Design for Testability: FSM with Scan Path

Ayan Garg
Roll No. B23484
Course: VL405
Design For Testability

Abstract

This report presents the design, implementation, and verification of a finite-state machine (FSM) for detecting the sequence 111, enhanced with a two-flip-flop scan path to facilitate design-for-testability (DFT). The flip-flops are modified with multiplexers to support both normal functional operation and scan mode, providing controllability and observability of internal states. The FSM is modeled at the RTL level in VHDL, and a dedicated testbench validates the design by performing scan chain verification, scanning in initial states, applying input vectors, and scanning out the next-state results. Simulation confirms correct sequence detection, proper scan operation, and the effectiveness of the scan path in testing the sequential circuit.

1 Introduction

Sequential digital circuits are inherently harder to test than purely combinational ones because internal flip-flops hide state—primary inputs cannot directly force every internal state (limited controllability) and primary outputs cannot directly reveal every internal node (limited observability). Many faults may never be activated or observed. Scan design addresses this by augmenting the circuit with a serial scan chain built from scan-capable flip-flops. Scan enables deterministic loading (scan-in) and deterministic observation (scan-out) of internal states.

2 Theory

An asynchronous sequential circuit consists of a combinational network and a set of flip-flops that store state information. The combinational circuit implements both the output and next-state functions, while the flip-flops hold the present-state variables between clock cycles.

The combinational network receives the primary inputs (w_1 through w_n) along with the present-state variables (y_1 through y_k), and produces the primary outputs (z_1 through z_m) and the next-state variables (Y_1 through Y_k).

To facilitate **testing and debugging**, a widely used design-for-testability (DFT) technique known as the *scan-path method* is employed. In this approach, multiplexers are connected to the flip-flop inputs, enabling the flip-flops to function either:

- normally, as part of the sequential circuit during regular operation, or
- as a **shift register** during testing.

The scan-path technique typically involves the following sequence of steps:

1. **Flip-flop testing:** A known bit pattern (e.g., `01011001`) is scanned through the chain of flip-flops. The output pattern at `Scan-out` is compared with the expected result to verify correct flip-flop operation.
2. **Combinational logic testing:**
   (a) Scan in the required present-state variables with `Normal/Scan = 1`.
   (b) Apply the desired input vector and perform one clock cycle of normal operation with `Normal/Scan = 0`.
   (c) Scan out the resulting next-state variables with `Normal/Scan = 1` to analyze the circuit's behavior.

This technique provides a systematic approach to testing sequential circuits by making them more controllable and observable during the test mode.

3 Explanation

3.1 Aim and Setup

We were given the following circuit:

