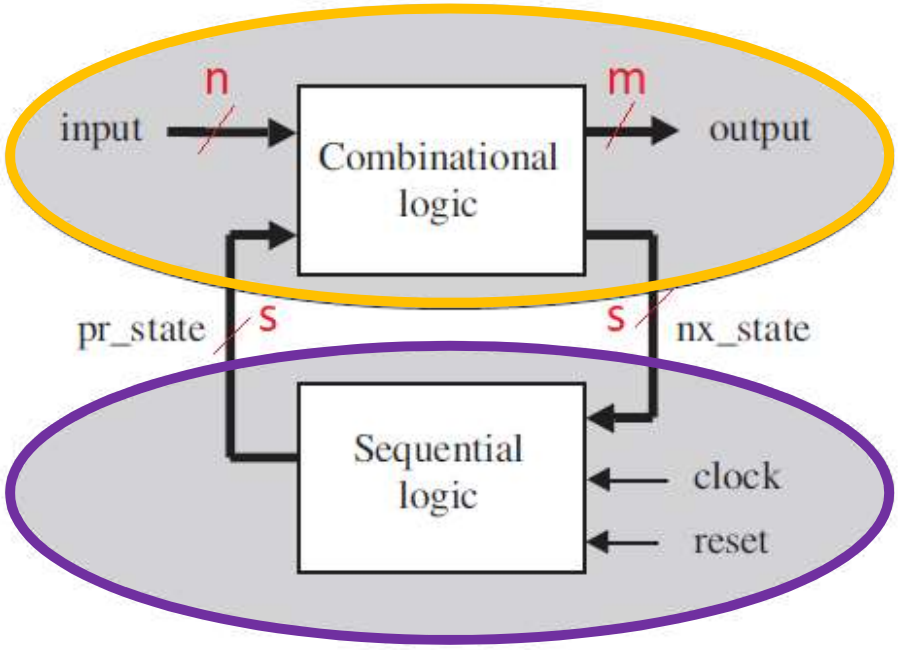
Finite State Machines modeling in VHDL

Introduction

- Finite state machines (FSM) constitute a special modeling technique for sequential logic circuits, as digital controllers and etc.
- Single-phase FSM (Mealy/Moore) diagram:

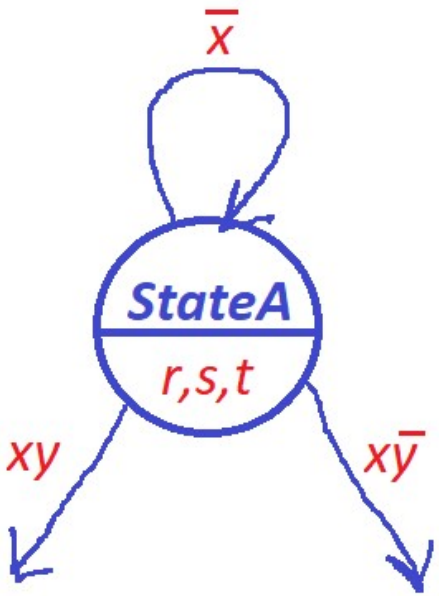


This upper section contains the combinational logic. From a VHDL perspective this part, being combinational (concurrent coding).

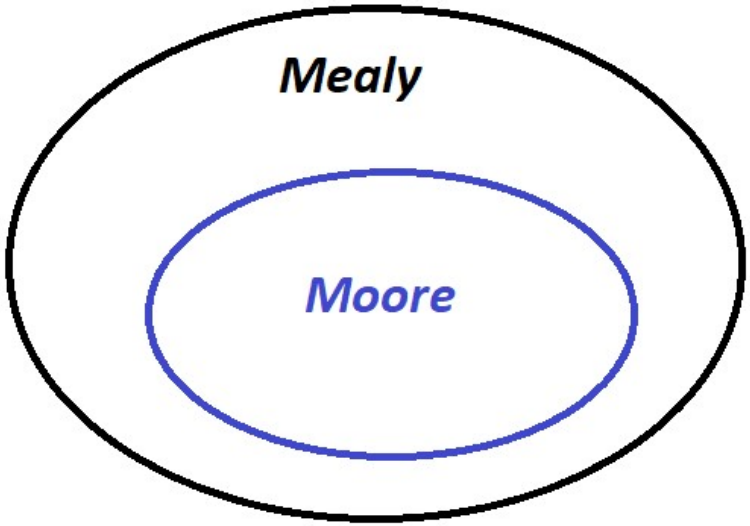
This lower section contains the sequential logic (flip-flops). *Since all flip-flops are in this part of the system,* ***clock*** *and* ***reset*** *must be connected to it. From a VHDL perspective this part, being sequential, will require a PROCESS. When reset is asserted, pr_state will be set to the system's initial state.*

