LAB 6: ALU CIRCUIT USING BEHAVIORAL-DATAFLOW

ARYA AGARWAL
Roll Number - 210070012

10 September 2022

1. Overview of the Experiment / Assignment:

- In this experiment, we have to watch the videos of behavioral structure.
- Then we were introduced to our lab 6 experiment. In this experiment we have to design an ALU circuit which performs various operations depending on the selection inputs.
- Then using Quartus and tracefile provided we have to write our VHDL code for our design.In this design we have to only use behavioral dataflow.
- After writing the VHDL code , we have to check our design using Scanchain.

2. Experiment Setup or Approach to the Assignment:

- In this experiment we have to take 8 inputs and we will correspondingly will get 8 outputs. In this ALU circuit we will perform 4 different kind of operations depending upon the selection input bits. We were told to take the 2 of our input bits as our selection input bits. Below is the table which tells which operation will be performed based on the selection input. Four operations performed were Rotation, Subtraction, Multiplication and bitwise XOR operation.
- We were given an incomplete code which contained some hints and basics of our VHDL code. We have to complete that code and then check our program using SCANCHAIN.

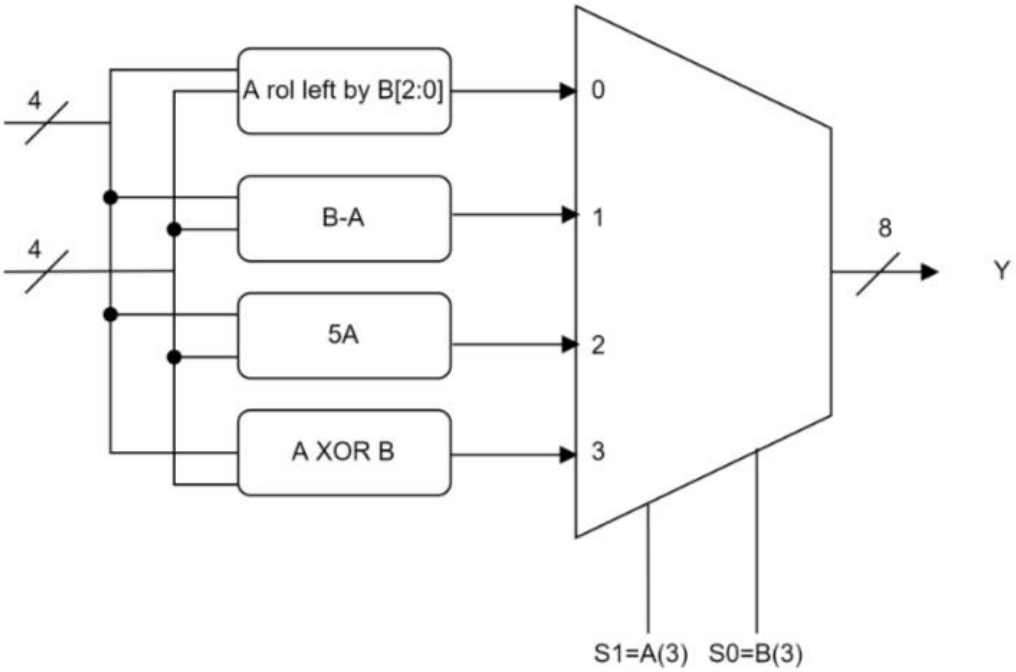


Figure 1: ALU with 4 functions

S1 S0	ALU Output
0 0	Rotate left A by B[2:0] number of bits
0 1	Performs B-A Operation
1 0	Produces output as 5*A
1 1	Performs bitwise A xor B Operation

2.1 Design Code and Documentation:

Now once we understood all the functions and their logic, it's time to describe the circuit using code in VHDL language in Quartus. The DUT, gates files will almost be same except some minor changes in both cases.

The code for the main file is written below which is used to describe the circuit.

