Experiment 2
Eight-Bit ALU Design

Amit Kumar,16D070034

March 5, 2018

1 Overview of the experiment

ALU (Arithmetic Logic Unit) is a importaant unit which can perform simple arithmetic and logical operations. In this experiment I have designed a 8-bit ALU which performs the following task :

- Addition
- Substraction
- Logical Right-Shift
- Logical Left-Shift

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

2 Experiment setup & approach

We implemented the above operations step by step starting from basic logic gates which is mentioned in particular sections later.

We have to do these operations on two eight numbers. So there are two 8-bit inputs X and Y, and it produces an 8-bit output Z. The operation to be

performed is selected by a 2-bit operation code (opcode). The functionality of the ALU based on the op code bits is shown in Table 1.

Op Code(opcode)	Operation	result
00	Addition	Z = X + Y
01	Substraction	Z = X - Y
10	Logical Right-Shift	Z = X >> Y
11	Logical Left-Shift	Z = X << Y

Table 1: Op-codes for the ALU

2.1 Addition

For Addition, I designed **half adder** which takes two bits as input and gives their sum bit and carry bit .
Logic :
$s_o = a \oplus b, c_o = a.b$

Half Adder

```
1  library std;
2  use std.standard.all;
3
4  library ieee;
5  use ieee.std_logic_1164.all;
6  ------------------------------------------------------------------------------------------------
7  package EE224_Components is
8
9          component AND_2 is
10         port (a, b: in std_logic; c : out std_logic);
11    end component;
12
13    component XOR_2 is
14         port (a, b: in std_logic; c : out std_logic);
15    end component;
16
17 end EE224_Components;
18 ------------------------------------------------------------------------------------------------
19
20
21 library ieee;
22 use ieee.std_logic_1164.all;
```

```
23 entity AND_2 is
24   port (a, b: in std_logic;
25         c: out std_logic);
26 end entity AND_2;
27 architecture Behave of AND_2 is
28 begin
29   c <= a and b;
30 end Behave;
31
32 library ieee;
33 use ieee.std_logic_1164.all;
34 entity XOR_2 is
35   port (a, b: in std_logic;
36         c: out std_logic);
37 end entity XOR_2;
38 architecture Behave of XOR_2 is
39 begin
40   c <= (a xor b);
41 end Behave;
42
43
44 library ieee;
45 -- std_logic type and associated functions.
46 use ieee.std_logic_1164.all;
47
48 library work;
49 -- package of component declarations..
50 use work.EE224_Components.all;
51
52 entity HalfBitAdder is
53    port(x0,y0: in std_logic;
54         s0,c0: out std_logic);
55 end entity;
56 architecture Struct of HalfBitAdder is
57   -- signal w, z: std_logic;
58 begin
59    output: XOR_2 port map (a => x0, b => y0, c => s0);
60
61    carry: AND_2 port map (a => x0, b => y0, c => c0);
62
63 end Struct;
```

I used two half adders to make a **full adder** which takes three inputs (two bits to be added and a carry$_{in}$ bit and gives their sum and carry bit. Logic :

$$s_o = a \oplus b \oplus c_{in}, c_o = ab + bc_{in} + ac_{in}$$

Full Adder

```
library std;
use std.standard.all;

library ieee;
use ieee.std_logic_1164.all;

entity FullBitAdder is
  port (a, b, cin: in std_logic;
          sum, cout: out std_logic);
end entity FullBitAdder;

architecture Struct of FullBitAdder is

component HalfBitAdder is
   port(x0,y0: in std_logic;
        s0,c0: out std_logic);
end component;
signal o1, c1, c2: std_logic;
begin
  HA1: HalfBitAdder port map ( x0 => a, y0 => b, s0 => o1 , c0 => c1) ;
  HA2: HalfBitAdder port map ( x0 => o1, y0 => cin, s0 => sum , c0 => c2) ;

cout <= c1 or c2 ;
end Struct;
```

Finally I used a series of full adders to implement add eight bits and outputs their sum.

Eight Bit Adder

```
library std;
use std.standard.all;

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

```
6
7  entity EightBitAdder is
8    port (x,y: in std_logic_vector(7 downto 0) ;
9          sum: out std_logic_vector(7 downto 0) );
10 end entity EightBitAdder;
11
12 architecture Struct of EightBitAdder is
13
14 component FullBitAdder is
15   port (a, b, cin: in std_logic;
16         sum, cout: out std_logic);
17 end component FullBitAdder;
18
19 signal co: std_logic_vector(6 downto 0);
20 signal cot: std_logic ;
21 begin
22 FA1: FullBitAdder port map (a =>x(0) , b =>y(0) , cin =>'0' ,sum =>sum(0) ,cout => co(0));
23 g1: for i in 0 to 5 generate
24 begin
25   FA:  FullBitAdder port map (a =>x(i+1) , b =>y(i+1) , cin => co(i) ,sum => sum(i+1),
26 cout => co(i+1) );
27 end generate g1;
28 FA2: FullBitAdder port map (a =>x(7) , b =>y(7) , cin => co(6) ,sum =>sum(7) ,cout => cot);
29 end Struct;
```

2.2 Substractor

For the Substractor, I have used the 2's complement approach which is done by using the previous eight bit adder with an additional bit '1' to cin of the first full adder.
Let a' be 2's complement of a, so
$b + a' = b + (2^n - 1 - a) = (b - a) + 2^n - 1$
Here additional one is added to care of -1 and 2^n is taken into account by taking n-bits only as it will appear at (n+1)th bit.

Substractor

