Integrated Circuits Design by FPGA

م.م. أحمد مؤيد عبدالحسين
جامعة الفرات الأوسط التقنية / الكلية التقنية الهندسية / نجف

Lecture 8

Finite State Machines (FSM)

Objectives of this Lecture

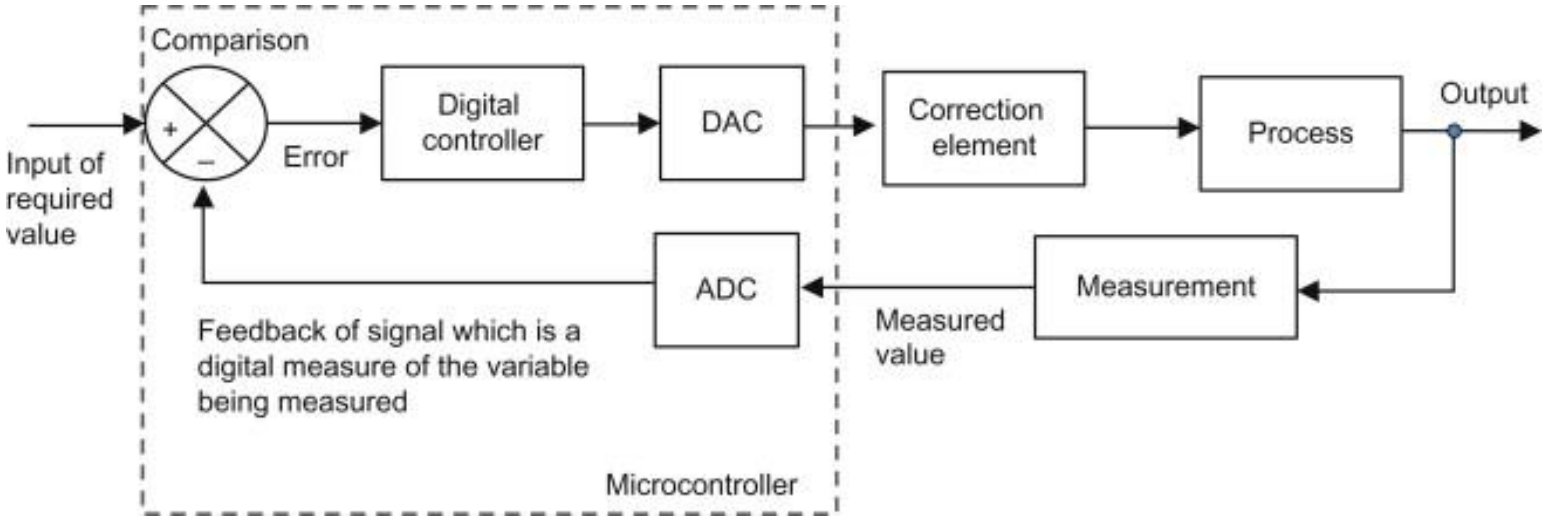
- To define the term **Finite State Machines** (**FSM**). And to recognize its principles.
- To implement digital systems using **FSM** method.
- To indicate the difference between **FSM style #1** and **FSM style #2**

Contents of this Lecture

- **FSM** introduction.
- **FSM** style #1
- **FSM** style #2

FSM introduction

- **FSM** can be very helpful in the design of certain types of systems, particularly those whose tasks form a well-defined sequence (digital controllers, for example).



FSM introduction

- Figure 8.1 shows the block diagram of a single-phase state machine. As indicated in the figure, the lower section contains the sequential logic (flip-flops), while the upper section contains the combinational logic (parallel VHDL).