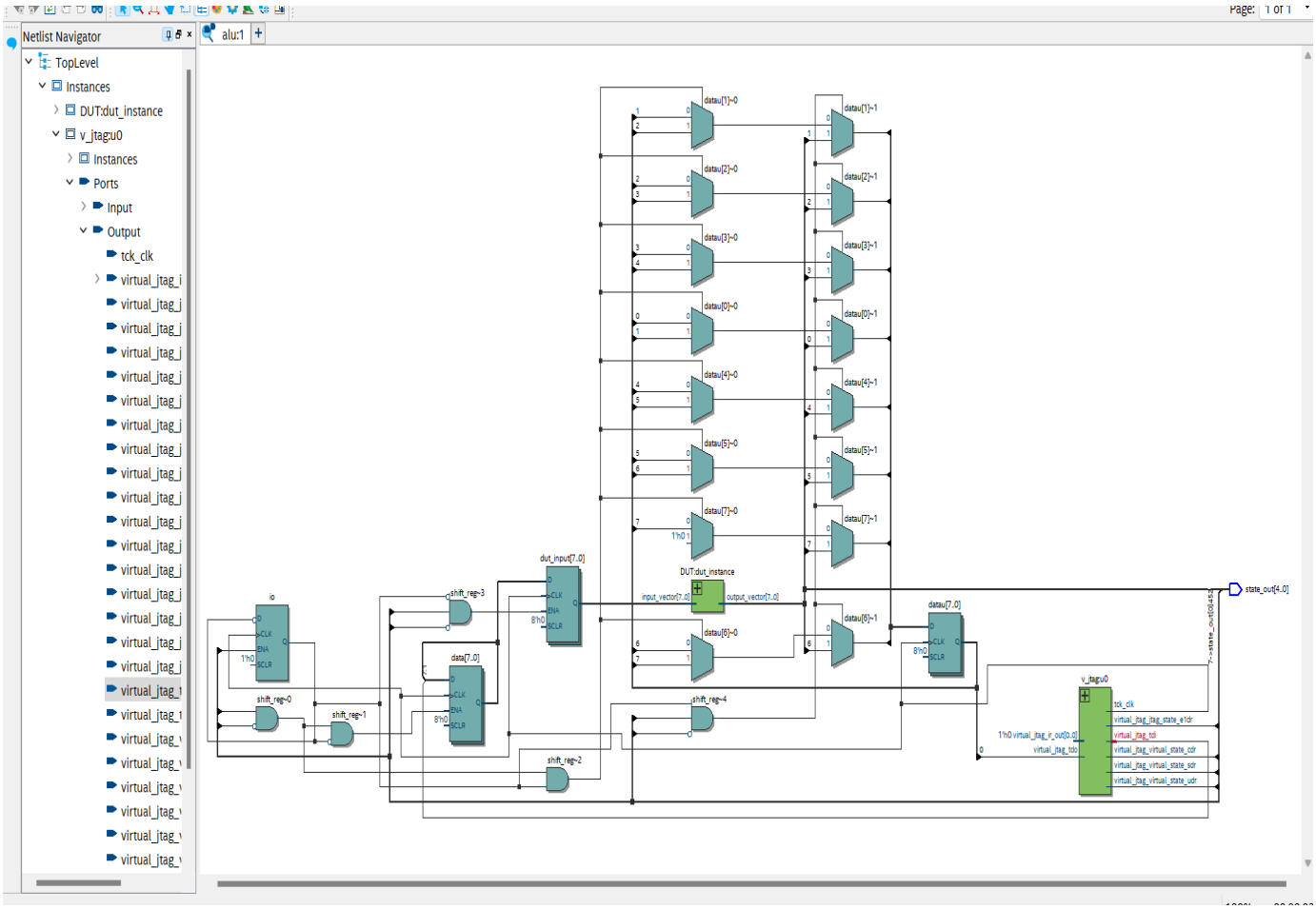
```
library ieee;
use ieee.std_logic_1164.all;


entity alu_beh is
generic(
operand_width : integer:=4);
port (
A: in std_logic_vector(operand_width-1 downto 0);
B: in std_logic_vector(operand_width-1 downto 0);
op: out std_logic_vector((operand_width*2)-1 downto 0)) ;
end alu_beh;


architecture a1 of alu_beh is


function sub(A: in std_logic_vector(operand_width-1 downto 0);
                B: in std_logic_vector(operand_width-1 downto 0))
return std_logic_vector is
-- declaring and initializing variables using aggregates
    variable diff : std_logic_vector(operand_width*2-1 downto 0):= (others=>'0');
    variable carry : std_logic:= '1';
begin
-- Hint: Use for loop to calculate value of "diff" and "carry" variable
-- Use aggregates to assign values to multiple bits
    Loop_1 : for i in 0 to 3 loop
            diff(i) := B(i) xor ('1' xor A(i)) xor carry;
            carry := (B(i) and carry) or (('1' xor A(i)) and (B(i) or carry));

    end loop Loop_1;


return diff;
end sub;
```

```
function rolf(A: in std_logic_vector(operand_width-1 downto 0);
                B: in std_logic_vector(operand_width-1 downto 0))

return std_logic_vector is
    variable shift : std_logic_vector((operand_width*2)-1 downto 0):= (others=>'0');
    variable r : integer := 0;
    variable temp : std_logic_vector((operand_width*2)-1 downto 0):= (others=>'0');
    begin

    shift(operand_width-1 downto 0):= A;