Mealy vs. Moore Machines

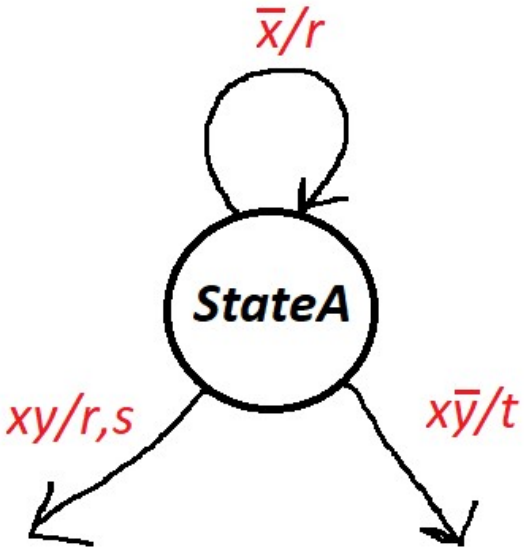
Moore



input=(x,y)
output=(r,s,t)



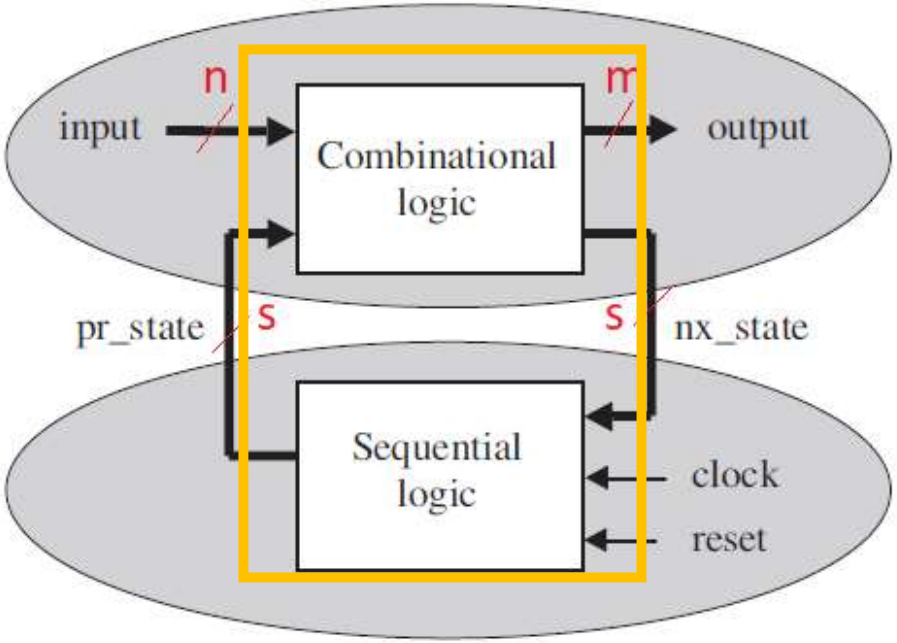
Mealy



input=(x,y)
output=(r,s,t)

Mealy Machines

If the output of the machine depends on the *present state* and the *current input*, then it is called **Mealy machine**.



$$output = f(input\ , PS)$$
$$NS = g(input\ , PS)$$

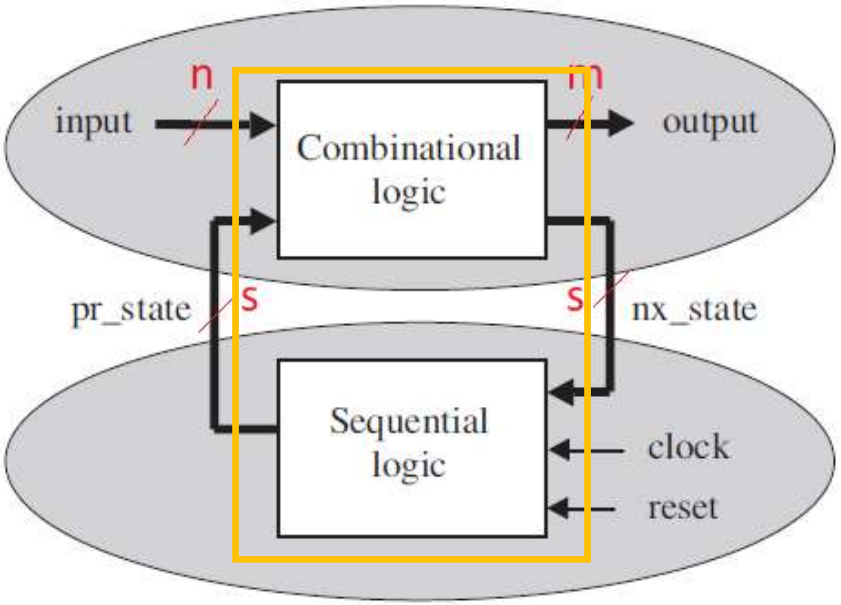
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------------
ENTITY <entity_name> IS
    PORT (  input: IN <data_type>;
            reset, clock: IN STD_LOGIC;
            output: OUT <data_type>);
END <entity_name>;
-------------------------------------------------
ARCHITECTURE <arch_name> OF <entity_name> IS
    TYPE state IS (state0, state1, state2, state3, ...);
    SIGNAL pr_state, nx_state: state;
BEGIN
---------- Lower section: -----------------------

