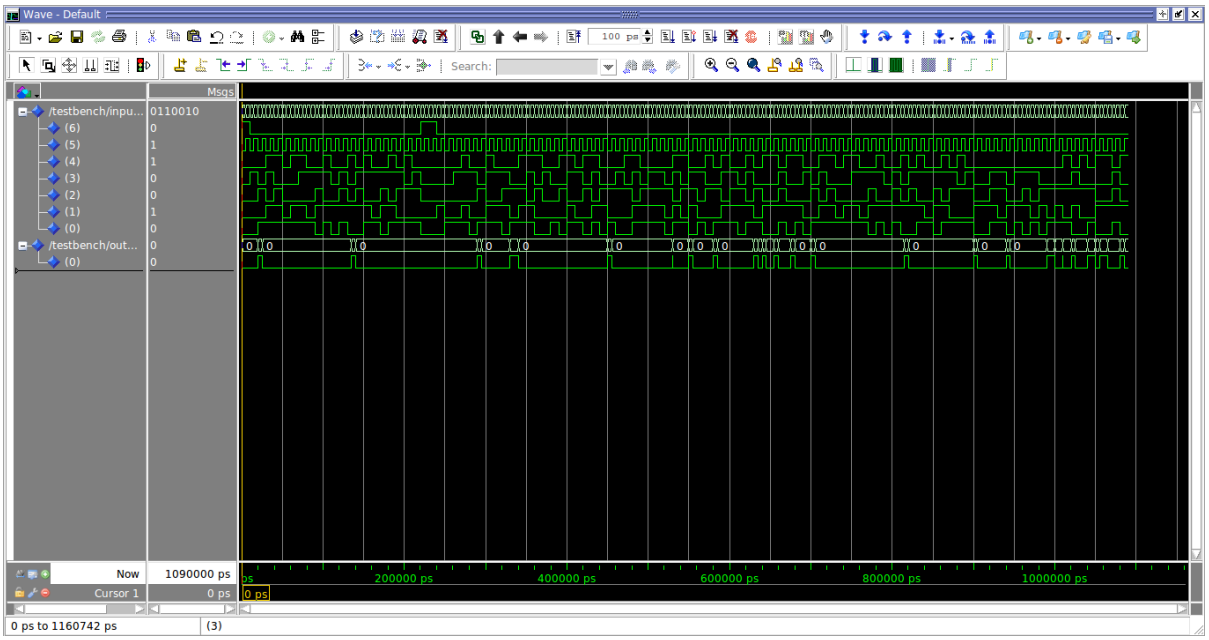

Figure 5: RTL Simulation of the String Detector

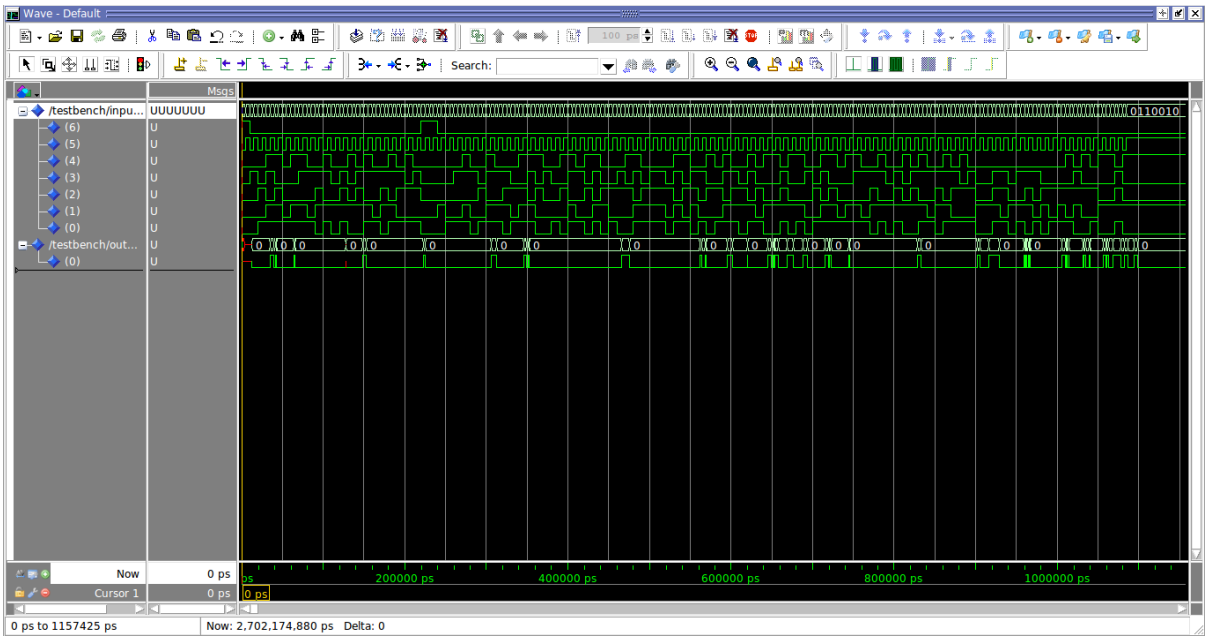

Figure 6: Gate-level Simulation of the String Detector

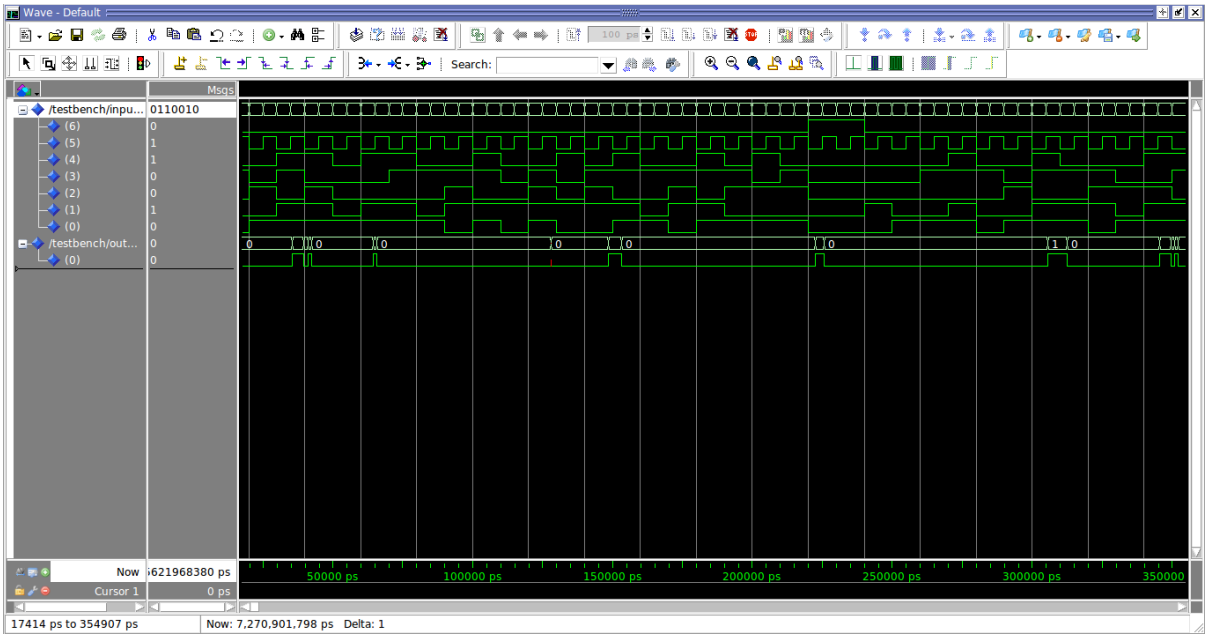

Figure 7: Gate-level Simulation Zoomed-in to a smaller time frame

3 Scan-Chain Tests

We have tested the logic using the RTL simulations, emulated the CPLD performance using the gate-level simulation and uploaded the code on the Krypton board. Next, we need to check that the code is actually running as it is expected to, on the board. We could do so manually but that is not feasible due to the following reasons.

