

CPE 462

VHDL: Simulation and Synthesis

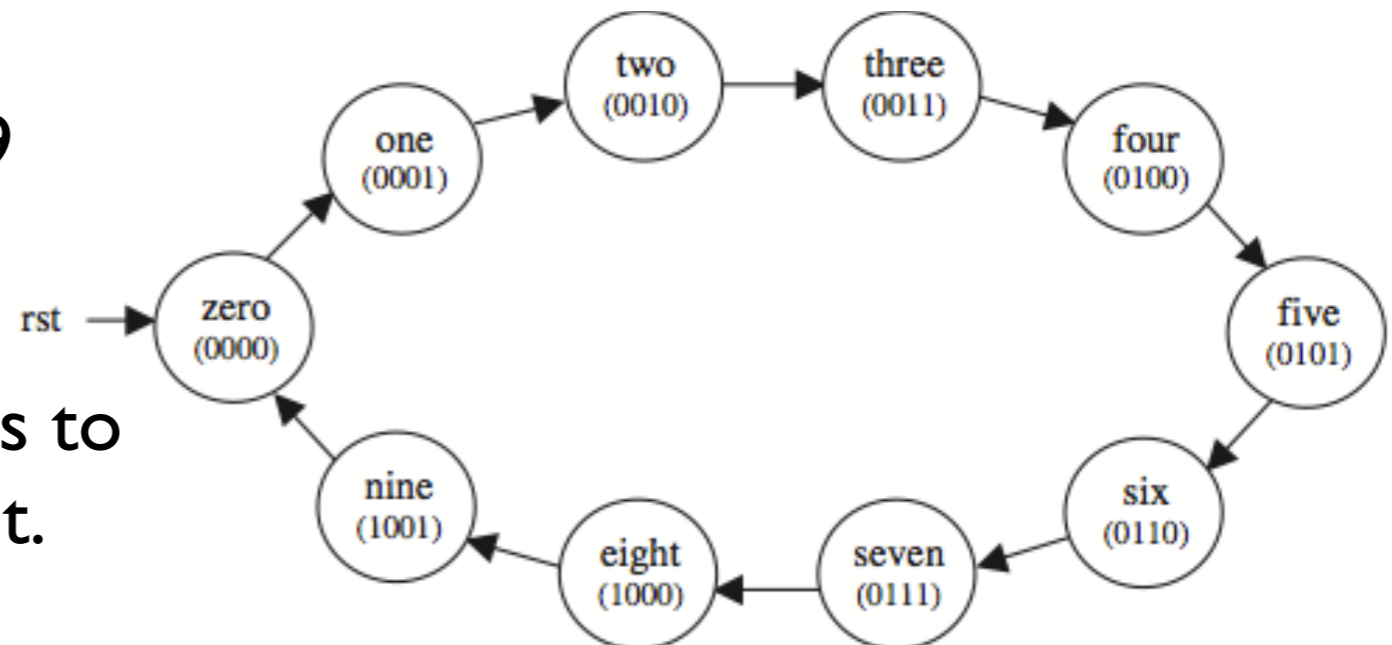
Topic #07 - b) VHDL implementations

When to use a FSM

- Any sequential circuit can in principle be modeled as a state machine, this is not always advantageous.
- For example, implementing a counter as a FSM will likely result in longer, more complex, and more error prone than in a conventional approach.
- The FSM approach is advisable in systems whose tasks constitute a well-structured list so all states can be easily enumerated.
- For example digital controllers, such as traffic light and elevator controllers.

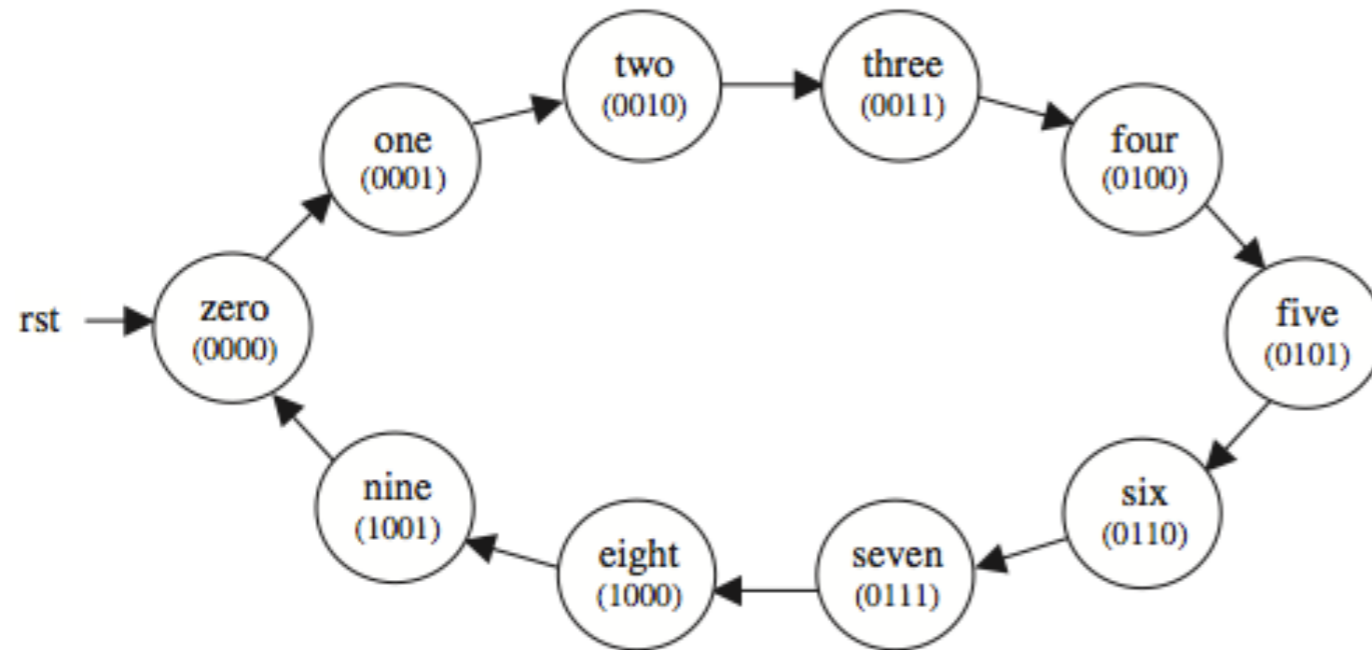
Example #1: BCD counter

- The state diagram of a 0-to-9 circular counter.



- Each state-name corresponds to the decimal value of the output.
- We could easily perform do this BCD counter using all that we learned thus far.
- However we are going to try to do this using a FSM, where each counter outcome is a particular state.
- As you can see, the output depends only on the current state.