```
1  library std;
2  use std.standard.all;
3
4  library ieee;
5  use ieee.std_logic_1164.all;
```

```
6
7  entity Substractor is
8    port (x,y: in std_logic_vector(7 downto 0) ;
9          sum: out std_logic_vector(7 downto 0) );
10 end entity Substractor;
11
12 architecture Struct of Substractor is
13
14 component FullBitAdder is
15   port (a, b, cin: in std_logic;
16         sum, cout: out std_logic);
17 end component FullBitAdder;
18
19 signal co: std_logic_vector(6 downto 0);
20 signal inv: std_logic_vector(7 downto 0);
21 signal cot: std_logic;
22 begin
23 inv <= (not y) ;
24 FA1: FullBitAdder port map (a =>x(0) , b =>(inv(0)) , cin =>'1' ,sum =>sum(0) ,cout => co(0));
25 g1: for i in 0 to 5 generate
26 begin
27   FA: FullBitAdder port map (a =>x(i+1), b =>(inv(i+1)), cin => co(i), sum => sum(i+1),
28 cout => co(i+1));
29 end generate g1;
30 FA2: FullBitAdder port map (a =>x(7) , b =>(inv(7)) , cin => co(6) ,sum =>sum(7) ,cout => cot);
31
32 end Struct;
```

2.3 Logical Right Shift

For this part, I perform shifts using logarithmic barrel shifting. An illustration of barrel-shifting is given in figure 1. Note that although both X, Y are 8-bit, if Y is more than 111, then the output would be monotonously zero. So, we need to implement only 3 stages.

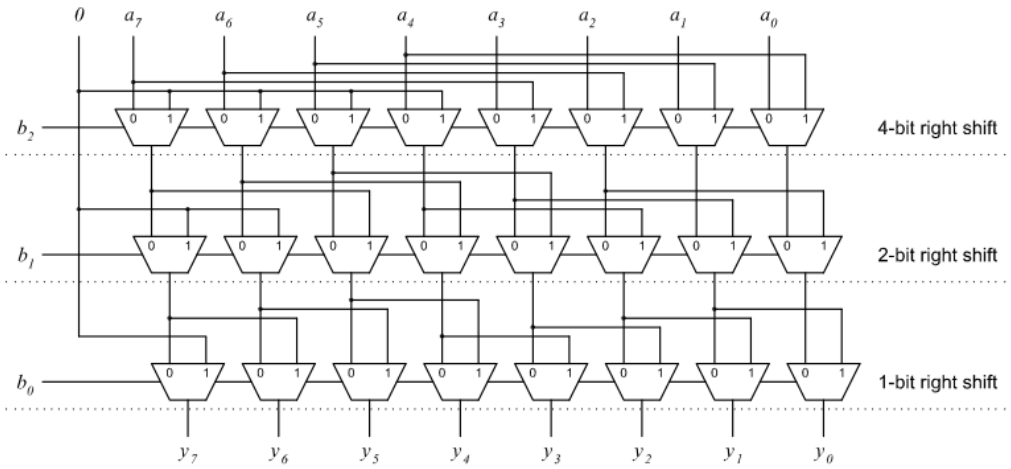

Figure 1: Logarithmic barrel shifter Logic for Right Shifter

So, I created a mux entity, which is then used to make the 8-mux chains, eventually making our shifter.
logic :

$$o = a * \bar{s} + b * s$$

MUX

```
library std;
use std.standard.all;

library ieee;
use ieee.std_logic_1164.all;

entity mux is
  port (a, b, s : in std_logic;
        o : out std_logic);
end entity mux ;

architecture Struct of mux is

--signal o1, c1, c2: std_logic;
```

```
begin
  o <= (a and (not s) ) or (s and b) ;

end Struct;
```

Then I implented 8-mux chains to shift 1 bit, 2 bits and bits respectively.

MUX chain for 1-bit shifting

```
library std;
use std.standard.all;

library ieee;
use ieee.std_logic_1164.all;

entity mux_chain_1bit is
  port (x: in std_logic_vector(7 downto 0);
        s1: in std_logic;
        o : out std_logic_vector(7 downto 0) );
end entity mux_chain_1bit ;

architecture Struct of mux_chain_1bit is
component mux is
  port (a, b, s : in std_logic;
        o : out std_logic);
end component mux ;

--signal s1: std_logic;
begin
 mx: mux port map ( a => x(0) , b => '0' , s => s1, o => o(0)) ;
chain2 : for i in 1 to 7 generate
begin
 mx1: mux port map ( a => x(i) , b => x(i-1) , s => s1, o => o(i)) ;
end generate chain2 ;
end Struct;
```

Implementation of mux chain for 2 bit shift

MUX chain for 2-bit shifting

```
library std;
use std.standard.all;
```

```

library ieee;
use ieee.std_logic_1164.all;

entity mux_chain_2bit is
  port (x: in std_logic_vector(7 downto 0);
        s1: in std_logic;
        o : out std_logic_vector(7 downto 0) );
end entity mux_chain_2bit ;

architecture Struct of mux_chain_2bit is
component mux is
  port (a, b, s : in std_logic;
        o : out std_logic);
end component mux ;

