Technische
Hochschule
Augsburg

VLSI Design

Finite State Machine Design

J Färber

SS2026

Contents

Finite State Machine Design

Learning Objectives

After completing this unit, you will be able to:

- Explain the Moore FSM model and state why it is preferred over the Mealy type in synchronous design
- Draw and interpret state diagrams using the mandatory legend (inputs, outputs)
- Declare an enumerated `state_type` and use it for `current_state` / `next_state` signals
- Apply the two-part FSM architecture: **state register** (concurrent `WHEN ... ELSE`) and **next-state and output logic** (PROCESS with CASE)
- Implement a rising-edge detector as a three-state Moore FSM in VHDL
- Implement a ‘101’-sequence detector as a four-state Moore FSM in VHDL
- Write a synchronous testbench for an FSM design and verify the state transitions

Block Diagram

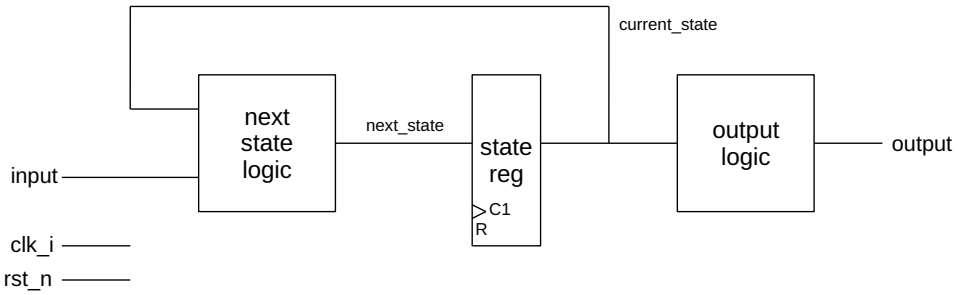

Figure 1: Moore Type Finite State Machine - Block Diagram

The **mandatory design rules** for synchronous circuits are extended by the rule of designing FSMs of type Moore only:

- There is only one common clock signal clk, which is connected to all clock inputs of n-bit storage elements
- A clock signal is never linked via logic elements.
- Each synchronous module has a reset signal.
- **Finite state machines** have to be of **type Moore only**

Rising Edge Detector

Schematic Symbol

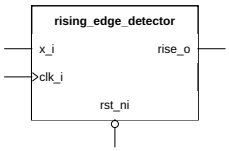

Figure 2: Rising Edge Detector - Schematic Symbol

Entity Declaration

```
ENTITY rising_edge_detector IS`
  PORT (
    clk_i  : IN  std_ulogic;
    rst_ni : IN  std_ulogic;
    x_i    : IN  std_ulogic;
    rise_o : OUT std_ulogic
    );
END rising_edge_detector;
```

Timing Diagram

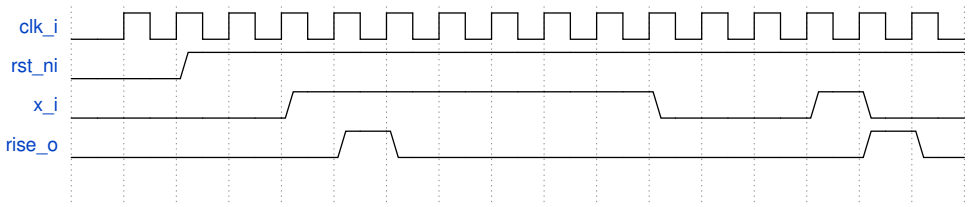

Figure 3: Rising Edge Detector - Timing Diagram

State Diagram

Figure 4: Rising Edge Detector - State Diagram

VHDL Model

```
ARCHITECTURE fsm OF rising_edge_detector IS

  TYPE state_type IS (zero, one, edge);
  SIGNAL next_state, current_state : state_type;

BEGIN

  fsm_state_register: current_state <=                WHEN rst_ni = '0' else
                                                      WHEN rising_edge(clk_i);

  fsm_next_state_and_output_logic: PROCESS
  BEGIN
    rise_o <= '0';
    next_state <= current_state;
    CASE current_state IS
      WHEN zero =>
        IF x_i = '1' THEN
          next_state <= edge;
        END IF;

    END CASE;
  END PROCESS;
END fsm;
```

Moore FSM — Recommended VHDL Design Pattern

Every Moore FSM architecture follows the same two-part structure.