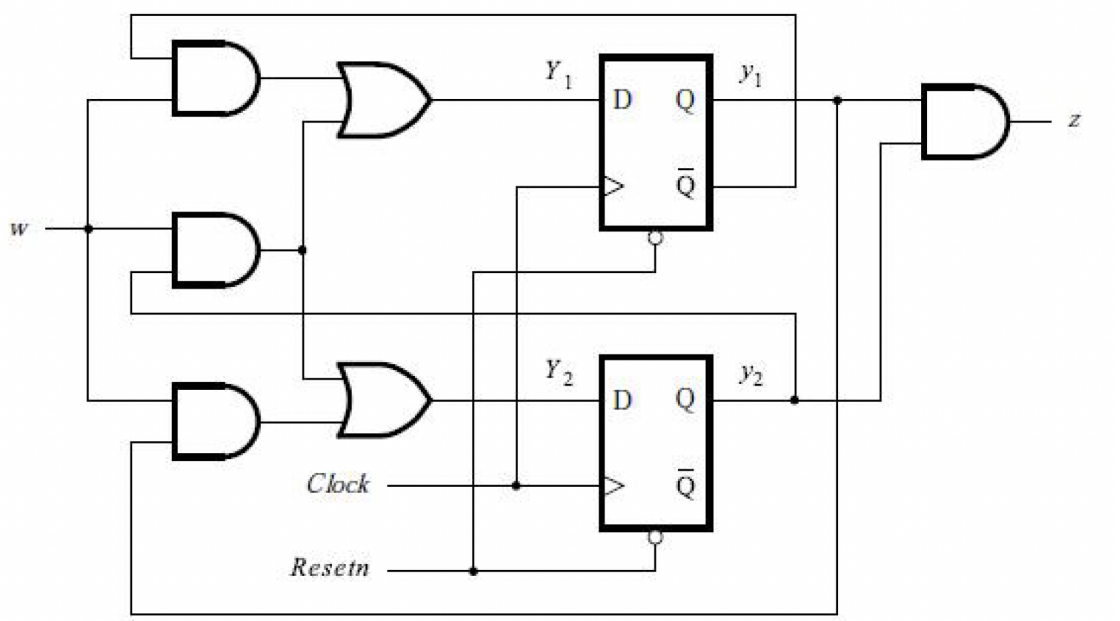

Figure 1: Input Circuit

In this circuit there are 2 flip flops, therefore there will be 4 states in the FSM. The truth table is as follows:

Table 1: State Table of Sequence Detector

Input	**Present State**		**Next State**		**Z**
	S_1	S_0	S_1	S_0	
0	0	0	0	0	0
1	0	0	1	0	0
0	0	1	0	0	0
1	0	1	1	1	0
0	1	0	0	0	0
1	1	0	0	1	0
0	1	1	0	0	1
1	1	1	1	1	1

Therefore the state transition diagram becomes: Assuming the following state encoding:

$$S_0 : 00, \quad S_1 : 01, \quad S_2 : 10, \quad S_3 : 11$$

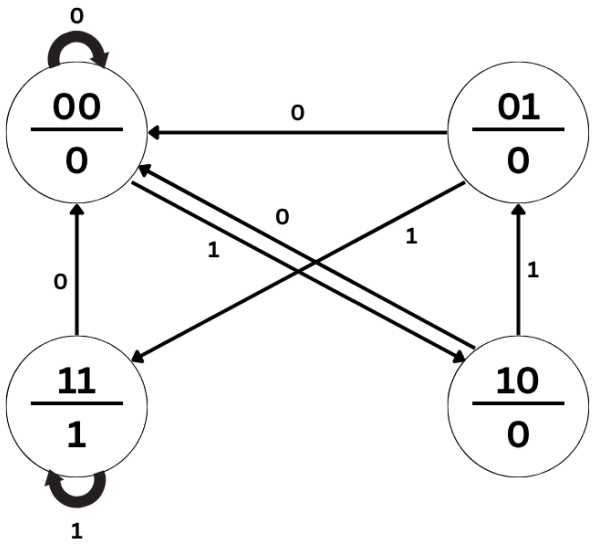

Figure 2: State Transition Diagram

It is clear from the above state table and state diagram that the given circuit is a overlapping '111' sequence detector. As we can see that the output depends only on the present state, so we can say that it is a Moore state Machine.

