University of Ioannina

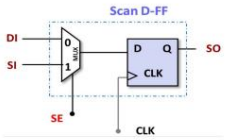
Department of Electronics Engineering
Computers & Informatics

Testing and Reliability of Electronic Systems

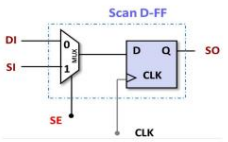
Spring Semester 2024

Scan Testing Circuit

Christos Dimitressis **4351** **cs04351@uoi.gr**



Contents



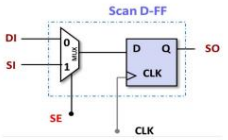
Question 1.1:
Implementing Testable-Ready Circuit Under Test

Creating a Scan D-Flip Flop

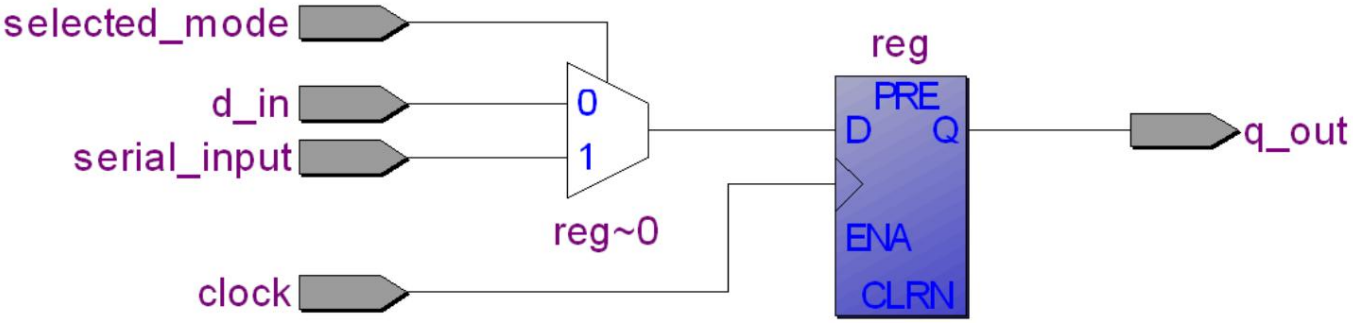
The code code_snippet_01 creates a Scan D Flip Flop in VHDL language with the requested functionality to be used later in the circuit testing using the Scan Testing method.

```
library IEEE; use
IEEE.std_logic_1164.all;

entity scanDFF is port( clock d_in serial_input                    : in std_logic; : in
                                                  q_out              std_logic;
                                                  selected_mode          : in std_logic; : out
                                                                     std_logic;
                                                                         : in std_logic);
end entity;

architecture behavior of scanDFF is

      signal reg : std_logic := 'X';

begin

      process(clock) begin if

      rising_edge(clock) then case selected_mode is when '0' =>
                        reg <= d_in; => reg <= serial_input; when
                              '1' when others => reg <= 'X'; end case? endif? end
                                                    process?




      q_out <= reg;

end architecture;
```

code_snippet_01



By synthesizing the circuit code_snippet_01 in quartus, the following circuit (screen_01) is obtained at the RTL level, which confirms the correctness of its design.



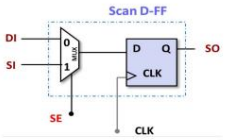
screen_01

—----------

Create Circuit Under Test (CUT)

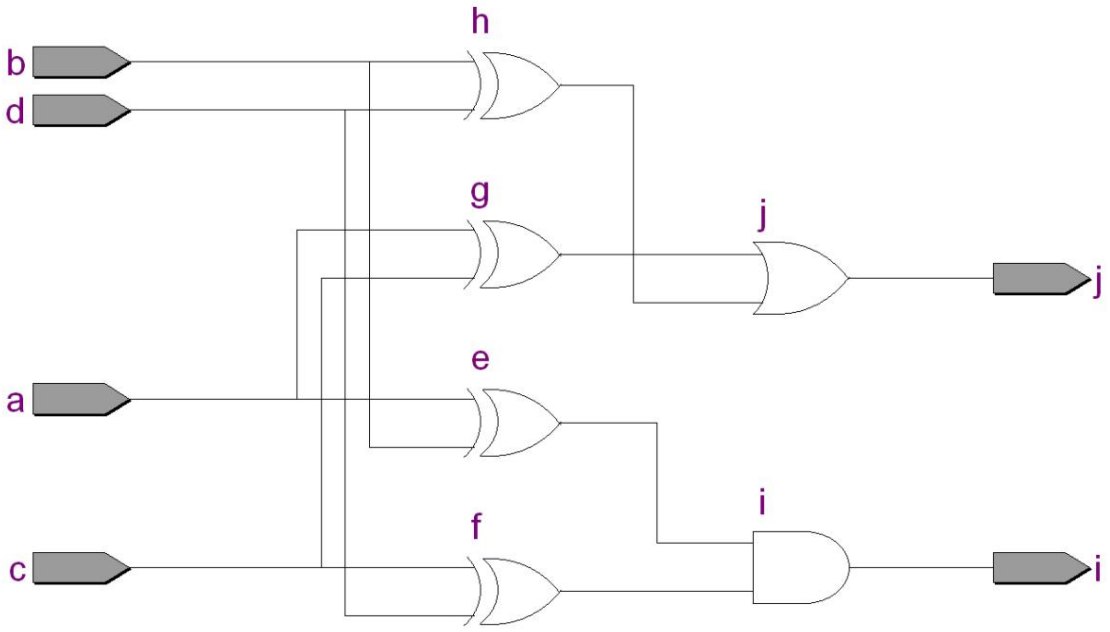
The code code_snippet_02 creates the requested CUT circuit

By synthesizing the circuit code_snippet_02 in quartus, the following design (screen_02) is obtained at the RTL level. The identification of the design at the RTL level of quartus with the image of the pronunciation confirms the correctness of the circuit.

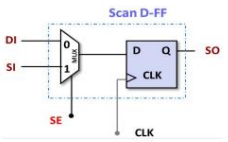



```
library IEEE; use
IEEE.std_logic_1164.all;

entity Circuit_Under_Test is port( a : in std_logic; b : in std_logic; c : in std_logic; d :
                                                                    in std_logic; i : out
                                                                    std_logic; j : out std_logic);



end entity;

architecture dataFlow of Circuit_Under_Test is

    signal e, f, g, h : std_logic;

begin

    -- First Level -- e <= a xor b; f <=
    c xor d; g <= a xor c; h
    <= b xor d;



    -- Second Level -- i <= e and f; j <=
    g or h?


end architecture dataFlow;
```

