Experiment 3
Combinatorial Circuits Using Quartus

Dhruv Ilesh Shah — 150070016

February 3, 2017

Overview

In this lab, I have implemented the following combinatorial circuits on VHDL.

- Two Bit Adder
- Two Bit Subtractor
- Priority Encoder

The code was compiled on Quartus Prime, and simulated using ModelSim. This was then uploaded to the *Krypton v1.1* 5M1270ZT144C5N CPLD-based board.

The codes and setup has been covered in section 1. Since the adder/subtractor have already been implemented by code in the previous labs, I shall skip the code details of those. The setup for the Priority Encoder, however, has been included. Section 2 includes the observations of the simulation and miscellaneous results.

1 Setup

For all the codes, Generic Testbench was used.

1.1 Testing the Krypton Board

The I/O pins on the Krypton board, and its basic functionality, was verified by running simple `svf` files on the board.

1.2 Two Bit Adder

The addition was carried out using $\oplus$ operations. Given input vectors x_1x_0 & y_1y_0, we have the output s_1s_0 given by

$$\begin{aligned} s_0 &= x_0 \oplus y_0 \\ s_1 &= x_1 \oplus y_1 \oplus x_0y_0 \end{aligned} \tag{1}$$

Given below are the DUT definitions and code for the Adder.

```
entity DUT is
   port(input_vector: in bit_vector(3 downto 0);
                output_vector: out bit_vector(1 downto 0));
end entity;
```

```
5
6  architecture DutWrap of DUT is
7     component TwoBitAdder is
8      port(x1,x0,y1,y0: in bit;
9                   s1,s0: out bit);
10    end component;
11 begin
12
13    add_instance: TwoBitAdder
14                          port map (
15                                          x0 => input_vector(0),
16                                          x1 => input_vector(1),
17                                          y0 => input_vector(2),
18                                          y1 => input_vector(3),
19                                          s0 => output_vector(0),
20                                          s1 => output_vector(1));
21
22 end DutWrap;
```

The code for the adder is as follows.

```
1  entity TwoBitAdder is
2          port(x1,x0,y1,y0: in bit;
3                  s1,s0: out bit);
4  end entity;
5
6  architecture Formulas of TwoBitAdder is
7     signal w, z: bit;
8  begin
9     s0 <= (y0 and (not x0)) or ((not y0) and x0); -- s0 <= (y0 xor x0)
10    w  <= (y1 and (not x1)) or ((not y1) and x1); -- w <= (y1 xor x1)
11    z  <= (y0 and x0);
12    s1 <= (w  and (not z)) or ((not w) and z); -- s1 <= (w xor z)
13 end Formulas;
```

1.3 Two Bit Subtractor

The two bit subtractor is a simple 4-input, 2-output combinatorial circuit. After simplification of the terms, the logic can be given as.

$$\begin{aligned} b_0 &= x_0 \oplus y_0 \\ b_1 &= x_1 \oplus y_1 \oplus \overline{x_0} y_0 \end{aligned} \tag{2}$$

As for the implementation, I also 2 intermediate signals $s_1 s_0$ defined as below.

$$\begin{aligned} s_0 &= x_1 \oplus y_1 \\ s_1 &= \neg x_0 \cdot y_0 \end{aligned} \tag{3}$$

And hence, the final output bit definitions look like:

$$\begin{aligned} b_0 &= x_0 \oplus y_0 \\ b_1 &= s_0 \oplus s_1 \end{aligned} \tag{4}$$

This can now be converted to VHDL code and simulated. In this experiment, I have used the `GenericTB` as testbench, which invokes a DUT entity, and hence the top-level entity would also need to be modified. Defining the DUT entity:

```
-- A DUT entity is used to wrap your design.
--  This example shows how you can do this for the
--  two-bit adder.
entity DUT is
   port(input_vector: in bit_vector(3 downto 0);
                output_vector: out bit_vector(1 downto 0));
end entity;

architecture DutWrap of DUT is
   component TwoBitSubtractor is
     port(x1,x0,y1,y0: in bit;
                  b1, b0: out bit);
   end component;
begin

   -- input/output vector element ordering is critical,
   -- and must match the ordering in the trace file!

   -- For this case, the port map has been altered.
   add_instance: TwoBitSubtractor
                        port map (
                                              x1 => input_vector(3),
                                              x0 => input_vector(2),
                                              y1 => input_vector(1),
                                              y0 => input_vector(0),
                                              b1 => output_vector(1),
                                              b0 => output_vector(0));

end DutWrap;
```