3.2 Circuit Implementation with Scan Cells

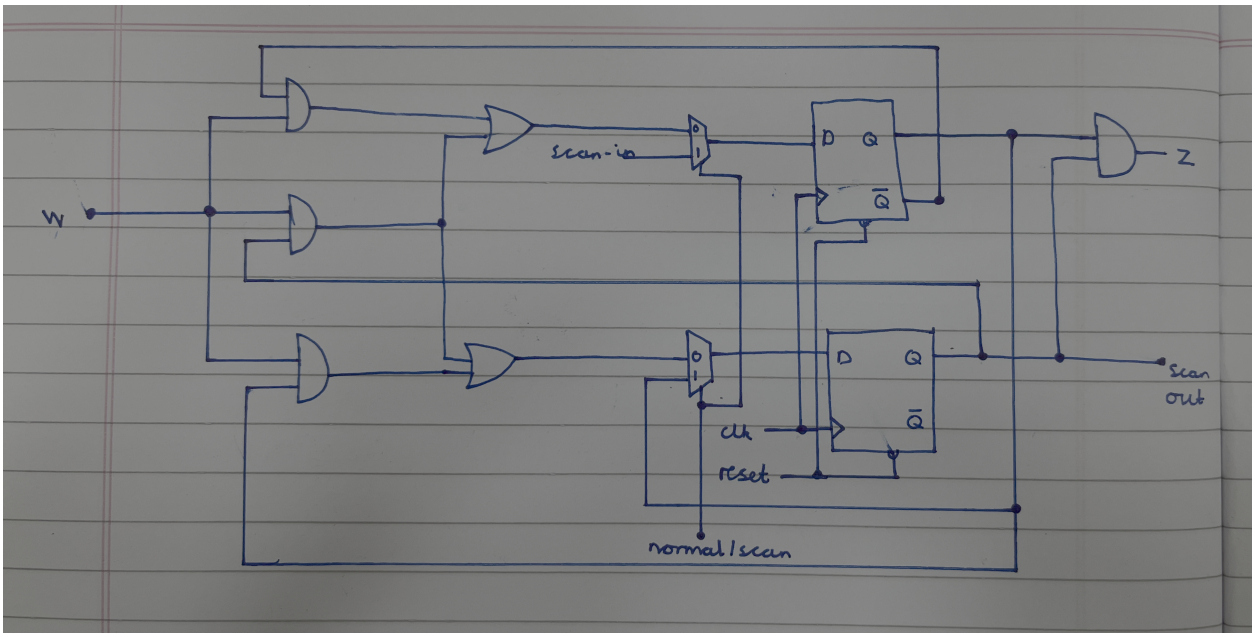

Figure 3: Circuit with scan cells

3.3 Boolean Equations for intermediate nodes

State diagram progression follows detecting 111:

$$
\begin{aligned}
a &= w \wedge \neg q_1 \\
b &= w \wedge q_2 \\
c &= a \vee b \\
d &= w \wedge q_1 \\
e &= d \vee b \\
f &= c \\
g &= e \\
z &= q_1 \wedge q_2
\end{aligned}
$$

4 Design Objectives

- Implement FSM that detects 111, asserts z.
- Add a two-flip-flop scan chain with scanen, scanin, scanout.
- Provide a testbench that verifies scan shifting, loads state, applies inputs, scans out next states.

5 RTL Description

The FSM is augmented with two scan flip-flops to form a scan chain. Multiplexers select mode. Signals q_1, q_2 represent FSM state; z asserts for the detected sequence.

5.1 Entity Ports

The entity of the sequence detector with scan-path support consists of the following ports:

- `clk` (*in*): Clock input signal.
- `rst` (*in*): Active-low reset signal.
- `w` (*in*): Serial input to the finite state machine (FSM).
- `scan_enable` (*in*): Scan enable control signal (`1` – scan mode, `0` – normal functional mode).
- `scan_in` (*in*): Input of the scan chain.
- `scan_out` (*out*): Output of the scan chain.
- `z` (*out*): Sequence-detected output signal.

5.2 VHDL RTL Example

5.2.1 D Flip Flop VHDL Code

```
    library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity dff is
    port (
        clk   : in  std_logic;
        rst_n : in  std_logic;
        d     : in  std_logic;
        q     : out std_logic
    );
end dff;

architecture behavior of dff is
    signal q_next : std_logic := '0';
begin
    process (clk, rst_n)
    begin
        if (rst_n = '0') then
            q_next <= '0';
        elsif rising_edge(clk) then
            q_next <= d;
        end if;
    end process;

    q <= q_next;
end behavior;
```

5.2.2 Implementation VHDL Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity test is
    port (
        clk         : in  std_logic;
        rst_n       : in  std_logic;
        w           : in  std_logic;
        scan_enable : in  std_logic;
        scan_in     : in  std_logic;
        scan_out    : out std_logic;
        z           : out std_logic
    );
end test;

architecture FSM of test is
    signal a, b, c, d, e, f, g : std_logic;
    signal q1, q2 : std_logic;
    signal s1, s2 : std_logic;
```

```
    signal d1, d2 : std_logic;
begin
    scan_ff1: entity work.dff
        port map (
            clk   => clk,
            rst_n => rst_n,
            d     => scan_in,
            q     => s1
        );

    scan_ff2: entity work.dff
        port map (
            clk   => clk,
            rst_n => rst_n,
            d     => s1,
            q     => s2
        );

    d1 <= f when scan_enable = '0' else s1;
    d2 <= g when scan_enable = '0' else s2;

    dff1: entity work.dff
        port map (
            clk   => clk,
            rst_n => rst_n,
            d     => d1,
            q     => q1
        );

    dff2: entity work.dff
        port map (
            clk   => clk,
            rst_n => rst_n,
            d     => d2,
            q     => q2
        );

    a <= w and (not q1);
    b <= w and q2;
    c <= a or b;
    d <= w and q1;
    e <= d or b;
    f <= c;
    g <= e;
    z <= q1 and q2;
    scan_out <= q2;
end FSM;
```

6 Testbench