code_snippet_02

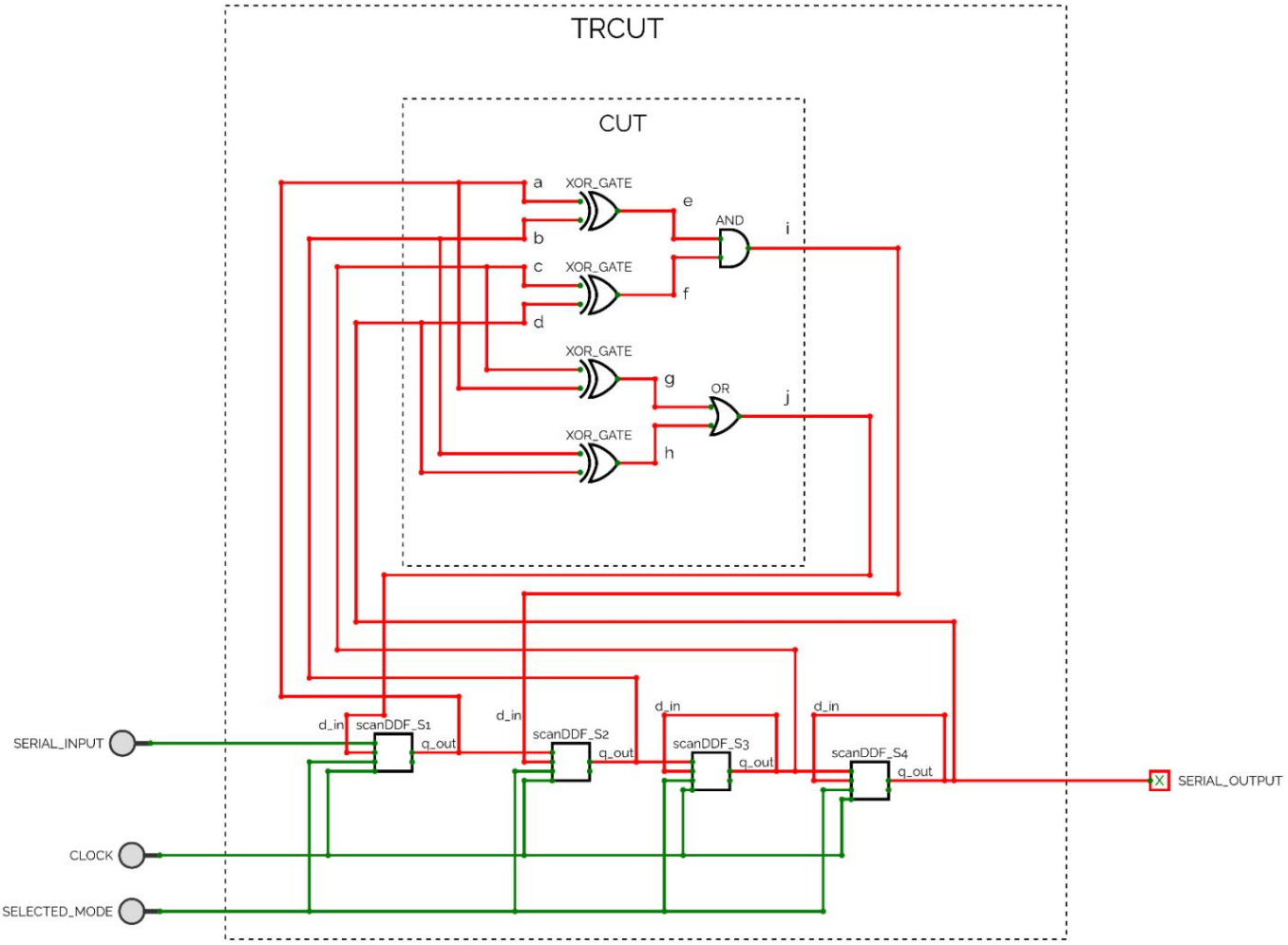


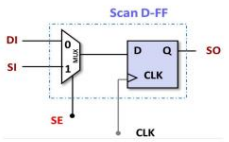
screen_02



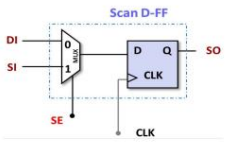
Creating a Testable-Ready Circuit Under Test (TRCUT)

The code code_snippet_03 creates the TRCUT circuit in VHDL with the functionality of the speech. The schematic below explains its topology. Essentially we connect the CUT we created previously with 4 scanDFFs of the form we created previously. That is, the TRCUT consists of the CUT connected appropriately with 4 scanDFFs which constitute the scan chain.



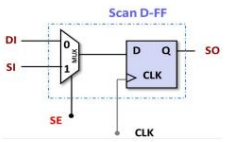



```
library IEEE; use
IEEE.std_logic_1164.all;

entity Test_Ready_Circuit_Under_Test is port( clock                                    : in std_logic;
                                                serial_input : in std_logic; selected_mode :
                                                in std_logic; serial_output : out std_logic);


end entity;


architecture dataFlow of Test_Ready_Circuit_Under_Test is

    component scanDFF is port
        ( clock :
            in std_logic; d_in : in std_logic;
            serial_input : in std_logic;
            q_out : out std_logic; selected_mode :
            in std_logic

        );
    end component;

    component Circuit_Under_Test is port ( a, b,
        c, d : in
            std_logic; i, j : out std_logic

        );
    end component;


    signal i_signal, j_signal, dff_q_out_1, dff_q_out_2,
    dff_q_out_3, dff_q_out_4 : std_logic := 'X';


begin

    Circuit_Under_Test_inst : Circuit_Under_Test port map ( a =>
        dff_q_out_1,
            b => dff_q_out_2, c =>
            dff_q_out_3, d =>
            dff_q_out_4, i =>
            i_signal, j => j_signal


        );
```

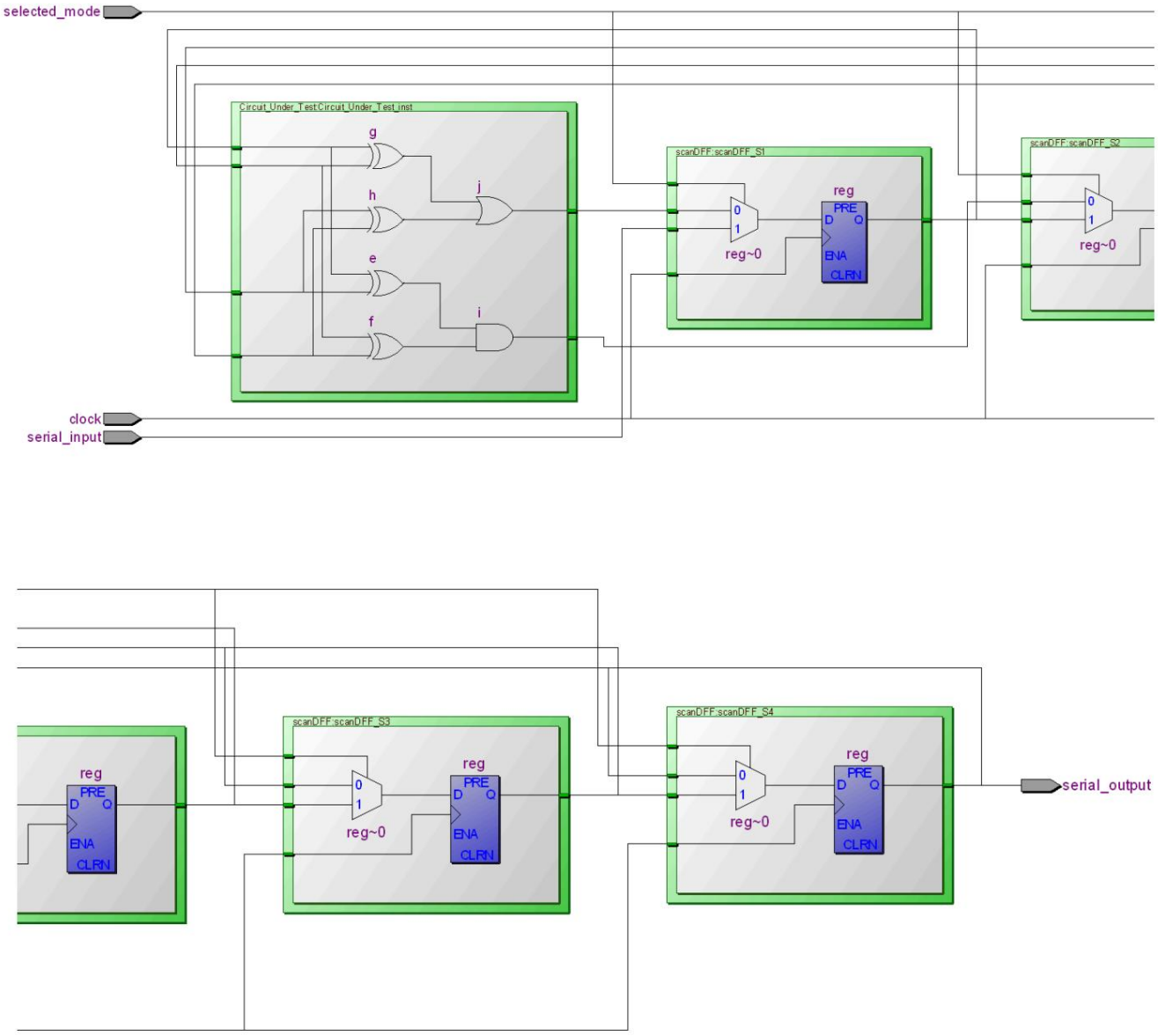



```
    scanDFF_S1 : scanDFF port
        map ( clock =>
                clock, d_in => j_signal,
                serial_input =>
                serial_input, q_out => dff_q_out_1,
                selected_mode => selected_mode

        );

    scanDFF_S2 : scanDFF port
        map ( clock =>
                clock, d_in => i_signal,
                serial_input =>
                dff_q_out_1, q_out => dff_q_out_2,
                selected_mode => selected_mode

        );

    scanDFF_S3 : scanDFF port
        map ( clock =>
                clock, d_in =>
                dff_q_out_3, serial_input =>
                dff_q_out_2, q_out => dff_q_out_3,
                selected_mode => selected_mode

        );

    scanDFF_S4 : scanDFF port
        map ( clock =>
                clock, d_in =>
                dff_q_out_4, serial_input =>
                dff_q_out_3, q_out => dff_q_out_4,
                selected_mode => selected_mode

        );

    serial_output <= dff_q_out_4;

end architecture;
```

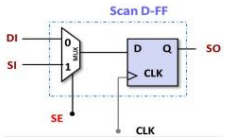
code_snippet_03



By synthesizing the circuit code_snippet_03 in quartus, the design screen_03 is obtained at the RTL level. Due to the simplicity of the circuit, we can easily visually conclude that it is identical to the design we made in the initial stage.



screen_03

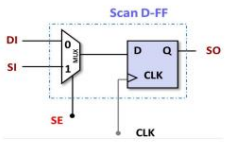


Question 1.2:
Testbench Implementation for TRCUT

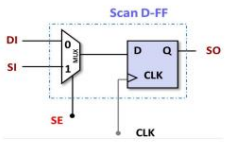
The following code (code_snippet_04) creates a testbench that serially loads the scanDFFs with the CUT truth table and, following the scan testing method, examines the correct operation of the circuit.