Example #1: BCD counter

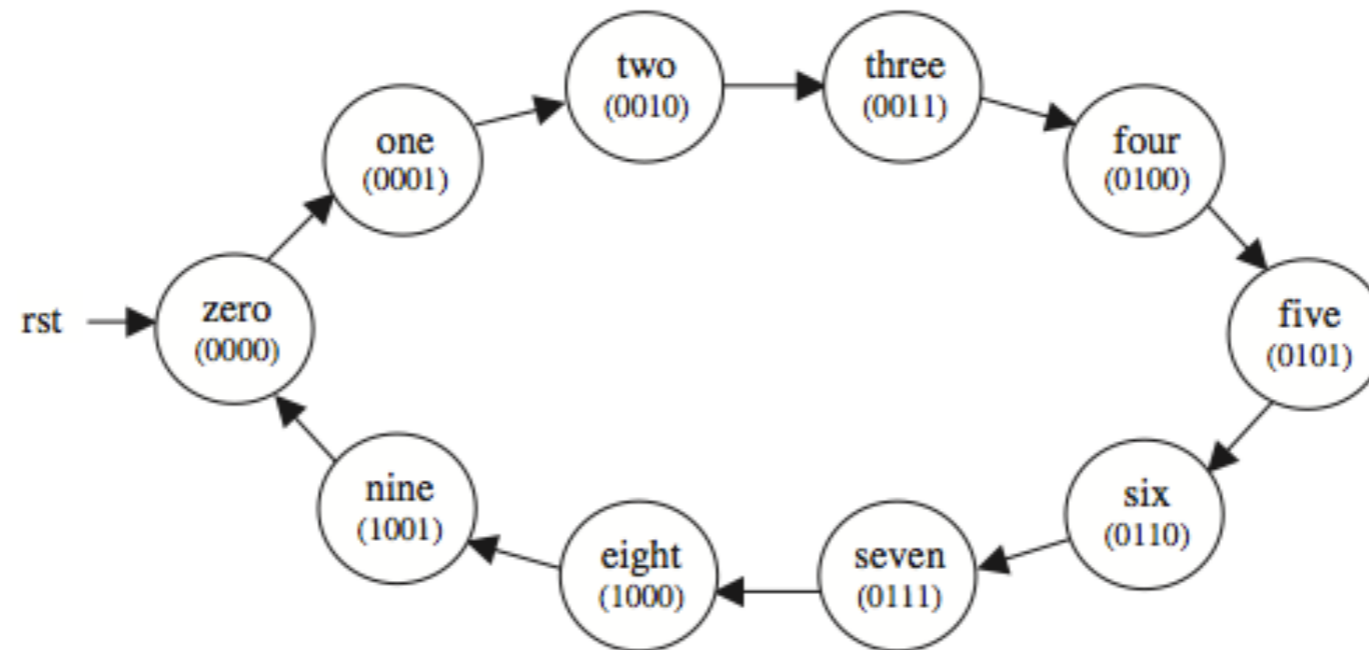


State name: three

Circuit output: 0011

- The problem with a FSM implementation is that when the number of states is large it becomes cumbersome to enumerate them all.
- This is no problem if we were trying to count without using a FSM.

Example #1: BCD counter

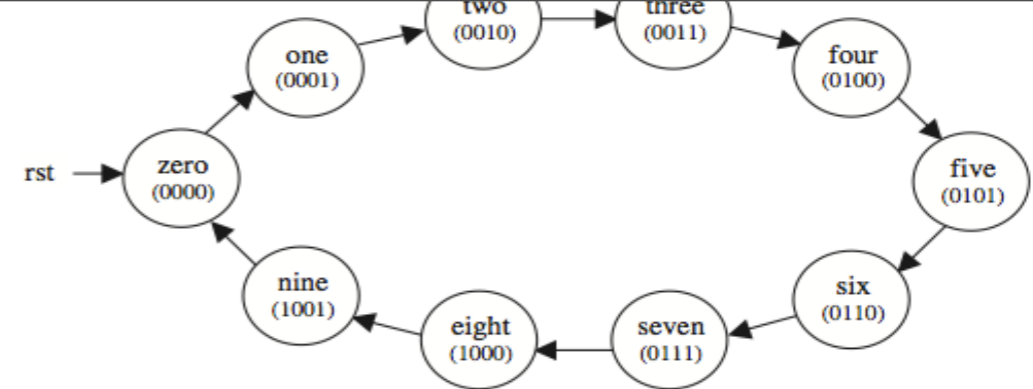


- The operation of this FSM is simple.
- At every rising-edge clock cycle it will change its state.
- When each state changes, the circuit will return a new output value.
- When **rst** is pressed, the present state will now be state zero.

```

1  -----
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  -----
5  ENTITY counter IS
6      PORT ( clk, rst: IN STD_LOGIC;
7              count: OUT STD_LOGIC_VECTOR (3 DOWNT0 0));
8  END counter;
9  -----
10 ARCHITECTURE state_machine OF counter IS
11     TYPE state IS (zero, one, two, three, four,
12                     five, six, seven, eight, nine);
13     SIGNAL pr_state, nx_state: state;
14 BEGIN
15     ----- Lower section: -----
16     PROCESS (rst, clk)
17     BEGIN
18         IF (rst='1') THEN
19             pr_state <= zero;
20         ELSIF (clk'EVENT AND clk='1') THEN
21             pr_state <= nx_state;
22         END IF;
23     END PROCESS;
24     ----- Upper section: -----
25     PROCESS (pr_state)
26     BEGIN
27         CASE pr_state IS
28             WHEN zero =>
29                 count <= "0000";
30                 nx_state <= one;
31             WHEN one =>
32                 count <= "0001";
33                 nx_state <= two;

```



```

34     WHEN two =>
35         count <= "0010";
36         nx_state <= three;
37     WHEN three =>
38         count <= "0011";
39         nx_state <= four;
40     WHEN four =>
41         count <= "0100";
42         nx_state <= five;
43     WHEN five =>
44         count <= "0101";
45         nx_state <= six;
46     WHEN six =>
47         count <= "0110";
48         nx_state <= seven;
49     WHEN seven =>
50         count <= "0111";
51         nx_state <= eight;
52     WHEN eight =>
53         count <= "1000";
54         nx_state <= nine;
55     WHEN nine =>
56         count <= "1001";
57         nx_state <= zero;
58     END CASE;
59     END PROCESS;
60 END state_machine;
61 -----

```

The **state** (enumerated data type) is defined, and used for the **pr_state** (present state) and **nx_state** (next state).

```
1  -----
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  -----
5  ENTITY counter IS
6      PORT ( clk, rst: IN STD_LOGIC;
7              count: OUT STD_LOGIC_VECTOR (3 DOWNT0 0));
8  END counter;
9  -----
10 ARCHITECTURE state_machine OF counter IS
11     TYPE state IS (zero, one, two, three, four,
12                    five, six, seven, eight, nine);
13     SIGNAL pr_state, nx_state: state;
14 BEGIN
15     ----- Lower section: -----
16     PROCESS (rst, clk)
17     BEGIN
18         IF (rst='1') THEN
19             pr_state <= zero;
20         ELSIF (clk'EVENT AND clk='1') THEN
21             pr_state <= nx_state;
22         END IF;
23     END PROCESS;
24     ----- Upper section: -----
25     PROCESS (pr_state)
26     BEGIN
27         CASE pr_state IS
28             WHEN zero =>
29                 count <= "0000";
30                 nx_state <= one;
31             WHEN one =>
32                 count <= "0001";
33                 nx_state <= two;
```

This is how you
use enumerated
data types.

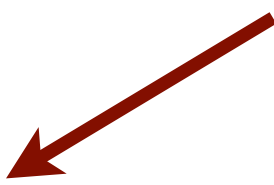
```
34         WHEN two =>
35             count <= "0010";
36             nx_state <= three;
37         WHEN three =>
38             count <= "0011";
39             nx_state <= four;
40         WHEN four =>
41             count <= "0100";
42             nx_state <= five;
43         WHEN five =>
44             count <= "0101";
45             nx_state <= six;
46         WHEN six =>
47             count <= "0110";
48             nx_state <= seven;
49         WHEN seven =>
50             count <= "0111";
51             nx_state <= eight;
52         WHEN eight =>
53             count <= "1000";
54             nx_state <= nine;
55         WHEN nine =>
56             count <= "1001";
57             nx_state <= zero;
58         END CASE;
59     END PROCESS;
60 END state_machine;
61 -----
```

```

1  -----
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  -----
5  ENTITY counter IS
6      PORT ( clk, rst: IN STD_LOGIC;
7              count: OUT STD_LOGIC_VECTOR (3 DOWNT0 0));
8  END counter;
9  -----
10 ARCHITECTURE state_machine OF counter IS
11     TYPE state IS (zero, one, two, three, four,
12                    five, six, seven, eight, nine);
13     SIGNAL pr_state, nx_state: state;
14 BEGIN
15     ----- Lower section: -----
16     PROCESS (rst, clk)
17     BEGIN
18         IF (rst='1') THEN
19             pr_state <= zero;
20         ELSIF (clk'EVENT AND clk='1') THEN
21             pr_state <= nx_state;
22         END IF;
23     END PROCESS;
24     ----- Upper section: -----
25     PROCESS (pr_state)
26     BEGIN
27         CASE pr_state IS
28             WHEN zero =>
29                 count <= "0000";
30                 nx_state <= one;
31             WHEN one =>
32                 count <= "0001";
33                 nx_state <= two;

