Custom Computing & Embedded Systems

FSM debug and implementation

1. **Thematic scope of the exercise:**
   - simulation using VHDL testbench,
   - methods of describing FSM finite machines,
   - synthesis of automata from the VHDL description.

2. **Methods of modeling finite state machines in VHDL**
   Most synthesis tools include special templates designed to describe finite state machines (FSMs). The basic way to describe an FSM in VHDL is to use the `process` statement. Below are examples of machine construction using a 1-, 2- and 3-process template [2].

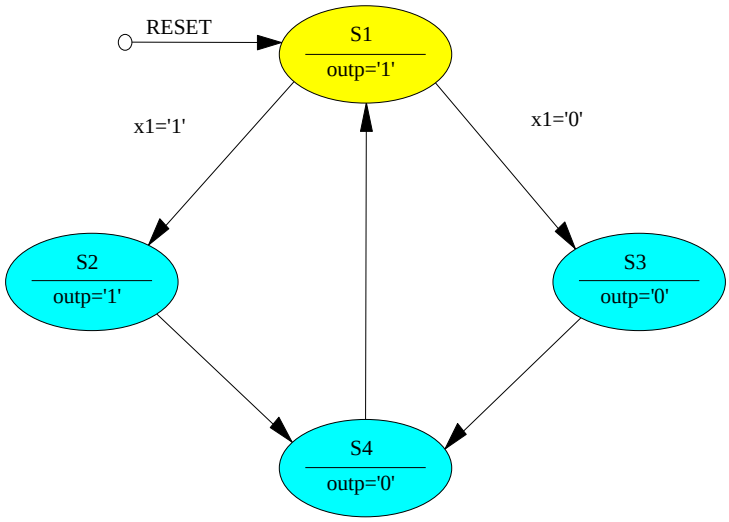


Fig.1. State diagram of the analyzed machine.

Code 1. 1-process description [2]

```
entity fsm_1 is
    port ( clk, reset, x1 : IN std_logic;
           outp           : OUT std_logic);
end entity;

architecture beh1 of fsm_1 is
    type state_type is (s1,s2,s3,s4);
    signal state: state_type;
begin

    process (clk,reset)
    begin
        if (reset ='1') then
            state <=s1;
            outp<='1';
        elsif rising_edge(clk) then
            case state is
                when s1 =>
```

```
                    if x1='1' then
                        state <= s2; outp <= '1';
                    else
                        state <= s3; outp <= '0';
                    end if;
                when s2 =>
                    state <= s4; outp <= '0';
                when s3 =>
                    state <= s4; outp <= '0';
                when s4 =>
                    state <= s1; outp <= '1';
            end case;
        end if;
    end process;
end beh1;
```

Code 2. 2-process description

```
entity fsm_2 is
    port ( clk, reset, x1 : IN std_logic;
           outp           : OUT std_logic);
end entity;

architecture beh1 of fsm_2 is
    type state_type is (s1,s2,s3,s4);
    signal c_state, n_state: state_type;
begin
    proc_fsm: process(c_state, x1) begin
            outp <= '0';
            case c_state is
                when s1 =>
                    outp <= '1';
                    if x1='1' then
                        n_state <= s2;
                    else
                        n_state <= s3;
                    end if;
                when s2 =>
                    outp <= '1';
                    n_state <= s4;
                when s3 =>
                    n_state <= s4;
                when s4 =>
                    n_state <= s1;
            end case;
    end process;

    proc_memory: process (clk,reset)
    begin
        if (reset ='1') then
            c_state <= s1;
        elsif rising_edge(clk) then
            c_state <= n_state;
        end if;
    end process;
end beh1;
```

Code 3. 3-process description

```
entity fsm_3 is
    port ( clk, reset, x1 : IN std_logic;
           outp           : OUT std_logic);
end entity;

architecture beh1 of fsm_3 is
    type state_type is (s1,s2,s3,s4);
    signal c_state, n_state: state_type;
begin
```

```
        proc_fsm: process(c_state, x1) begin
                case c_state is
                    when s1 =>
                        if x1='1' then
                            n_state <= s2;
                        else
                            n_state <= s3;
                        end if;
                    when s2 =>
                        n_state <= s4;
                    when s3 =>
                        n_state <= s4;
                    when s4 =>
                        n_state <= s1;
                end case;
        end process;

        proc_outs : process (c_state)
        begin
            case c_state is
                when s1 => outp <= '1';
                when s2 => outp <= '1';
                when s3 => outp <= '0';
                when s4 => outp <= '0';
            end case;
        end process;

        proc_memory: process (clk,reset)
        begin
            if (reset ='1') then
                c_state <= s1;
            elsif rising_edge(clk) then
                c_state <= n_state;
            end if;
        end process;
end beh1;
```