A detailed explanation of the simulation stages will be provided by providing snapshots of signal responses after the testbench execution.

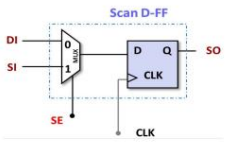
```
LIBRARY ieee ; LIBRARY
std ; USE
ieee.std_logic_1164.all ; USE ieee.std_logic_arith.all ;
USE ieee.std_logic_textio.all ; USE
ieee.STD_LOGIC_UNSIGNED.all ; USE
ieee.std_logic_unsigned.all ; USE std.textio.all ;



entity scan_testing_testbench is end entity;



architecture testbench of scan_testing_testbench is

    component Test_Ready_Circuit_Under_Test is
        port( clock
                                        : in std_logic;
            serial_input : in std_logic; selected_mode : in std_logic;
            serial_output : out std_logic

        );
    end component;

    signal serial_input, selected_mode, serial_output : std_logic := 'X'; signal clock : std_logic := '1';


    type std_logic_vector_array is array (0 to 15) of std_logic_vector(3 downto 0); signal truth_table : std_logic_vector_array;


    type std_logic_vector_array1 is array (0 to 15) of std_logic_vector(3 downto 0); signal serial_output_log : std_logic_vector_array1;



    signal tb_line: std_logic_vector(3 downto 0) := "0000";
```




```
    signal first_execution : boolean := true;

    signal correct_circuit_output : std_logic := 'X';

    signal test_bit : std_logic;

    signal a,b,c,d,e,f,g,h,ii,j : std_logic := 'X';

begin

    DUT : Test_Ready_Circuit_Under_Test port map ( clock => clock,
            serial_input =>
                    serial_input, selected_mode => selected_mode,
                    serial_output => serial_output);



    clock <= not clocked after 2.5 ns;



    The process is
    beginning.


            if first_execution then first_execution <= false;


                    -- truth table initialization for i in 0 to 15 loop truth_table(i)
                    <= tb_line; tb_line <= tb_line + '1'; wait
                            for 5 ns? end loop?




                    -- Checking serial is working wait for 2.5 ns?
                    serial_input <= '0';
                    selected_mode <= '1'; wait for 5
                    ns?


                    serial_input <= not serial_input; wait for 5 ns?


                    serial_input <= not serial_input; wait for 5 ns?


                    serial_input <= not serial_input; wait for 5 ns?
```

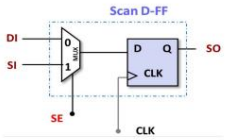



```
                serial_input <= 'X'; wait for 5 ns?


                -- Scan Testing for i in 0 to
                15 loop
                        for j in 0 to 3 loop serial_input <=
                                truth_table(i)(j); wait for 5 ns? end loop? selected_mode <=
                                '0'; wait for 5 ns?
                        selected_mode
                        <= '1'; end loop?




                serial_input <= 'X';


        else
                wait for 5 ns;
        end if; wait;
        end
process;

The process is
beginning.

        wait for 135 ns;

        -- serial_output_log_fill for i in 0 to 15 loop

                for j in 0 to 3 loop serial_output_log(i)
                        (j) <= serial_output; wait for 5 ns? end loop? --ignore last one bit from
                        serial wait for 5 ns? end
                loop?




        --- results confirmed

        for i in 0 to 15 loop a <= truth_table(i)
                (3); b <= truth_table(i)(2); c <=
                truth_table(i)(1); d <= truth_table(i)(0);



                wait for 5 ns;

                e <= a xor b; f <= c
                xor d;
```




```
                    g <= a xor c; h <= b
                    xor d;

                    wait for 5 ns;

                    ii <= e and f; j <= g or h;


                    wait for 5 ns;


                    if (serial_output_log(i)(3) = j) and (serial_output_log(i)(2) = ii)
then
                        correct_circuit_output <= '1'; else correct_circuit_output <=
                    '0';
                        endif?


                    wait for 5 ns; end loop;


            correct_circuit_output <= 'X'; wait? end process?



end architecture;
```

code_snippet_04



Testbench screenshots and explanation

The signals that interest us in our simulation are the following:

- **clock :** The clock signal for TRCUT. A clock period of 5 ns has been selected and the clock starts initialized at logic '1'. So in our simulation starting from time t = 5 ns and every 5 ns thereafter, a positive edge of the clock is assigned.

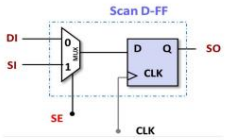
- **selected_mode :** The signal that determines the operation of the scan chain. When the signal is set to logic '0' and we are on a positive edge of the clock, the DFFs that make up the scan chain either take as input D the outputs of the circuit (CUT) or recycle the value I already had. Specifically, since our circuit has 2 outputs and the scan chain has 4 scanDFFs, **when selected_mode is at logic '0' the first DFF in the chain is selected to accept as input the value of signal j , the second scanDFF to accept as input the value while the remaining two recycle their value.** When the signal **of the signal i ,** has been set to logical , a right shift of the logical values located is performed '1' stored in the respective flip-flop.

- **serial_input :** The signal whose logic value is connected to the serial input of the first scanDFF, i.e. the input of the scan chain. Through this signal we load the scan chain with the desired logic values.

- **serial_output:** The signal whose logical value corresponds to the serial output of the scan chain, that is, at the Q output of the last scanDFF of our serial chain. Through this signal we get the responses of our circuit which we will use to confirm its operation at a later time.

- **truth_table:** This signal is essentially a 16-position table of 4-bit signals. As its name suggests, it is used to generate all possible input combinations for our circuit and load them through the scan chain to test the correct operation of the CUT.

- **scanDFFregs:** In this signal, all the scanDFFs of our scan chain are grouped in the order they are in the scan chain, so that we can visually see them as a 4-bit register and can easily see which input vector for our circuit is loaded into the scan chain at any given time, as well as whether the cut responses were successfully stored at the desired time within the scan chain.



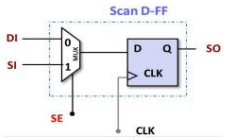
- **serial_output_log:** This signal is essentially a 16-position array which stores the logical values of the serial output of the scan chain in order to later verify the responses of our circuit. What the values it stores correspond to and how they are collected will be explained below.