Architecture Overview

Part	Function	VHDL style
State register	stores `current_state`; synchronous load, asynchronous reset	concurrent `WHEN … ELSE`
Next-state and output logic	computes `next_state` and all outputs from `current_state` and inputs	PROCESS with `CASE / IF` Outputs are a function of **`current_state` only** — never of inputs directly.

State Type and Signal Declarations

```
TYPE state_type IS (state_a, state_b, state_c, ...);
SIGNAL current_state, next_state : state_type;
```

State Register

```
state_register: current_state <= reset_state WHEN rst_ni = '0' ELSE
                                 next_state   WHEN rising_edge(clk_i);
```

Next-State and Output Logic

```
next_state_and_output_logic: PROCESS (current_state, input_signals)
BEGIN
  -- Default assignments — prevent unintended latches
  output_signals <= default_values;
  next_state     <= current_state;

  CASE current_state IS
    WHEN state_a =>
      -- First: Assign Moore outputs
      output <=
      -- Then: Transitions for state_a
      IF input_signals THEN
        ...
      END IF;
```

```
    WHEN state_b =>
      -- First: Assign Moore outputs
      -- Then: Transitions for state_b
    -- ...
    WHEN OTHERS =>
      next_state <= reset_state;    -- safe recovery from undefined states
  END CASE;
END PROCESS;
```

Design Rules

- **Default assignments** are placed *before* the CASE statement — every signal is driven on every path through the process, which prevents synthesis tools from inferring latches
- **`WHEN OTHERS`** is mandatory — it covers unused state encodings that can occur after power-on or a single-event upset
- **Outputs depend on `current_state` only** — input-dependent assignments inside a WHEN branch would produce Mealy behaviour and must be avoided in design files

Sequence Detector

Schematic Symbol

Function

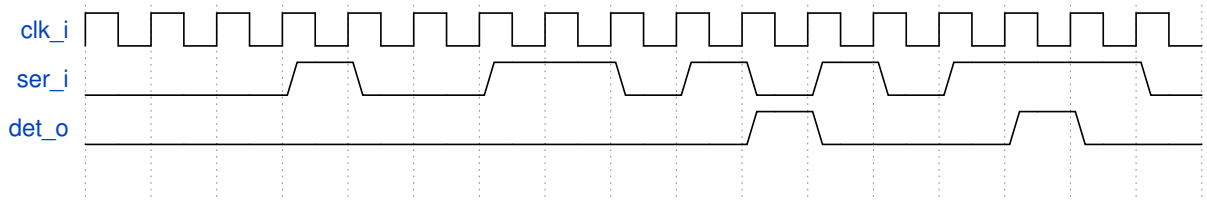

Figure 5: ‘101’-Sequence Detector - Timing Diagram

State Diagram

The mandatory legend of a state diagram includes input and ouput signals:

```
Input(s):   ser_i
Output(s):  det_o
```

Figure 6: ‘101’-Sequence Detector - State Diagram

VHDL RTL Model

```
ARCHITECTURE fsm OF sequence_detector IS

TYPE state_type IS (idle, one, zero, detected);
SIGNAL next_state, current_state : state_type;

BEGIN

  state_register: current_state <=            WHEN rst_ni='0' ELSE
                                  next_state WHEN rising_edge(clk_i);


  next_state_and_output_logic: PROCESS (current_state,       )
  BEGIN
    det_o <= '0';
    next_state <= current_state;
    CASE current_state IS
      WHEN idle =>









    END CASE;
  END PROCESS;

END fsm;
```