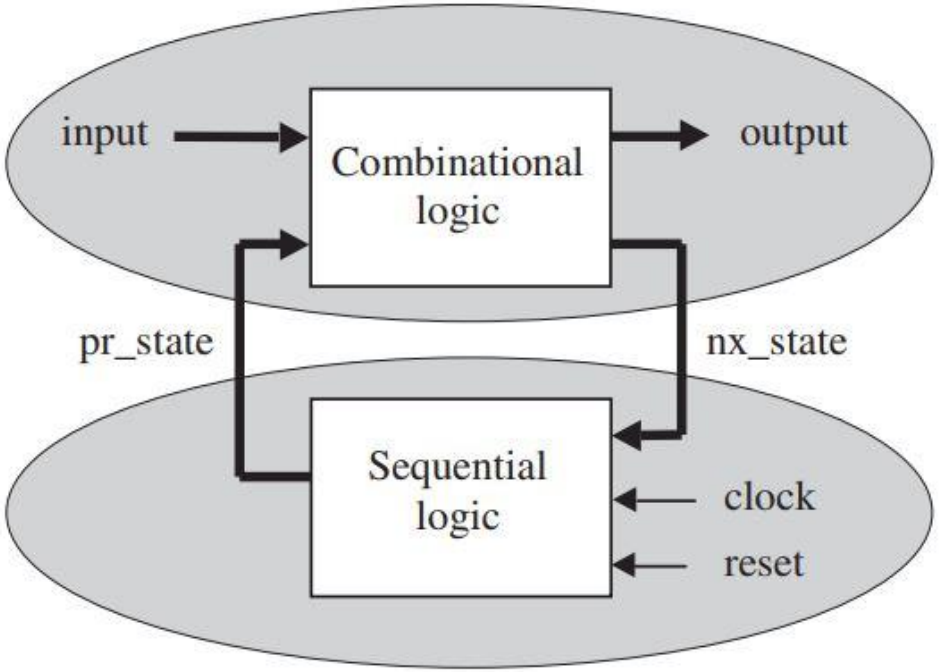


Figure 8.1
Mealy (Moore) state machine diagram.

FSM introduction

- The combinational (upper) section has two inputs, being one **pr_state** (present state) and the other the external input proper. It has also two outputs, **nx_state** (next state) and the external output proper.

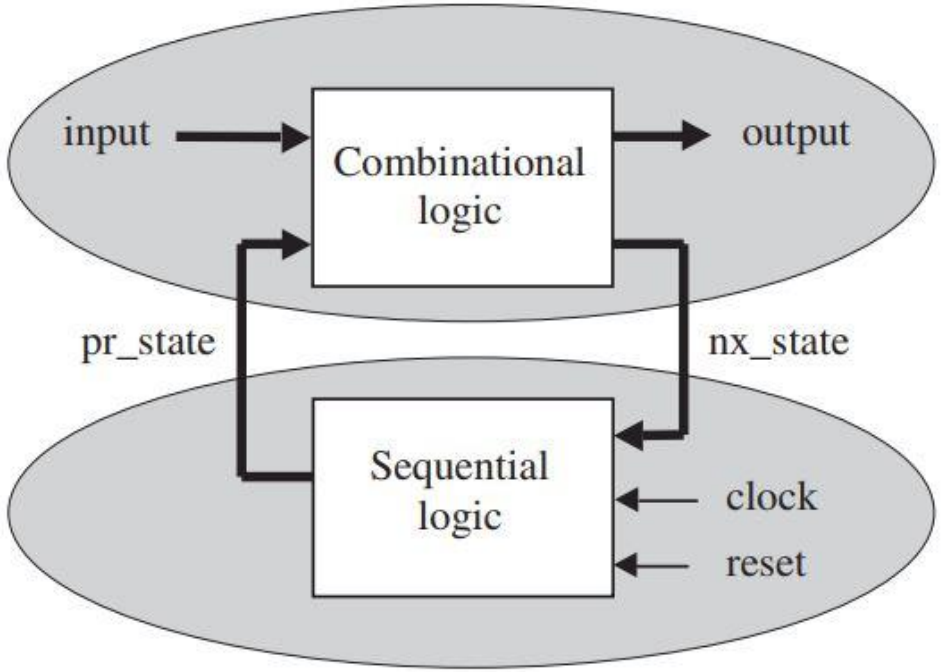


Figure 8.1
Mealy (Moore) state machine diagram.

FSM introduction

- The sequential (lower) section has three inputs (clock, reset, and nx_state), and one output (pr_state). Since all flip-flops are in this part of the system, clock and reset must be connected to it.

