Experiment 5
8-Bit ALU on the Krypton CPLD

Dhruv Ilesh Shah — 150070016

February 17, 2017

Overview

The ALU, or Arithmetic Logic Unit, is a very crucial part of any processor. By definition, it must be able to perform simple arithmetic and logical operations. In this experiment, I have implemented an 8-bit ALU which can perform the following operations.

1. Addition
2. Subtraction
3. Left-Shift
4. Right-Shift

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 ALU piece-wise, 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 aim is to build an ALU with 4 different purposes. At a basic level, we can build 4 different modules and then use a 4-bit multiplexer to choose the final output of the ALU.

op_code	**Operation**	**Result**
00	Addition	$Z = X + Y$
01	Subtraction	$Z = X - Y$
10	Logical Right Shift	$Z = X >> Y$
11	Logical Left Shift	$Z = X << Y$

Sticking to the design guidelines, the logic has been implemented using *only* two-input AND, OR and NOT gates. In this section, I build the elements piece-wise in the earlier parts, and put them all together in the end to form the entity `alu`.

Before entering specific implementation, we see that the *XOR* operation can be useful in logic, and hence I made an entity `myXOR` which can be used later on.

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

entity myXOR is
        port(x,y: in std_logic;
                s: out std_logic);
end entity;

architecture Form of myXOR is
begin
        s <= ((x and (not y)) or (y and (not x)));
end Form;
```

This entity will be used multiple times in the various implementation below.

1.1 Addition

The addition operation can be carried out in multiple ways, but given the scale of the problem, and the possibility of scaling, reusability of the code is essential. I have implemented the 8-bit adder using 8 units of 1-bit *full adders*. This can be extended using 2-bit full adders etc, to reduce the delays, but the complexity of implementation remains $\mathcal{O}(n)$. For a problem this size, the logarithmic adder would become too complex in space, and hence I stuck to the simpler model.

Full Adder

Consider a block with inputs c_i, x_i, y_i and outputs c_o, s_o.

$$\begin{aligned} s_o &= c_i \oplus x_i \oplus y_i \\ c_o &= x_i y_i + y_i c_i + c_i x_i \end{aligned} \tag{1}$$

The implementation is given below as the entity `full_adder`.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity full_adder is
5          port(ci,xi,yi: in std_logic;
6                  co,so: out std_logic);
7  end entity;
8
9  architecture Form of full_adder is
10         signal xor_i: std_logic;
11         component myXOR is
12                 port(x,y: in std_logic;
13                         s: out std_logic);
14         end component;
15 begin
16         add_instance_s0_1: myXOR
17                 port map (xi,yi,xor_i);
18
19         add_instance_s0_2: myXOR
```

```
                port map (xor_i,ci,so);

        co <= ((xi and yi) or (yi and ci) or (ci and xi));
end Form;
```

Eight Bit Adder

Using this module, we can easily define the 8-bit addition operation as below. *(Note that I have avoided using loops as a structure, to keep the code raw, and close to how it is actually implemented in hardware. The redundancy can be easily be replaced by calling a process and iterating.)*

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

entity EightBitAdder is
        --constant size : integer := 8;
        port(x,y: in std_logic_vector(7 downto 0);
                z: out std_logic_vector(7 downto 0));
end entity;

architecture Summer of EightBitAdder is
        signal c : std_logic_vector(7 downto 0);
        signal etc: std_logic;
        component full_adder is
                port(ci,xi,yi: in std_logic;
                        co,so: out std_logic);
        end component;

begin
        c(0) <= '0';
        bit_1: full_adder
                port map (c(0),x(0),y(0),c(1),z(0));
        bit_2: full_adder
                port map (c(1),x(1),y(1),c(2),z(1));
        bit_3: full_adder
                port map (c(2),x(2),y(2),c(3),z(2));
        bit_4: full_adder
                port map (c(3),x(3),y(3),c(4),z(3));
        bit_5: full_adder
                port map (c(4),x(4),y(4),c(5),z(4));
        bit_6: full_adder
                port map (c(5),x(5),y(5),c(6),z(5));
        bit_7: full_adder
                port map (c(6),x(6),y(6),c(7),z(6));
        bit_8: full_adder
                port map (c(7),x(7),y(7),etc,z(7));

end Summer;
```

1.2 Subtractor

Approaching the subtractor in a manner similar to the adder, I have used a 1-bit full subtractor, which is then used to create an 8-bit subtractor.

Full Subtractor

Consider a block with inputs d_i, x_i, y_i and outputs d_o, s_o.

$$\begin{aligned} s_o &= c_i \oplus x_i \oplus y_i \\ c_o &= \overline{x_i} y_i + d_i(\overline{x_i \oplus y_i}) \end{aligned} \tag{2}$$

The implementation of the above is given below.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity subtractor is
5          port(di,xi,yi: in std_logic;
6                  do,so: out std_logic);
7  end entity;
8
9  architecture Form of subtractor is
10         signal xor_i: std_logic;
11         signal hold: std_logic;
12         component myXOR is
13                 port(x,y: in std_logic;
14                         s: out std_logic);
15         end component;
16 begin
17         out_instance_1: myXOR
18                 port map(xi, yi, xor_i);
19         out_instance_2: myXOR
20                 port map(xor_i, di, so);
21
22         do <= (((not xi) and (yi)) or (di and (not xor_i)));
23
24 end Form;
```

Eight Bit Subtractor

By using the above unit entity, and extending the definition of the subtractor to 8 bits, we have the following.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity EightBitSub is
5          port(x,y: in std_logic_vector(7 downto 0);
6                  z: out std_logic_vector(7 downto 0));
7  end entity;
8
9  architecture Diff of EightBitSub is
```