Note the use of the enumerated type to define the states of the machine. This approach facilitates testing of different types of state encodings at the synthesis and implementation stages. I recommend using the description from Code 2. It allows a clear separation between the combinational part (proc_fsm) and the sequential part (proc_memory). The sensitivity list in proc_fsm contains all signals on the right side of the expressions. All signals set in the process have a defined default value (`outp <= '0'` after `process begin`). This guarantees that the synthesis will not produce latches that have an adverse effect on the operation of the device. The second process in Code 2 describes all the operations that are to be edge-dependent on the clock signal.

3. Automation simulation

[!] Create a new Questa simulator project and add . **vhd** from the course server

[!] Compile from the command line with the command:

```
vcom -2008 -autoorder -fsmdebug -fsmverbose *.vhd
```

Check the meaning of the `fsmdebug` and `fsmverbose` in the simulator manual [1]
Run the testbench simulation with the command:

```
vsim -voptargs=+acc -fsmdebug work.fsm_2_tb
```

Note: in analyzing the operation of the system, you can use the graphical presentation of the machine (FSM View window). All FSMs detected in the current simulation can be accessed via the *View menu -> FSM List* .

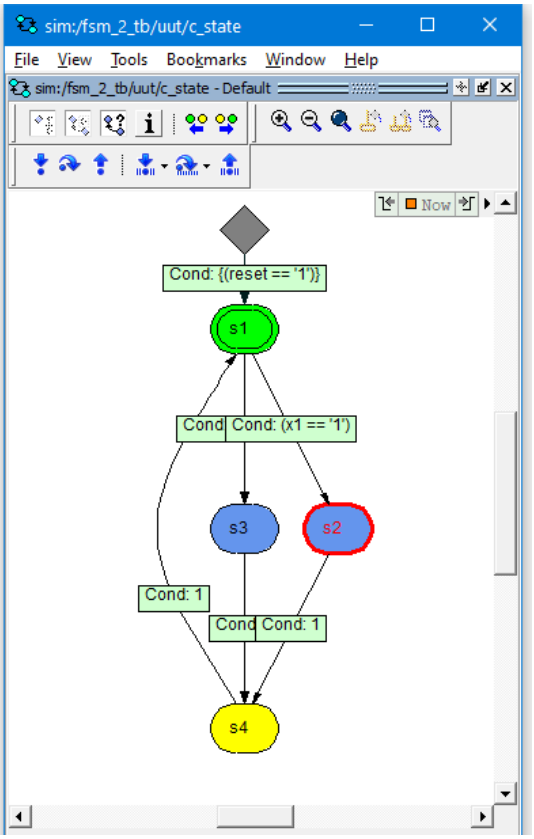


Fig.2. FSM View .

[!] Analyze how the stimuli are defined in the `fsm_2_tb.vhd` file
What changes should be made to the testbench code to test the system's double reset during the first 200ns of operation? For this purpose, use the procedures from the `pkg_symuli`.

4. Designing a FSM

The project defined sources for building a simple frequency meter (Fig. 3).