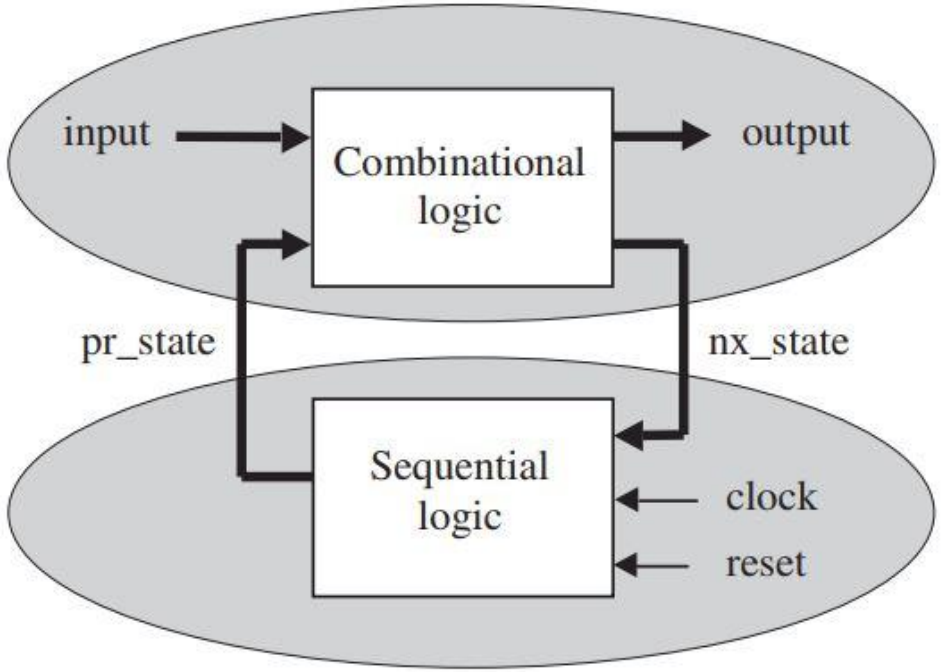


Figure 8.1
Mealy (Moore) state machine diagram.

FSM introduction

- From a VHDL perspective, it is clear that the lower part, being sequential, will require a PROCESS, while the upper part, being combinational, will not.
- However, it is also possible to use PROCESS inside the upper part to implement combinational (parallel) design.

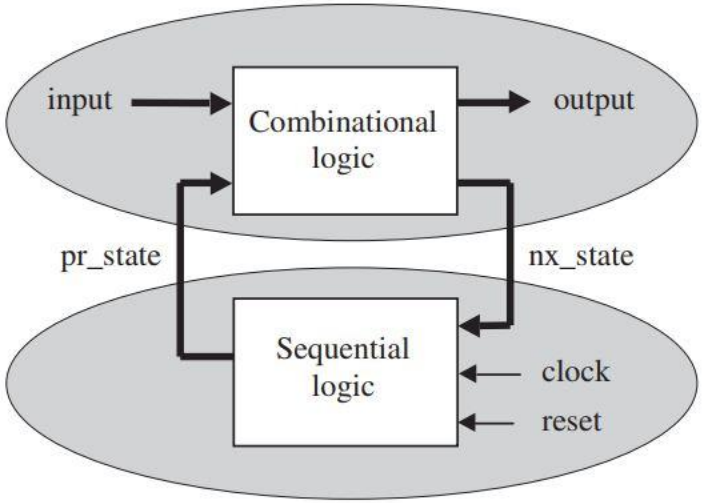


Figure 8.1
Mealy (Moore) state machine diagram.

FSM introduction

- One important aspect related to the **FSM** approach is that, though any sequential circuit can in principle be modeled as a state machine, this is not always advantageous. The reason is that the code might become longer, more complex, and more error prone than in a conventional approach. This is often the case with simple registered circuits, like counters.

- The **FSM** approach is advisable in systems whose tasks constitute a well - structured list so all states can be easily enumerated. That is, in a typical state machine implementation, we will encounter, at the beginning of the ARCHITECTURE, a user-defined enumerated data type, containing a list of all possible system states.

FSM style #1

- Counter Example

- **FSM** method could be used to implement a counter circuit. The problem with the **FSM** is that when the number of states is large it becomes cumbersome to enumerate them all, a problem easily avoided using the LOOP statement or other conventional approaches.

- The state diagram of a 0-to-9 circular counter is shown in figure 8.2. The states were called **zero**, **one**, . . . , **nine**, each name corresponding to the decimal value of the output.