```
        signal d : std_logic_vector(7 downto 0);
        signal etc: std_logic;
        component subtractor is
                port(di,xi,yi: in std_logic;
                        do,so: out std_logic);
        end component;

begin
        d(0) <= '0';
        bit_1: subtractor
                port map (d(0),x(0),y(0),d(1),z(0));
        bit_2: subtractor
                port map (d(1),x(1),y(1),d(2),z(1));
        bit_3: subtractor
                port map (d(2),x(2),y(2),d(3),z(2));
        bit_4: subtractor
                port map (d(3),x(3),y(3),d(4),z(3));
        bit_5: subtractor
                port map (d(4),x(4),y(4),d(5),z(4));
        bit_6: subtractor
                port map (d(5),x(5),y(5),d(6),z(5));
        bit_7: subtractor
                port map (d(6),x(6),y(6),d(7),z(6));
        bit_8: subtractor
                port map (d(7),x(7),y(7),etc,z(7));

end Diff;
```

1.3 Logical Right Shift

The logical right shift is an essential operation for an ALU. One of the simplest interpretation can be division by 2. This can give an easy way to implement multiplication/division operations. Hardwiring is always an option, but a more general and reusable algorithm would be preferred.

For this purpose, 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 (not a rotator). Hence we need to implement only 3 stages.

We notice that an important feature of this implementation is the multiplexer. So, I begin by creating a `mux` entity, which is then used to make the 8-mux chains, eventually making our shifter.

MUX

A multiplexer with inputs n_1, n_0, s and output b follows:

$$b = sn_1 + \bar{s}n_0$$

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

```

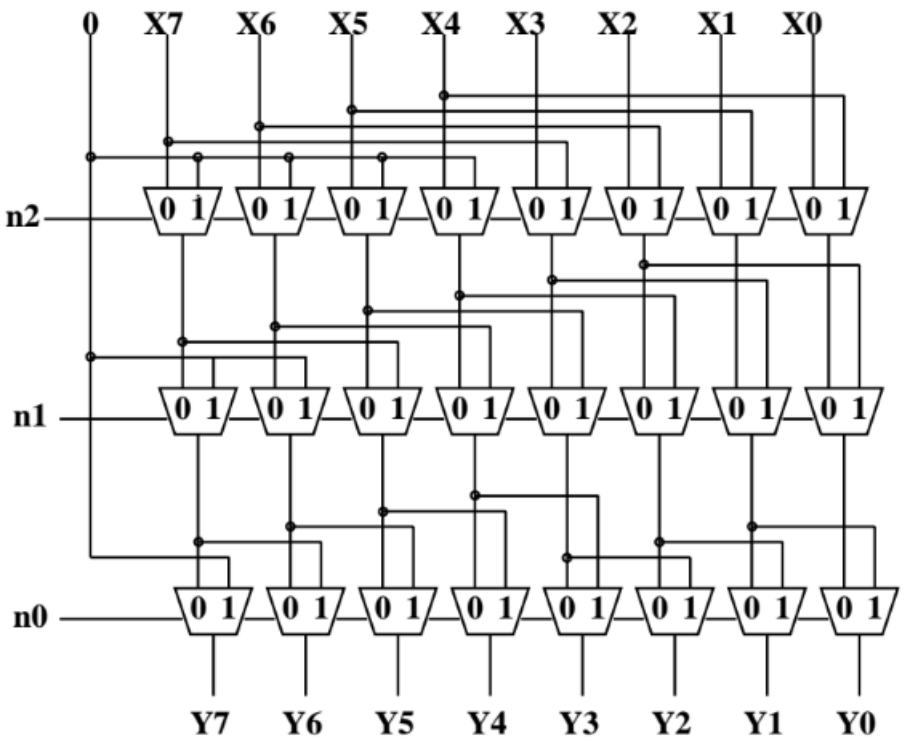

Figure 1: Logical Right Shift (*Barrel Shift*)

```
entity mux is
        port(n1,n0,s: in std_logic;
                b: out std_logic);
end entity;

architecture choose of mux is
begin
        b <= ((s and n1) or ((not s) and n0));
end choose;
```

MUX Chains

In principle, we can use a generic 8-unit chain of multiplexers with 16 inputs and 8 outputs, working the way we like. Such an implementation is given below.

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

entity mux8 is
        port(mx, my: in std_logic_vector(7 downto 0);
                        mo: out std_logic_vector(7 downto 0);
                        sel: in std_logic);
end entity;

architecture choose of mux8 is
```

```
component mux is
        port(n1,n0,s: in std_logic;
                b: out std_logic);
end component;
begin
        mux_7: mux port map(mx(7),my(7),sel, mo(7));
        mux_6: mux port map(mx(6),my(6),sel, mo(6));
        mux_5: mux port map(mx(5),my(5),sel, mo(5));
        mux_4: mux port map(mx(4),my(4),sel, mo(4));
        mux_3: mux port map(mx(3),my(3),sel, mo(3));
        mux_2: mux port map(mx(2),my(2),sel, mo(2));
        mux_1: mux port map(mx(1),my(1),sel, mo(1));
        mux_0: mux port map(mx(0),my(0),sel, mo(0));

end choose;
```

In this implementation, we would have to give the input vector in the order they enter, and hence the code would end up messy anyway. Instead, I made three different entities `mux_chain_1` `mux_chain_2` `mux_chain_3` customised to the chains seen in the diagram.

MUX Chain 1 (n_2)

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


entity mux_chain_1 is
        port (inp: in std_logic_vector(7 downto 0);
                sel: in std_logic;
                outp: out std_logic_vector(7 downto 0));
end entity;

architecture chain of mux_chain_1 is

component mux is
        port(n1,n0,s: in std_logic;
                b: out std_logic);
end component;
begin
        mux_7: mux port map('0',inp(7),sel, outp(7));
        mux_6: mux port map('0',inp(6),sel, outp(6));
        mux_5: mux port map('0',inp(5),sel, outp(5));
        mux_4: mux port map('0',inp(4),sel, outp(4));
        mux_3: mux port map(inp(7),inp(3),sel, outp(3));
        mux_2: mux port map(inp(6),inp(2),sel, outp(2));
        mux_1: mux port map(inp(5),inp(1),sel, outp(1));
        mux_0: mux port map(inp(4),inp(0),sel, outp(0));
end chain;
```

MUX Chain 2 (n_1)