--signal s1: std_logic;
begin
chain1: for i in 0 to 1 generate
 begin
  mx: mux port map ( a => x(i) , b => '0' , s => s1, o => o(i)) ;
 end generate chain1;
chain2 : for i in 2 to 7 generate
begin
 mx1: mux port map ( a => x(i) , b => x(i-2) , s => s1, o => o(i)) ;
end generate chain2 ;
end Struct;
```

Implementation of mux chain for 4 bit shift

MUX chain for 4-bit shifting

```

library std;
use std.standard.all;

library ieee;
use ieee.std_logic_1164.all;

entity mux_chain_4bit is
  port (x: in std_logic_vector(7 downto 0);
        s1: in std_logic;
        o : out std_logic_vector(7 downto 0) );
end entity mux_chain_4bit ;

```

```
architecture Struct of mux_chain_4bit is
component mux is
  port (a, b, s : in std_logic;
        o : out std_logic);
end component mux ;

--signal s1: std_logic;
begin
chain1: for i in 0 to 3 generate
 begin
  mx: mux port map ( a => x(i) , b => '0' , s => s1, o => o(i) ) ;
 end generate chain1;
chain2 : for i in 4 to 7 generate
begin
 mx1: mux port map ( a => x(i) , b => x(i-4) , s => s1 , o => o(i) ) ;
end generate chain2 ;
end Struct;
```

Implementation of an additional mux chain to convert the output to zero when number to shift is greater than three bits.

MUX chain for implementation with 8 bit input

```
library std;
use std.standard.all;

library ieee;
use ieee.std_logic_1164.all;

entity mux_ALU is
  port (x,b: in std_logic_vector(7 downto 0);
        --s1: in std_logic;
        o : out std_logic_vector(7 downto 0) );
end entity mux_ALU ;

architecture Struct of mux_ALU is
component mux is
  port (a, b, s : in std_logic;
        o : out std_logic);
end component mux ;

signal s1: std_logic;
begin
s1 <= (b(7) or b(6) or b(5) or b(4) or b(3)) ;
```

```
chain1: for i in 0 to 7 generate
 begin
  mx: mux port map ( a => x(i) , b => '0' , s => s1, o => o(i) ) ;
 end generate chain1;
end Struct;
```

Implementation of an additional mux chain to reverse the bits from LSB side to MSB side for implentation of both Lshift and Rshift with same logic.

MUX chain for reversing bits

```
library std;
use std.standard.all;

library ieee;
use ieee.std_logic_1164.all;

entity mux_change is
  port (x: in std_logic_vector(7 downto 0);
         --s1: in std_logic;
         o : out std_logic_vector(7 downto 0) );
end entity mux_change ;

architecture Struct of mux_change is
component mux is
  port (a, b, s : in std_logic;
         o : out std_logic);
end component mux ;

--signal s1: std_logic;
begin
--s1 <= '1' ;
chain1: for i in 0 to 7 generate
 begin
  mx: mux port map ( a => x(i) , b => x(7-i) , s => '1', o => o(i) ) ;
 end generate chain1;
end Struct;
```

Final implementation of Right shifter using above components.

Right Shifter

```
library std;
use std.standard.all;
library ieee;
use ieee.std_logic_1164.all;
entity Rshift is
port (a: in std_logic_vector(7 downto 0);
       b: in std_logic_vector(7 downto 0);
       z: out std_logic_vector(7 downto 0));
end entity Rshift;
architecture behave of Rshift is

component mux_chain_4bit is
  port (x: in std_logic_vector(7 downto 0);
         s1: in std_logic;
         o : out std_logic_vector(7 downto 0) );
end component mux_chain_4bit ;

component mux_chain_2bit is
  port (x: in std_logic_vector(7 downto 0);
         s1: in std_logic;
         o : out std_logic_vector(7 downto 0) );
end component mux_chain_2bit ;

component mux_chain_1bit is
  port (x: in std_logic_vector(7 downto 0);
         s1: in std_logic;
         o : out std_logic_vector(7 downto 0) );
end component mux_chain_1bit ;

component mux_ALU is
  port (x,b: in std_logic_vector(7 downto 0);
         --s1: in std_logic;
         o : out std_logic_vector(7 downto 0) );
end component mux_ALU ;

component mux_change is
        port ( x: in std_logic_vector(7 downto 0);
         --s1: in std_logic;
         o : out std_logic_vector(7 downto 0)
               );
end component mux_change;

signal st0,st1,st2,st3,st4: std_logic_vector(7 downto 0);

Begin
```

```
stage0 :mux_change port map ( x => a, o => st0  ) ;
stage1 :mux_chain_4bit port map ( x => st0 , s1 => b(2) , o => st1  ) ;
stage2 :mux_chain_2bit port map ( x => st1 , s1 => b(1) , o => st2  ) ;
stage3 :mux_chain_1bit port map ( x => st2 , s1 => b(0) , o => st3   ) ;
stage4 :mux_ALU port map ( x => st3, b => b , o => st4  ) ;
stage5 :mux_change port map ( x => st4, o => z  ) ;
end behave;
```

2.4 Logical Left Shift

It is implemented with same logic as right shift. Just I havereversed the incoming bits and the respective output .

Left Shifter