FSM style #1

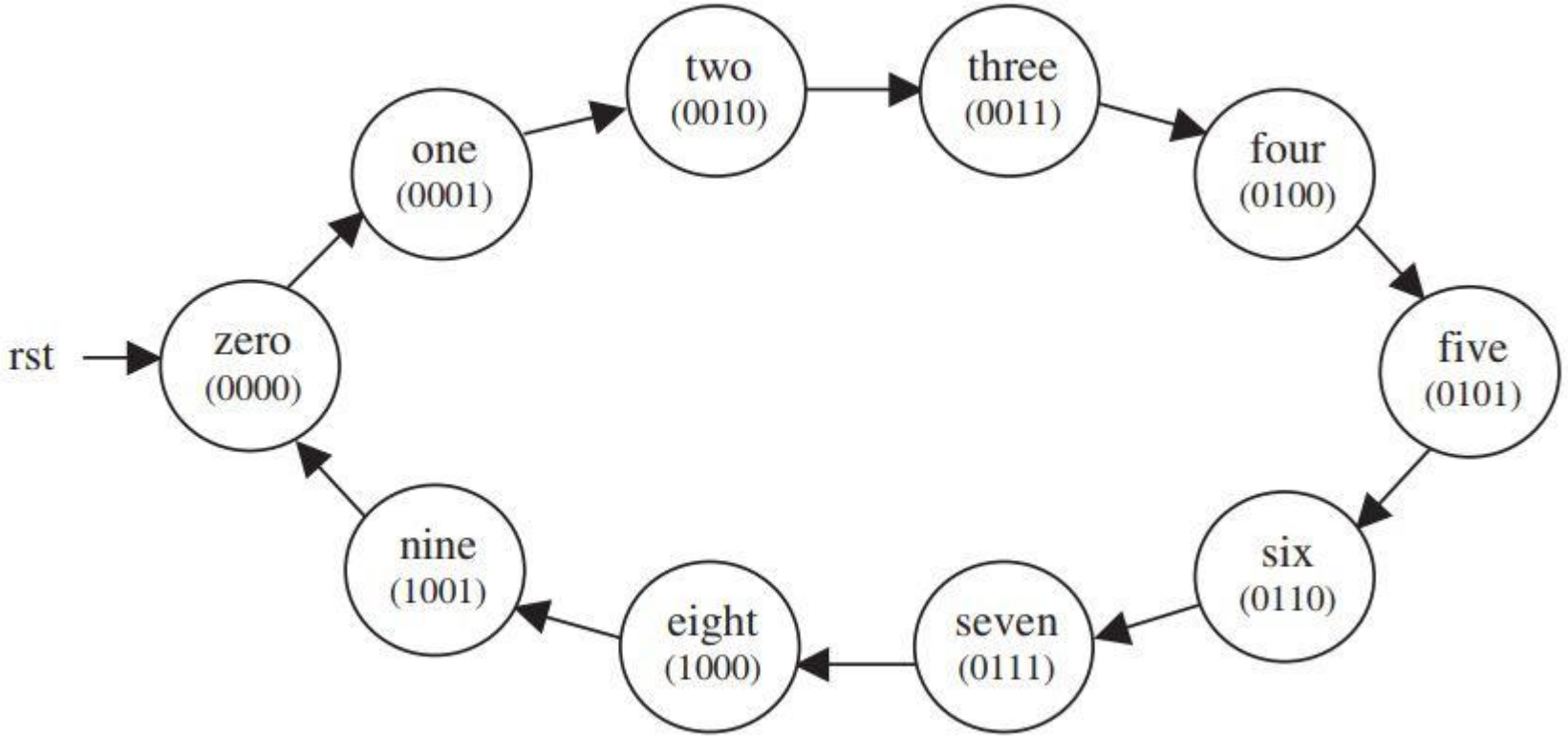


Figure 8.2
States diagram of example 8.1.

FSM style #1

```
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;
```

FSM style #1

```
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 ------------------------------
```