```

The output **count** is the circuit output at each stage.



```

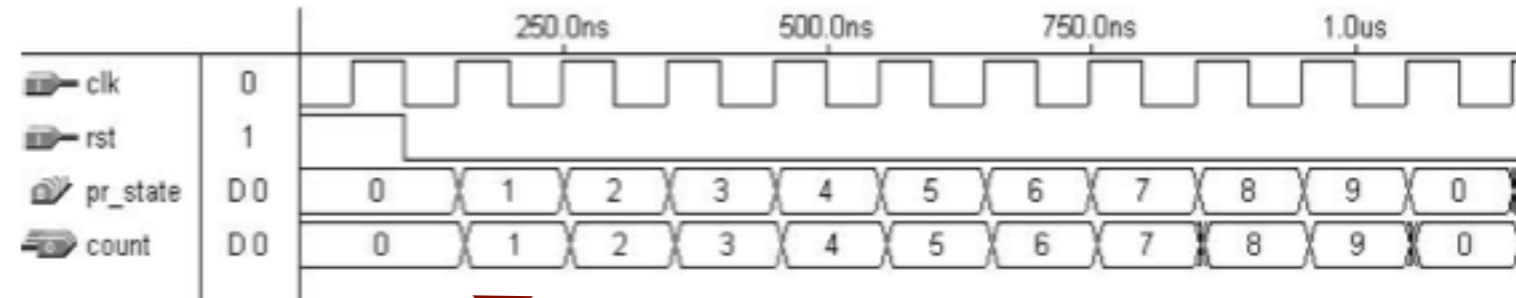
34         WHEN two =>
35             count <= "0010";
36             nx_state <= three;
37         WHEN three =>
38             count <= "0011";
39             nx_state <= four;
40         WHEN four =>
41             count <= "0100";
42             nx_state <= five;
43         WHEN five =>
44             count <= "0101";
45             nx_state <= six;
46         WHEN six =>
47             count <= "0110";
48             nx_state <= seven;
49         WHEN seven =>
50             count <= "0111";
51             nx_state <= eight;
52         WHEN eight =>
53             count <= "1000";
54             nx_state <= nine;
55         WHEN nine =>
56             count <= "1001";
57             nx_state <= zero;
58         END CASE;
59     END PROCESS;
60 END state_machine;
61 -----

```

```

1  -----
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  -----
5  ENTITY counter IS
6      PORT ( clk, rst: IN STD_LOGIC;
7              count: OUT STD_LOGIC_VECTOR (3 DOWNTO 0));
8  END counter;
9  -----
10 ARCHITECTURE state_machine OF counter IS
11     TYPE state IS (zero, one, two, three, four,
12                    five, six, seven, eight, nine);
13     SIGNAL pr_state, nx_state: state;
14 BEGIN
15     ----- Lower section: -----
16     PROCESS (rst, clk)
17     BEGIN
18         IF (rst='1') THEN
19             pr_state <= zero;
20         ELSIF (clk'EVENT AND clk='1') THEN
21             pr_state <= nx_state;
22         END IF;
23     END PROCESS;
24     ----- Upper section: -----
25     PROCESS (pr_state)
26     BEGIN
27         CASE pr_state IS
28             WHEN zero =>
29                 count <= "0000";
30                 nx_state <= one;
31             WHEN one =>
32                 count <= "0001";
33                 nx_state <= two;

```



```

34     WHEN two =>
35         count <= "0010";
36         nx_state <= three;
37     WHEN three =>
38         count <= "0011";
39         nx_state <= four;
40     WHEN four =>
41         count <= "0100";
42         nx_state <= five;
43     WHEN five =>
44         count <= "0101";
45         nx_state <= six;
46     WHEN six =>
47         count <= "0110";
48         nx_state <= seven;
49     WHEN seven =>
50         count <= "0111";
51         nx_state <= eight;
52     WHEN eight =>
53         count <= "1000";
54         nx_state <= nine;
55     WHEN nine =>
56         count <= "1001";
57         nx_state <= zero;
58     END CASE;
59     END PROCESS;
60 END state_machine;
61 -----

```

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;

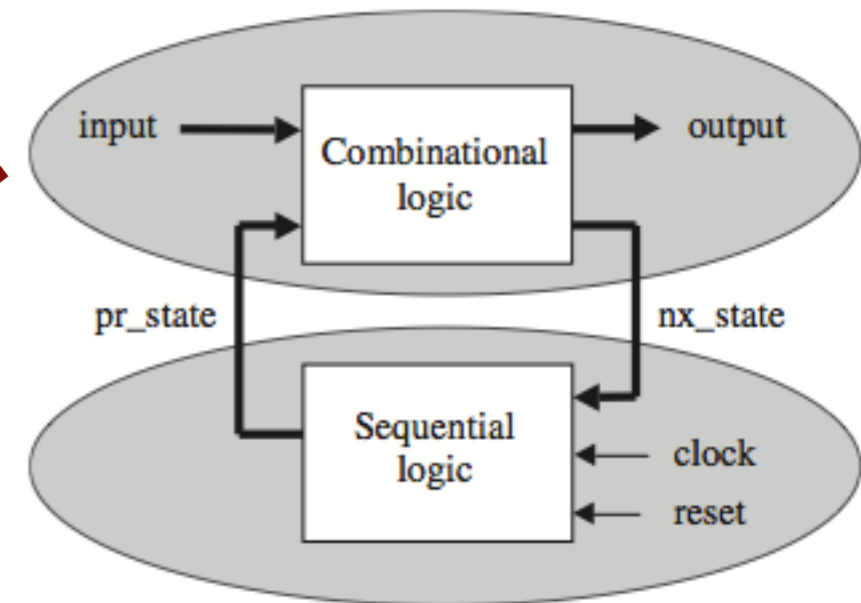
-----
ENTITY <entity_name> IS
    PORT ( input: IN <data_type>;
          reset, clock: IN STD_LOGIC;
          output: OUT <data_type>);
END <entity_name>;

-----
ARCHITECTURE <arch_name> OF <entity_name> IS
    TYPE state IS (state0, state1, state2, state3, ...);
    SIGNAL pr_state, nx_state: state;
BEGIN
    ----- Lower section: -----
    PROCESS (reset, clock)
    BEGIN
        IF (reset='1') THEN
            pr_state <= state0;
        ELSIF (clock'EVENT AND clock='1') THEN
            pr_state <= nx_state;
        END IF;
    END PROCESS;
    ----- Upper section: -----
    PROCESS (input, pr_state)
    BEGIN
        CASE pr_state IS
            WHEN state0 =>
                IF (input = ...) THEN
                    output <= <value>;
                    nx_state <= state1;
                ELSE ...
                END IF;
            WHEN state1 =>
                IF (input = ...) THEN
                    output <= <value>;
                    nx_state <= state2;
                ELSE ...
                END IF;
            WHEN state2 =>
                IF (input = ...) THEN
                    output <= <value>;
                    nx_state <= state3;
                ELSE ...
                END IF;
            ...
        END CASE;
    END PROCESS;
END <arch_name>;

```

FSM template #1

Remember, in a FSM the next state depends on the current input and the current state.



- This template allows for the design of any FSM.
- The design of the lower section of the state machine is completely separated from that of the upper section.

```

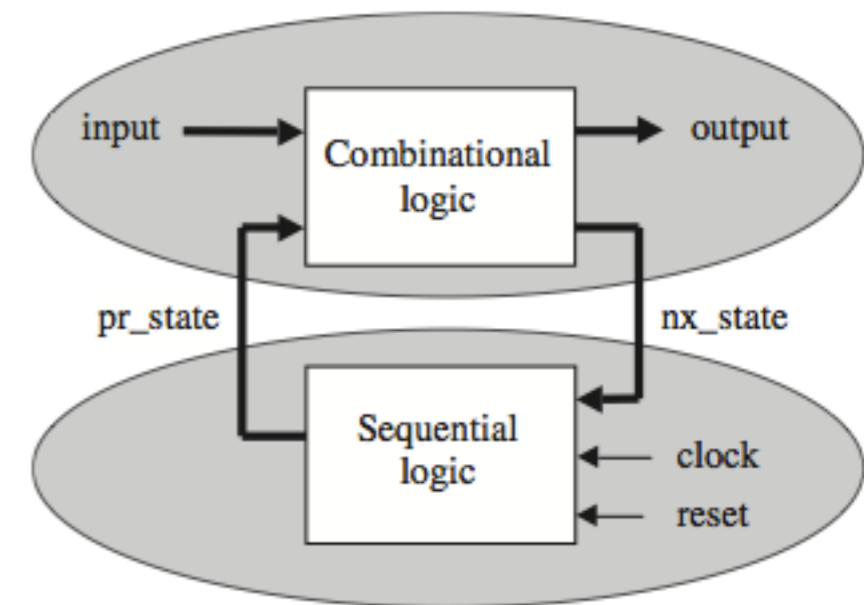
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-----
ENTITY <entity_name> IS
    PORT ( input: IN <data_type>;
          reset, clock: IN STD_LOGIC;
          output: OUT <data_type>);
END <entity_name>;

-----
ARCHITECTURE <arch_name> OF <entity_name> IS
    TYPE state IS (state0, state1, state2, state3, ...);
    SIGNAL pr_state, nx_state: state;
BEGIN
    ----- Lower section: -----
    PROCESS (reset, clock)
    BEGIN
        IF (reset='1') THEN
            pr_state <= state0;
        ELSIF (clock'EVENT AND clock='1') THEN
            pr_state <= nx_state;
        END IF;
    END PROCESS;
    ----- Upper section: -----
    PROCESS (input, pr_state)
    BEGIN
        CASE pr_state IS
            WHEN state0 =>
                IF (input = ...) THEN
                    output <= <value>;
                    nx_state <= state1;
                ELSE ...
                END IF;
            WHEN state1 =>
                IF (input = ...) THEN
                    output <= <value>;
                    nx_state <= state2;
                ELSE ...
                END IF;
            WHEN state2 =>
                IF (input = ...) THEN
                    output <= <value>;
                    nx_state <= state3;
                ELSE ...
                END IF;
            ...
        END CASE;
    END PROCESS;
END <arch_name>;

```

FSM template #1



- All states of the machine are always explicitly declared using an enumerated data type.