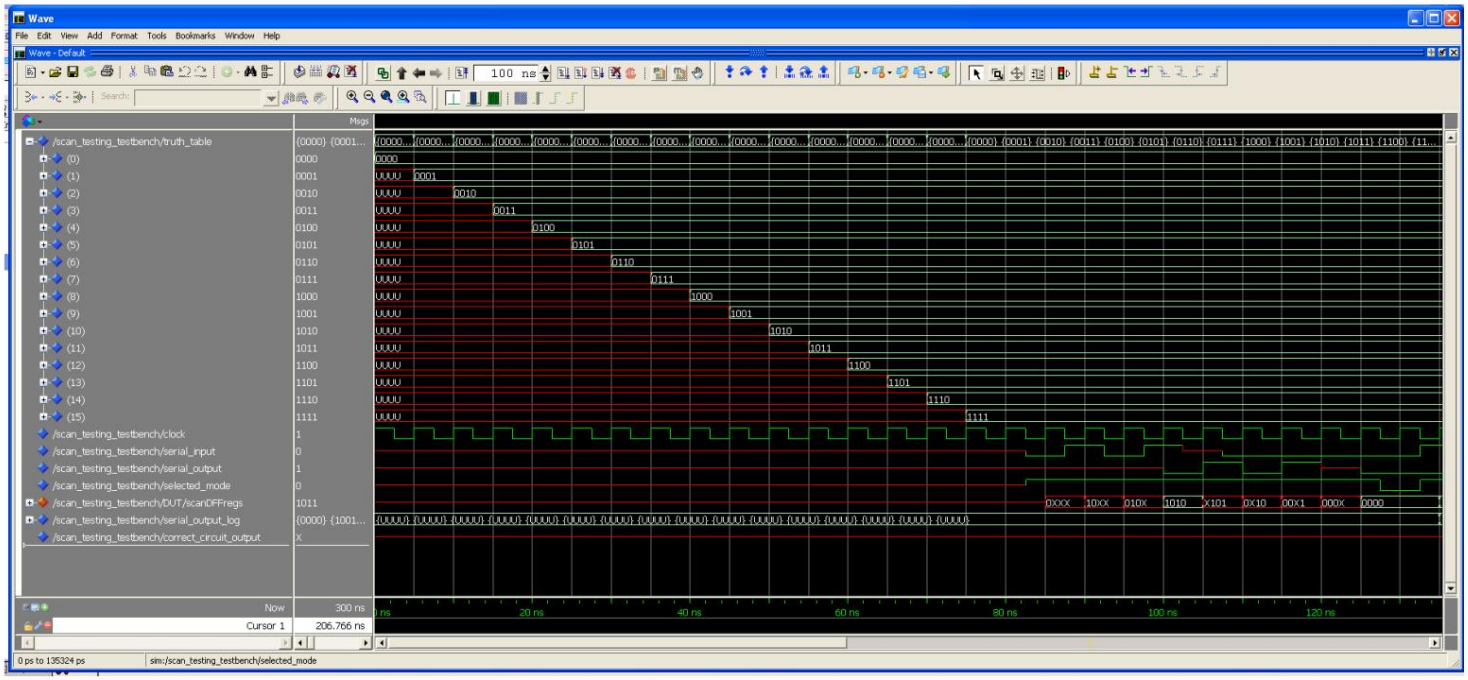
- **correct_circuit_output :** This signal is used to indicate that the values collected by the scan chain as a result of loading it with the truth table of CUT are correct. Specifically, when the scan testing is finished and the CUT responses have been collected from the scan chain, we compare the results we collected from the scan chain with the CUT outputs, receiving as input the same vectors that we loaded into the scan chain. If the CUT responses for the same vector as input are identical, we set the relevant signal to logical '1', otherwise we set it to logical '0'. Obviously, the specific signal for all the comparisons that will be made is required to be at logical '1'.

Next, snapshots from the circuit simulation using testbench will be presented.

All snapshots come from the same simulation and each is the continuation in time of the previous one.

The scale of the simulation has been chosen to have a step of 5 ns (each white line) to coincide with the operating period of the clock and to make it easy to visually understand what is happening.

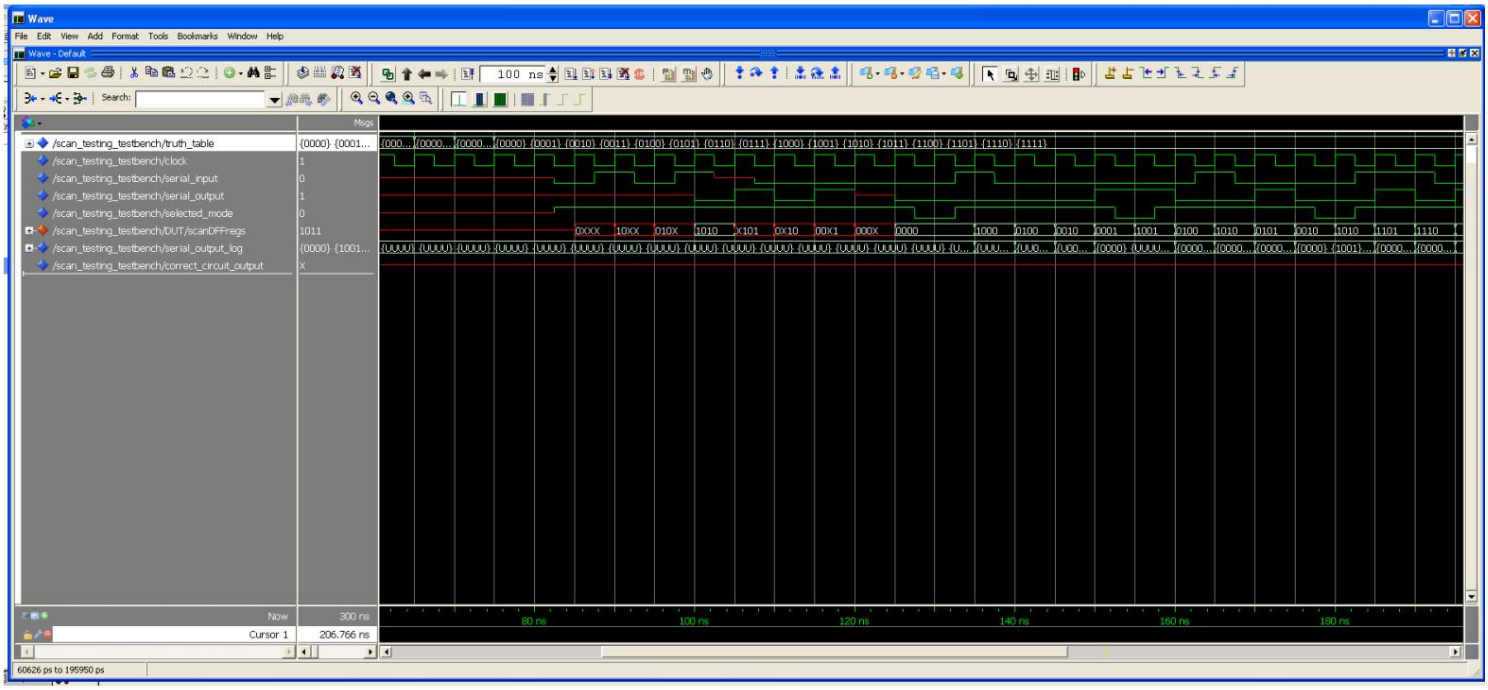




Snapshot 1

The above screenshot shows the beginning of our simulation. From time t0 = 0ns to time t1 = 75 ns, the truth table of our circuit is created and stored in the signal truth table as a table of std_logic_vectors(3 downto 0).

In code_snippet_04 , below the comment #truth table initialization, you will find the code that corresponds to the above operation.



Snapshot 2

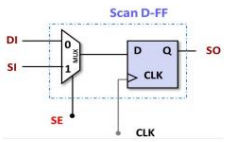
In the above snapshot at time t = 80 ns the truth table creation has been completed. The next step is to check if the right shift of our scan chain is working properly before loading the truth table and starting scan testing.

So:

At time t = 82.5 ns we set selected_mode <= '1' and the signal connected to the serial input of the scan chain to serial_input <= '0'.

So on the next positive edge of the clock at time t = 85 ns, the logical values of the serial_input signal will begin to be loaded into the scan chain.

With appropriate changes to the serial_input signal (with a phase difference of 2.5 ns from the positive edge of the clock) we load the logical vector “1010” into the scan chain, which becomes evident by observing the logical values stored due to the right shift in the scanDFFregs signal after 4 clock cycles.



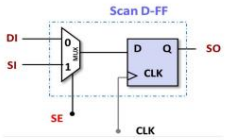
After this 4-bit vector has been loaded, we set serial_input<='X' for 5 ns (i.e. for a positive clock edge) in order to separate it and see it serially exiting the scan chain. In code_snippet_04 , below the comment #Checking serial is working, we find the code corresponding to the above operation.

—-------

After 5 ns where the serial input <= 'X' (at time t= 107.5 ns i.e.) and while it remains in selected mode <= '1' (right shift) and after we have visually confirmed that the scan chain regarding the serial shift is working as expected, we begin to load into the scan chain bit by bit vector lines of the truth table we created previously.

For each vector of the truth table of 4 bits every 5ns and having a phase difference with the clock of 2.5 ns, we set the serial_input to be equal to each bit of this vector starting from the least significant bit, that is, we load each vector of the truth table serially from right to left.

In order to understand the operation of the scan chain, the loading of the logical vector '0000' into the scan chain will be explained step by step, how the CUT response is recorded, and how the result is output for processing at a later time.