FSM style #1

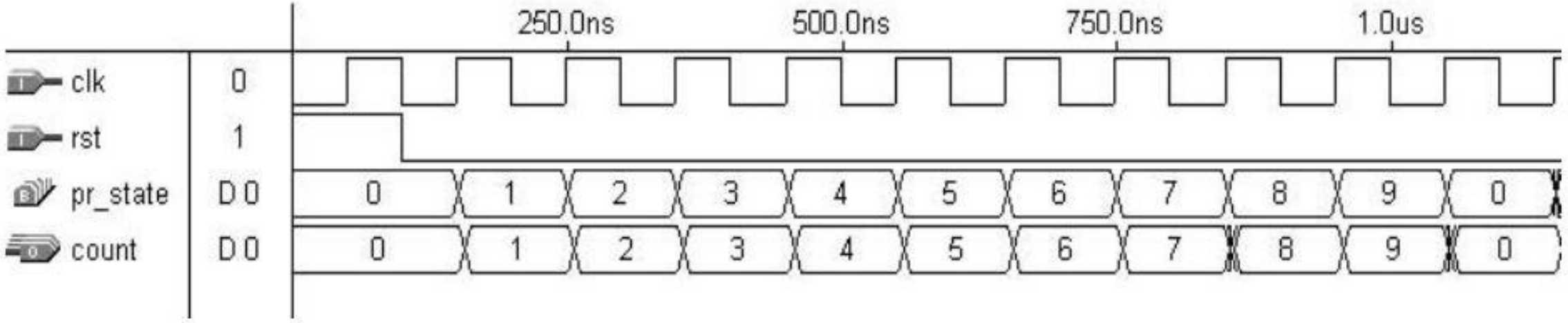


Figure 8.3
Simulation results of example 8.1.