Next up, we declare the entity `TwoBitSubtractor` and its functions. This has been shown below.

```
entity TwoBitSubtractor is
        port(x1,x0,y1,y0: in bit;
                b1,b0: out bit);
end entity;
architecture Formulae of TwoBitSubtractor is
        signal s0, s1: bit;
begin
        b0 <= (x0 xor y0);
        s0 <= (x1 xor y1);
        s1 <= ((not x0) and y0);
        b1 <= (s0 xor s1);
end Formulae;
-- For the logic of b1, the bit b1 is unaffected by x0, y0 as long
-- as there is no carry-over in the subtraction. Look below:
```

```
-- x0          y0          diff
-- 0           1           -1
-- 1           1           0
-- 1           0           1
-- 0           0           0
-- Thus, if the case is the first, then we flip  (x1 xor y1), and
-- in other cases, just let it be.
```

1.4 Priority Encoder

The 8x3 priority encoder is a combinatorial circuit that identifies the highest significance bit that is 1. This can be visualised as the truth table below, with *dont-cares*.

Inputs								Outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	Y_2	Y_1	Y_0
1	0	0	0	0	0	0	0	0	0	0
×	1	0	0	0	0	0	0	0	0	1
×	×	1	0	0	0	0	0	0	1	0
×	×	×	1	0	0	0	0	0	1	1
×	×	×	×	1	0	0	0	1	0	0
×	×	×	×	×	1	0	0	1	0	1
×	×	×	×	×	×	1	0	1	1	0
×	×	×	×	×	×	×	1	1	1	1

In addition to the 3 output bits, we also have an extra bit, say N, which is 1 if all the input bits are 0. For an input bit string $x_7x_6x_5x_4x_3x_2x_1x_0$, the output bit string $s_2s_1s_0N$ can be given by the relations

$$
\begin{aligned}
N &= \overline{x_7 + x_6 + x_5 + x_4 + x_3 + x_2 + x_1 + x_0} \\
s_0 &= x_1 \cdot \overline{x_0} + x_3 \cdot \overline{x_2} \cdot \overline{x_1} \cdot \overline{x_0} + x_5 \cdot \overline{x_4} \cdot \overline{x_3} \cdot \overline{x_2} \cdot \overline{x_1} \cdot \overline{x_0} + \\
&\quad x_7 \cdot \overline{x_6} \cdot \overline{x_5} \cdot \overline{x_4} \cdot \overline{x_2 3} \cdot \overline{x_2} \cdot \overline{x_1} \cdot \overline{x_0} \\
s_1 &= x_2 \cdot \overline{x_1} \cdot \overline{x_0} + x_3 \cdot \overline{x_2} \cdot \overline{x_1} \cdot \overline{x_0} + x_6 \cdot \overline{x_5} \cdot \overline{x_4} \cdot \overline{x_2 3} \cdot \overline{x_2} \cdot \overline{x_1} \cdot \overline{x_0} + \\
&\quad x_7 \cdot \overline{x_6} \cdot \overline{x_5} \cdot \overline{x_4} \cdot \overline{x_2 3} \cdot \overline{x_2} \cdot \overline{x_1} \cdot \overline{x_0} \\
s_2 &= x_4 \cdot \overline{x_3} \cdot \overline{x_2} \cdot \overline{x_1} \cdot \overline{x_0} + x_5 \cdot \overline{x_4} \cdot \overline{x_3} \cdot \overline{x_2} \cdot \overline{x_1} \cdot \overline{x_0} + \\
&\quad x_6 \cdot \overline{x_5} \cdot \overline{x_4} \cdot \overline{x_2 3} \cdot \overline{x_2} \cdot \overline{x_1} \cdot \overline{x_0} + + x_7 \cdot \overline{x_6} \cdot \overline{x_5} \cdot \overline{x_4} \cdot \overline{x_2 3} \cdot \overline{x_2} \cdot \overline{x_1} \cdot \overline{x_0}
\end{aligned}
\tag{5}
$$

The VHDL implementation of this circuit has been given below. First, we define the container entity `DUT` as follow.

```
entity DUT is
   port(input_vector: in bit_vector(7 downto 0);
                output_vector: out bit_vector(3 downto 0));
end entity;
```

```
5
6  architecture DutWrap of DUT is
7      component PriorityEncoder is
8      port ( x7 , x6 , x5 , x4 , x3 , x2 , x1 , x0 :in bit ;
9      s2 , s1 , s0 , N :out bit ) ;
10    end component;
11 begin
12    add_instance: PriorityEncoder
13                          port map (
14           x7 => input_vector(7),
15           x6 => input_vector(6),
16           x5 => input_vector(5),
17           x4 => input_vector(4),
18           x3 => input_vector(3),
19           x2 => input_vector(2),
20           x1 => input_vector(1),
21           x0 => input_vector(0),
22           s0 => output_vector(1),
23           s1 => output_vector(2),
24           s2 => output_vector(3),
25           N => output_vector(0));
26
27 end DutWrap;
```