Explanation of the scan chain operation for the first input vector -> “0000”

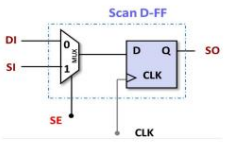
1. Positive clock edge at time t = 110 ns
- **The first bit of the vector “0000” is loaded with the right slippage in the scanning chain.**

2. Positive clock edge at time t = 115 ns
- **The second bit of the vector “0000” is loaded with the right slippage in the scanning chain.**

3. Positive clock edge at time t = 120 ns
- **The third bit of the vector “0000” is loaded with the right slippage in the scanning chain.**

4. Positive clock edge at time t = 125 ns
- **The fourth bit of the vector “0000” is loaded with the right slip in the scan chain. 5. Time instant ÿ = 127.5 ns • We change the selected_mode signal to logic '0' so that on the next positive clock edge we collect and store the CUT response to the first 2 scanFs of the scan chain. 6. Positive clock edge at time t = 130 ns • The scan chain is updated with the result of the circuit response. In the first scanDFF (having as a reference the input of the scan chain) the value of the output j of the CUT is stored while in the second the value of the output i. In the remaining 2 scanDFFs their value has simply been recycled. • Because as will be seen later, the output of CUT when it receives the logical vector “0000” as input (i.e. when a=b=c=d= '0') is j = 0 and j = 0, no visual change is seen in the values stored in the scanDFFregs signal, but the first 2 zeros correspond to the CUT response and are not the values of the input vector! ,**

7. Time instant 132.5 ns • We change the selected_mode signal to logic '1' so that on the next positive edge of the clock the next line of the truth table will start to be loaded while at the same time the least significant bit of the first vector of the truth table will start to come out of the scan chain with a right shift, "carrying" within it in the first 2 bits the response of the circuit to it. That is, for every 4 bits that come out of the scan chain, the first 2 in the output vector are the response of the CUT to some vector that was accepted as input.

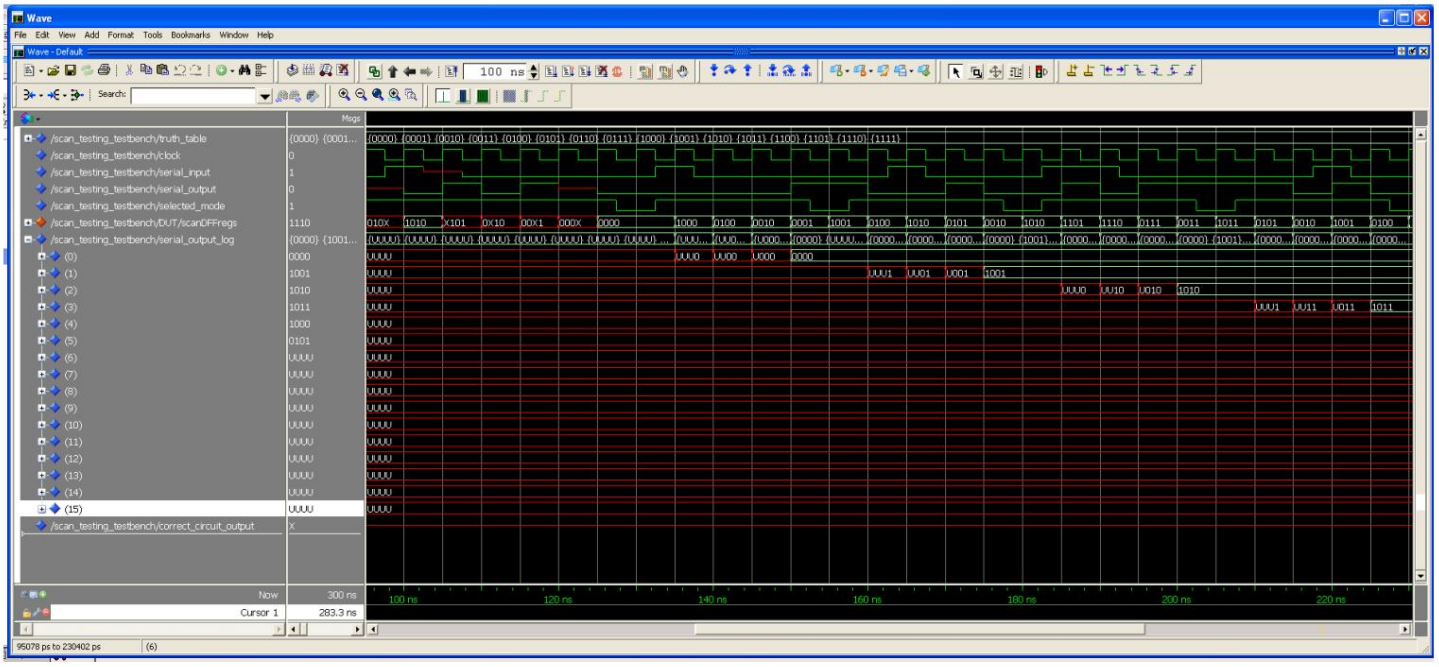


8. Time instant t = 135 ns
   - At this point the process is repeated as followed previously, starting again from step 1, with the next truth table vector, i.e. “0001”, as input to the chain until we have loaded all the truth table vectors required for a 4-input circuit.

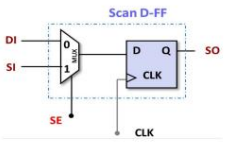
With appropriate synchronization in our test bench, we have taken care of the positive edge of the clock at time t = 135 ns and store the serial output of the
after,
of the scan chain on every positive edge of the clock except for the positive edge of the clock at any time when the chain is in capture_mode and therefore no right shift occurs but the previous value that it had in the last scanDFF as output remains constant again. That is, practically, only when a right shift occurs do we store the output of the scan chain.

In code_snippet_04 at the point after the comment #serial_output_log_fill is the code corresponding to the above operation.

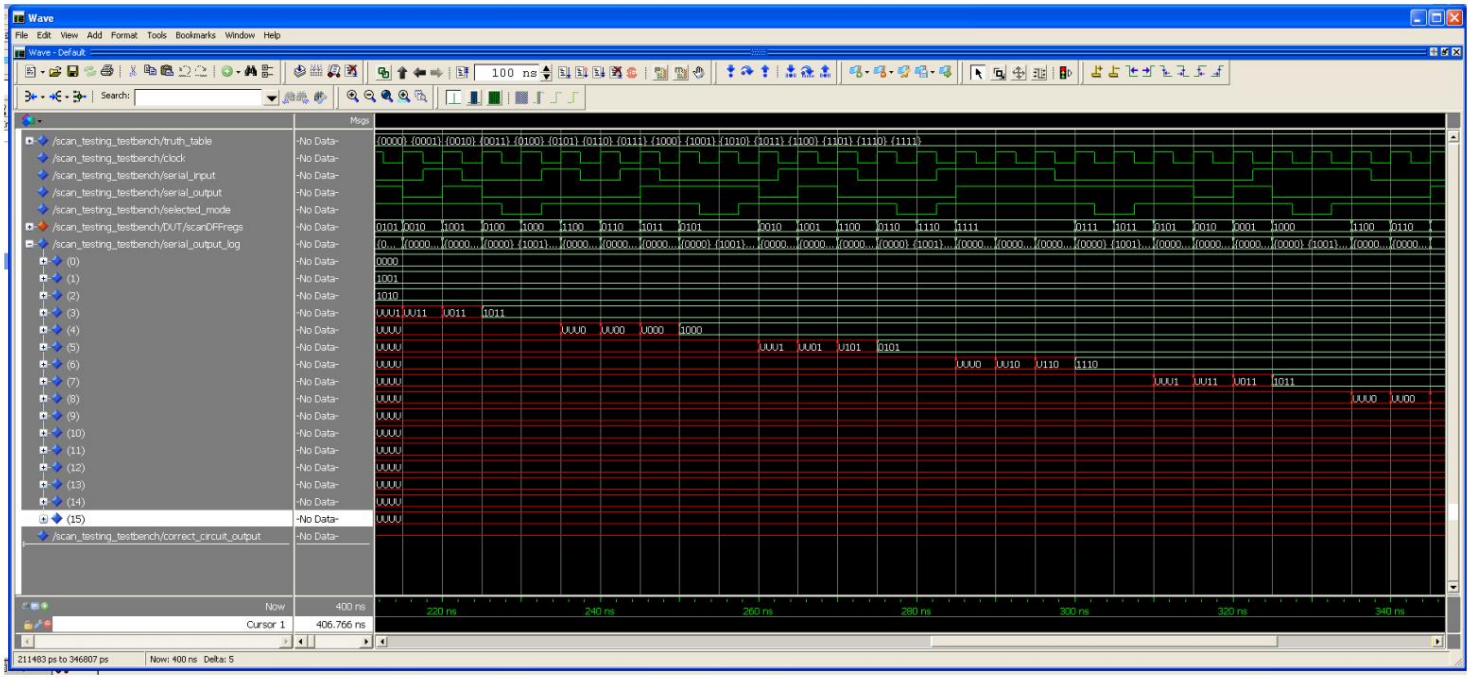
The image below shows in detail how the serial output of the scan chain is stored in the serial_output_log vector, but also how the signal at its output is ignored for one clock cycle, when we are in capture mode.