Example 8.1 Counter

FSM style #1

Example 8.2 : Simple FSM #1

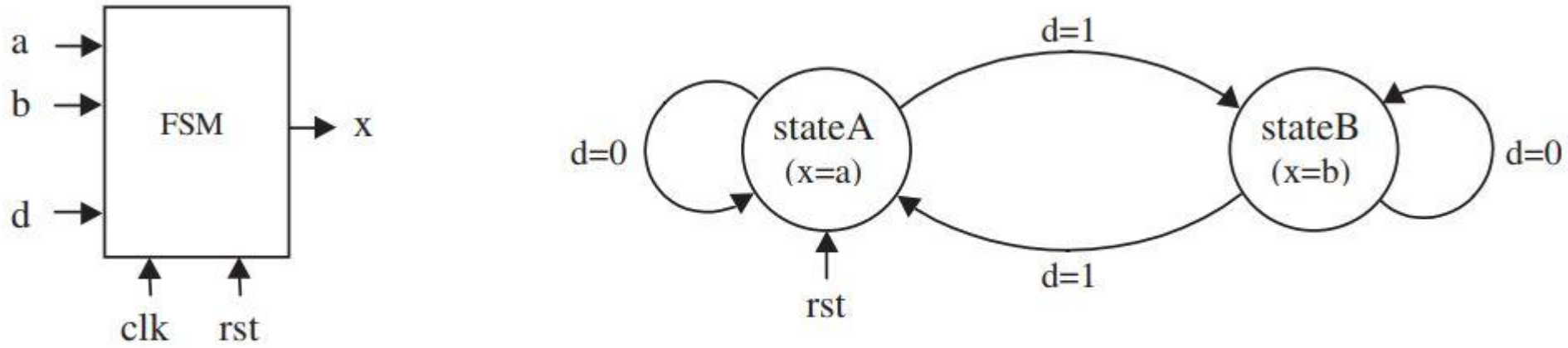


Figure 8.4
State machine of **example 8.2**

FSM style #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  ----------------------------------------
```

FSM style #2

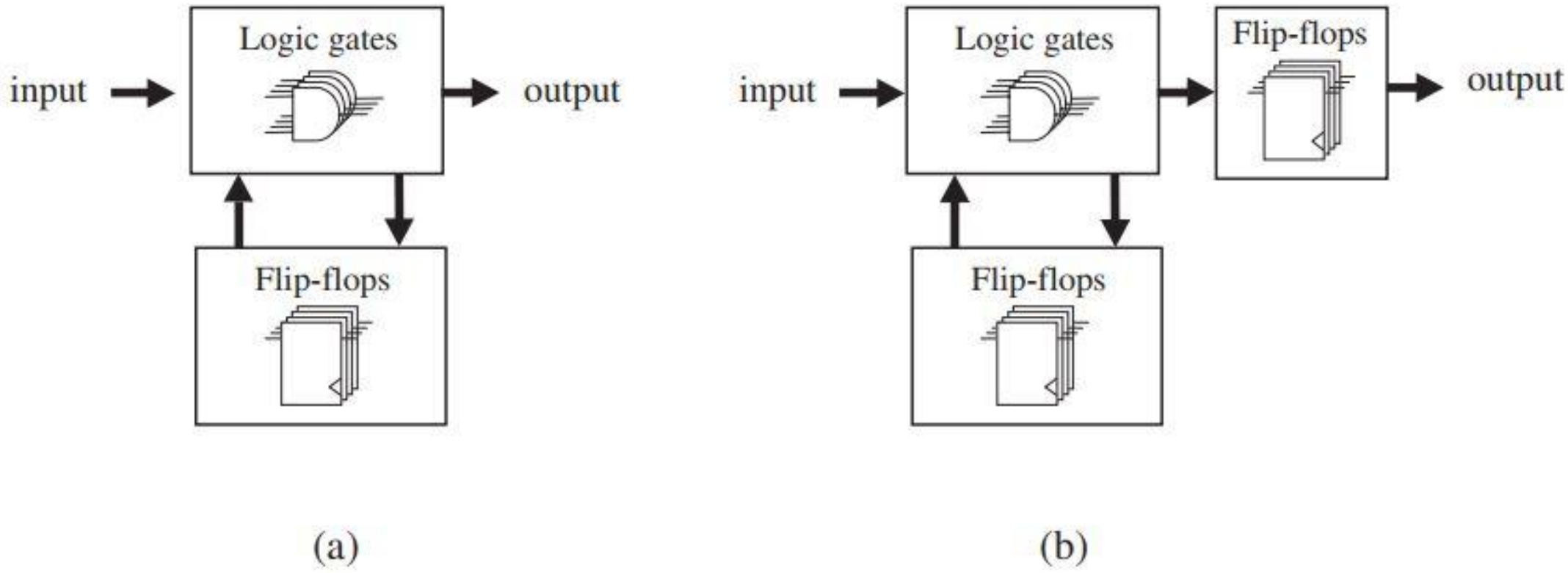
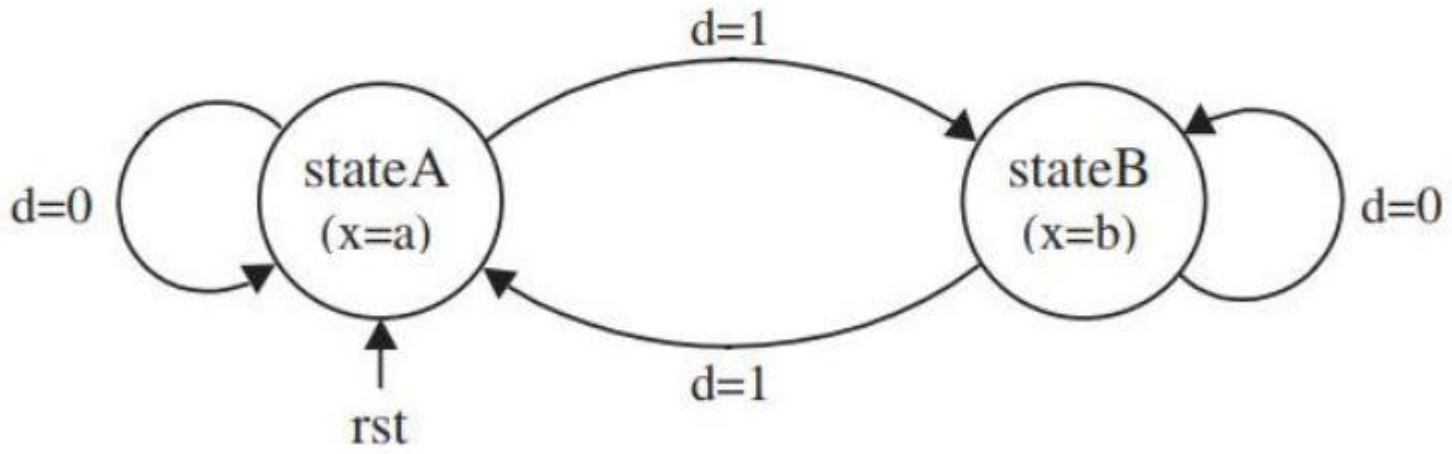


Figure 8.6
Circuit diagrams for (a) Design Style #1 and (b) Design Style #2.

The only difference is ?

FSM style #2

Example 8.3 : Simple FSM #2



FSM style #2