    Loop_3 : for i in 0 to 2 loop

        temp := shift;

        if B(i) = '1' then
            r := (2**i);
        else
            r := 0;
        end if;

        Loop_4 : for j in 0 to 7 loop
            if (j+r)>7 then
                shift(j+r-8) := temp(j);
            else
                shift(j+r) := temp(j);

            end if;

        end loop Loop_4;

    end loop Loop_3;
```

```
-- Hint: use for loop to calculate value of shift variable
-- shift(____ downto ___) & shift(__ downto _____)
-- to calculate exponent, you can use double asterisk. ex: 2**i

return shift;
end rolf;


begin
alu : process( A, B)
begin
-- complete VHDL code for various outputs of ALU based on select lines
-- Hint: use if/else statement

    F1 : if ((A(3)='0')and(B(3)='0')) then
                op <= rolf (A,B);

            elsif ((A(3)='0')and(B(3)='1')) then
                op <= sub (A,B);

            elsif ((A(3)='1')and(B(3)='0')) then
                op <= (others => '0');
                op(0) <= A(0);
                op(1) <= A(1);
                op(2) <= A(0) xor A(2);
                op(3) <= A(1) xor A(3) xor (A(0) and A(2));
                op(4) <= A(2) xor ((A(1) and A(3))or(A(3) and A(0) and A(2))or(A(1) and A(0) and A(2)));
                op(5) <= A(3) xor (A(2) and ((A(1) and A(3))or(A(3) and A(0) and A(2))or(A(1) and A(0) and A(2))));
                op(6) <= A(3) and (A(2) and ((A(1) and A(3))or(A(3) and A(0) and A(2))or(A(1) and A(0) and A(2))));
```