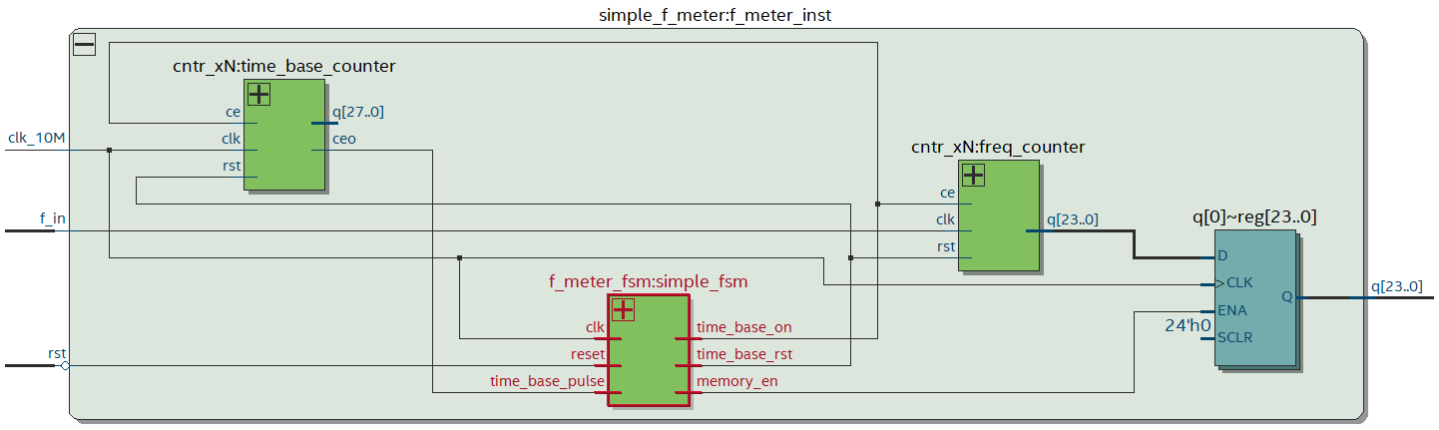


Fig.3. simple_f_meter

The measurement with this meter is based on counting the pulses of the signal measured per unit of time. The counting time is determined by the time base generator (time_base_counter). In the discussed case, it is a 7-decade parallel counter - it determines the time equal to 1sec based on the clock clk_10M=10MHz. After 1sec, we receive the signal frequency f_in expressed in Hertz at the output of the freq_counter. Then the result is written to the reg[23..0] register, the counters are cleared and the next measurement cycle begins. The operation of the meter is

managed by a simple automaton responsible for generating the appropriate sequence of control signals.

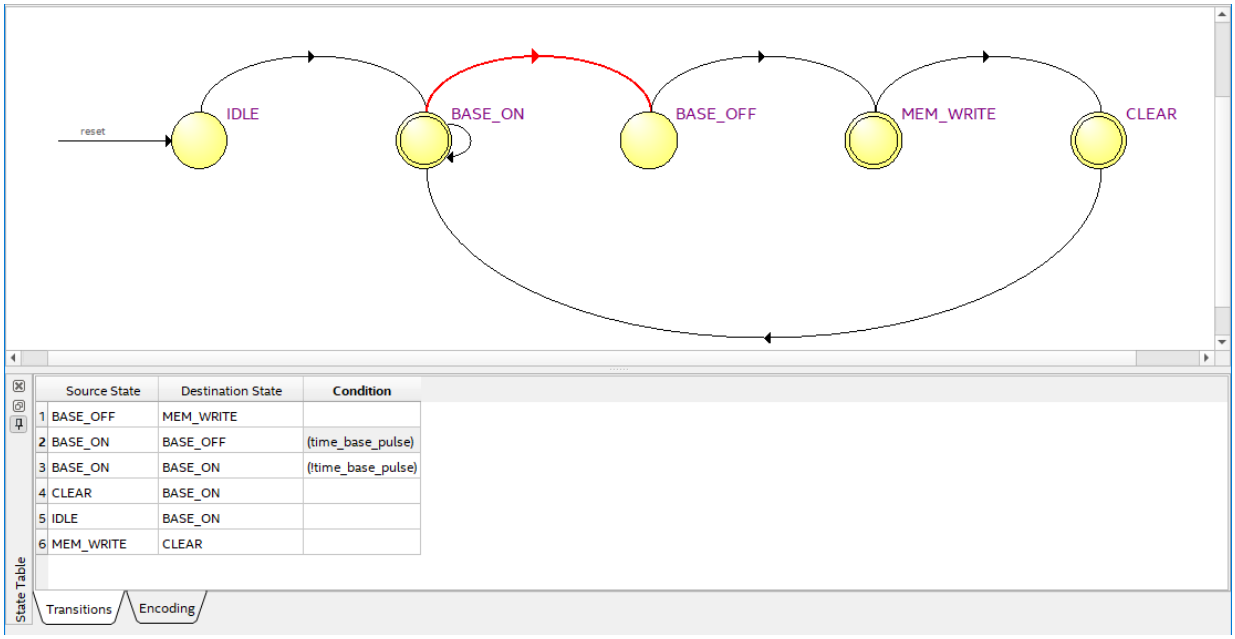


Fig.4. State diagram of the simple_fsm control machine

[!] Compile from the command line with the command:

```
vcom -2008 -autoorder -fsmdebug -fsmverbose *.vhd
```

[!] Run a simulation **of a few seconds** of frequency meter operation with the command:

```
vsim -t 1ps -fsmdebug work.simple_f_meter_tb
```

Analyze the operation of the control machine using the generated state diagram - **View** menu -> **FSM List** -> **View FSM**

[!] Prepare a frequency meter for implementation on the **DE10-Lite** prototype board. Take the measured signal from GPIO(0), the reset signal from KEY(0).