```
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;
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 ----------------------------------------
```

FSM style #1

Example 8.4 : String detector

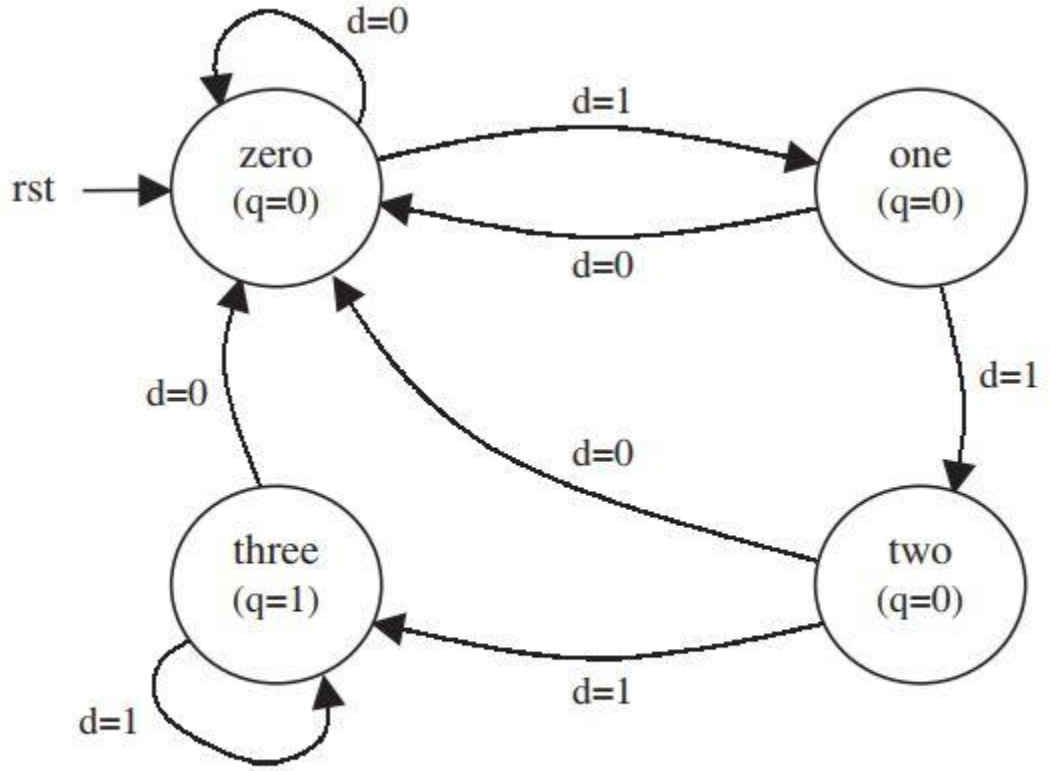


Figure 8.8
States diagram for example 8.4.

FSM style #1

```
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;
```

FSM style #1

```
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  ---------------------------------------
```

FSM style #1

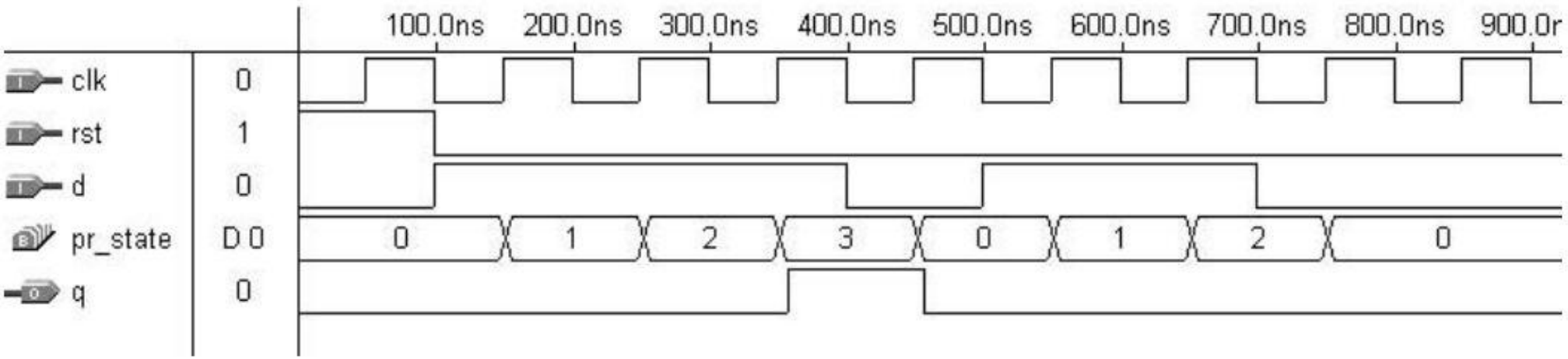


Figure 8.9
Simulation results of example 8.4.

Example 8.4 : String detector

Assignments

- The assignments will be attached to your class room.

End of lecture 8

Any Questions ?