Next up, we have the main segment of the code, as given in the file `PriorityEncoder.vhd`

```
1  library ieee ;
2  use ieee.std_logic_1164.all ;
3
4
5  entity PriorityEncoder is
6          port ( x7 , x6 , x5 , x4 , x3 , x2 , x1 , x0 :in bit ;
7          s2 , s1 , s0 , N :out bit ) ;
8  end PriorityEncoder;
9
10 architecture comb of PriorityEncoder is
11 begin
12         N <= not( x7 or x6 or x5 or x4 or x3 or x2 or x1 or x0 ) ;
13
14         s0 <= ( x1 and not x0 ) or
15                 (x3 and not x2 and not x1 and not x0) or
16                 ( x5 and not x4 and not x3 and not x2 and
17                 not x1 and not x0 ) or
18                 ( x7 and not x6 and not x5 and not x4
19                 and not x3 and not x2 and not x1
20                 and not x0 ) ;
21         s1 <= ( x2 and not x1 and not x0 ) or
22                 ( x3 and not x2 and not x1 and not x0 ) or
23                 ( x6 and not x5 and not x4 and not x3 and
24                 not x2 and not x1 and not x0 ) or
25                 ( x7 and not x6 and not x5 and not x4 and
```

```
                not x3 and not x2 and not x1 and not x0 ) ;
        s2 <= ( x4 and not x3 and not x2 and
                not x1 and not x0 ) or
                ( x5 and not x4 and not x3 and not x2 and
                not x1 and not x0 ) or
                ( x6 and not x5 and not x4 and not x3
                and not x2 and not x1 and not x0 ) or
                ( x7 and not x6 and not x5 and not x4 and not x3
                and not x2 and not x1 and not x0 ) ;
end comb ;
```

Another essential part of the implementation is the testbench. Instead of using a custom testbench for the process, I have modified and used the `Generic Testbench` for an 8x4 implementation. Below is the code for the testbench. Main changes were made in lines *9 - 18*.

```
1  library std;
2  use std.textio.all;
3
4  entity Testbench is
5  end entity;
6  architecture Behave of Testbench is
7
8    ----------------------------------------------------------------
9    ----------------------------------------------------------------
10   constant number_of_inputs  : integer := 8;  -- # input bits to your design.
11   constant number_of_outputs : integer := 4;  -- # output bits from your design.
12
13   -- component port widths..
14   component DUT is
15    port(input_vector: in bit_vector(7 downto 0);
16                 output_vector: out bit_vector(3 downto 0));
17   end component;
18
19   ----------------------------------------------------------------
20   ----------------------------------------------------------------
21
22   signal input_vector  : bit_vector(number_of_inputs-1 downto 0);
23   signal output_vector : bit_vector(number_of_outputs-1 downto 0);
24
25   -- create a constrained string outof
26   function to_string(x: string) return string is
27       variable ret_val: string(1 to x'length);
28       alias lx : string (1 to x'length) is x;
29   begin
30       ret_val := lx;
31       return(ret_val);
32   end to_string;
33
34 begin
35   process
36     variable err_flag : boolean := false;
```

```
37      File INFILE: text open read_mode is "<path>/PriorityEncoder/TRACEFILE.txt";
38      FILE OUTFILE: text  open write_mode is "<path>/PriorityEncoder/OUTPUTS.txt";
39
40      ------------------------------------------------
41      ------------------------------------------------
42      variable input_vector_var: bit_vector (number_of_inputs-1 downto 0);
43      variable output_vector_var: bit_vector (number_of_outputs-1 downto 0);
44      variable output_mask_var: bit_vector (number_of_outputs-1 downto 0);
45      variable output_comp_var: bit_vector (number_of_outputs-1 downto 0);
46      constant ZZZZ : bit_vector(number_of_outputs-1 downto 0) := (others => '0');
47      ------------------------------------------------
48
49      variable INPUT_LINE: Line;
50      variable OUTPUT_LINE: Line;
51      variable LINE_COUNT: integer := 0;
52
53
54    begin
55      while not endfile(INFILE) loop
56            -- will read a new line every 5ns, apply input,
57            -- wait for 1 ns for circuit to settle.
58            -- read output.
59
60
61            LINE_COUNT := LINE_COUNT + 1;
62
63
64            -- read input at current time.
65            readLine (INFILE, INPUT_LINE);
66            read (INPUT_LINE, input_vector_var);
67            read (INPUT_LINE, output_vector_var);
68            read (INPUT_LINE, output_mask_var);
69
70            -- apply input.
71            input_vector <= input_vector_var;
72
73            -- wait for the circuit to settle
74            wait for 1 ns;
75
76            -- check output.
77            output_comp_var := (output_mask_var and (output_vector xor output_vector_var));
78            if (output_comp_var  /= ZZZZ) then
79               write(OUTPUT_LINE,to_string("ERROR: line "));
80               write(OUTPUT_LINE, LINE_COUNT);
81               writeline(OUTFILE, OUTPUT_LINE);
82               err_flag := true;
83            end if;
84
85            write(OUTPUT_LINE, input_vector);
86            write(OUTPUT_LINE, to_string(" "));
```

```
87              write(OUTPUT_LINE, output_vector);
88              writeline(OUTFILE, OUTPUT_LINE);
89
90              -- advance time by 4 ns.
91              wait for 4 ns;
92      end loop;
93
94      assert (err_flag) report "SUCCESS, all tests passed." severity note;
95      assert (not err_flag) report "FAILURE, some tests failed." severity error;
96
97      wait;
98    end process;
99
100   dut_instance: DUT
101               port map(input_vector => input_vector, output_vector => output_vector);
102
103 end Behave;
```