Task A

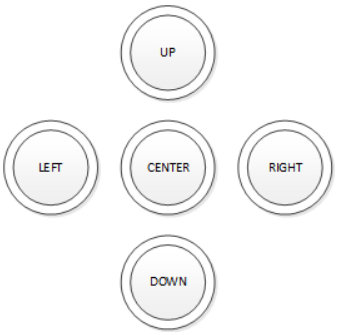
Design a multi-range frequency meter according to the following assumptions:

1. Presentation of the result on 4 digits of a 7-segment display
2. Controlling the position of the decimal point of the displays to increase the accuracy of the displayed measurement result
3. Presentation of the measurement range on two characters of a 7-segment display

    - Hz,  - kilo Hz,  - mega Hz
4. Automatic change of the measurement range

[!] In connection with assumption 4, create a new model of the meter control machine enabling automatic range change.
[!] Prepare a testbench for meter simulation with automatic change of measurement range. The testbench should examine the range change up and down and report the input and measured frequency value.
[!] Prepare the simulation compilation/configuration macro.
[!] Perform gate-level simulation using the prepared testbench .

Task B



Make a model of the machine controlling the settings of the multi-decade counter (see Fig.5). The system is controlled using five monostable buttons as shown in the figure below. By default (after turning on the power and after activating `rst`), the control system is in the idle state (IDLE). The executive system (counter Cntr_Nbcd_load from Fig.6) after triggering (`right`) is incremented until the end of counting is reached. The executive system can be paused or reset after using the appropriate buttons (`left, down`). In the edit mode (after pressing `center`), you can define a new value of the end of counting for the actuator using the `up, down, left, right` buttons and confirm by pressing `center` again. When exiting the edit mode, new settings may be abandoned (RETURN →IDLE) or values may be loaded into the executive system (via the LOAD state). In addition to editing, content can be forced to reload with the `up` button.

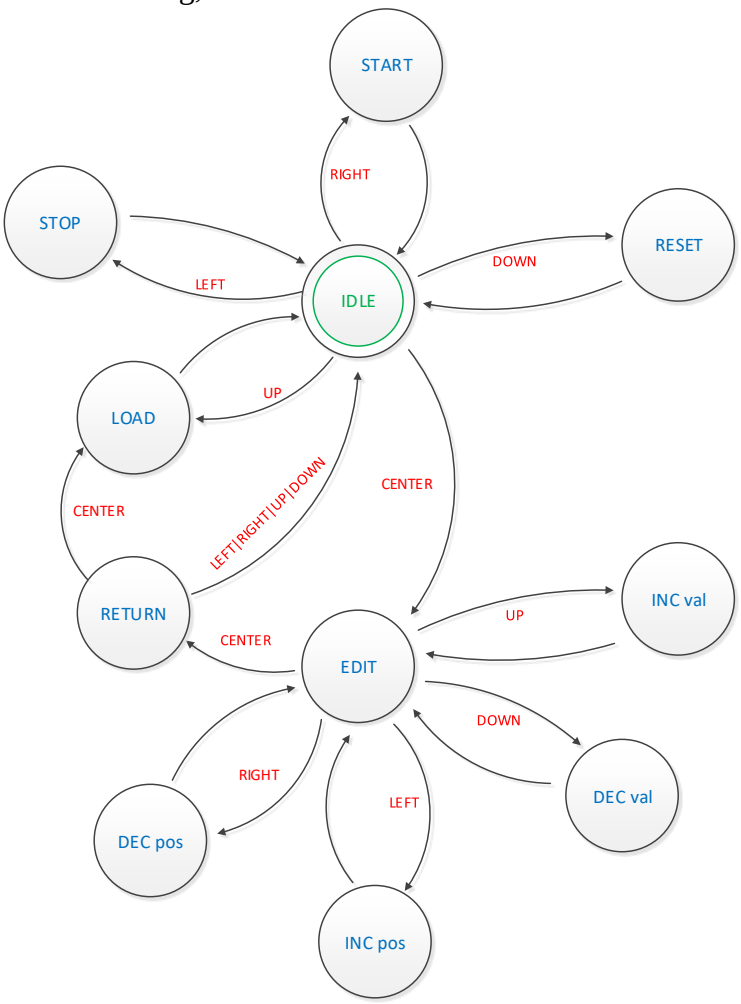


Fig.5. State diagram of the key_fsm machine.