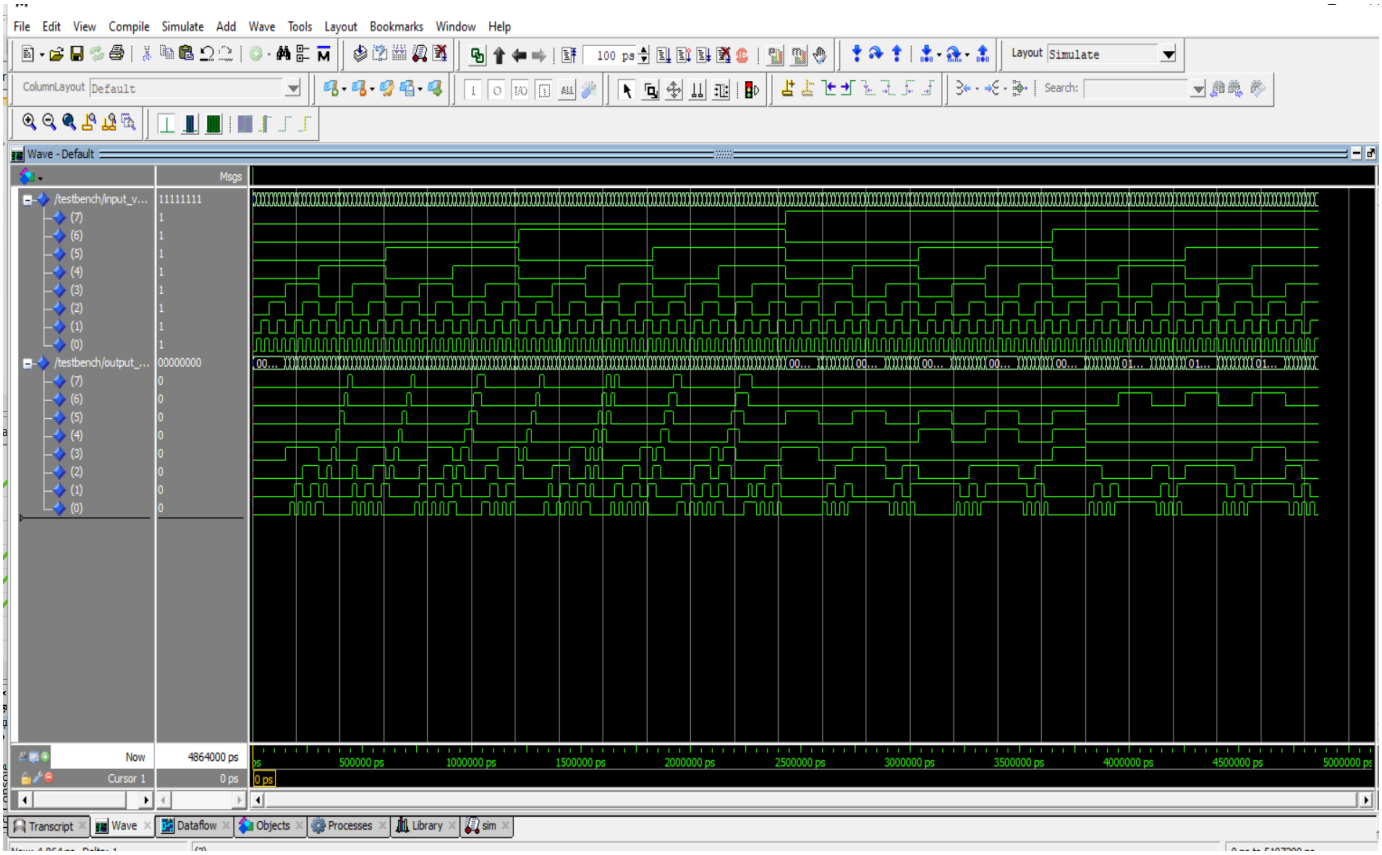
```
            else
                op <= (others => '0');
                op(0) <= A(0) xor B(0);
                op(1) <= A(1) xor B(1);
                op(2) <= A(2) xor B(2);
                op(3) <= A(3) xor B(3);


            end if;
--
-- sub function usage :
-- signal_name <= sub(A,B)
-- variable_name := sub(A,B)
--
-- concatenate operator usage:
-- "0000"&A
end process ; --alu
end a1 ; -- a1
```