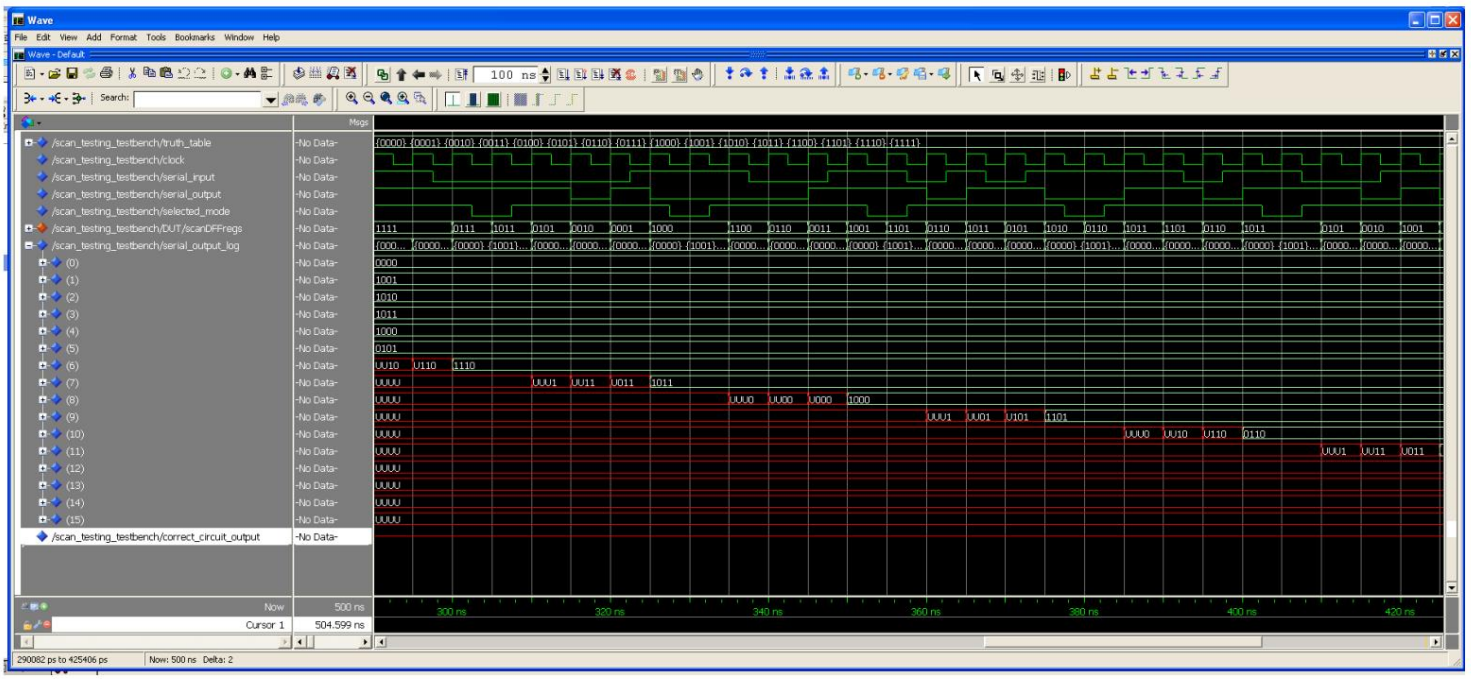
Snapshot 4



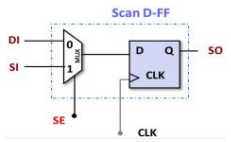
The following screenshots show the operation of the scan chain, performing all the functionality mentioned previously, for all the remaining truth table vectors.