Notes: the output settings are not marked in the diagram fig.5 - this is a part to be developed by the designer. Actions taken in the RESET state concern deleting the actuator system, not the control system.

For design purposes, we assume that the input control signals (`left, right, up, down, center`) **last exactly one period of the clock** (`clk`). Transitions not described in the diagram are performed unconditionally (under the influence of the rising edge of the clock). When the device is started, the counter (executive system) does not count - it starts the first counting after receiving

a command from the control machine (`right`). In the EDIT mode, the executive system (counter) is blocked (this means that the `ce` input of the counter is disabled);

Interface of the designed device `key_fsm` :

```
12  entity key_fsm is
13    port (clk: in std_logic;
14          rst: in std_logic; -- synch, high
15          left, right, up, down, center: in std_logic; -- keys
16          data_out: out std_logic_vector(31 downto 0);
17          cntr_en: out std_logic;
18          cntr_rst: out std_logic;
19          cntr_load: out std_logic;
20          edit_en_out: out std_logic);
21  end entity;
```

`data_out` : output BCD-coded, can be parameterized by the number of counter digits;
`cntr_en` : ce signal output for bcd counter;
`cntr_rst` : clear signal output for bcd counter;
`cntr_load` : load signal output for bcd counter;
`edit_en_out` : an output indicating that the machine is in edit mode (in a later implementation it can be used as a display switching signal);

Requirements for task B

[!] design an automaton controlling an 8-decade counter
[!] prepare simulation **macro ;**
[!] simulate the prepared architecture using testbench **key_fsm_tb** ;
[!] perform **"gate-level"** simulation

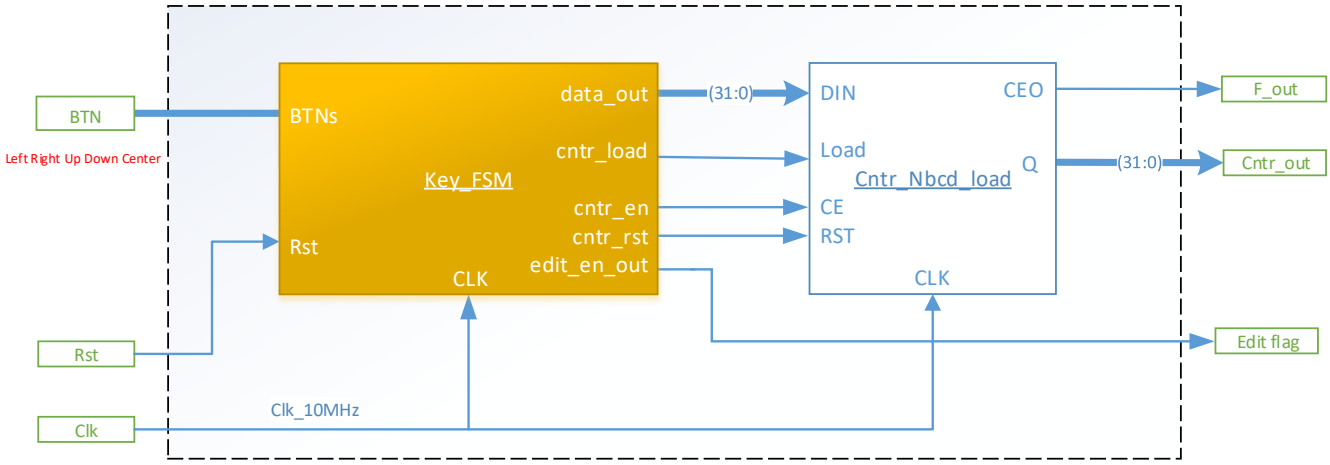


Fig.6. Key_FSM machine with executive system (Cntr_Nbcd_load).

[1] Siemens EDA, Questa® SIM Command Reference Manual Including Support for Questa Base, Software Version 2023.3, Document Revision 8.3, © 2023 Siemens.
[2] Synthesis and Simulation Design Guide, ISE Version 9.2i, Xilinx Corporation 2007.