Simulation Environment

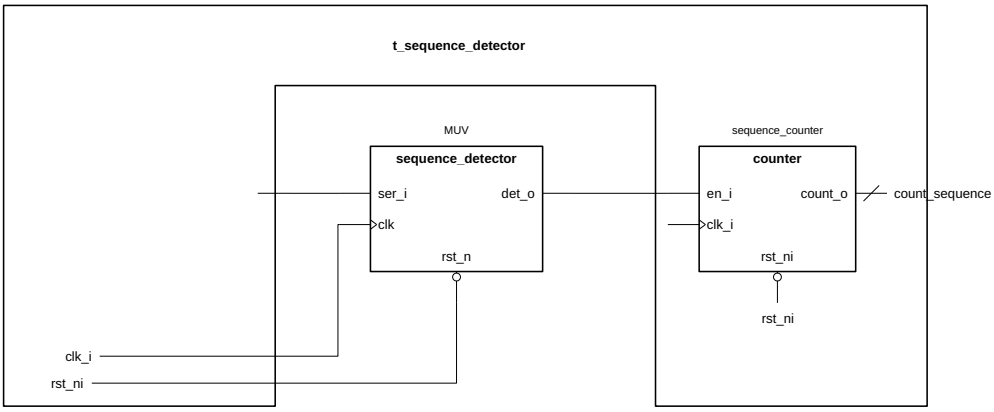

Figure 7: Testbench of Sequence Detector

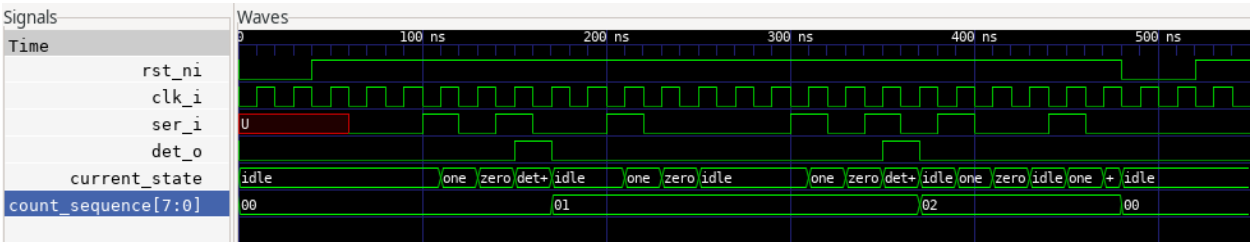

Figure 8: ‘101’-Sequence Detector - Simulation Result

Simple Testbench with Observer

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY t_sequence_detector IS
END t_sequence_detector;

ARCHITECTURE tbench OF t_sequence_detector IS
  -- Component declaration for the Unit Under Test (UUT)
  COMPONENT sequence_detector
    PORT (
      clk_i  : IN  std_ulogic;
      rst_ni : IN  std_ulogic;
      ser_i  : IN  std_ulogic;
      det_o  : OUT std_ulogic
      );
  END COMPONENT;

  -- Signals for connecting to the UUT
  SIGNAL clk_i  : std_ulogic;
  SIGNAL rst_ni : std_ulogic;
  SIGNAL ser_i  : std_ulogic;
  SIGNAL det_o  : std_ulogic;

  -- Switch for clock generator
  SIGNAL clken_p : boolean := true;

  -- Clock period constant
  CONSTANT period : TIME := 20 ns;

  SIGNAL count_sequence : unsigned(7 DOWNTO 0);

BEGIN
  -- Instantiate the Unit Under Test (UUT)
  MUV: sequence_detector
    PORT MAP (
      clk_i  => clk_i,
      rst_ni => rst_ni,
      ser_i  => ser_i,
      det_o  => det_o
      );

  -- Clock generation process with enable switch
  clk_process: PROCESS
  BEGIN
    WHILE clken_p LOOP
      clk_i <= '0';
      WAIT FOR period / 2;
      clk_i <= '1';
      WAIT FOR period / 2;
    END LOOP;
    WAIT;
  END PROCESS;
```

```
  -- Stimulus process
  stimulus_process: PROCESS
  BEGIN
    -- Initial state
    REPORT "Starting test bench simulation..." SEVERITY NOTE;

    -- Apply reset
    rst_ni <= '0';
    WAIT FOR 2 * period;
    rst_ni <= '1';
    WAIT FOR period;

    -- Test case 1: No input (idle -> idle)
    REPORT "Test case 1: ser_i = '0', no sequence started" SEVERITY NOTE;
    ser_i <= '0';
    WAIT FOR 2 * period;