Another important step in the evaluation of the simulation is the `TRACEFILE`. For this, I used the Python script given below.

```
1  def tobinary(n, width):
2      s = ''
3      for i in range(width):
4          s = s + str(n % 2)
5          n = n / 2
6          s = s[:: -1]
7      return s
8
9  f = open('TRACEFILE.txt', 'w')
10 f. seek(0)
11 f. truncate()
12 for i in range(256):
13     x = tobinary(i, 8)
14     s = '000'
15     if x == '00000000':
16         N = '1'
17     else:
18         N = '0'
19
20     if x[7] == '1':
21         s = '000'
22     elif x[6] == '1':
23         s = '001'
24     elif x[5] == '1':
25         s = '010'
26     elif x[4] == '1':
27         s = '011'
28     elif x[3] == '1':
29         s = '100'
30     elif x[2] == '1':
```

```
31          s = '101'
32      elif x[1] == '1':
33          s = '110'
34      elif x[0] == '1':
35          s = '111'
36      f. write(x + " " + s + N +" " + "1111" +  "\n")
37  f. close()
```

This marks the end of the setup etc. We will now run RTL and Gate-level simulations on these codes, before deploying it on hardware.

2 Observations

In this section, I shall display the results of the running RTL and Gate-level simulations on the above implementations. Screenshots have been attached for each part.

2.1 Two Bit Adder

Figures 1-3 show the results of the simulation.

```
# ** Note: SUCCESS, all tests passed.
#    Time: 80 ns  Iteration: 0  Instance: /testbench
```

Figure 1: Evaluation of the 2^4 test cases for the Two Bit Adder.