```
library std;
use std.standard.all;
library ieee;
use ieee.std_logic_1164.all;
entity Lshift is
port (a: in std_logic_vector(7 downto 0);
       b: in std_logic_vector(7 downto 0);
       z: out std_logic_vector(7 downto 0));
end entity Lshift;
architecture behave of Lshift is

component mux_chain_4bit is
  port (x: in std_logic_vector(7 downto 0);
         s1: in std_logic;
         o : out std_logic_vector(7 downto 0) );
end component mux_chain_4bit ;

component mux_chain_2bit is
  port (x: in std_logic_vector(7 downto 0);
         s1: in std_logic;
         o : out std_logic_vector(7 downto 0) );
end component mux_chain_2bit ;

component mux_chain_1bit is
  port (x: in std_logic_vector(7 downto 0);
         s1: in std_logic;
         o : out std_logic_vector(7 downto 0) );
end component mux_chain_1bit ;
```

```
29
30 component mux_ALU is
31   port (x,b: in std_logic_vector(7 downto 0);
32         --s1: in std_logic;
33         o : out std_logic_vector(7 downto 0) );
34 end component mux_ALU ;
35
36 signal st1,st2,st3: std_logic_vector(7 downto 0);
37
38
39 Begin
40 stage1 :mux_chain_4bit port map ( x => a , s1 => b(2) , o => st1  ) ;
41 stage2 :mux_chain_2bit port map ( x => st1 , s1 => b(1) , o => st2  ) ;
42 stage3 :mux_chain_1bit port map ( x => st2 , s1 => b(0) , o => st3   ) ;
43 stage0 :mux_ALU port map ( x => st3, b => b , o => z  ) ;
44
45 end behave;
```

3 ALU

Integrating all the above components and testbench used are given below.

ALU

```
1  ----------------------------------------------
2  library std;
3  use std.standard.all;
4
5  library ieee;
6  use ieee.std_logic_1164.all;
7  ----------------------------------------------
8  entity alu is
9          port( X,Y : in std_logic_vector(7 downto 0); x0,x1 : in std_logic ;
10                 Z : out std_logic_vector(7 downto 0));
11 end entity;
12
13 architecture behave of alu is
14         signal sig1,sig2,sig3,sig4 : std_logic_vector(7 downto 0);
15
16     component EightBitAdder is
17     port (x,y: in std_logic_vector(7 downto 0) ;
18          sum: out std_logic_vector(7 downto 0) );
19     end component EightBitAdder;
```

```
20
21      component Substractor is
22              port (x,y: in std_logic_vector(7 downto 0) ;
23           sum: out std_logic_vector(7 downto 0)
24              );
25      end component Substractor;
26
27      component Lshift is
28      port (a: in std_logic_vector(7 downto 0);
29         b: in std_logic_vector(7 downto 0);
30         z: out std_logic_vector(7 downto 0));
31      end component Lshift;
32
33      component Rshift is
34      port (a: in std_logic_vector(7 downto 0);
35         b: in std_logic_vector(7 downto 0);
36         z: out std_logic_vector(7 downto 0));
37      end component Rshift;
38  ------------------------------------------------
39  begin
40  a: EightBitAdder port map(x => X, y => Y, sum => sig1);
41  b: Lshift port map(a => X, b => Y, z => sig4);
42  c: Rshift port map(a => X, b => Y, z => sig3);
43  d: Substractor  port map(x => X, y => Y, sum => sig2);
44  ------------------------------------------------
45
46  process(x0, x1,sig1, sig2, sig3, sig4)
47  begin
48  ------------------------------
49  if (x0 = '0' and x1 = '0') then
50  z<= sig1;
51  elsif(x0 = '1') and (x1 = '0') then
52  z<= sig2;
53  elsif(x0 = '0') and (x1 = '1') then
54  z<= sig3;
55  else
56  z<= sig4;
57  end if;
58  ------------------------------
59  end process;
60
61  end behave;
```

Implementation of final DUT to be tested by testbench

DUT

```
-- A DUT entity is used to wrap your design.
--  This example shows how you can do this for the
--  two-bit adder.
library std;
use std.standard.all;

library ieee;
use ieee.std_logic_1164.all;

entity DUT is
   port(input_vector: in std_logic_vector(17 downto 0);
   ---Note: for alu testing (17 downto 0) for others (15 downto 0)
               output_vector: out std_logic_vector(7 downto 0));
end entity;

architecture DutWrap of DUT is

        component alu is
        port( X,Y : in std_logic_vector(7 downto 0); x0,x1 : in std_logic ; Z :
           out std_logic_vector(7 downto 0));
    end component;


begin

dut: alu port map( X => input_vector(15 downto 8), Y => input_vector(7 downto 0) ,
             x0 => input_vector(16) , x1 => input_vector(17), Z => output_vector);
end DutWrap;
```

Testbench