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


entity mux_chain_2 is
        port (inp2: in std_logic_vector(7 downto 0);
                sel2: in std_logic;
                outp2: out std_logic_vector(7 downto 0));
end entity;

architecture chain of mux_chain_2 is
component mux is
        port(n1,n0,s: in std_logic;
                b: out std_logic);
end component;
begin
        mux_7: mux port map('0',inp2(7),sel2, outp2(7));
        mux_6: mux port map('0',inp2(6),sel2, outp2(6));
        mux_5: mux port map(inp2(7),inp2(5),sel2, outp2(5));
        mux_4: mux port map(inp2(6),inp2(4),sel2, outp2(4));
        mux_3: mux port map(inp2(5),inp2(3),sel2, outp2(3));
        mux_2: mux port map(inp2(4),inp2(2),sel2, outp2(2));
        mux_1: mux port map(inp2(3),inp2(1),sel2, outp2(1));
        mux_0: mux port map(inp2(2),inp2(0),sel2, outp2(0));
end chain;
```

MUX Chain 3 (n_0)

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


entity mux_chain_3 is
        port (inp3: in std_logic_vector(7 downto 0);
                sel3: in std_logic;
                outp3: out std_logic_vector(7 downto 0));
end entity;

architecture chain of mux_chain_3 is
component mux is
        port(n1,n0,s: in std_logic;
                b: out std_logic);
end component;
begin
        mux_7: mux port map('0',inp3(7),sel3, outp3(7));
        mux_6: mux port map(inp3(7),inp3(6),sel3, outp3(6));
        mux_5: mux port map(inp3(6),inp3(5),sel3, outp3(5));
        mux_4: mux port map(inp3(5),inp3(4),sel3, outp3(4));
```

```
        mux_3: mux port map(inp3(4),inp3(3),sel3, outp3(3));
        mux_2: mux port map(inp3(3),inp3(2),sel3, outp3(2));
        mux_1: mux port map(inp3(2),inp3(1),sel3, outp3(1));
        mux_0: mux port map(inp3(1),inp3(0),sel3, outp3(0));
end chain;
```

Right Barrel Shifter

Now that we have these components ready, we can code up the shifter.

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

entity right_shifter is
        port (x,y: in std_logic_vector(7 downto 0);
                s: out std_logic_vector(7 downto 0));
end entity;
architecture logical_right of right_shifter is
        signal zero_check: std_logic;
        signal level1, level2, level3: std_logic_vector(7 downto 0);
        component mux_chain_1 is
                port (inp: in std_logic_vector(7 downto 0);
                        sel: in std_logic;
                        outp: out std_logic_vector(7 downto 0));
        end component;
        component mux_chain_2 is
                port (inp2: in std_logic_vector(7 downto 0);
                        sel2: in std_logic;
                        outp2: out std_logic_vector(7 downto 0));
        end component;
        component mux_chain_3 is
                port (inp3: in std_logic_vector(7 downto 0);
                        sel3: in std_logic;
                        outp3: out std_logic_vector(7 downto 0));
        end component;
        component mux8 is
                port(mx, my: in std_logic_vector(7 downto 0);
                                mo: out std_logic_vector(7 downto 0);
                                sel: in std_logic);
        end component;
begin
        lev1: mux_chain_1
                port map(x,y(2),level1);
        lev2: mux_chain_2
                port map(level1, y(1), level2);
        lev3: mux_chain_3
                port map(level2, y(0), level3);
        zero_check <= (not ((y(3)) or (y(4)) or (y(5)) or (y(6)) or (y(7))));
        final: mux8
                port map(level3, "00000000", s, zero_check);
end logical_right;
```

1.4 Logical Left Shifter

Instead of designing the left shifter from scratch, we can use symmetry arguments to ease our process. Given a right shifter, feeding it with the reverse of the input vector, and reading the reverse of its output vector gives the same result as an intended left shifter! This means that all we need is a way to reverse the vector. The rest is straightforward.

Instead of writing the reverse operations within the main body, I have used another entity `reverse` for clarity.

8-bit Reverse

The logic is straightforward, as given below.

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

entity reverse is
        port (x: in std_logic_vector(7 downto 0);
                y: out std_logic_vector(7 downto 0));
end entity;

architecture reverse of reverse is
begin
        y(0) <= x(7);
        y(1) <= x(6);
        y(2) <= x(5);
        y(3) <= x(4);
        y(4) <= x(3);
        y(5) <= x(2);
        y(6) <= x(1);
        y(7) <= x(0);

end reverse;
```

Left Barrel Shifter

Given the `right_shifter` from above, and the `reverse` entity, we can perform this very easily.

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

entity left_shifter is
        port (x,y: in std_logic_vector(7 downto 0);
                s: out std_logic_vector(7 downto 0));
end entity;

architecture rtl of left_shifter is
        signal x_r, y_r: std_logic_vector(7 downto 0);

        component reverse is
```

```
13                  port (x: in std_logic_vector(7 downto 0);
14                          y: out std_logic_vector(7 downto 0));
15          end component;
16
17          component right_shifter is
18                  port (x,y: in std_logic_vector(7 downto 0);
19                          s: out std_logic_vector(7 downto 0));
20          end component;
21  begin
22          reverse_input: reverse
23                  port map(x, x_r);
24          right_shift: right_shifter
25                  port map(x_r,y,y_r);
26
27          reverse_output: reverse
28                  port map(y_r,s);
29  end rtl;
```

1.5 The ALU

Putting together the whole code, we declare the `alu` entity as follows.
(Most of this segment was provided by WEL and hence the use of `if...else`. This can easily be replaced by multiplexers.)

```
1   library std;
2   use std.standard.all;
3   library ieee;
4   use ieee.std_logic_1164.all;
5
6   entity alu is
7           port( X,Y : in std_logic_vector(7 downto 0);
8                   x0,x1 : in std_logic ;
9                   Z : out std_logic_vector(7 downto 0));
10  end entity;
11
12  architecture behave of alu is
13          signal sig1,sig2,sig3,sig4 : std_logic_vector(7 downto 0);
14
15          component EightBitAdder is
16                  port(x,y: in std_logic_vector(7 downto 0);
17                          z: out std_logic_vector(7 downto 0));
18          end component;
19
20          component EightBitSub is
21                  port(x,y: in std_logic_vector(7 downto 0);
22                          z: out std_logic_vector(7 downto 0));
23          end component;
24
25          component left_shifter is
26                  port (x,y: in std_logic_vector(7 downto 0);
27                          s: out std_logic_vector(7 downto 0));
```