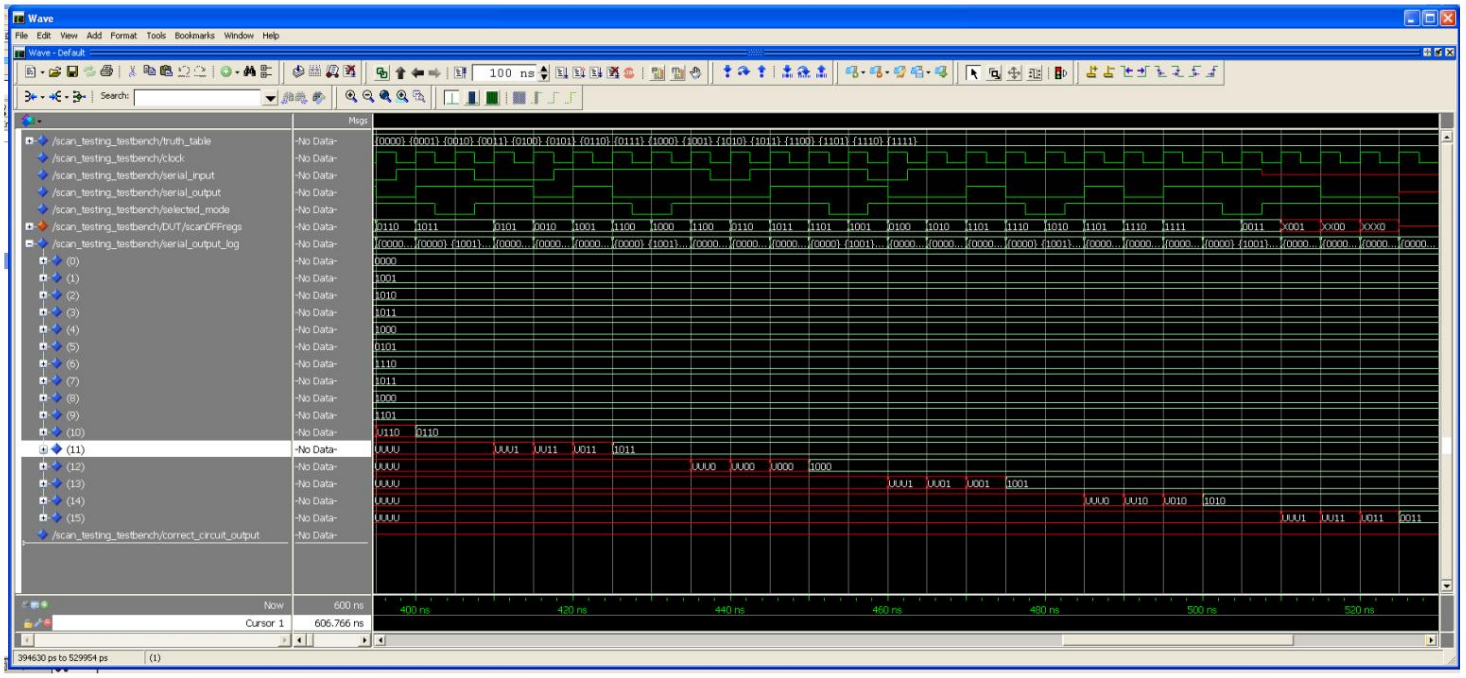


Snapshot 5

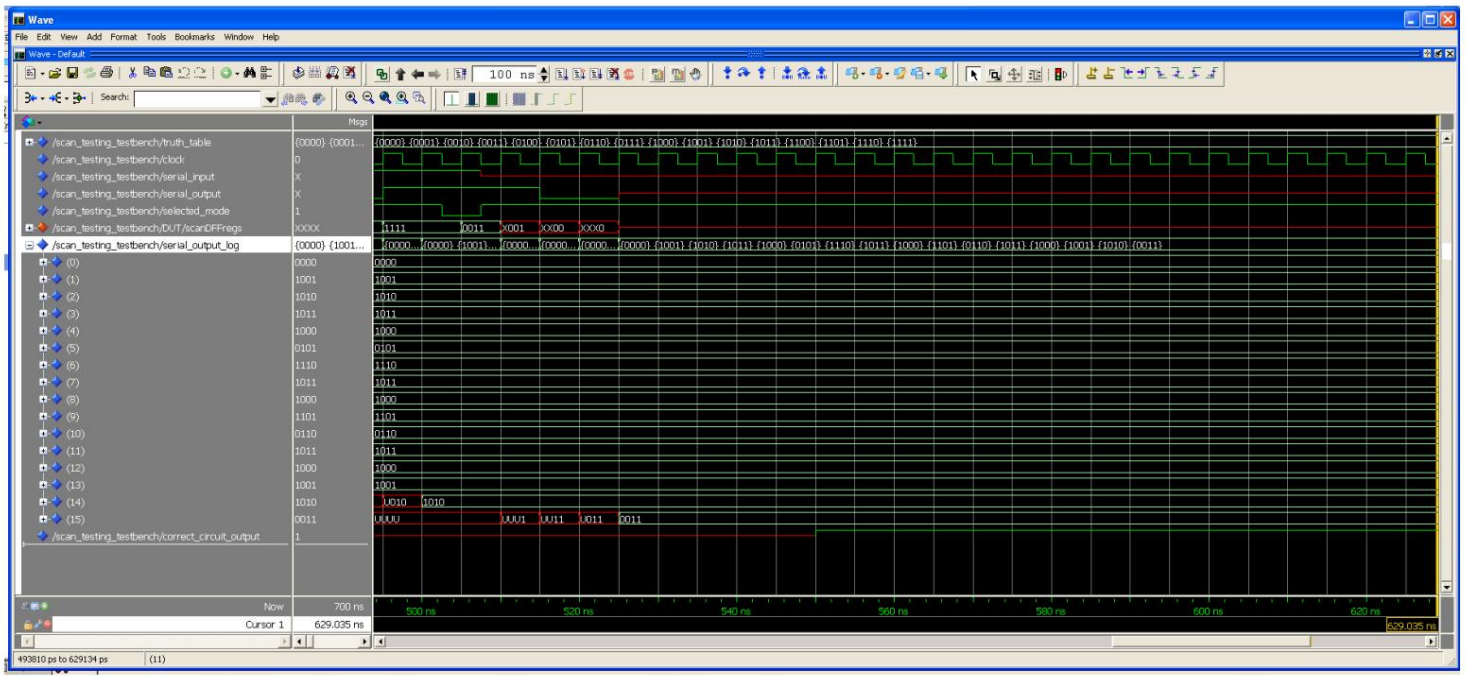


Snapshot 6

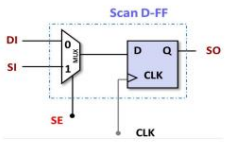




Snapshot 7



Snapshot 8



At time t = 500 ns we observe that the last vector of the truth table has been loaded into the scan chain, while at time ÿ = 525 ns all the vectors of interest have been “exited” and collected from the scan chain.

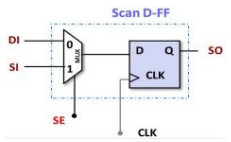
To make it more obvious where the scan chain stops accepting truth table vectors as input, after the last vector “1111” we set the serial input of the scan chain to accept the signal 'X' as input.

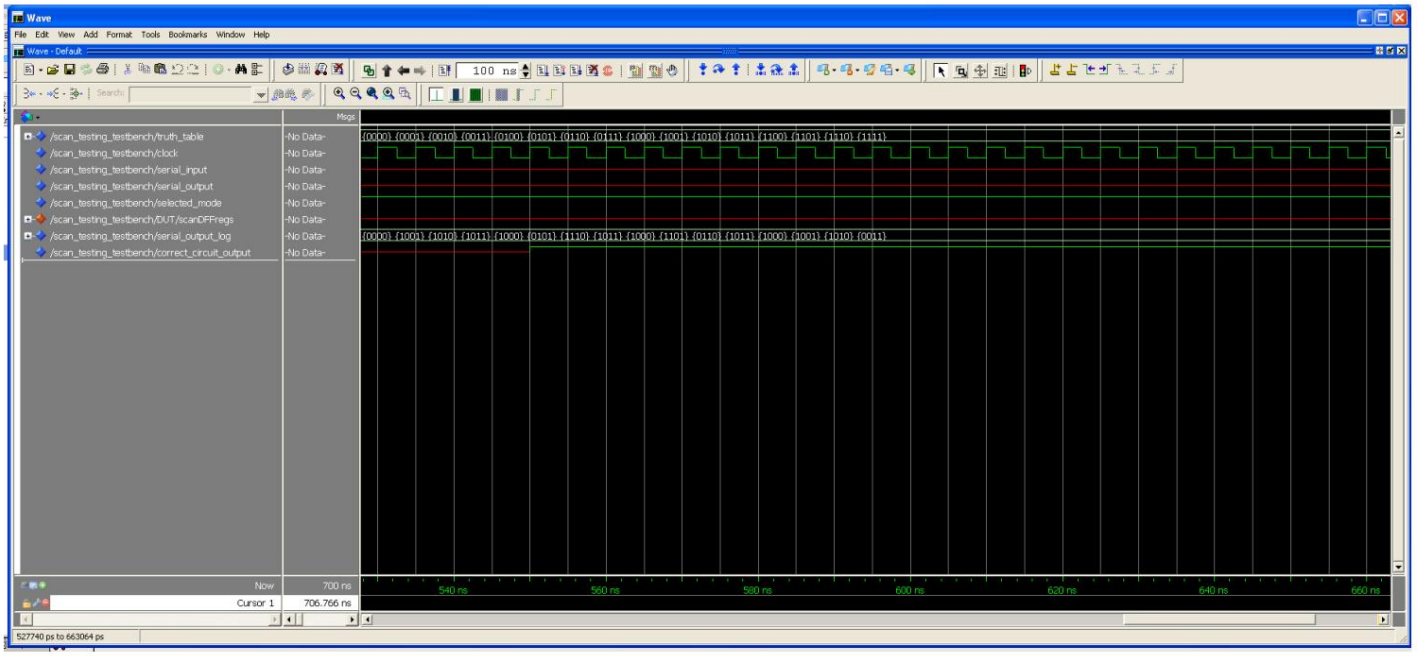
Finally, with code in the testbench, we compare the responses of the scan chain with the expected responses of the circuit and if the result is correct, we set the correct_circuit_output signal to logical '1'.

In code_snippet_04 , at the point below the comment #results confirmation, you will find the code corresponding to the above operation.

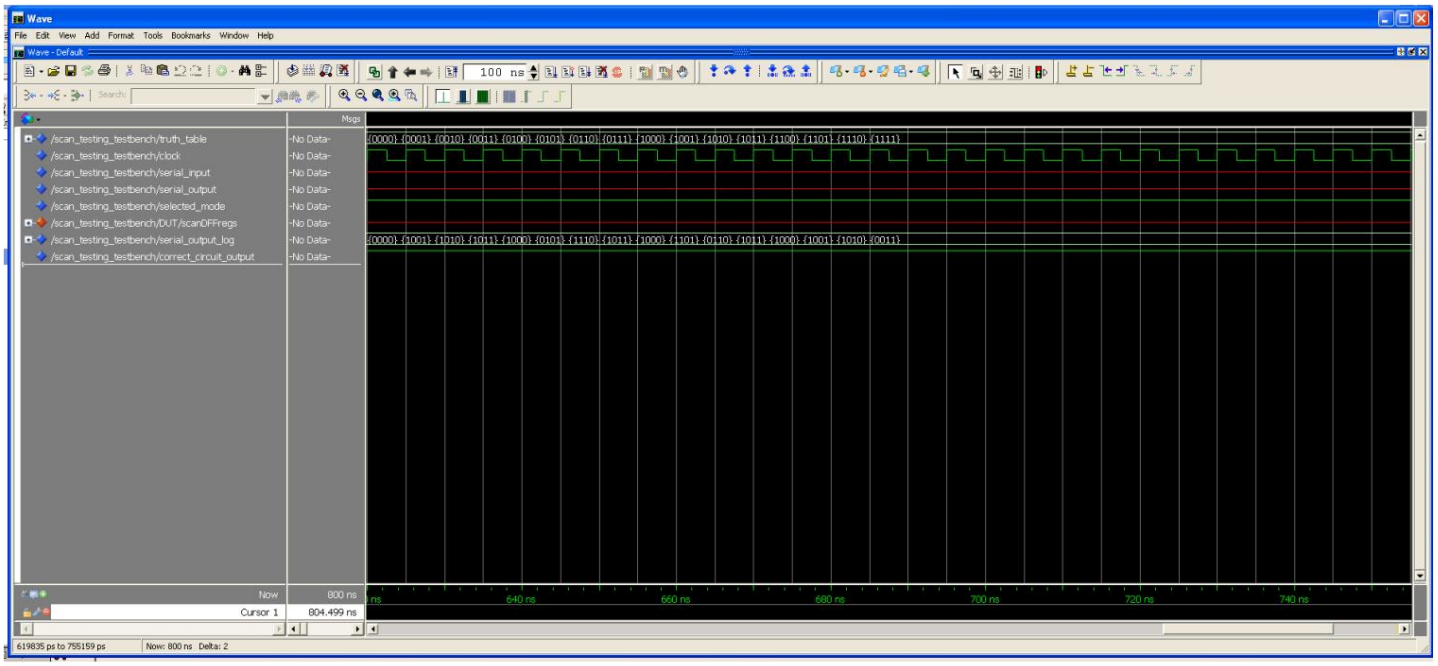
From time t = 550 ns to time t = 855 ns (i.e., the time it took to verify all the logical values of the truth table with the circuit responses from the seria_output_log table) the correct_circuit_output signal remains constantly at logical 1, which indicates the correctness of our scanning chain and the simulation.