```
library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;

entity FSM_tb is
end FSM_tb;

architecture sim of FSM_tb is
    signal clk, rst_n, w, scan_enable, scan_in, scan_out, z : std_logic := '0';

    component test
        port (
            clk         : in  std_logic;
            rst_n       : in  std_logic;
            w           : in  std_logic;
            scan_enable : in  std_logic;
            scan_in     : in  std_logic;
            scan_out    : out std_logic;
            z           : out std_logic
        );
    end component;

begin
    uut: test
        port map (
            clk         => clk,
            rst_n       => rst_n,
            w           => w,
            scan_enable => scan_enable,
            scan_in     => scan_in,
            scan_out    => scan_out,
            z           => z
        );

    clk_process: process
    begin
        while true loop
            clk <= '0'; wait for 5 ns;
            clk <= '1'; wait for 5 ns;
        end loop;
    end process;

    stim_proc: process
    begin
        rst_n <= '0'; wait for 20 ns;
        rst_n <= '1'; wait for 10 ns;

        scan_enable <= '1';
        scan_in <= '1'; wait for 10 ns;
        scan_in <= '0'; wait for 10 ns;
        scan_in <= '1'; wait for 10 ns;
        scan_in <= '0'; wait for 10 ns;
        scan_in <= '0'; wait for 40 ns;

        scan_enable <= '1';
        scan_in <= '1'; wait for 10 ns;
```

```
        scan_in <= '0'; wait for 10 ns;

        scan_enable <= '0';  -- functional mode
        w <= '1'; wait for 10 ns;

        scan_enable <= '1';
        scan_in <= '0'; wait for 20 ns;

        scan_enable <= '1';
        scan_in <= '1'; wait for 10 ns;
        scan_in <= '1'; wait for 10 ns;

        scan_enable <= '0';
        w <= '1'; wait for 10 ns;

        scan_enable <= '1';
        scan_in <= '0'; wait for 20 ns;

        wait;
    end process;

end sim;
```

7 Results

The simulation was carried out using ModelSim. In scan mode (scan mode = 1), the flip-flops acted as a serial shift register, confirming proper scan chain operation. In functional mode (scan mode = 0), the FSM transitions occurred according to the input pattern, and the output detect z asserted high when the desired pattern was detected.

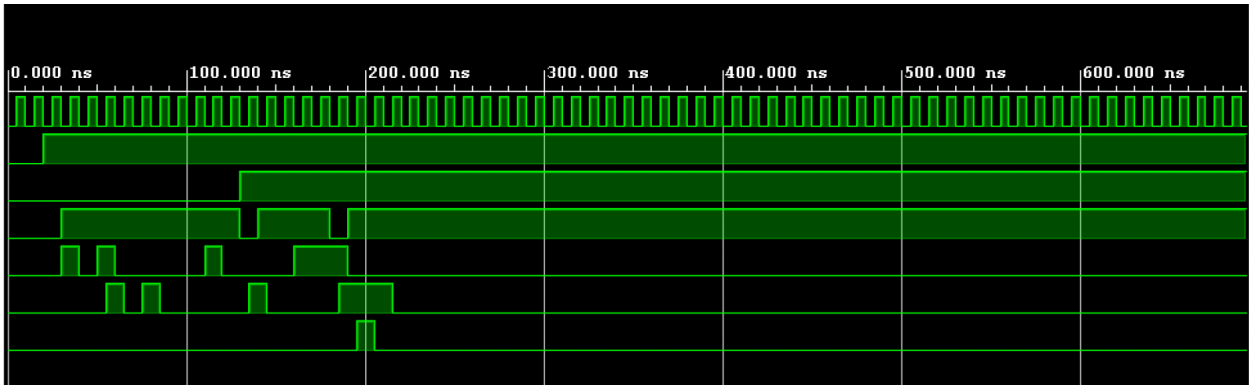

Figure 4: Simulation Output

8 Conclusion

We successfully designed and verified an FSM for detecting the sequence 111, incorporating a two-flip-flop scan chain. The testbench confirmed the correct operation of the scan path,

proper state loading, and accurate scan-out of the next state, demonstrating the advantages of scan-based design for design-for-testability (DFT).