```
        end component;

        component right_shifter is
                port (x,y: in std_logic_vector(7 downto 0);
                        s: out std_logic_vector(7 downto 0));
        end component;
-----------------------------------------------
begin
a: EightBitAdder       port map(x => X, y => Y, z => sig1);
b: left_shifter               port map(x => X, y => Y, s => sig4);
c: right_shifter          port map(x => X, y => Y, s => sig3);
d: EightBitSub  port map(x => X, y => Y, z => sig2);
-----------------------------------------------

process(x0, x1,sig1, sig2, sig3, sig4)
begin
------------------------------
if (x0 = '0' and x1 = '0') then
z<= sig1;
elsif(x0 = '1') and (x1 = '0') then
z<= sig2;
elsif(x0 = '0') and (x1 = '1') then
z<= sig3;
else
z<= sig4;
end if;
------------------------------
end process;
end behave;
```

The **Testbench** used for this entire setup is also 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

  constant number_of_inputs  : integer := 18;  -- # input bits to your design.
  constant number_of_outputs : integer := 8;  -- # output bits from your design.

  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));
```

```
20    end component;
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    signal std_output_vector : std_logic_vector(number_of_outputs-1 downto 0);
25
26
27    function to_string(x: string) return string is
28        variable ret_val: string(1 to x'length);
29        alias lx : string (1 to x'length) is x;
30    begin
31        ret_val := lx;
32        return(ret_val);
33    end to_string;
34
35  begin
36    process
37      variable err_flag : boolean := false;
38      File INFILE: text open read_mode is "~/tracefiles/alu_TRACEFILE.txt";
39      FILE OUTFILE: text  open write_mode is "~/tracefiles/alu_OUTPUTS.txt";
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      variable INPUT_LINE: Line;
48      variable OUTPUT_LINE: Line;
49      variable LINE_COUNT: integer := 0;
50
51
52    begin
53      while not endfile(INFILE) loop
54
55            LINE_COUNT := LINE_COUNT + 1;
56
57            readLine (INFILE, INPUT_LINE);
58            read (INPUT_LINE, input_vector_var);
59            read (INPUT_LINE, output_vector_var);
60            read (INPUT_LINE, output_mask_var);
61
62            input_vector <= input_vector_var;
63
64            wait for 1 ns;
65
66            output_comp_var := (output_mask_var and (output_vector xor output_vector_var));
67            if (output_comp_var  /= ZZZZ) then
68               write(OUTPUT_LINE,to_string("ERROR: line "));
69               write(OUTPUT_LINE, LINE_COUNT);
```

```
70              writeline(OUTFILE, OUTPUT_LINE);
71              err_flag := true;
72          end if;
73
74          write(OUTPUT_LINE, input_vector);
75          write(OUTPUT_LINE, to_string(" "));
76          write(OUTPUT_LINE, output_vector);
77          writeline(OUTFILE, OUTPUT_LINE);
78
79          wait for 4 ns;
80      end loop;
81
82      assert (err_flag) report "SUCCESS, all tests passed." severity note;
83      assert (not err_flag) report "FAILURE, some tests failed." severity error;
84
85      wait;
86    end process;
87
88          output_vector <= to_bitvector(std_output_vector);
89    dut_instance: DUT
90              port map(input_vector => to_stdlogicvector(input_vector), output_vector =>std_
91
92  end Behave;
```

2 Observations

In this section, we simulate the above codes to test logic and observe the delay characteristics.

2.1 Addition

Figures 2-3 show the results of the simulation. *(Since the number of possible test cases is too large (2^{16}), I have showed a small section of the waveform.)*