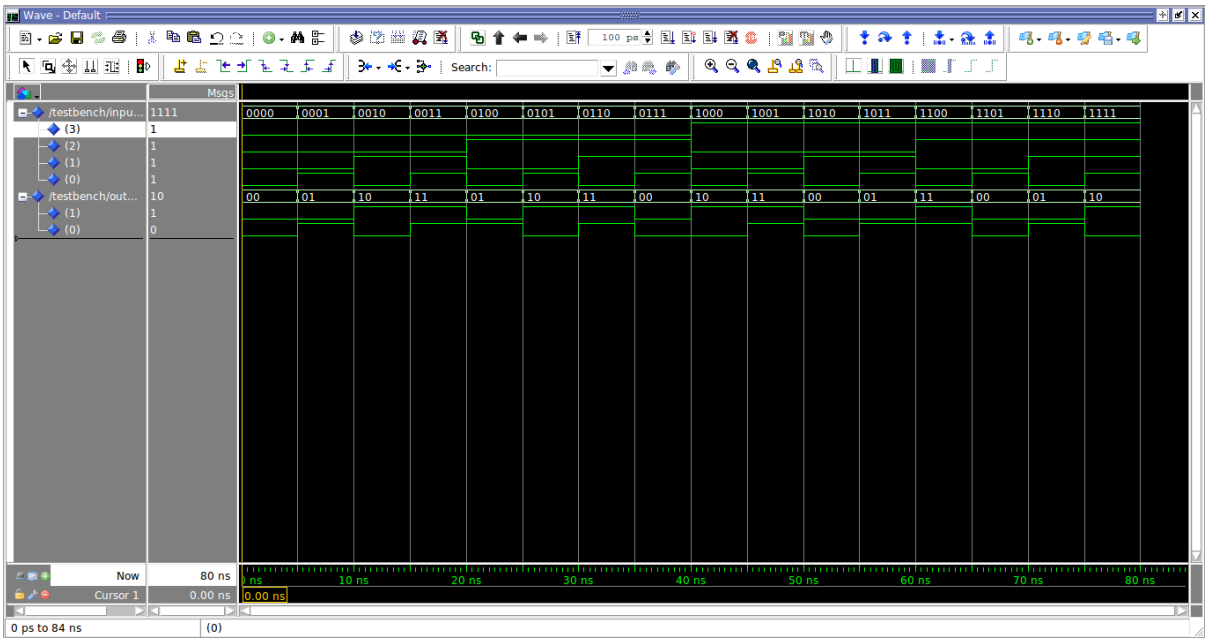

Figure 2: RTL Simulation of the Two Bit Adder

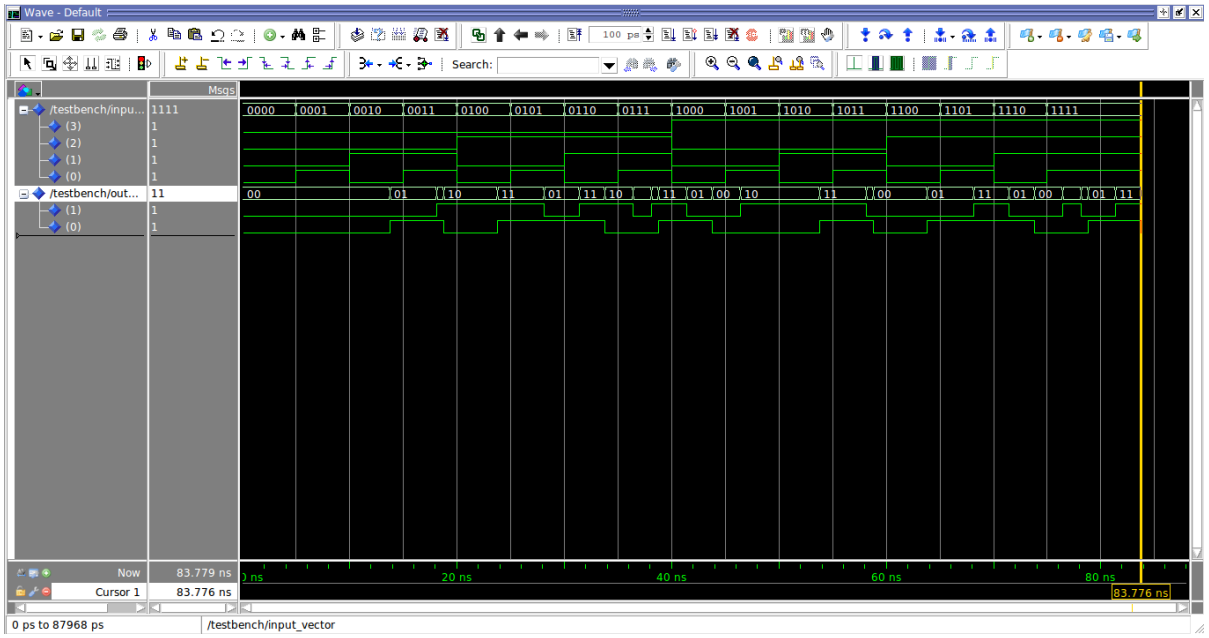

Figure 3: Gate-level Simulation of the Two Bit Adder

2.2 Two Bit Subtractor

Figures 4-6 show the results of the simulation.

```
# ** Note: SUCCESS, all tests passed.
#    Time: 80 ns  Iteration: 0  Instance: /testbench
```

Figure 4: Evaluation of the 2^4 test cases for the Two Bit Subtractor.