For visual reasons, after the verification is complete the correct_circuit_output signal is set to 'X'

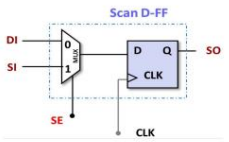


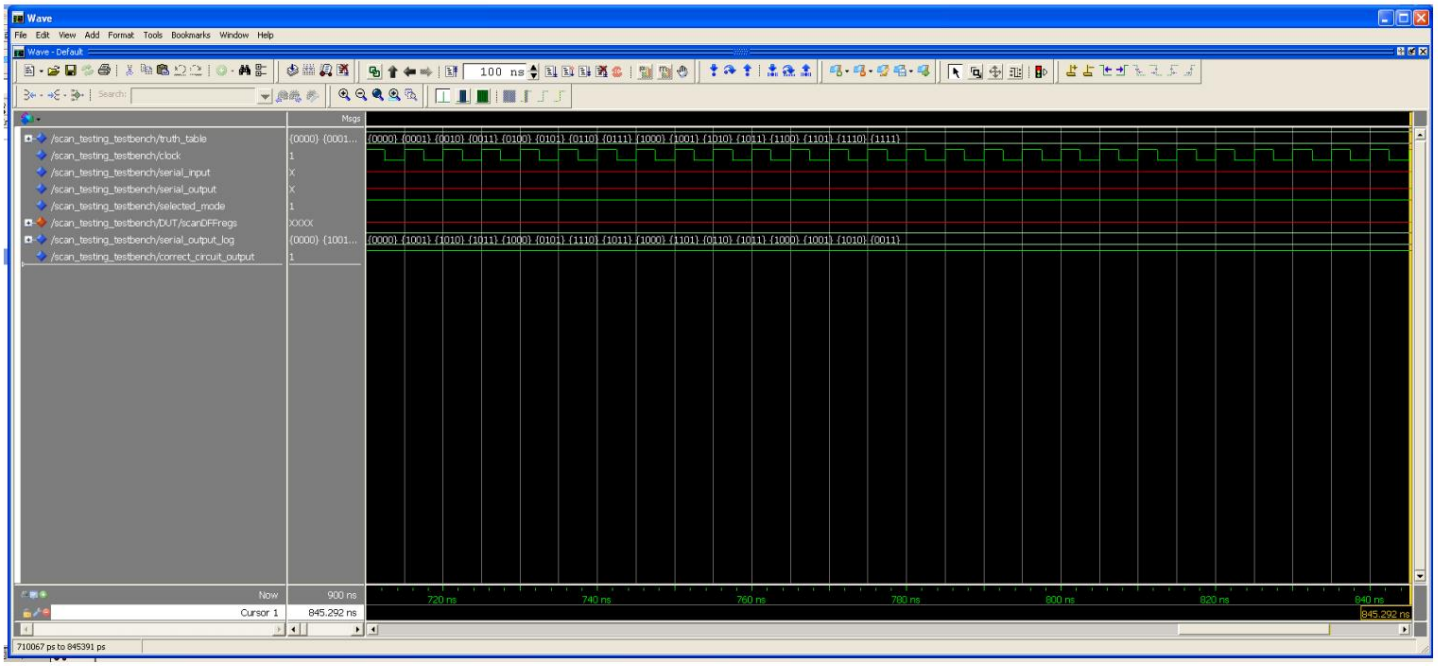


Snapshot 9

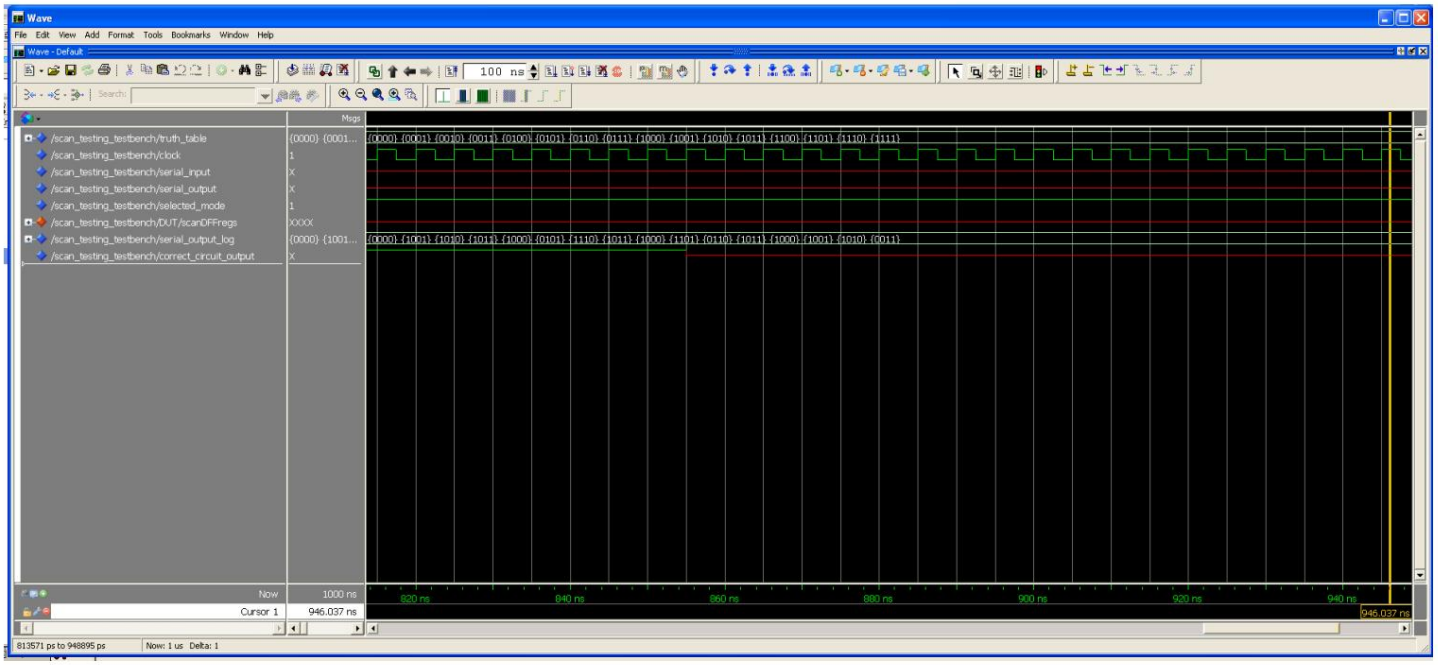


Snapshot 10

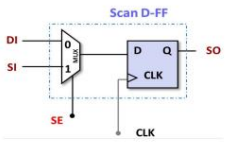




Snapshot 11



Snapshot 12



Question 1.3:
Scan Testing Time

Given the previous questions, we can conclude that a scan chain to perform a check of the correct operation of a circuit using its truth table takes time:

t = t1*t2 + t3

Where

t1 = 2^(Number_of_CUT_entries)

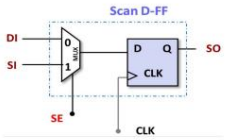
Number of columns of the truth table.

t2 = clokc_cycle*(Number_of_CUT_inputs + 1)

Clock cycles required to load a line of the truth table into the scan chain plus one clock cycle corresponding to the capture of the line.

t3 = clock_cycle*Number_of_CUT_inputs

Clock cycles required to output the last line of the truth table (which will also contain the circuit response within it) from the scan chain.



For clock frequency = 10 Mhz = 10 * 10 **Hz we have clock period t =** 1/10 * 10 **second =** 10 **seconds = 100 ns clock period.**

Using the above formula, it is easily obtained that if N = number of circuit inputs that:

N=10 => Scan testing time = 2 * 10 * (10 + 1) + 10 ***10 = 0.0011274 seconds**

N=20 => Scan testing time = 2 * (20 + 1) + 10 ***20 = 2.2020116 s**

N=30 => Scan testing time = 2 * 10 * (30 + 1) + 10 ***30 = 3328.59965 seconds**

N=40 => Scan testing time 2 * (40 + 1) + 10 ***40 = 4507997.6738856 s**

It is evident that for N > 20 inputs to a circuit, examining all the vectors of its truth table using the scan testing method ends up being impractical due to time constraints.