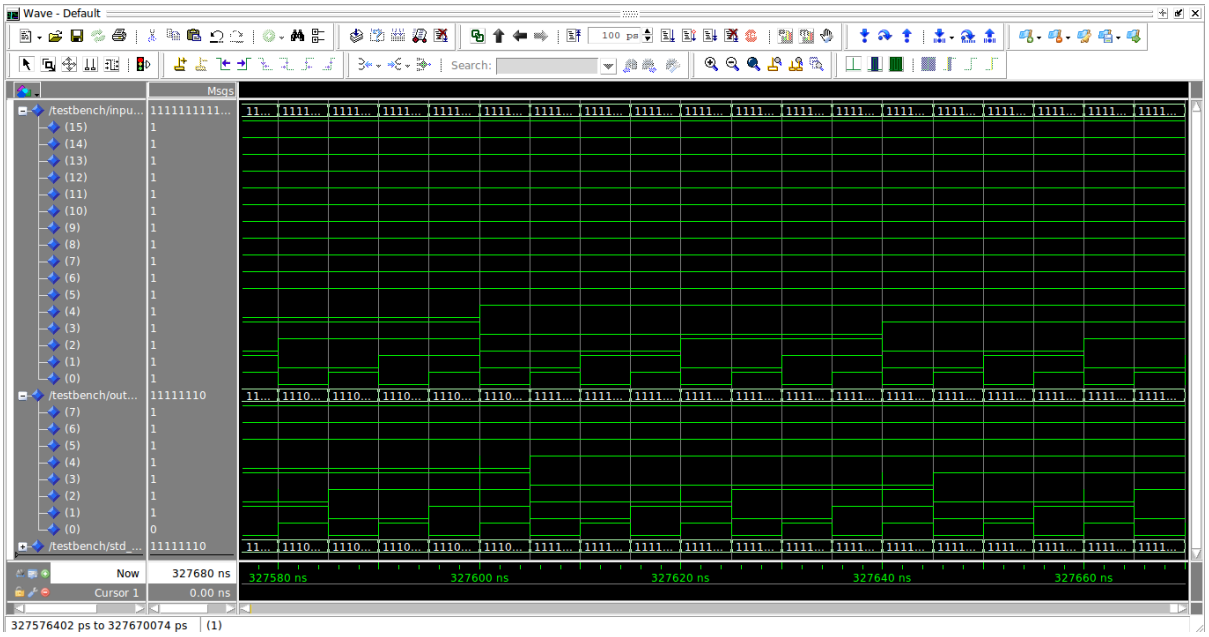

Figure 2: RTL Simulation of the Addition operation

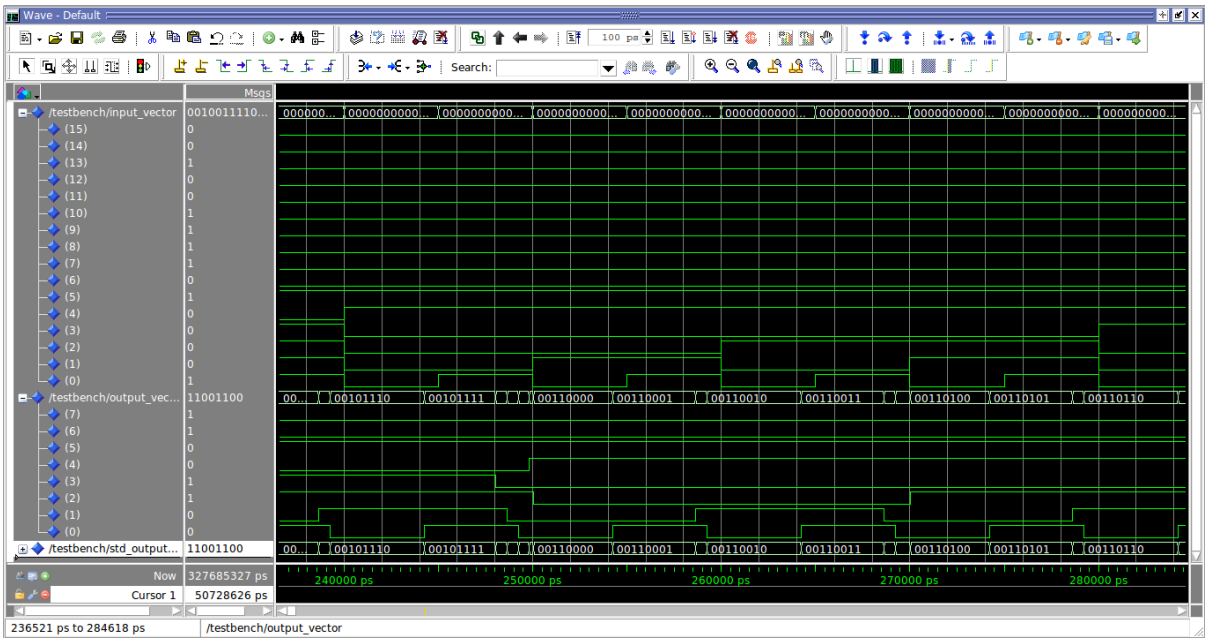

Figure 3: Gate-level Simulation of the Addition operation

2.2 Subtraction

Figures 4-5 show the results of the simulation. *(Since the number of possible test cases is too large (2^{16}), I have showed a small section of the waveform.)*

