Experiment 2
Combinatorial Circuits Using VHDL
Two Bit Subtractor

Dhruv Ilesh Shah — 150070016

January 20, 2017

Overview

In this lab, I have implemented a combinatorial circuit - the two bit subtractor, using VHDL. Given two input strings of two bits each (say, x_1x_0 & y_1y_0), the device computes the difference in modulo 2 sense.

Section 1 contains the approach and main code used for the simulation. Section 2 contains the observations and output of the simulation. For the simulation, I have used GHDL, which is an open-source simulator for the VHDL language.

1 Setup

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

$$\begin{aligned} b_0 &= x_0 \oplus y_0 \\ b_1 &= x_1 \oplus y_1 \oplus \overline{x_0}y_0 \end{aligned} \tag{1}$$

As for the implementation, I also 2 intermediate signals s_1s_0 defined as below.

$$\begin{aligned} s_0 &= x_1 \oplus y_1 \\ s_1 &= \neg x_0 \cdot y_0 \end{aligned} \tag{2}$$

And hence, the final output bit definitions look like:

$$\begin{aligned} b_0 &= x_0 \oplus y_0 \\ b_1 &= s_0 \oplus s_1 \end{aligned} \tag{3}$$

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:

```
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
   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.
```

The generic testbench used is also given below.

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

entity Testbench is
end entity;
architecture Behave of Testbench is

```

```
8     ------------------------------------------------------------------
9     constant number_of_inputs  : integer := 4;  -- # input bits to your design.
10    constant number_of_outputs : integer := 2;  -- # output bits from your design.
11
12    -- component port widths..
13    component DUT is
14     port(input_vector: in bit_vector(3 downto 0);
15                  output_vector: out bit_vector(1 downto 0));
16    end component;
17
18    ------------------------------------------------------------------
19    ------------------------------------------------------------------
20
21    signal input_vector  : bit_vector(number_of_inputs-1 downto 0);
22    signal output_vector : bit_vector(number_of_outputs-1 downto 0);
23
24    -- create a constrained string outof
25    function to_string(x: string) return string is
26        variable ret_val: string(1 to x'length);
27        alias lx : string (1 to x'length) is x;
28    begin
29        ret_val := lx;
30        return(ret_val);
31    end to_string;
32
33  begin
34    process
35      variable err_flag : boolean := false;
36      File INFILE: text open read_mode is "/home/dhruv-shah/Desktop/IIT Bombay 2015-19/Sem 4/
37      FILE OUTFILE: text  open write_mode is "/home/dhruv-shah/Desktop/IIT Bombay 2015-19/Sem
38
39      ------------------------------------------------
40      ------------------------------------------------
41      variable input_vector_var: bit_vector (number_of_inputs-1 downto 0);
42      variable output_vector_var: bit_vector (number_of_outputs-1 downto 0);
43      variable output_mask_var: bit_vector (number_of_outputs-1 downto 0);
44      variable output_comp_var: bit_vector (number_of_outputs-1 downto 0);
45      constant ZZZZ : bit_vector(number_of_outputs-1 downto 0) := (others => '0');
46      ------------------------------------------------
47
48      variable INPUT_LINE: Line;
49      variable OUTPUT_LINE: Line;
50      variable LINE_COUNT: integer := 0;
51
52
53    begin
54      while not endfile(INFILE) loop
55            -- will read a new line every 5ns, apply input,
56            -- wait for 1 ns for circuit to settle.
57            -- read output.
```

```
58
59
60            LINE_COUNT := LINE_COUNT + 1;
61
62
63            -- read input at current time.
64            readLine (INFILE, INPUT_LINE);
65            read (INPUT_LINE, input_vector_var);
66            read (INPUT_LINE, output_vector_var);
67            read (INPUT_LINE, output_mask_var);
68
69            -- apply input.
70            input_vector <= input_vector_var;
71
72            -- wait for the circuit to settle
73            wait for 1 ns;
74
75            -- check output.
76            output_comp_var := (output_mask_var and (output_vector xor output_vector_var));
77            if (output_comp_var  /= ZZZZ) then
78               write(OUTPUT_LINE,to_string("ERROR: line "));
79               write(OUTPUT_LINE, LINE_COUNT);
80               writeline(OUTFILE, OUTPUT_LINE);
81               err_flag := true;
82            end if;
83
84            write(OUTPUT_LINE, input_vector);
85            write(OUTPUT_LINE, to_string(" "));
86            write(OUTPUT_LINE, output_vector);
87            writeline(OUTFILE, OUTPUT_LINE);
88            -- advance time by 4 ns.
89            wait for 4 ns;
90      end loop;
91
92      assert (err_flag) report "SUCCESS, all tests passed." severity note;
93      assert (not err_flag) report "FAILURE, some tests failed." severity error;
94
95      wait;
96    end process;
97
98    dut_instance: DUT
99                port map(input_vector => input_vector, output_vector => output_vector);
100
101 end Behave;
```

For validating, we would need a list of expected values for all the possible input combinations. This can be found as the `TRACEFILE.txt` below.

$x_1x_0y_1y_0$	b_1b_0	E
0000	00	11
0001	11	11
0010	10	11
0011	01	11
0100	01	11
0101	00	11
0110	11	11
0111	10	11
1000	10	11
1001	01	11
1010	00	11
1011	11	11
1100	11	11
1101	10	11
1110	01	11
1111	00	11

Table 1: Tracefile for the Two Bit Subtractor

2 Observations

On compiling and executing the `Testbench`, the circuit was validated with all test cases successful. A snapshot of the same is given below (Figure 1). Test cases and the obtained outputs (ignoring gate delays) can be observed on `GTKWave` (Figure 2).

The output file generated by the testbench is given below.