```
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;
```

```
12  architecture Behave of Testbench is
13
14    ----------------------------------------------------------------
15    --  edit the following lines to set the number of i/o's of your
16    --  DUT.
17    ----------------------------------------------------------------
18    constant number_of_inputs  : integer := 18;  -- # input bits to your design.
19    constant number_of_outputs : integer := 8;  -- # output bits from your design.
20
21    -- component port widths..
22    component DUT is
23     port(input_vector: in std_logic_vector(number_of_inputs-1  downto 0);
24                  output_vector: out std_logic_vector(number_of_outputs-1 downto 0));
25    end component;
26
27    -- end editing.
28    ----------------------------------------------------------------
29    ----------------------------------------------------------------
30
31    signal input_vector  : bit_vector(number_of_inputs-1 downto 0);
32    signal output_vector : bit_vector(number_of_outputs-1 downto 0);
33    signal std_output_vector : std_logic_vector(number_of_outputs-1 downto 0);
34
35    -- create a constrained string outof
36    function to_string(x: string) return string is
37        variable ret_val: string(1 to x'length);
38        alias lx : string (1 to x'length) is x;
39    begin
40        ret_val := lx;
41        return(ret_val);
42    end to_string;
43
44  begin
45    process
46      variable err_flag : boolean := false;
47      File INFILE: text open read_mode is
48           "/home/amit/Desktop/Digital_lab/LAB/tracefiles_entity/tracefiles/alu_TRACEFILE.txt";
49      FILE OUTFILE: text  open write_mode is
50           "/home/amit/Desktop/Digital_lab/LAB/tracefiles_entity/tracefiles/OUTPUTS.txt";
51
52      -------------------------------------------------
53      -- edit the next two lines to customize
54      -------------------------------------------------
55      variable input_vector_var: bit_vector (number_of_inputs-1 downto 0);
56      variable output_vector_var: bit_vector (number_of_outputs-1 downto 0);
```

```
57      variable output_mask_var: bit_vector (number_of_outputs-1 downto 0);
58      variable output_comp_var: bit_vector (number_of_outputs-1 downto 0);
59      constant ZZZZ : bit_vector(number_of_outputs-1 downto 0) := (others => '0');
60      ---------------------------------------------------
61
62      variable INPUT_LINE: Line;
63      variable OUTPUT_LINE: Line;
64      variable LINE_COUNT: integer := 0;
65
66
67    begin
68      while not endfile(INFILE) loop
69            -- will read a new line every 5ns, apply input,
70            -- wait for 1 ns for circuit to settle.
71            -- read output.
72
73
74            LINE_COUNT := LINE_COUNT + 1;
75
76
77            -- read input at current time.
78            readLine (INFILE, INPUT_LINE);
79            read (INPUT_LINE, input_vector_var);
80            read (INPUT_LINE, output_vector_var);
81            read (INPUT_LINE, output_mask_var);
82
83            -- apply input.
84            input_vector <= input_vector_var;
85
86            -- wait for the circuit to settle
87            wait for 150 ns;
88
89            -- check output.
90            output_comp_var := (output_mask_var and (output_vector xor output_vector_var));
91            if (output_comp_var  /= ZZZZ) then
92               write(OUTPUT_LINE,to_string("ERROR: line "));
93               write(OUTPUT_LINE, LINE_COUNT);
94               writeline(OUTFILE, OUTPUT_LINE);
95               err_flag := true;
96            end if;
97
98            write(OUTPUT_LINE, input_vector);
99            write(OUTPUT_LINE, to_string(" "));
100           write(OUTPUT_LINE, output_vector);
101           writeline(OUTFILE, OUTPUT_LINE);
```