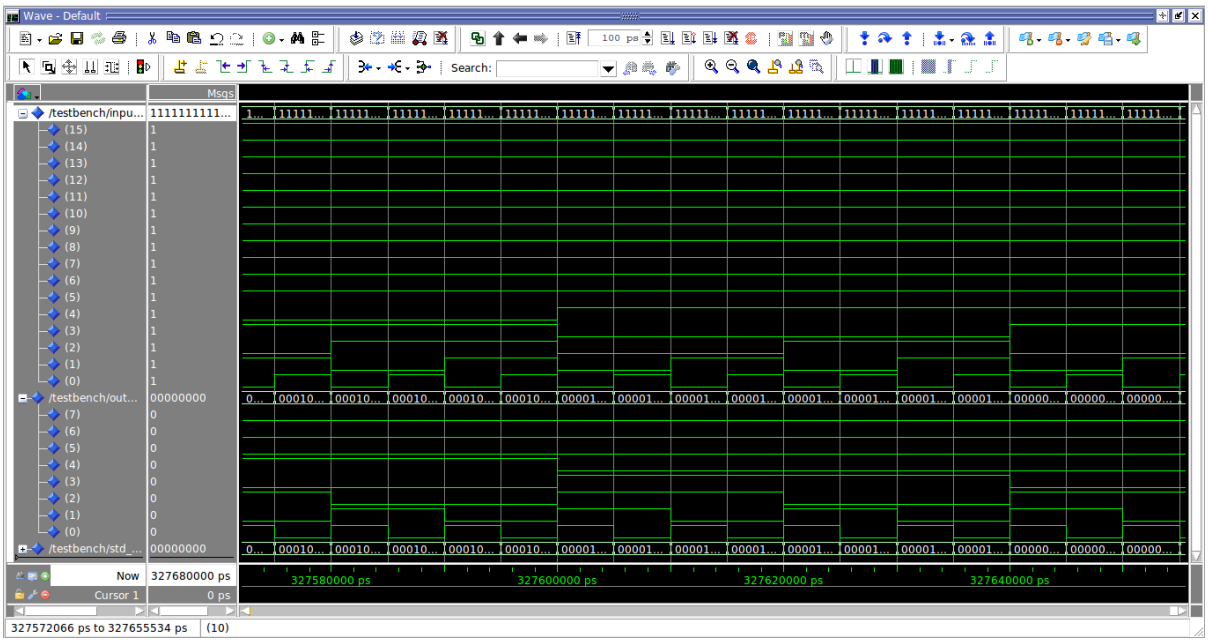

Figure 4: RTL Simulation of the Subtraction operation

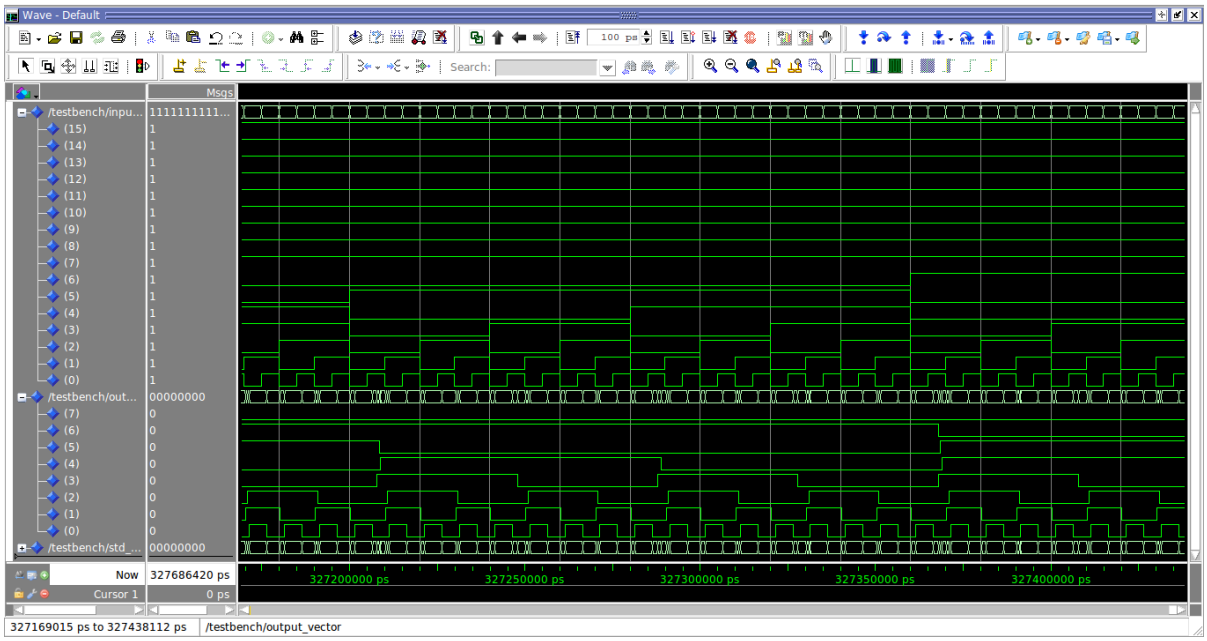

Figure 5: Gate-level Simulation of the Subtraction operation

2.3 Logical Right Shift

Figures 6-7 show the results of the simulation. *(Since the number of possible test cases is too large (2^{16}), I have showed a small section of the waveform.)*

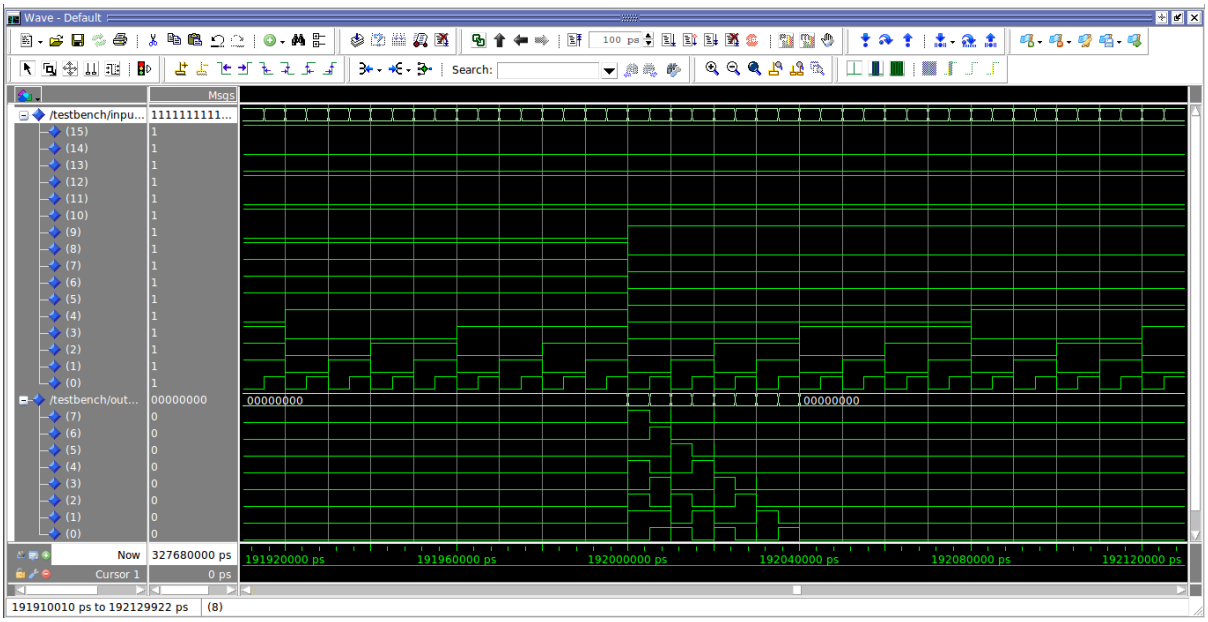

Figure 6: RTL Simulation of the Logical Right Shift operation

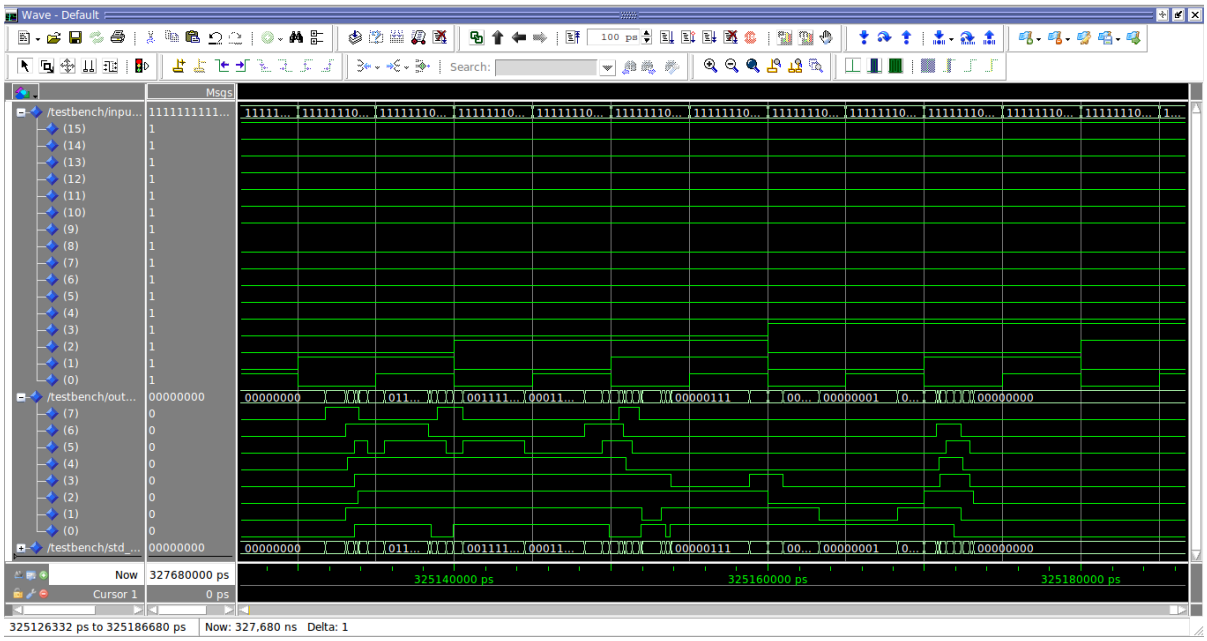

Figure 7: Gate-level Simulation of the Logical Right Shift operation

2.4 Logical Left Shift

Figures 8-9 show the results of the simulation. *(Since the number of possible test cases is too large (2^{16}), I have showed a small section of the waveform.)*