    -- Test case 2: Complete sequence '1','0','1'
    REPORT "Test case 2: sequence 1-0-1, det_o should pulse" SEVERITY NOTE;
    ser_i <= '1';
    WAIT FOR period;
    ser_i <= '0';
    WAIT FOR period;
    ser_i <= '1';
    WAIT FOR period;
    ser_i <= '0';
    WAIT FOR 2 * period;

    -- Test case 3: Incomplete sequence '1','0','0' (no detection)

    -- Test case 5: Reset during operation

    -- End simulation
    REPORT "Test bench simulation completed." SEVERITY NOTE;
    clken_p <= false;
    WAIT;
  END PROCESS;

  -- Observer process
  observer_process: PROCESS
  BEGIN
    -- WAIT ON clk_i, rst_ni, ser_i, det_o;
    WAIT UNTIL rising_edge(clk_i);

    -- Log current signal states
    REPORT "Time: " & TIME'IMAGE(NOW) &
           " RST_N: " & std_ulogic'IMAGE(rst_ni) &
           " SER_I: " & std_ulogic'IMAGE(ser_i) &
           " DET_O: " & std_ulogic'IMAGE(det_o)
    SEVERITY NOTE;

    -- Check for invalid conditions
    IF rst_ni = '0' AND det_o = '1' THEN
      REPORT "Error: Output active during reset" SEVERITY ERROR;
    END IF;

  END PROCESS;
END tbench;
```

```
ghdl.log
:77:5:@0ms:(report note): Starting test bench simulation...
:152:5:@10ns:(report note): Time: 10000000 fs RST_N: '0' SER_I: 'U' DET_O: '0'
:152:5:@30ns:(report note): Time: 30000000 fs RST_N: '0' SER_I: 'U' DET_O: '0'
:152:5:@50ns:(report note): Time: 50000000 fs RST_N: '1' SER_I: 'U' DET_O: '0'
:86:5:@60ns:(report note): Test case 1: ser_i = '0', no sequence started
:152:5:@70ns:(report note): Time: 70000000 fs RST_N: '1' SER_I: '0' DET_O: '0'
:152:5:@90ns:(report note): Time: 90000000 fs RST_N: '1' SER_I: '0' DET_O: '0'
:91:5:@100ns:(report note): Test case 2: sequence 1-0-1, det_o should pulse
:152:5:@110ns:(report note): Time: 110000000 fs RST_N: '1' SER_I: '1' DET_O: '0'
:152:5:@130ns:(report note): Time: 130000000 fs RST_N: '1' SER_I: '0' DET_O: '0'
:152:5:@150ns:(report note): Time: 150000000 fs RST_N: '1' SER_I: '1' DET_O: '0'
:152:5:@170ns:(report note): Time: 170000000 fs RST_N: '1' SER_I: '0' DET_O: '1'
:152:5:@190ns:(report note): Time: 190000000 fs RST_N: '1' SER_I: '0' DET_O: '0'
:102:5:@200ns:(report note): Test case 3: sequence 1-0-0, no detection expected
:152:5:@210ns:(report note): Time: 210000000 fs RST_N: '1' SER_I: '1' DET_O: '0'
:152:5:@230ns:(report note): Time: 230000000 fs RST_N: '1' SER_I: '0' DET_O: '0'
:152:5:@250ns:(report note): Time: 250000000 fs RST_N: '1' SER_I: '0' DET_O: '0'
:152:5:@270ns:(report note): Time: 270000000 fs RST_N: '1' SER_I: '0' DET_O: '0'
:152:5:@290ns:(report note): Time: 290000000 fs RST_N: '1' SER_I: '0' DET_O: '0'
...
```

Alternative Stimulus Process

```
  -- using a constant as an input bit stream

  stimulus_process: PROCESS
    CONSTANT ser_stream : std_ulogic_vector := "00101001000010101001000000";
     ↪
  BEGIN
    REPORT "Starting test bench simulation..." SEVERITY NOTE;

    -- Apply reset
    rst_ni <= '0';
    WAIT FOR 2 * period;
    rst_ni <= '1';
    WAIT FOR period;