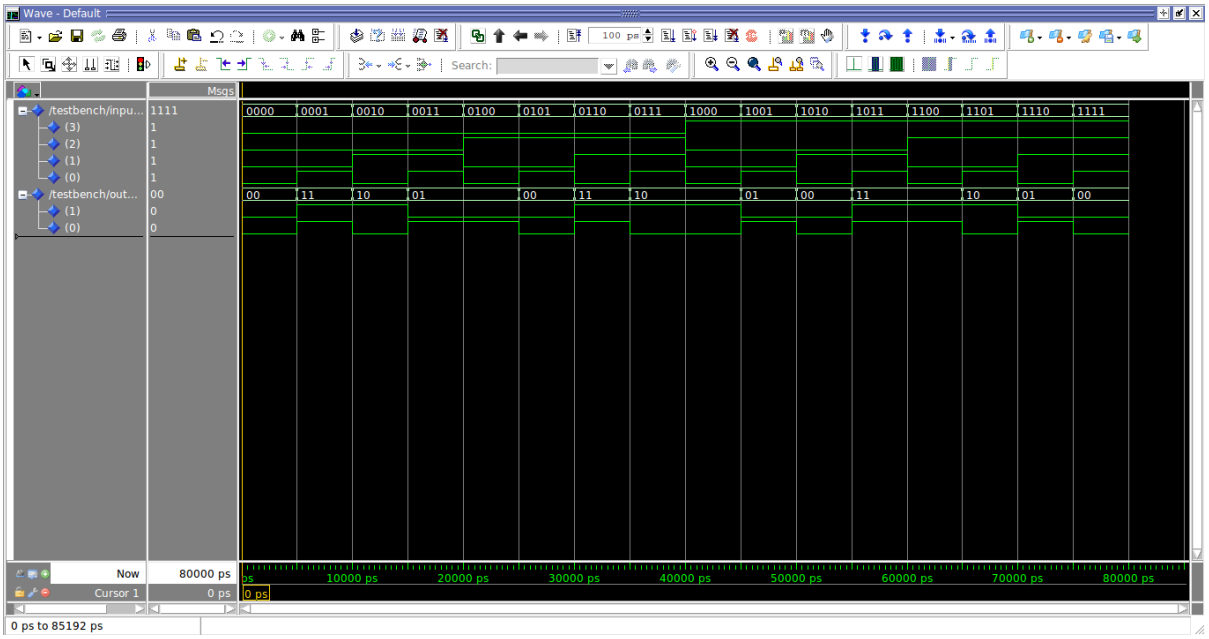

Figure 5: RTL Simulation of the Two Bit Subtractor

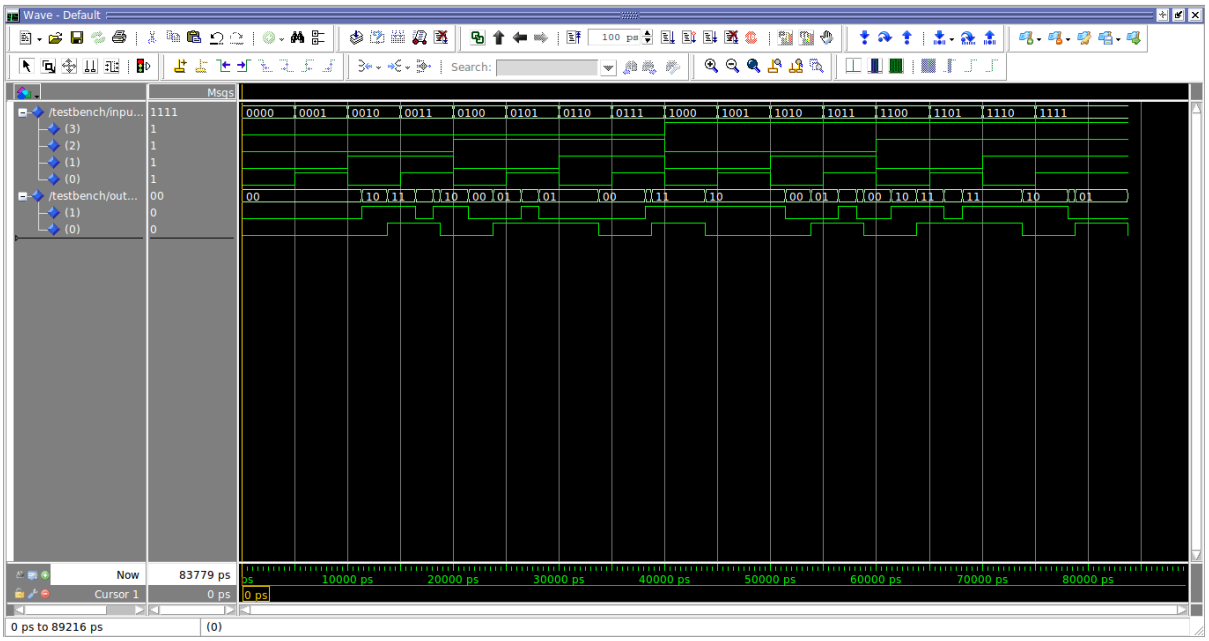

Figure 6: Gate-level Simulation of the Two Bit Subtractor

2.3 Priority Encoder

Figures 7-10 show the results of the simulation

```
# ** Note: SUCCESS, all tests passed.
#    Time: 1280 ns  Iteration: 0  Instance: /testbench
```