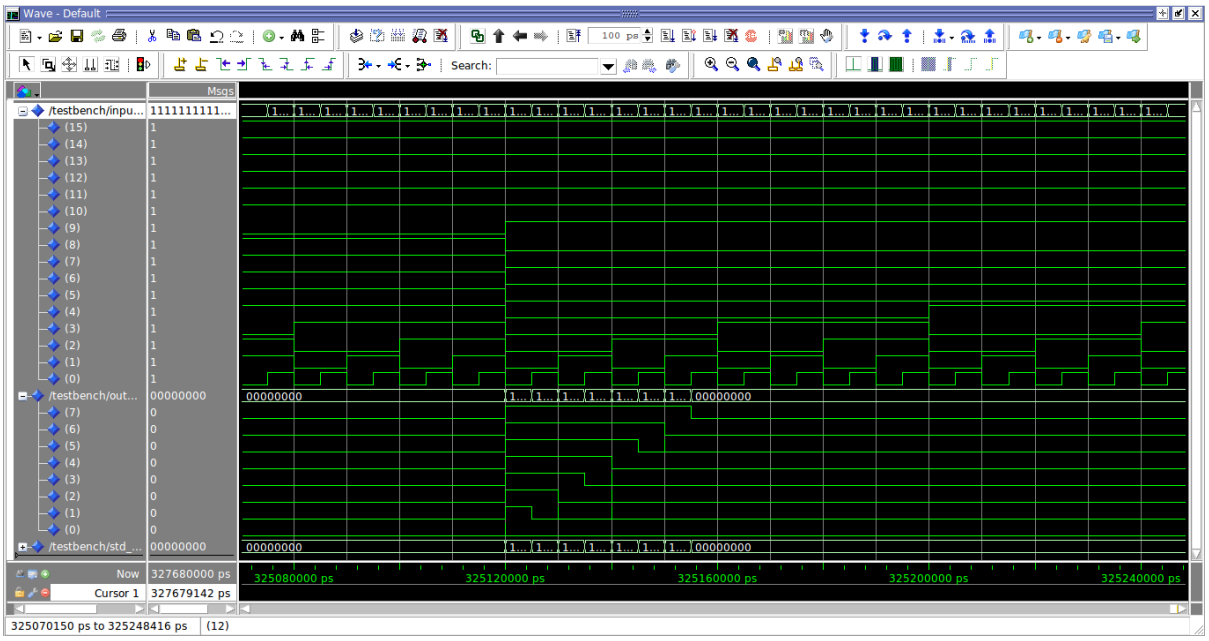

Figure 8: RTL Simulation of the Logical Left Shift operation

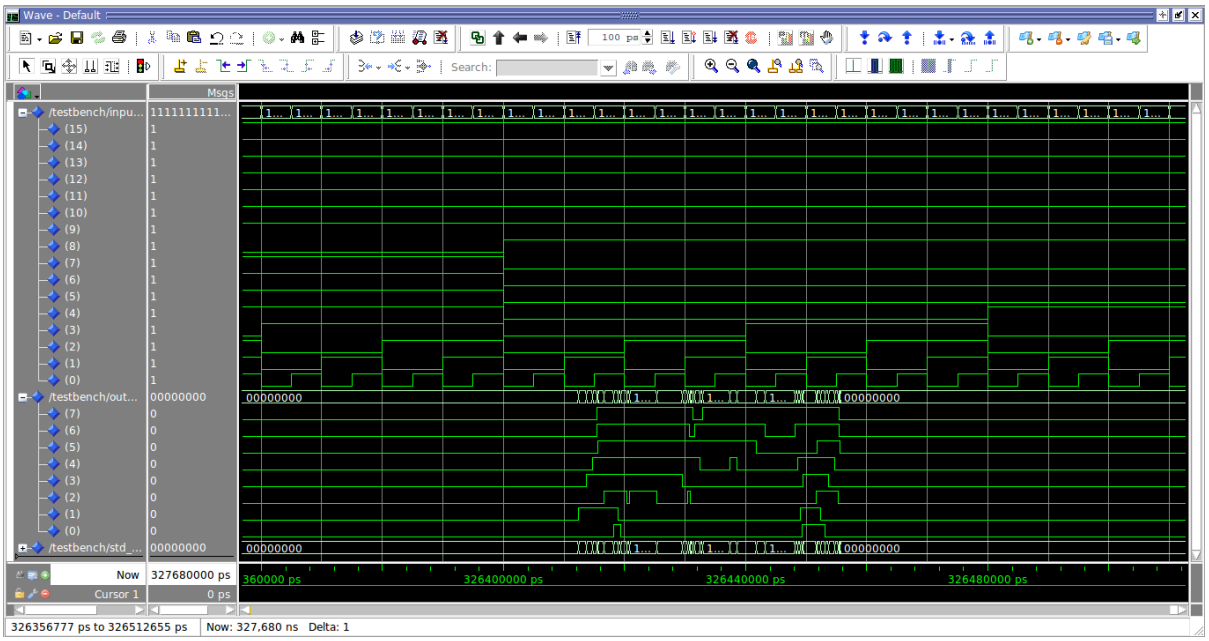

Figure 9: Gate-level Simulation of the Logical Left Shift operation

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 18 control switches. 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 first on a small sample of test cases to check common logic errors. Later on, exhaustive tests (2^{18}) were performed to check for any issues on the overall performance.