```

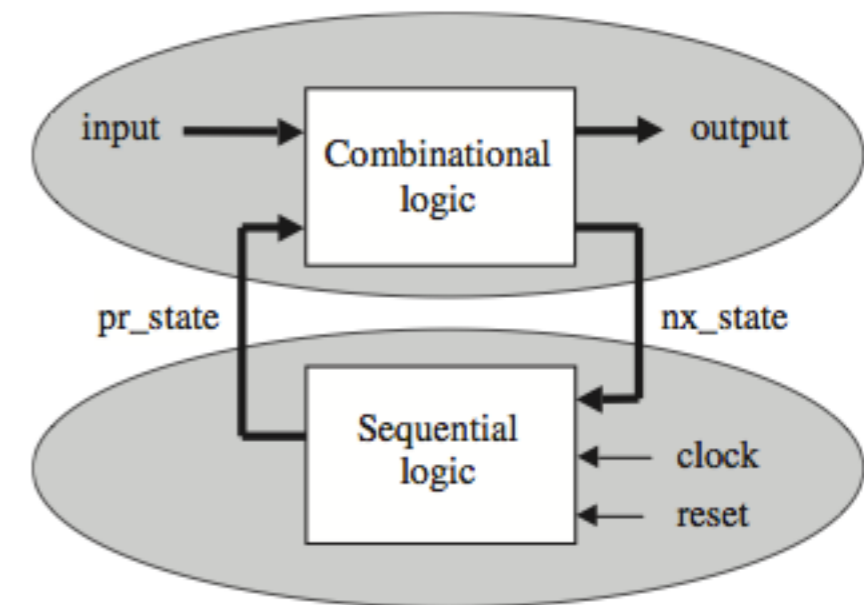
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-----
ENTITY <entity_name> IS
    PORT ( input: IN <data_type>;
          reset, clock: IN STD_LOGIC;
          output: OUT <data_type>);
END <entity_name>;

-----
ARCHITECTURE <arch_name> OF <entity_name> IS
    TYPE state IS (state0, state1, state2, state3, ...);
    SIGNAL pr_state, nx_state: state;
BEGIN
    ----- Lower section: -----
    PROCESS (reset, clock)
    BEGIN
        IF (reset='1') THEN
            pr_state <= state0;
        ELSIF (clock'EVENT AND clock='1') THEN
            pr_state <= nx_state;
        END IF;
    END PROCESS;
    ----- Upper section: -----
    PROCESS (input, pr_state)
    BEGIN
        CASE pr_state IS
            WHEN state0 =>
                IF (input = ...) THEN
                    output <= <value>;
                    nx_state <= state1;
                ELSE ...
                END IF;
            WHEN state1 =>
                IF (input = ...) THEN
                    output <= <value>;
                    nx_state <= state2;
                ELSE ...
                END IF;
            WHEN state2 =>
                IF (input = ...) THEN
                    output <= <value>;
                    nx_state <= state3;
                ELSE ...
                END IF;
            ...
        END CASE;
    END PROCESS;
END <arch_name>;

```

FSM template #1



- The flip-flops are in the lower section, so clock and reset are connected to it.
- The other lower section's input is nx_state (next state), while pr_state (present state) is its only output.
- The circuit of the lower section is sequential, so a PROCESS is required.

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;

-----
ENTITY <entity_name> IS
    PORT ( input: IN <data_type>;
          reset, clock: IN STD_LOGIC;
          output: OUT <data_type>);
END <entity_name>;

-----
ARCHITECTURE <arch_name> OF <entity_name> IS
    TYPE state IS (state0, state1, state2, state3, ...);
    SIGNAL pr_state, nx_state: state;
BEGIN
    ----- Lower section: -----
    PROCESS (reset, clock)
    BEGIN
        IF (reset='1') THEN
            pr_state <= state0;
        ELSIF (clock'EVENT AND clock='1') THEN
            pr_state <= nx_state;
        END IF;
    END PROCESS;
    ----- Upper section: -----
    PROCESS (input, pr_state)
    BEGIN
        CASE pr_state IS
            WHEN state0 =>
                IF (input = ...) THEN
                    output <= <value>;
                    nx_state <= state1;
                ELSE ...
                END IF;
            WHEN state1 =>
                IF (input = ...) THEN
                    output <= <value>;
                    nx_state <= state2;
                ELSE ...
                END IF;
            WHEN state2 =>
                IF (input = ...) THEN
                    output <= <value>;
                    nx_state <= state3;
                ELSE ...
                END IF;
            ...
        END CASE;
    END PROCESS;
END <arch_name>;

```

FSM template #1

- Here is some typical code for the lower section:

```

PROCESS (reset, clock)
BEGIN
    IF (reset='1') THEN
        pr_state <= state0;
    ELSIF (clock'EVENT AND clock='1') THEN
        pr_state <= nx_state;
    END IF;
END PROCESS;

```

- It has an, asynchronous reset, which determines the initial state of the system (state0),
- ... Followed by the synchronous storage of nx_state (at the positive transition of clock), which will produce pr_state at the lower section's output.

```

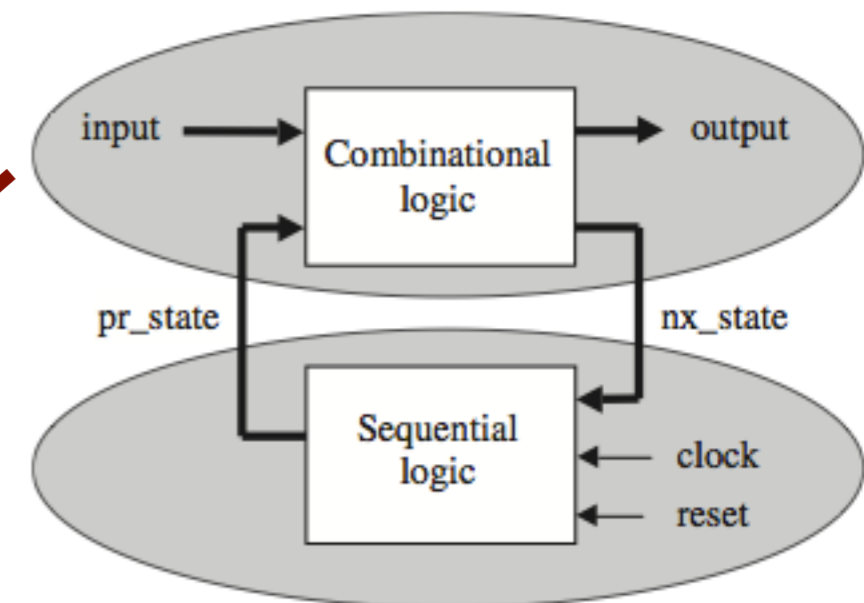
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-----
ENTITY <entity_name> IS
    PORT ( input: IN <data_type>;
          reset, clock: IN STD_LOGIC;
          output: OUT <data_type>);
END <entity_name>;

-----
ARCHITECTURE <arch_name> OF <entity_name> IS
    TYPE state IS (state0, state1, state2, state3, ...);
    SIGNAL pr_state, nx_state: state;
BEGIN
    ----- Lower section: -----
    PROCESS (reset, clock)
    BEGIN
        IF (reset='1') THEN
            pr_state <= state0;
        ELSIF (clock'EVENT AND clock='1') THEN
            pr_state <= nx_state;
        END IF;
    END PROCESS;
    ----- Upper section: -----
    PROCESS (input, pr_state)
    BEGIN
        CASE pr_state IS
            WHEN state0 =>
                IF (input = ...) THEN
                    output <= <value>;
                    nx_state <= state1;
                ELSE ...
                END IF;
            WHEN state1 =>
                IF (input = ...) THEN
                    output <= <value>;
                    nx_state <= state2;
                ELSE ...
                END IF;
            WHEN state2 =>
                IF (input = ...) THEN
                    output <= <value>;
                    nx_state <= state3;
                ELSE ...
                END IF;
            ...
        END CASE;
    END PROCESS;
END <arch_name>;

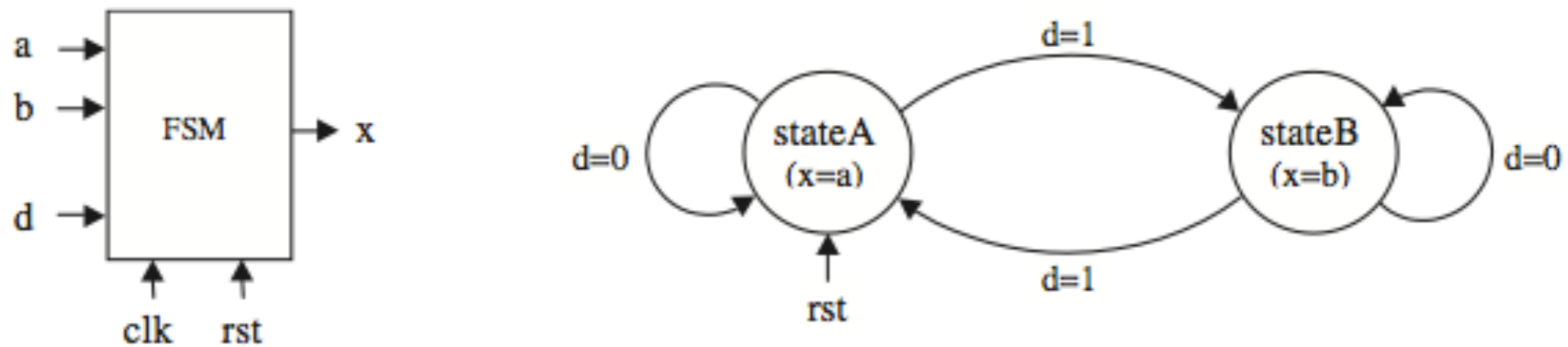
```