Figure 7: Evaluation of the 2^8 test cases for the Priority Encoder.

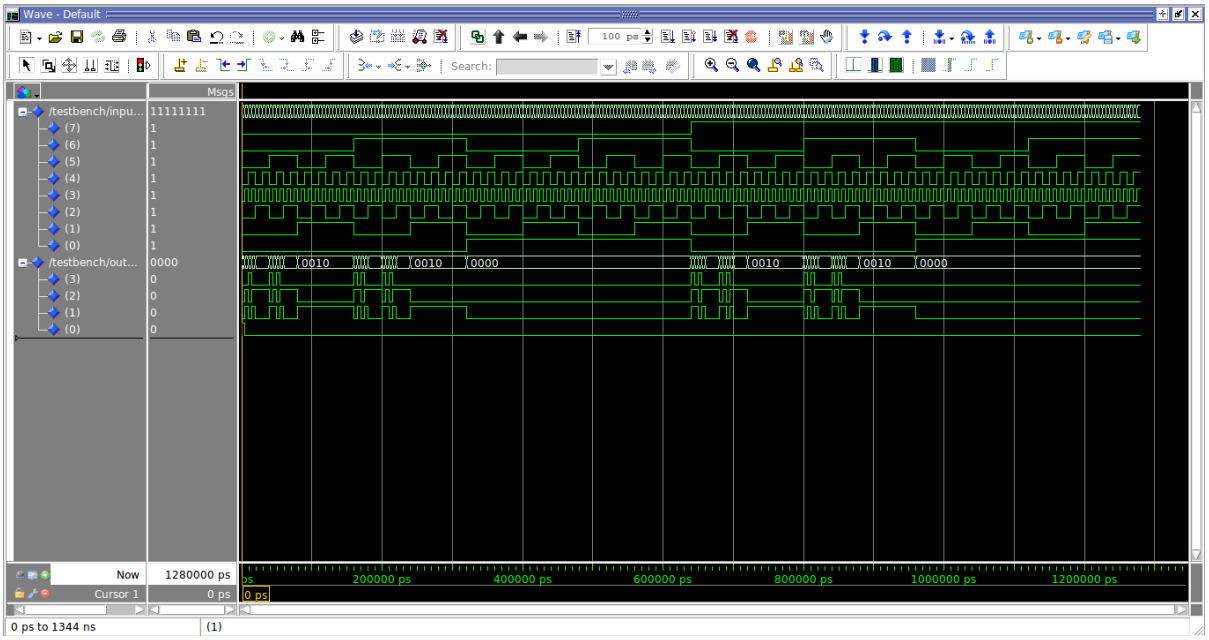

Figure 8: RTL Simulation of the Priority Encoder

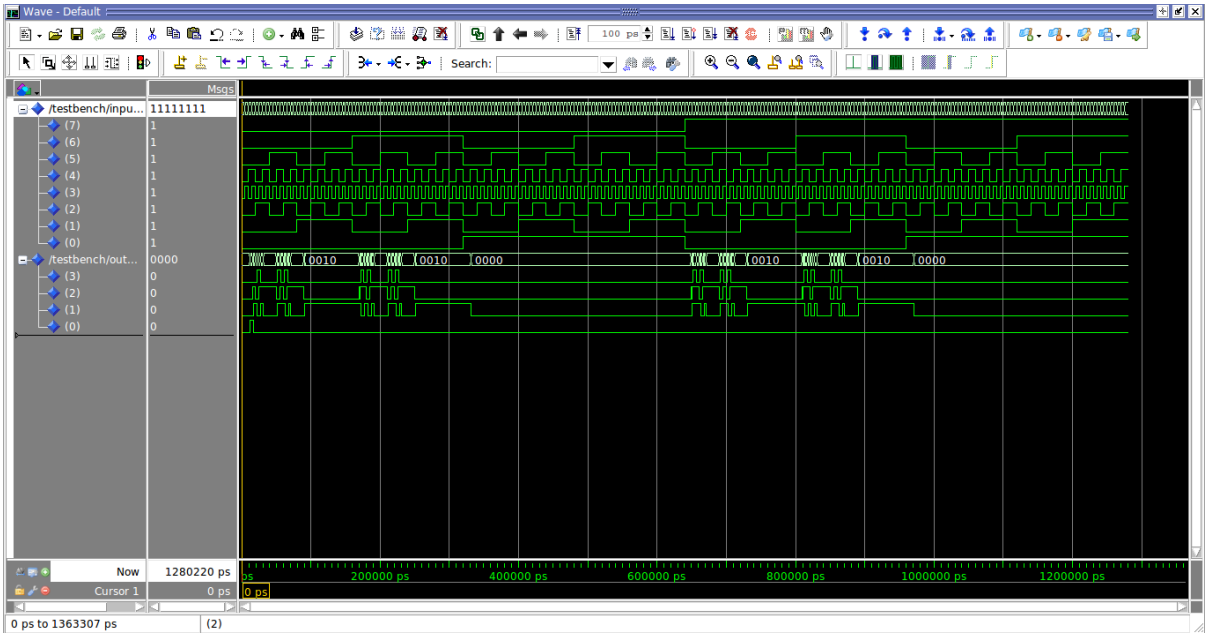

Figure 9: Gate-level Simulation of the Priority Encoder

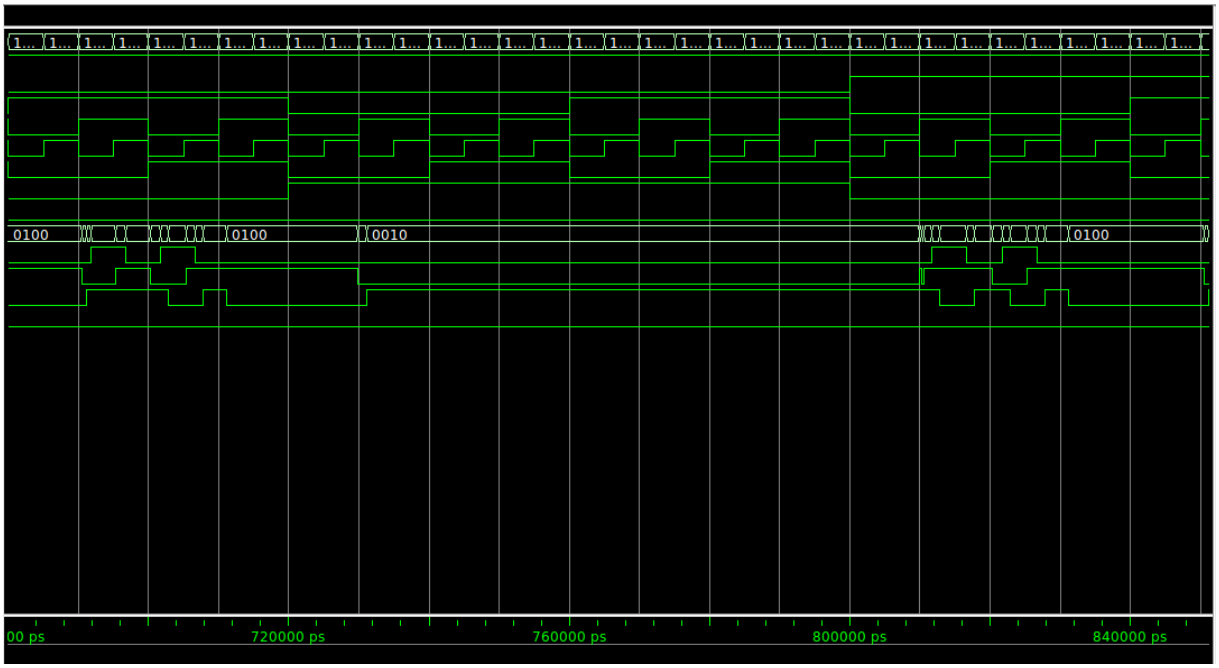

Figure 10: Gate-level Simulation of the Priority Encoder, 760ns - 850ns

Conclusion

We observe that the designs work well in the RTL simulations, but not in Gate level simulations. The gate level simulation emulated the gate delays, as they would exist in real hardware, and hence demonstrate a crucial bottleneck of the device. For ideal performance, the time between states should be increased so that the transient outputs can settle. Once this is done, the implementation can be uploaded on the CPLD for required purposes.