2.2 RTL VIEW

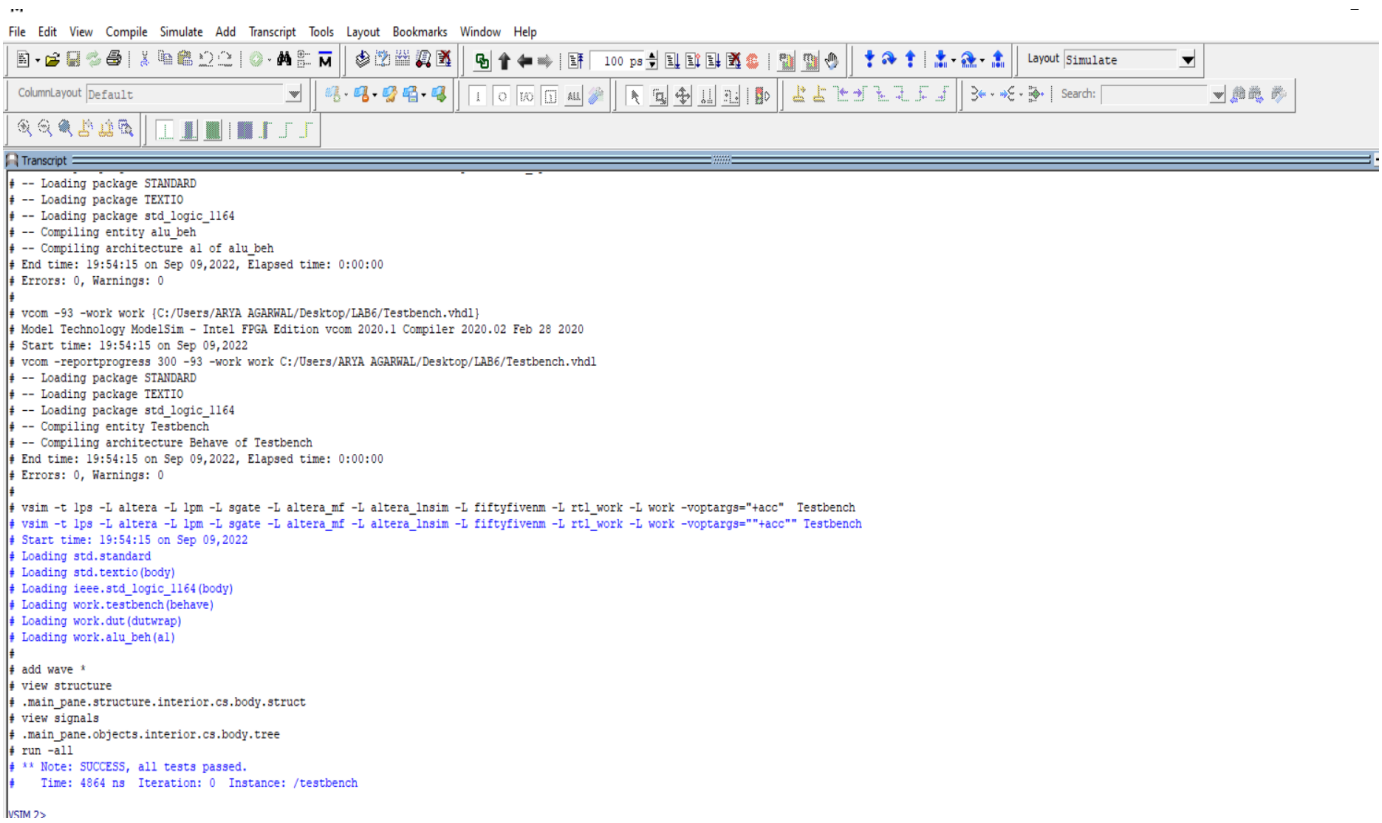


3. Observations:

After you run the analysis of code with no error. The next step is to run the simulation. For this we again use the MODELSIM - Altera Software. We also need the Testbench and the TRACEFILE (already provided) to run the simulation. The simulation is shown below.



3.1 TRANSCRIPT(all test cases passed successfully)



4. SCANCHAIN

After running the code successfully, we have to connect our design to Xenon. Then we were provided with the SCANCHAIN files. After adding the Top-level file from the SCANCHAIN files to our main code file. Then making some minor changes in the top level file and then set it as top level entity. Then Compile the whole design. Once the design is compiled make the .svf file and throw it in our URJTAG after connecting the Xen10 board to our PC. Before Using the Xenon Board, it should be checked thoroughly. We have to perform all the tests before we use it. Once the Testing is done we can throw our svf file in our Xen10 board using URJTAG. Then using SCANCHAIN we can check our logic.

The benefit of using SCANCHAIN over Xen10 is that if we want to check a 9 output bit device we can't check it using Xen10 board because it only contains 8 LEDs. Another benefit is that SCANCHAIN is much faster than Xen10.