FSM template #1



- The upper section is fully combinational, so its code does not need to be sequential; concurrent code can be used as well.
- The CASE statement plays a the central role.
- This code does two things: (a) it assigns the output value and (b) it establishes the next state.

Example #2: Simple FSM



- The system has two states (stateA and stateB), and must change from one to the other every time $d = '1'$ is received.
- The desired output is $x=a$ when the machine is in stateA, or $x=b$ when in stateB. The initial (reset) state is stateA.

Implemented using FSM template #1

```

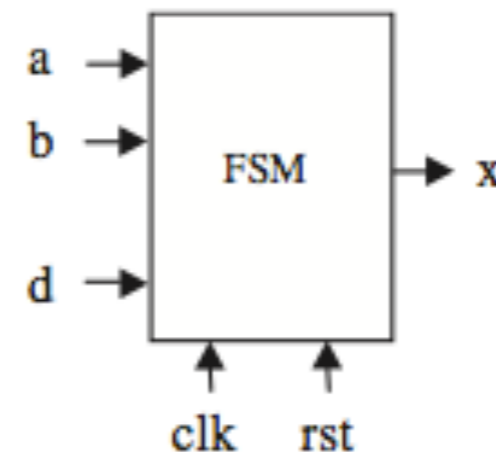
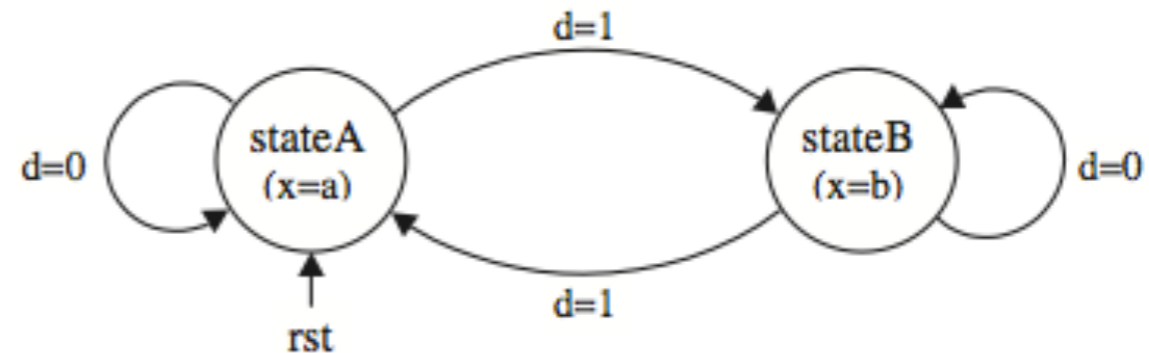
1  -----
2  ENTITY simple_fsm IS
3      PORT ( a, b, d, clk, rst: IN BIT;
4              x: OUT BIT);
5  END simple_fsm;
6  -----
7  ARCHITECTURE simple_fsm OF simple_fsm IS
8      TYPE state IS (stateA, stateB);
9      SIGNAL pr_state, nx_state: state;
10 BEGIN
11     ----- Lower section: -----
12     PROCESS (rst, clk)
13     BEGIN
14         IF (rst='1') THEN
15             pr_state <= stateA;
16         ELSIF (clk'EVENT AND clk='1') THEN
17             pr_state <= nx_state;
18         END IF;
19     END PROCESS;
20     ----- Upper section: -----
21     PROCESS (a, b, d, pr_state)
22     BEGIN
23         CASE pr_state IS
24             WHEN stateA =>
25                 x <= a;
26                 IF (d='1') THEN nx_state <= stateB;
27                 ELSE nx_state <= stateA;
28                 END IF;
29             WHEN stateB =>
30                 x <= b;

```

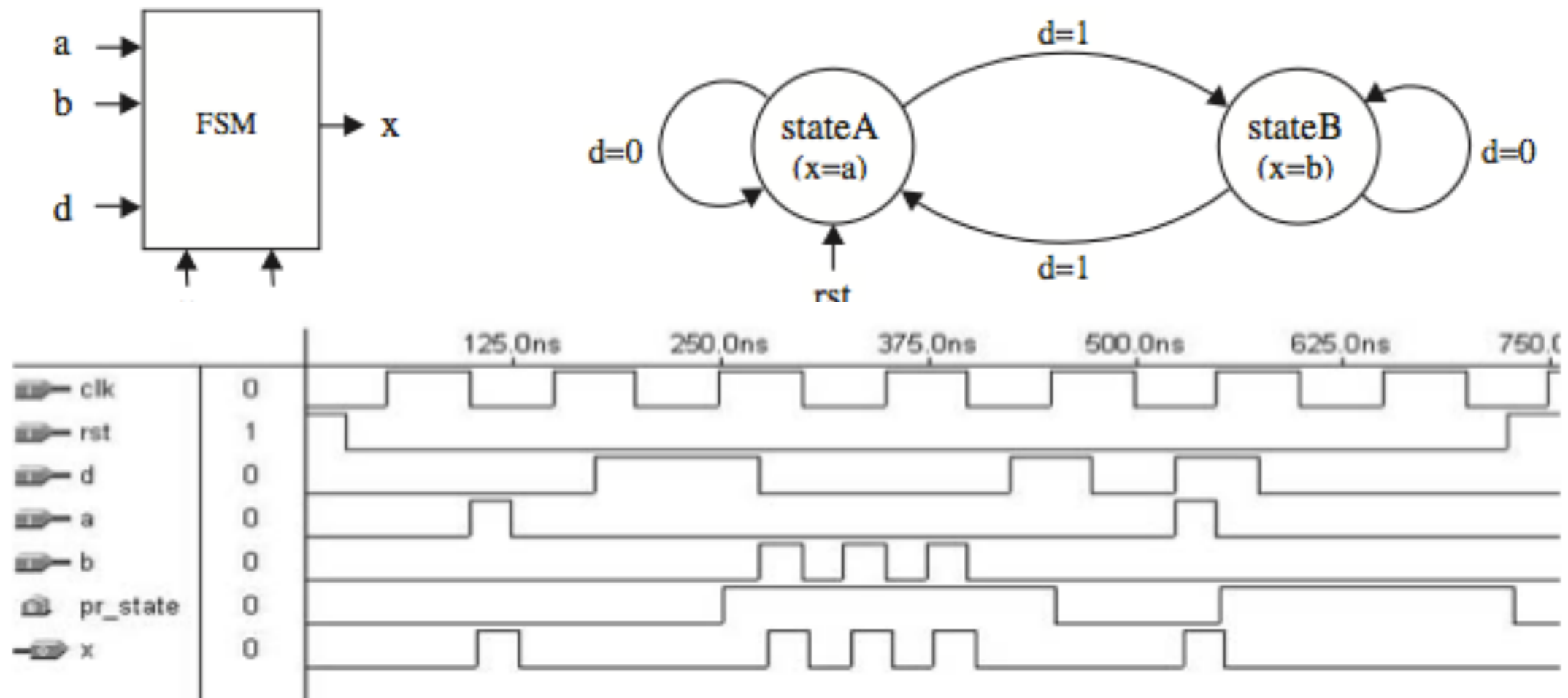
```

31             IF (d='1') THEN nx_state <= stateA;
32             ELSE nx_state <= stateB;
33             END IF;
34         END CASE;
35     END PROCESS;
36 END simple_fsm;
37 -----