    -- Feed bit stream
    FOR i IN ser_stream'RANGE LOOP
      ser_i <= ser_stream(i);
      WAIT FOR period;
    END LOOP;

    ser_i <= '0';
    WAIT FOR 2 * period;

    REPORT "Test bench simulation completed." SEVERITY NOTE;
    clken_p <= false;
    WAIT;
  END PROCESS;
```

Testbench with Noise Generator

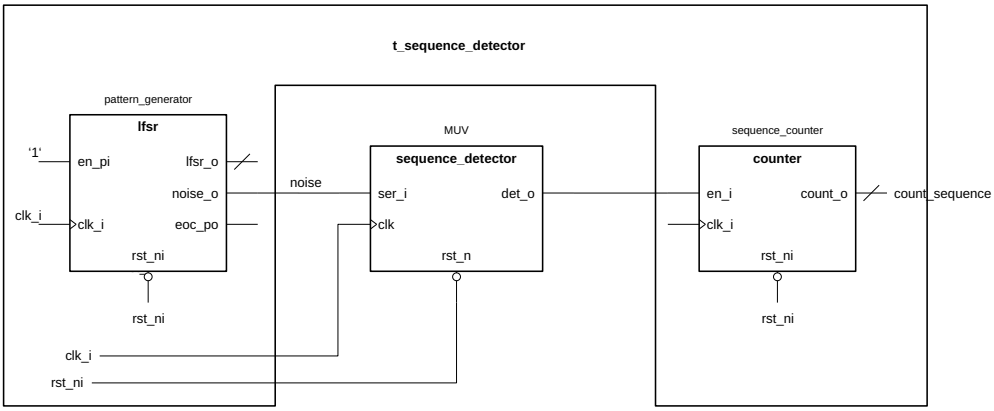

Figure 9: Testbench of Sequence Detector using Noise Generator

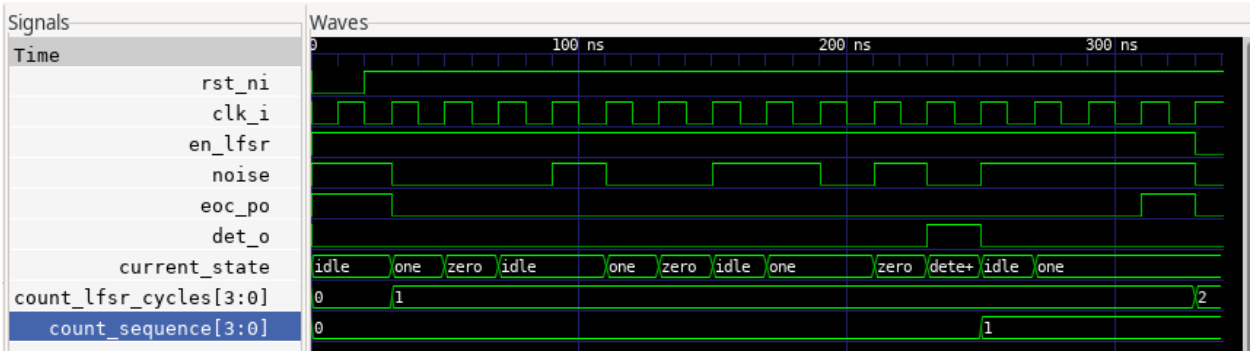

Figure 10: '101'-Sequence Detector - Simulation Result

Summary

- A **Moore FSM** produces outputs that are a function of the current state only
- Every Moore FSM is modelled with an enumerated **state type** and two named signals: `current_state` (state register output) and `next_state` (combinational next value)
- The **state register** uses a `WHEN ... ELSE` concurrent signal assignment — asynchronous reset to the defined initial state, synchronous load of `next_state` on the rising clock edge
- The **next-state and output logic** uses a combinational `PROCESS` with a `CASE` statement — default assignments before the `CASE` prevent latches; `WHEN OTHERS` ensures safe recovery from undefined states
- The **state diagram** is the primary design document; the mandatory legend must list all input and output signals

Reference

- [Mea25] Mealy, B., Tappero, F.: *Free Range VHDL*, 2025
  - Chapter 8