```
102
103             -- advance time by 4 ns.
104             wait for 4 ns;
105       end loop;
106
107       assert (err_flag) report "SUCCESS, all tests passed." severity note;
108       assert (not err_flag) report "FAILURE, some tests failed." severity error;
109
110       wait;
111     end process;
112
113           output_vector <= to_bitvector(std_output_vector);
114     dut_instance: DUT
115                 port map(input_vector => to_stdlogicvector(input_vector),
116                       output_vector =>std_output_vector );
117
118 end Behave;
```

4 Observations

All te relevant screenshota and results are shown below:

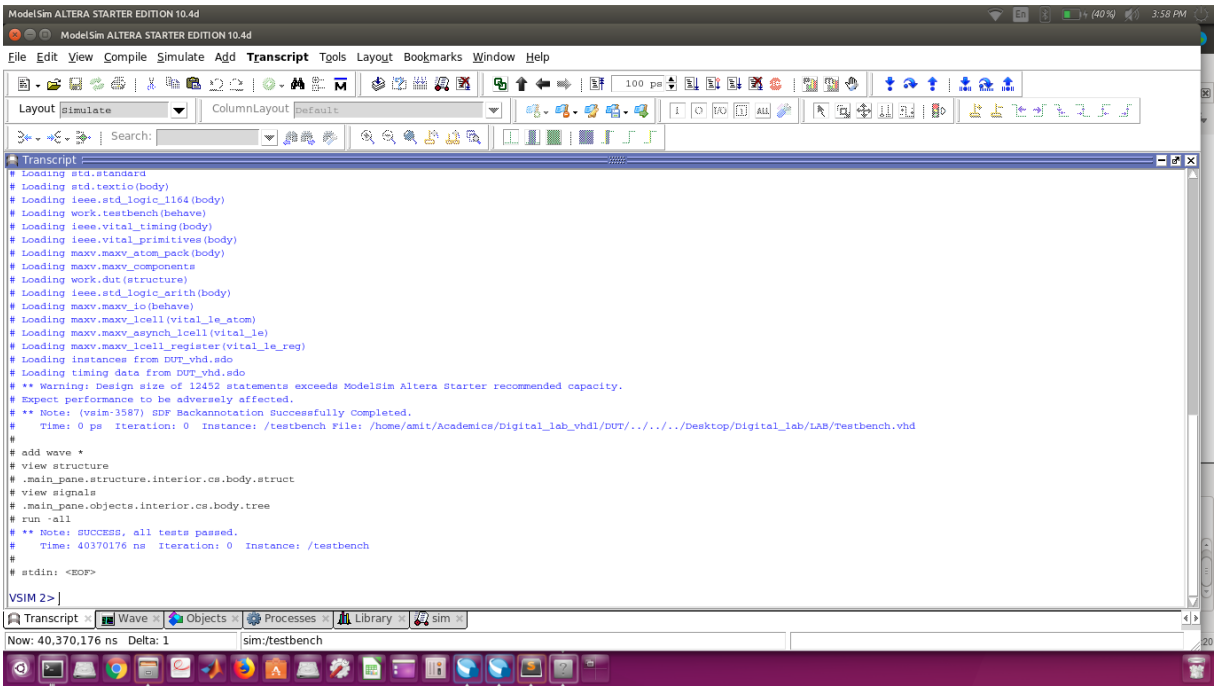

Figure 2: Transcript window of all test case passed with GATE level simulation

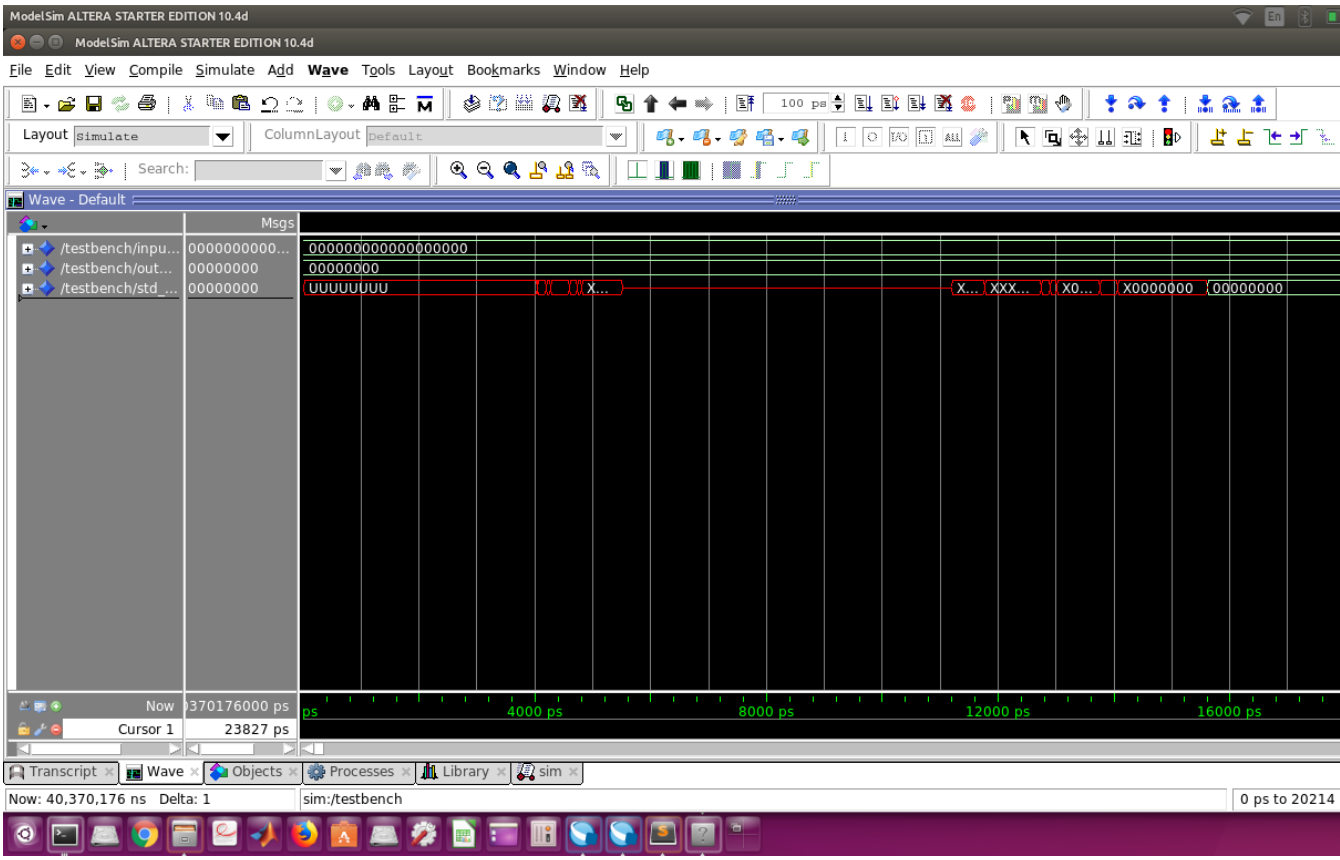