```



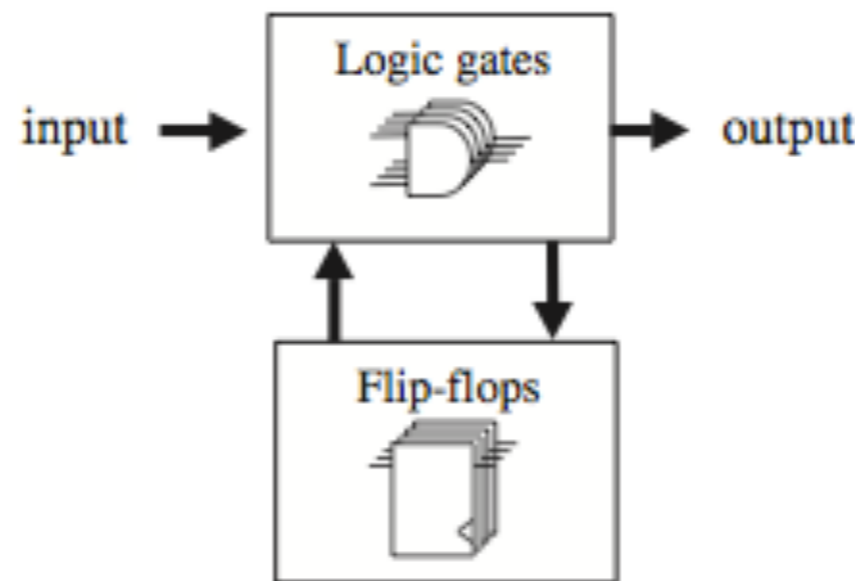
Example #2: Simple FSM



- State only changes on the rising edge of CLK
- The output (*x*), which in this case **does** depend on the inputs (*a* or *b*, depending on which state the machine is in), varies when *a* or *b* vary, regardless of *clk*.

Using FSM template #1

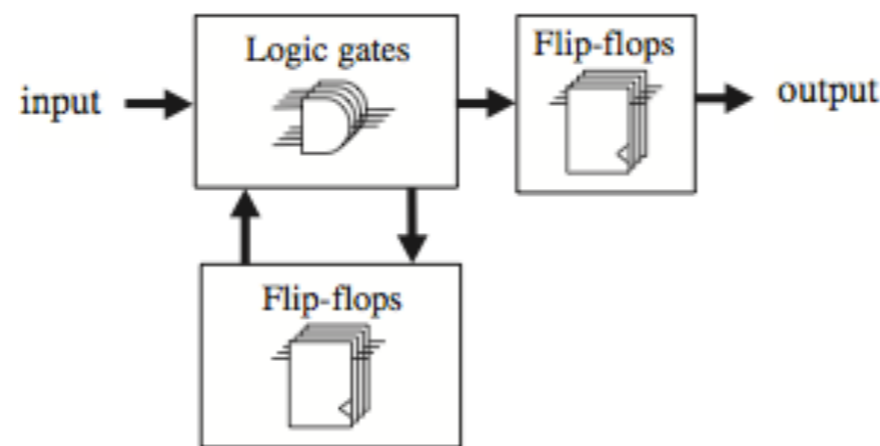
- In FSM template #1 only **pr_state** is stored.
- Therefore, the overall circuit can be summarized as follows:



- Notice that in this case the output might change when the input changes (asynchronous output).

What if we want a synchronous output?

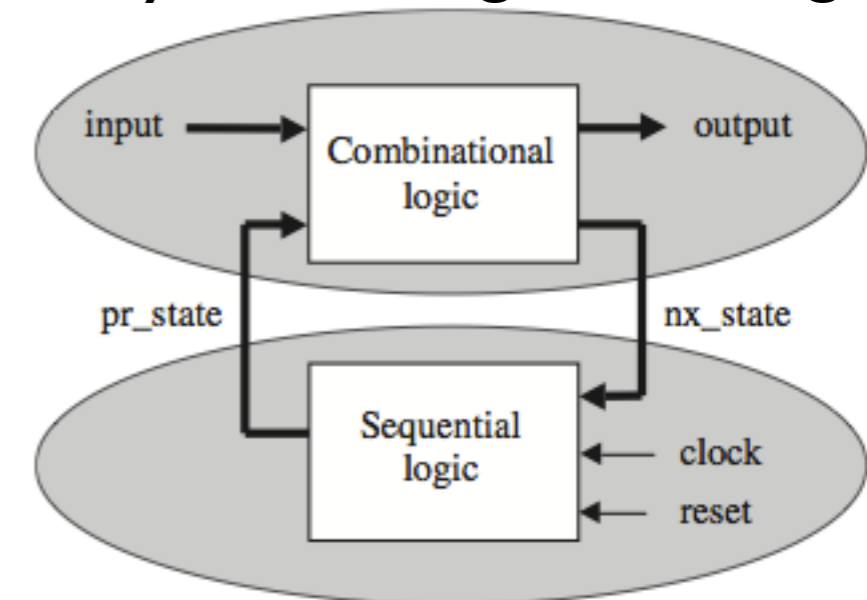
- In many applications, the signals are required to be synchronous, so the output should be updated **only** when the proper clock edge occurs.
- To make FSM machines synchronous, the output must be stored as well.



- This is the goal of the FSM template #2.

FSM template #2

Where output is **ONLY** updated only at a rising clock edge



- Very few modifications from template #1 are needed.
- For example, we can use an additional signal (say, temp) to compute the output value (upper section), but only pass its value to the actual output signal when a clock event occurs (lower section).

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;

-----
ENTITY <ent_name> IS
    PORT (input: IN <data_type>;
          reset, clock: IN STD_LOGIC;
          output: OUT <data_type>);
END <ent_name>;

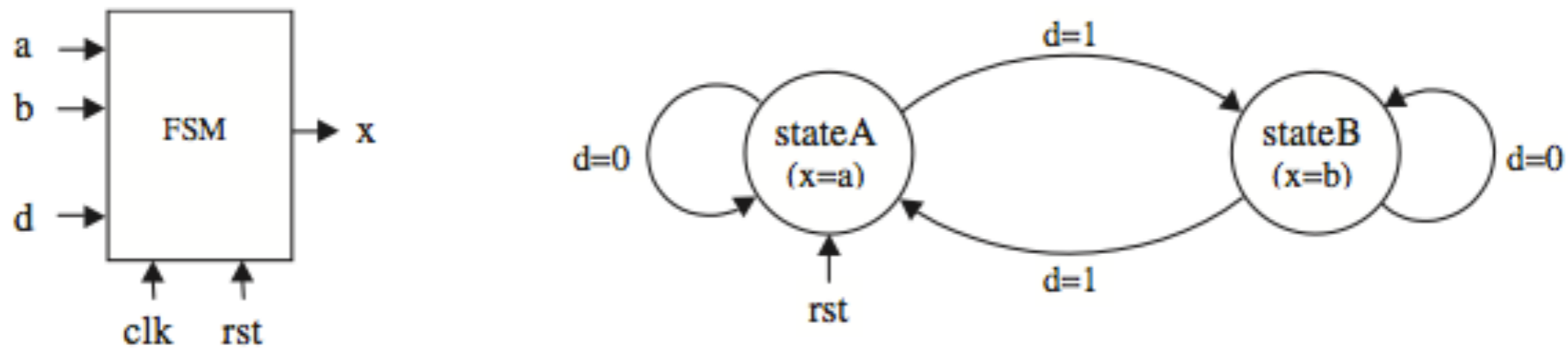
-----
ARCHITECTURE <arch_name> OF <ent_name> IS
    TYPE states IS (state0, state1, state2, state3, ...);
    SIGNAL pr_state, nx_state: states;
    SIGNAL temp: <data_type>;
BEGIN
    ----- Lower section: -----
    PROCESS (reset, clock)
    BEGIN
        IF (reset='1') THEN
            pr_state <= state0;
        ELSIF (clock'EVENT AND clock='1') THEN
            output <= temp;
            pr_state <= nx_state;
        END IF;
    END PROCESS;