- Our current circuit requires 7 control switches, including a clock. In any given setup, it *may* not be possible to allocate as many I/O pins. As the complexity increases, it will indeed not be possible to allocate so many pins.
- Even if the above is possible, the total number of test cases is *exponential* in the size of the input and it is impractical to perform each of this manually.[1]

Hence, we test the uploaded code on the hardware using the scan-chain setup, as suggested in the manual. This setup was run on a set of two collections of text which has occurrences of the concerned string.

[1]It is not wise to skip any case because, say, we do miss out a failed case it can cascade into unimaginable consequences, which can become difficult to debug.

Results

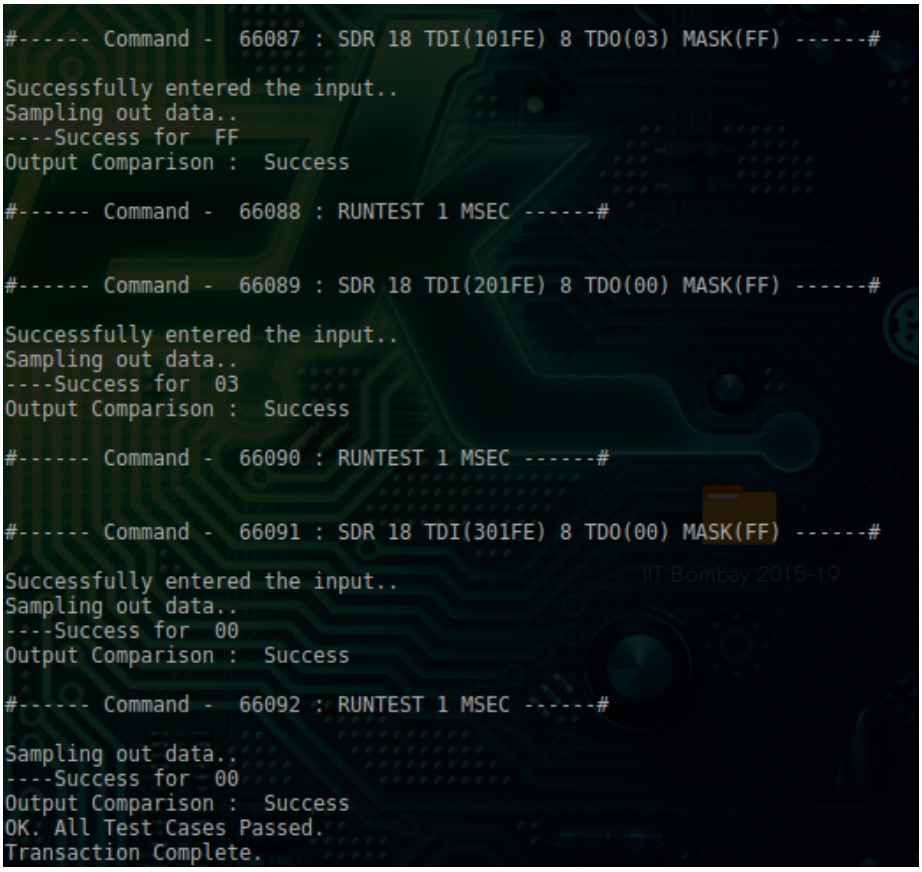

Figure 8: Results of the scan-chain test for random sample test cases.

Conclusion

Starting from the very scratch, in this report, I have presented the logic and code for a sequential implementation of a string recognizer. The logic was tested using RTL simulation, followed by the gate-level simulation for delay analysis and emulating the CPLD. This was followed by an actual rigorous test on the CPLD board after burning the code on it, using the *TIVA-C* microcontroller.

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