Figure 3: Initial waveforms obtained from gate level simulation

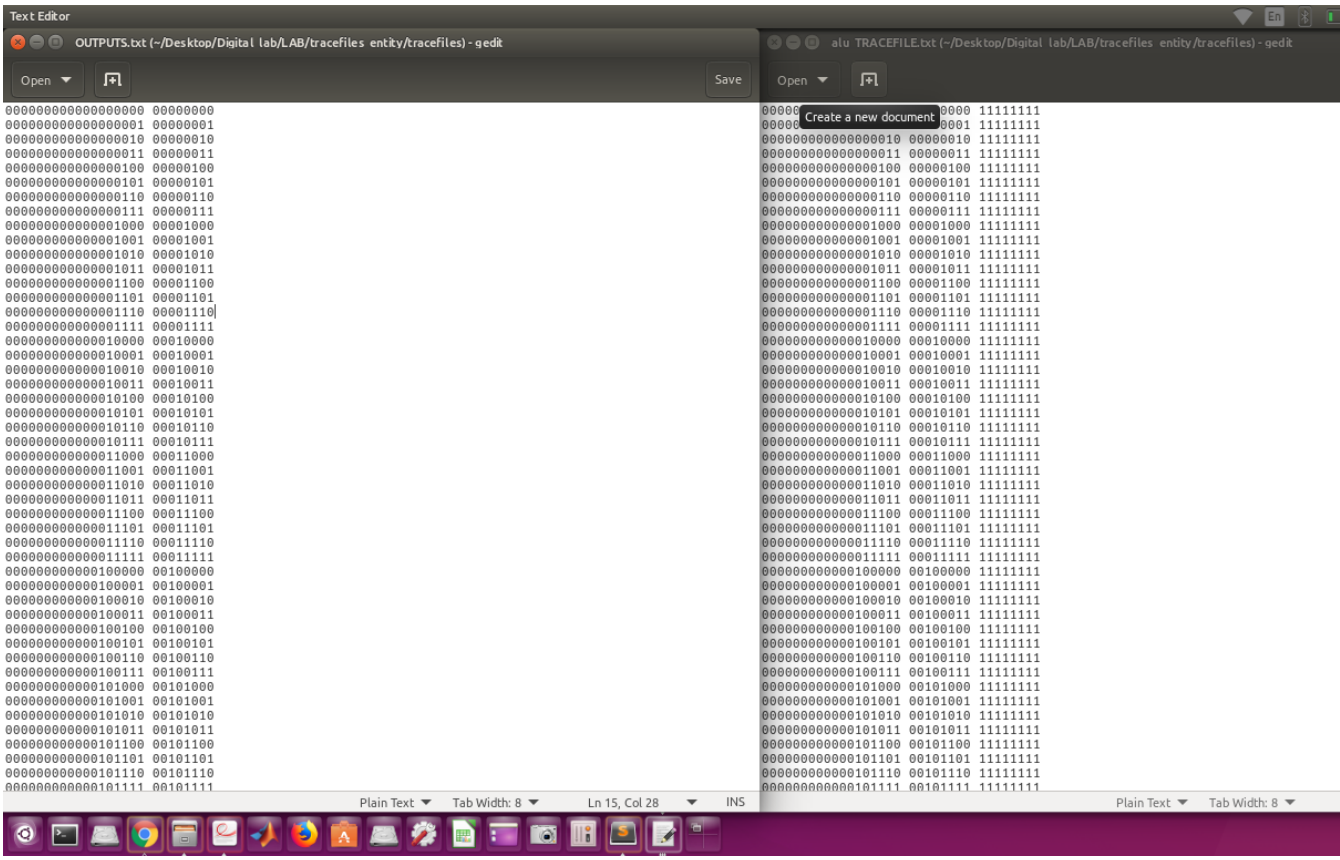

Figure 4: Comparison of output obtained by testbench and tracefiles for ALU

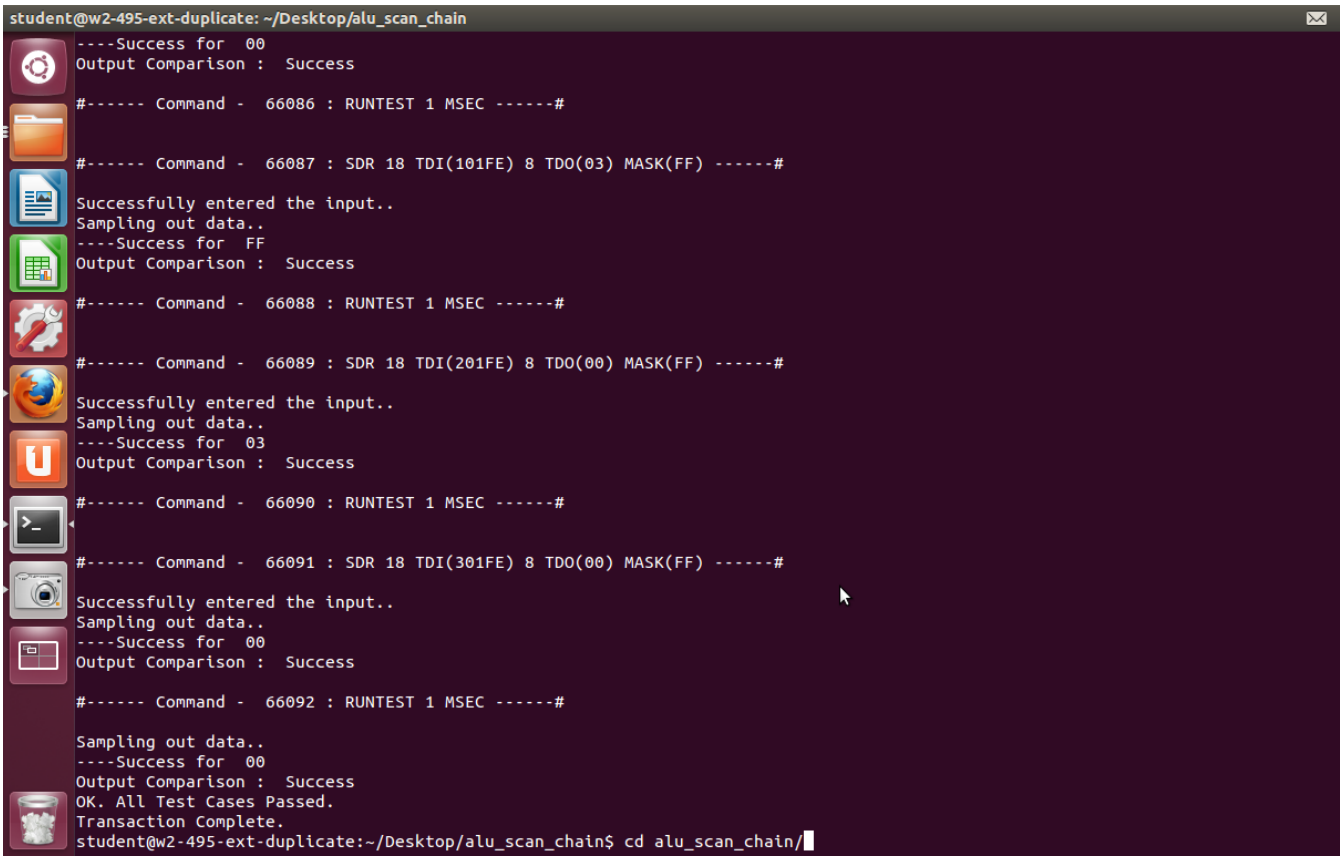

Figure 5: Screenshot of all test case passed using Scan chain

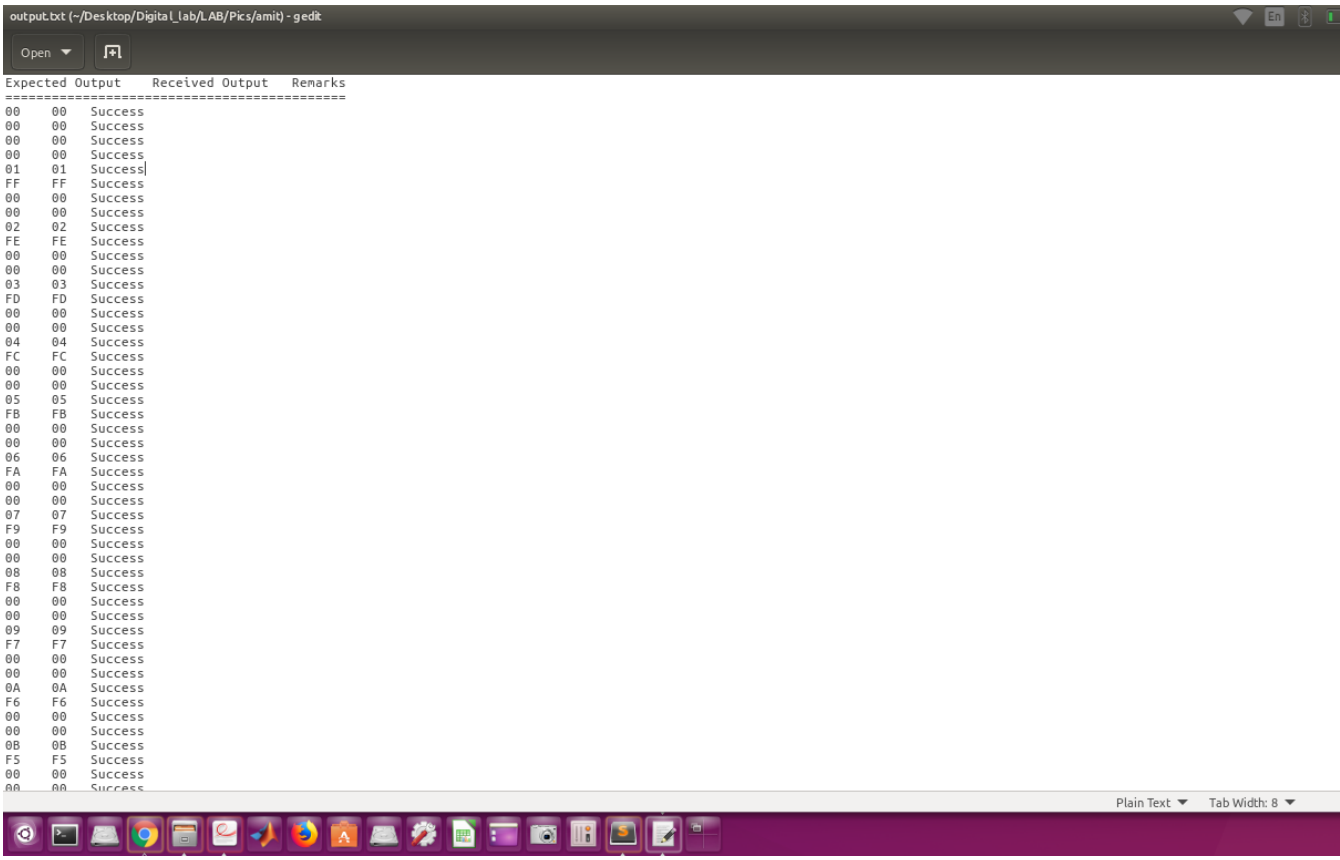

Figure 6: Output file obtained by scan chain