    ----- Upper section: -----
    PROCESS (pr_state)
    BEGIN
        CASE pr_state IS
            WHEN state0 =>
                temp <= <value>;
                IF (condition) THEN nx_state <= state1;
                ...
            END IF;
            WHEN state1 =>
                temp <= <value>;
                IF (condition) THEN nx_state <= state2;
                ...
            END IF;
            WHEN state2 =>
                temp <= <value>;
                IF (condition) THEN nx_state <= state3;
                ...
            END IF;
            ...
        END CASE;
    END PROCESS;
END <arch_name>;
    
```

FSM output is here

Storing a temp!

Example #3: Simple FSM whose output only changes on a rising clock!



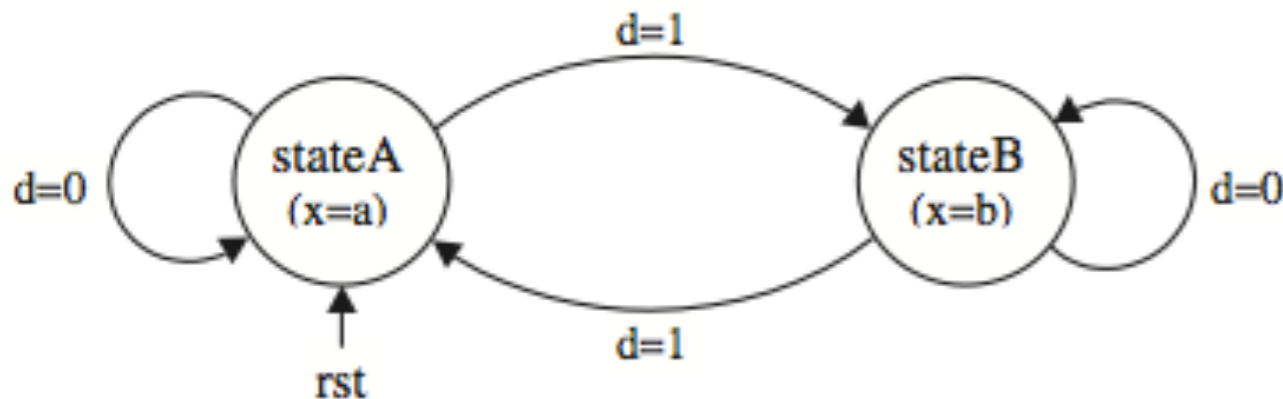
- The system has two states (stateA and stateB), and must change from one to the other every time $d = '1'$ is received.
- The desired output is $x=a$ when the machine is in stateA **and** on a rising clock edge, or $x=b$ when in stateB **and** on a rising clock edge. The initial (reset) state is stateA.

```

1  -----
2  ENTITY simple_fsm IS
3      PORT ( a, b, d, clk, rst: IN BIT;
4              x: OUT BIT);
5  END simple_fsm;
6  -----
7  ARCHITECTURE simple_fsm OF simple_fsm IS
8      TYPE state IS (stateA, stateB);
9      SIGNAL pr_state, nx_state: state;
10     SIGNAL temp: BIT;

```

- If the input (a or b) changes during between two consecutive clock edges, the change might not be observed by the circuit; moreover, when observed, it will be delayed with respect to the input.

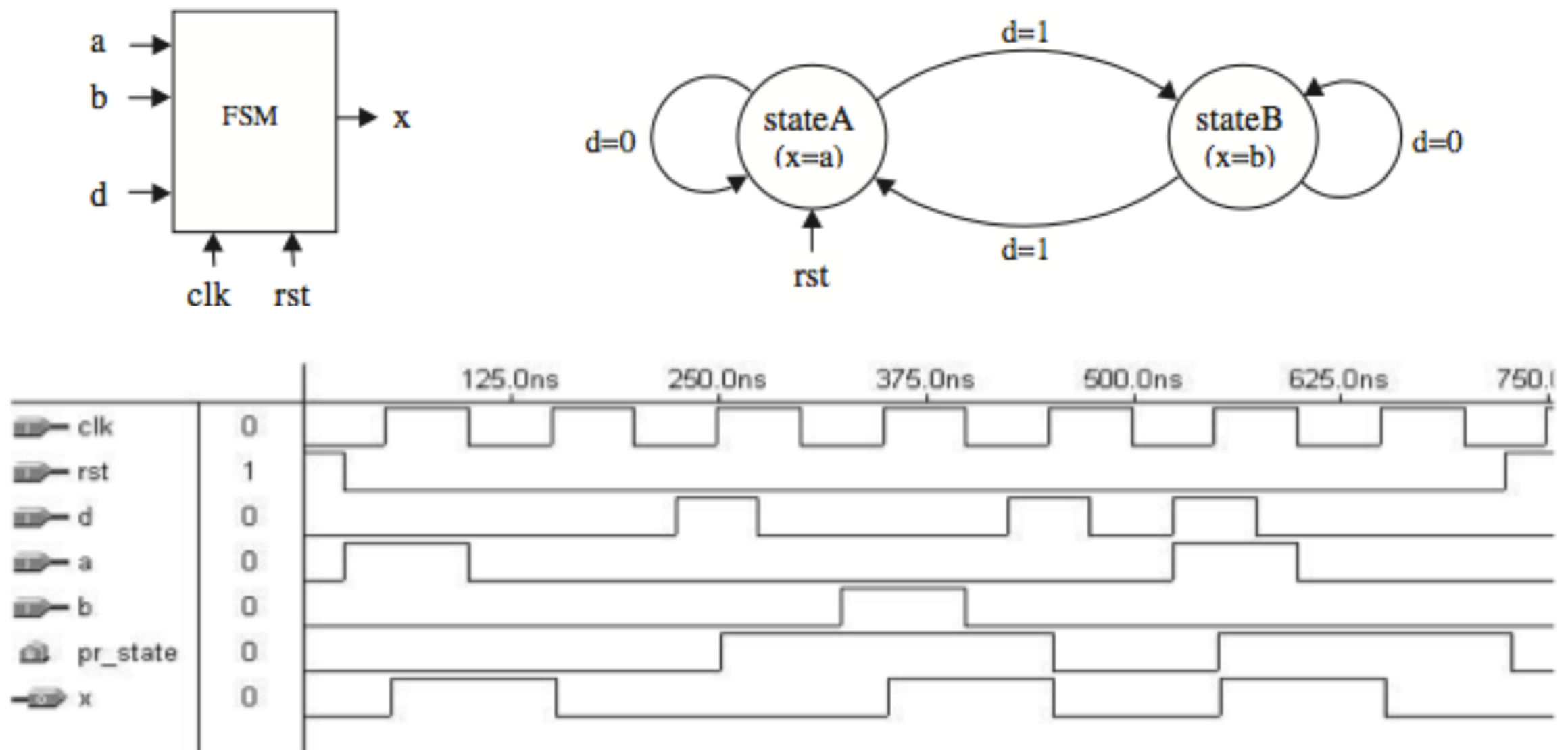