Results

Sample Tests

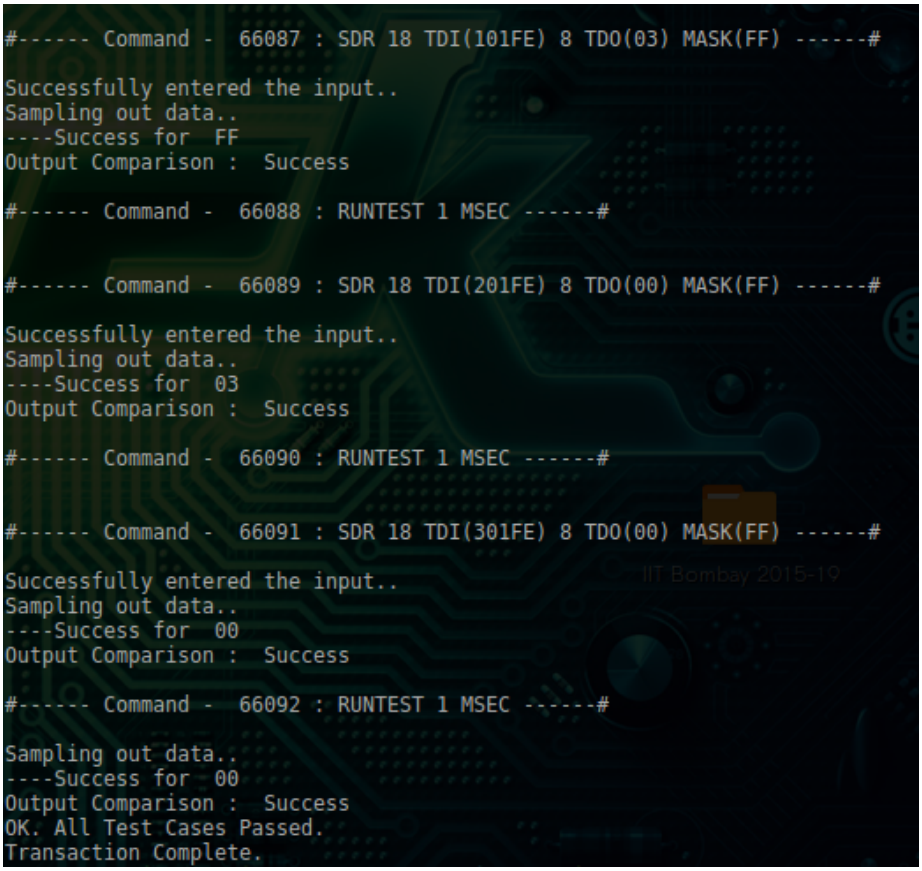

Figure 10: Results of the scan-chain test for random sample test cases.

[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.

Exhaustive Tests

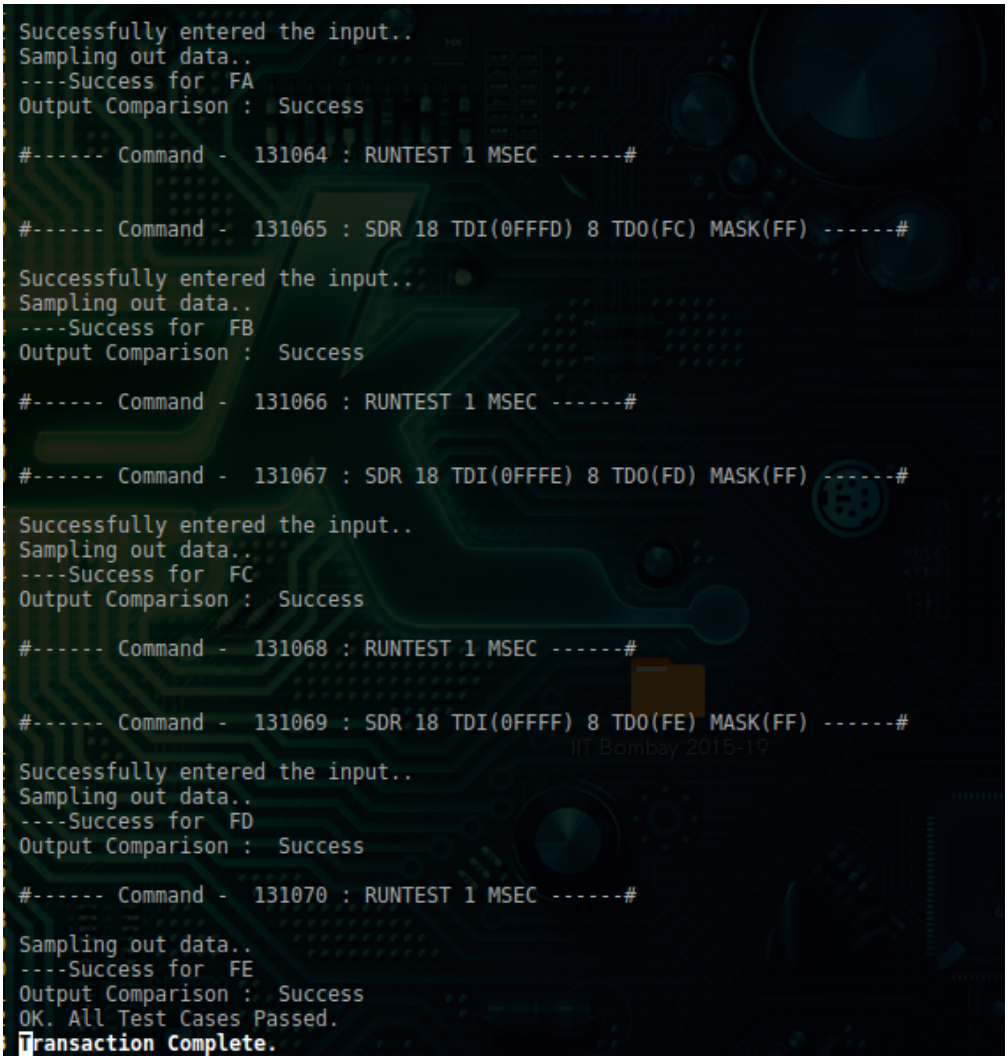

Figure 11: Results of the scan-chain test on an exhaustive set of inputs.

Conclusion

Starting from the very scratch, in this report, I have presented the logic and code for an 8-bit ALU as required. 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 ALU can be used in hardware, as required.