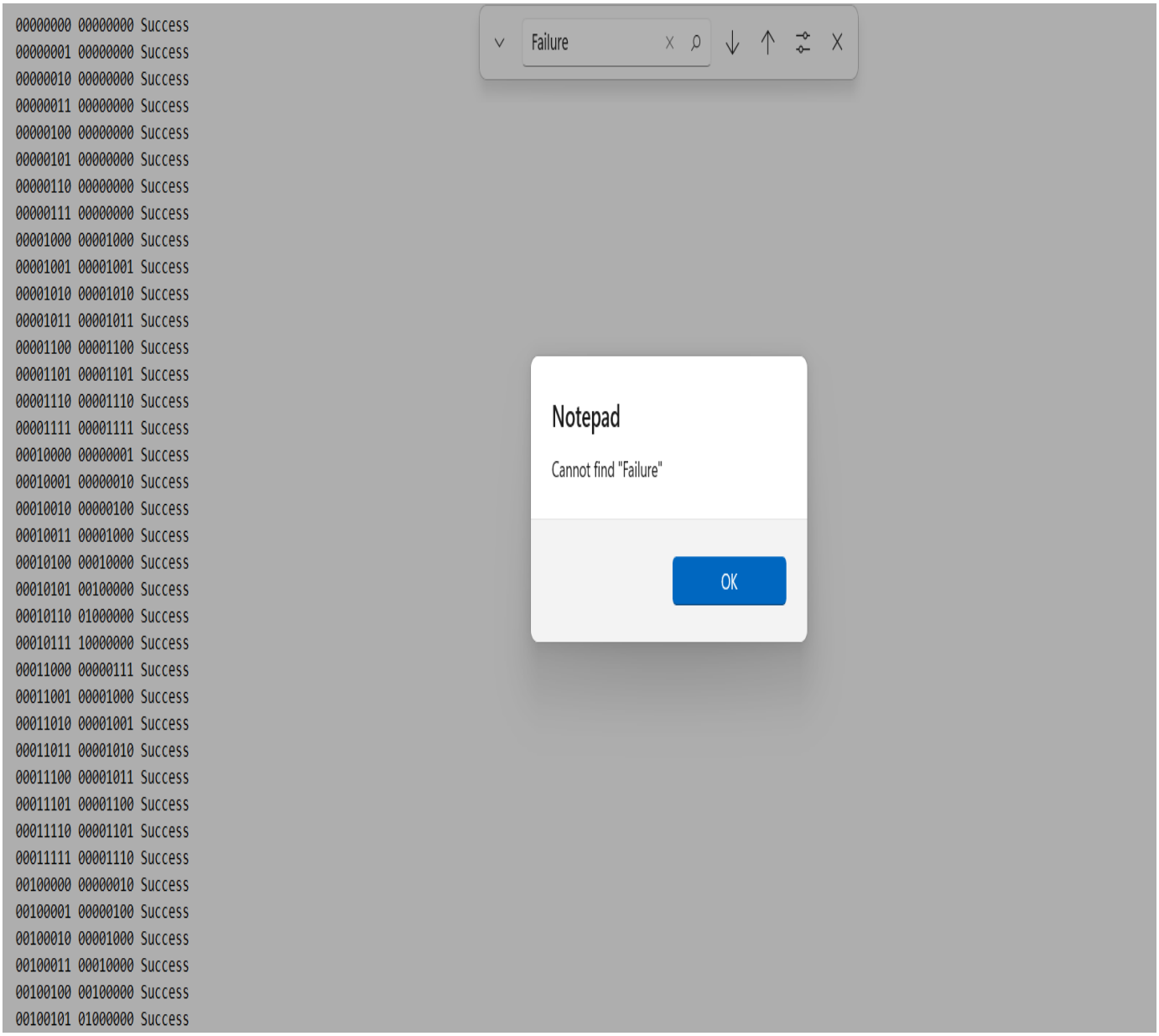
After doing all the necessary steps SCANCHAIN will create an output file which will show how many cases passed.

Here is my output file. My design did not contain any test case failure.

File Edit View

Failure

```
00000000 00000000 Success
00000001 00000000 Success
00000010 00000000 Success
00000011 00000000 Success
00000100 00000000 Success
00000101 00000000 Success
00000110 00000000 Success
00000111 00000000 Success
00001000 00001000 Success
00001001 00001001 Success
00001010 00001010 Success
00001011 00001011 Success
00001100 00001100 Success
00001101 00001101 Success
00001110 00001110 Success
00001111 00001111 Success
00010000 00000001 Success
00010001 00000010 Success
00010010 00000100 Success
00010011 00001000 Success
00010100 00010000 Success
00010101 00100000 Success
00010110 01000000 Success
00010111 10000000 Success
00011000 00000111 Success
00011001 00001000 Success
00011010 00001001 Success
00011011 00001010 Success
00011100 00001011 Success
00011101 00001100 Success
00011110 00001101 Success
00011111 00001110 Success
00100000 00000010 Success
00100001 00000100 Success
00100010 00001000 Success
00100011 00010000 Success
00100100 00100000 Success
00100101 01000000 Success
00100110 10000000 Success
00100111 00000001 Success
```


00000000 00000000 Success
00000001 00000000 Success
00000010 00000000 Success
00000011 00000000 Success
00000100 00000000 Success
00000101 00000000 Success
00000110 00000000 Success
00000111 00000000 Success
00001000 00001000 Success
00001001 00001001 Success
00001010 00001010 Success
00001011 00001011 Success
00001100 00001100 Success
00001101 00001101 Success
00001110 00001110 Success
00001111 00001111 Success
00010000 00000001 Success
00010001 00000010 Success
00010010 00000100 Success
00010011 00001000 Success
00010100 00010000 Success
00010101 00100000 Success
00010110 01000000 Success
00010111 10000000 Success
00011000 00000111 Success
00011001 00001000 Success
00011010 00001001 Success
00011011 00001010 Success
00011100 00001011 Success
00011101 00001100 Success
00011110 00001101 Success
00011111 00001110 Success
00100000 00000010 Success
00100001 00000100 Success
00100010 00001000 Success
00100011 00010000 Success
00100100 00100000 Success
00100101 01000000 Success
Failure
Notepad
Cannot find "Failure"
OK