```

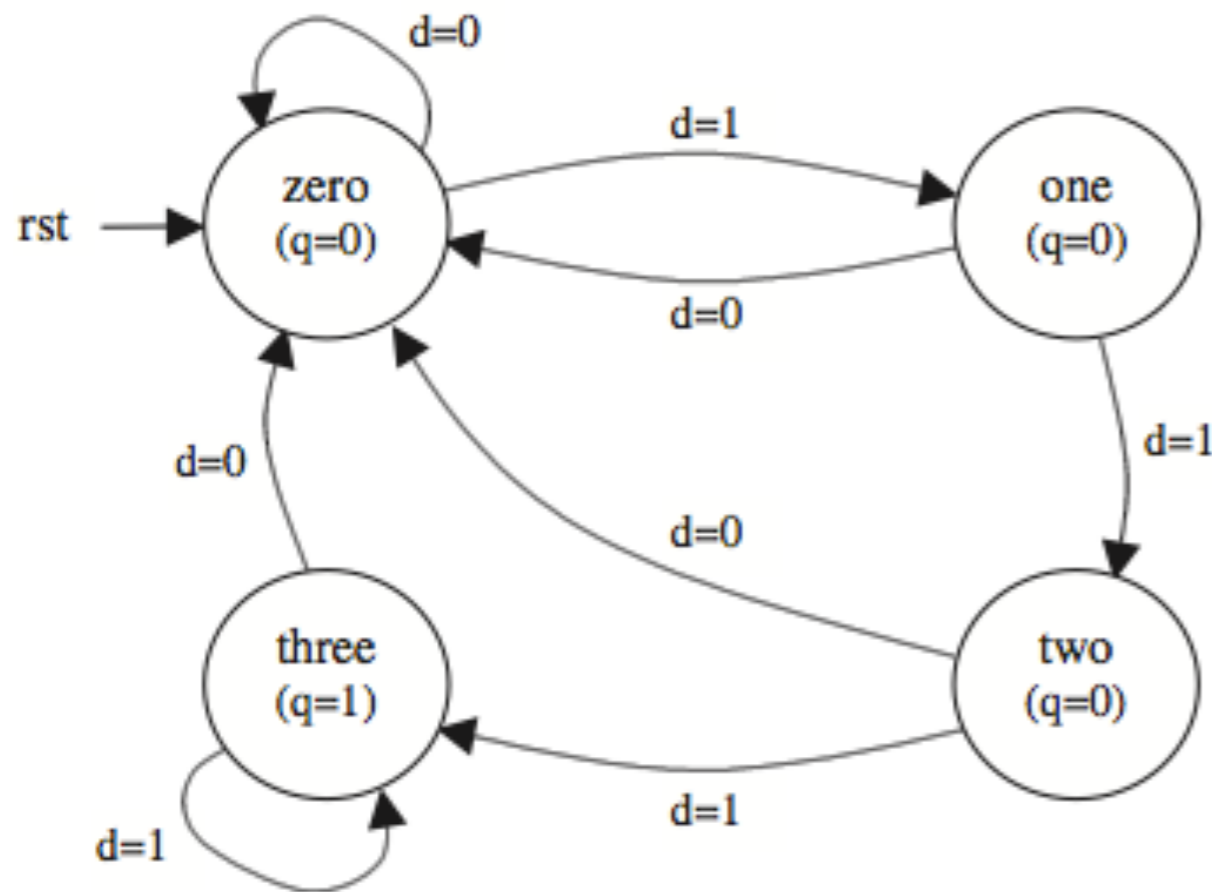
11 BEGIN
12     ----- Lower section: -----
13     PROCESS (rst, clk)
14     BEGIN
15         IF (rst='1') THEN
16             pr_state <= stateA;
17         ELSIF (clk'EVENT AND clk='1') THEN
18             x <= temp;
19             pr_state <= nx_state;
20         END IF;
21     END PROCESS;
22     ----- Upper section: -----
23     PROCESS (a, b, d, pr_state)
24     BEGIN
25         CASE pr_state IS
26             WHEN stateA =>
27                 temp <= a;
28                 IF (d='1') THEN nx_state <= stateB;
29                 ELSE nx_state <= stateA;
30                 END IF;
31             WHEN stateB =>
32                 temp <= b;
33                 IF (d='1') THEN nx_state <= stateA;
34                 ELSE nx_state <= stateB;
35                 END IF;
36         END CASE;
37     END PROCESS;
38 END simple_fsm;
39 -----

```

Example #3: Simulation



Example #4 : String detector



We want to design a circuit that takes as input a serial bit stream and outputs a '1' whenever the sequence "111" occurs.

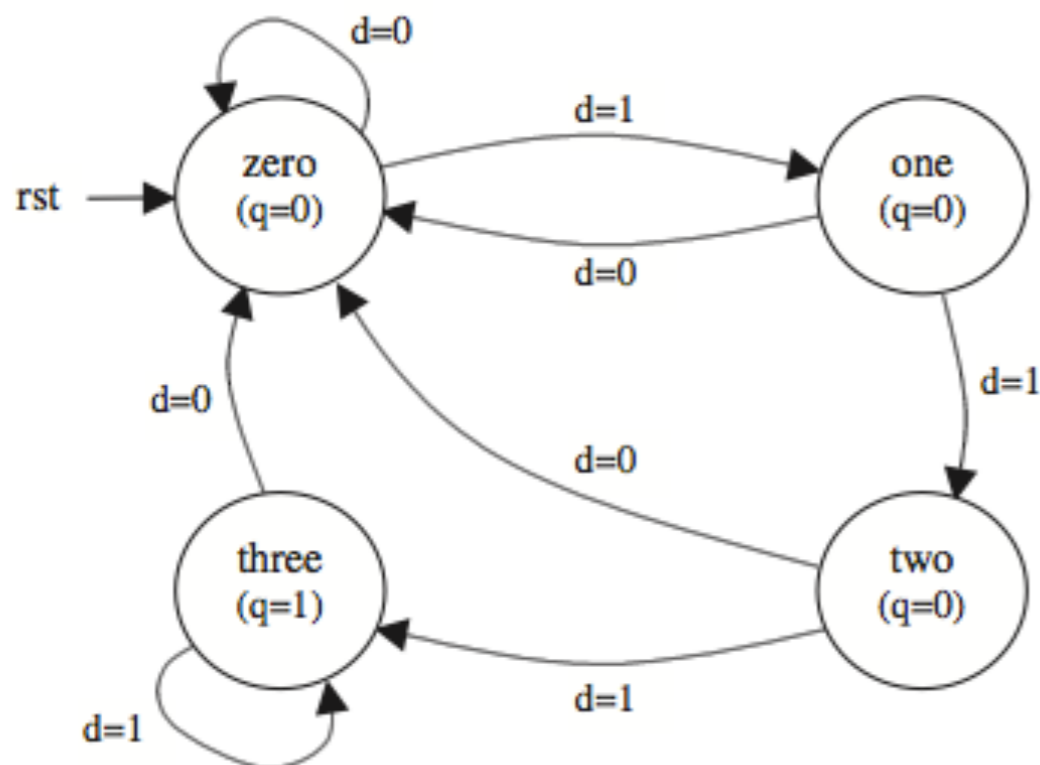
Overlaps must also be considered, that is, if ...011110... occurs, than the output should remain active for three consecutive clock cycles.

There are four states, which we called zero, one, two, and three, with the name corresponding to the number of consecutive '1's detected.

```

1  -----
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  -----
5  ENTITY string_detector IS
6      PORT ( d, clk, rst: IN BIT;
7              q: OUT BIT);
8  END string_detector;
9  -----
10 ARCHITECTURE my_arch OF string_detector IS
11     TYPE state IS (zero, one, two, three);
12     SIGNAL pr_state, nx_state: state;
13 BEGIN
14     ----- Lower section: -----
15     PROCESS (rst, clk)
16     BEGIN
17         IF (rst='1') THEN
18             pr_state <= zero;
19         ELSIF (clk'EVENT AND clk='1') THEN
20             pr_state <= nx_state;
21         END IF;
22     END PROCESS;

```



```

23 ----- Upper section: -----
24 PROCESS (d, pr_state)
25 BEGIN
26     CASE pr_state IS
27         WHEN zero =>
28             q <= '0';
29             IF (d='1') THEN nx_state <= one;
30             ELSE nx_state <= zero;
31             END IF;
32         WHEN one =>
33             q <= '0';
34             IF (d='1') THEN nx_state <= two;
35             ELSE nx_state <= zero;
36             END IF;
37         WHEN two =>
38             q <= '0';
39             IF (d='1') THEN nx_state <= three;
40             ELSE nx_state <= zero;
41             END IF;
42         WHEN three =>
43             q <= '1';
44             IF (d='0') THEN nx_state <= zero;
45             ELSE nx_state <= three;
46             END IF;
47     END CASE;
48 END PROCESS;
49 END my_arch;
50 -----

```

Even if **d** changes, the upper section process will still use the **pr_state** to make a decision. So, **q** will only change on rising edge.

Example #4 : String detector