```
0000 00
ERROR: line 2
0001 00
ERROR: line 3
0010 00
ERROR: line 4
0011 11
ERROR: line 5
0100 10
ERROR: line 6
0101 01
ERROR: line 7
0110 01
ERROR: line 8
0111 00
ERROR: line 9
1000 11
ERROR: line 10
1001 10
ERROR: line 11
1010 10
ERROR: line 12
1011 01
```

```
PrieureDeSion @ {TwoBitSubtractor} slayin' $ ./testbench --stop-time=100ns --vcd=subtractor.vcd
Testbench.vhd:98:5:@80ns:(assertion note): SUCCESS, all tests passed.
```

Figure 1: Evaluating Testbench

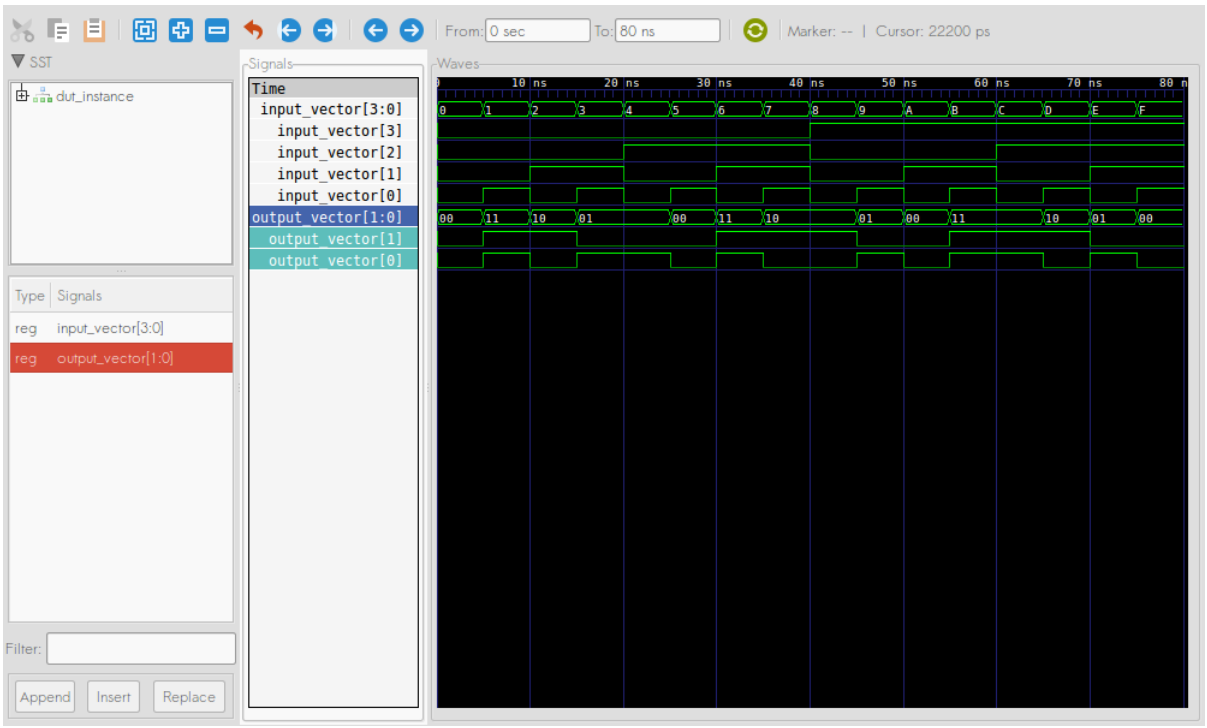

Figure 2: Visualising the waveforms on `GTKWave`

```
ERROR: line 13
1100 00
ERROR: line 14
1101 11
ERROR: line 15
1110 11
ERROR: line 16
1111 10
```

Conclusion

Simple combinatorial logic can be implemented and verified very easily by simulating.