---------- Upper section: -----------------------

END <arch_name>;
```

Moore Machines

If the output of the machine depends only on the *current state*, it is called **Moore machine**.



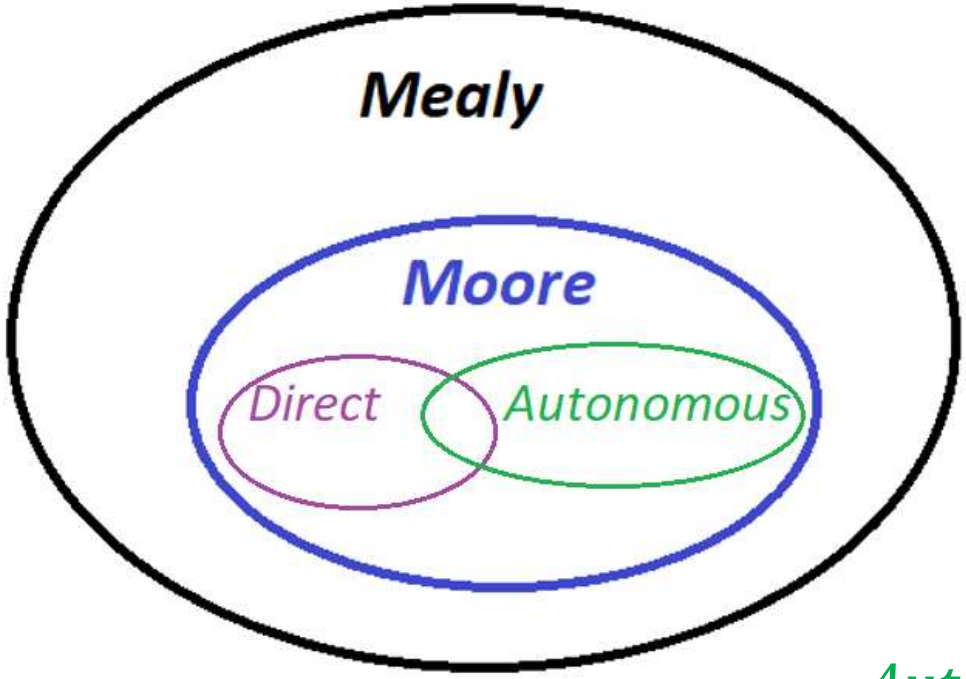
$$output = f(PS)$$
$$NS = g(input\ , PS)$$

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------------
ENTITY <entity_name> IS
    PORT (  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: -----------------------

---------- Upper section: -----------------------

END <arch_name>;
```

Moore Machines



Direct

$output = PS$

$NS = g(input\ , PS)$

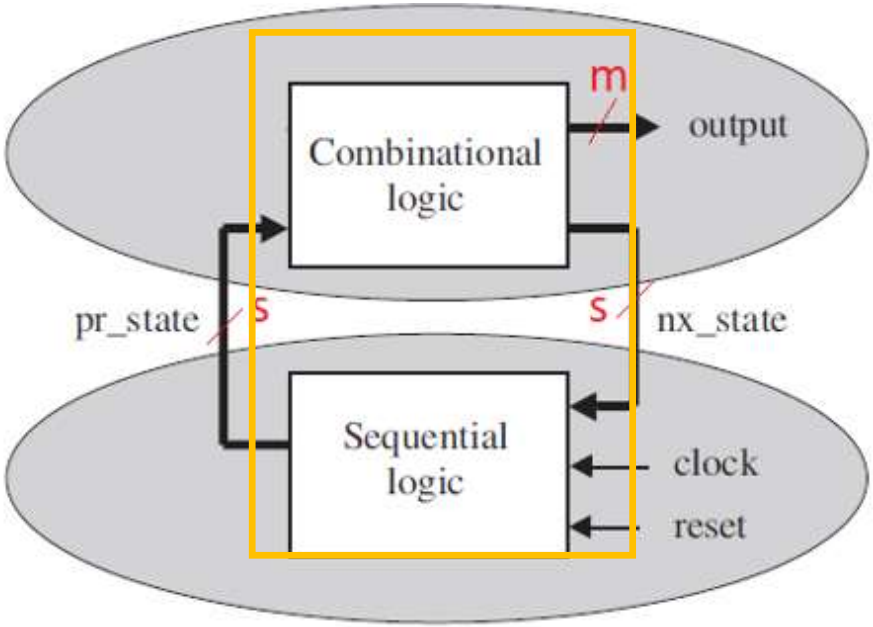
Autonomous

$output = f(PS)$

$NS = g(PS)$

Autonomous Moore Machine

If the output of the machine depends only on the *current state*, it is called a **Moore Autonomous machine**.



$$output = f(PS)$$
$$NS = g(PS)$$

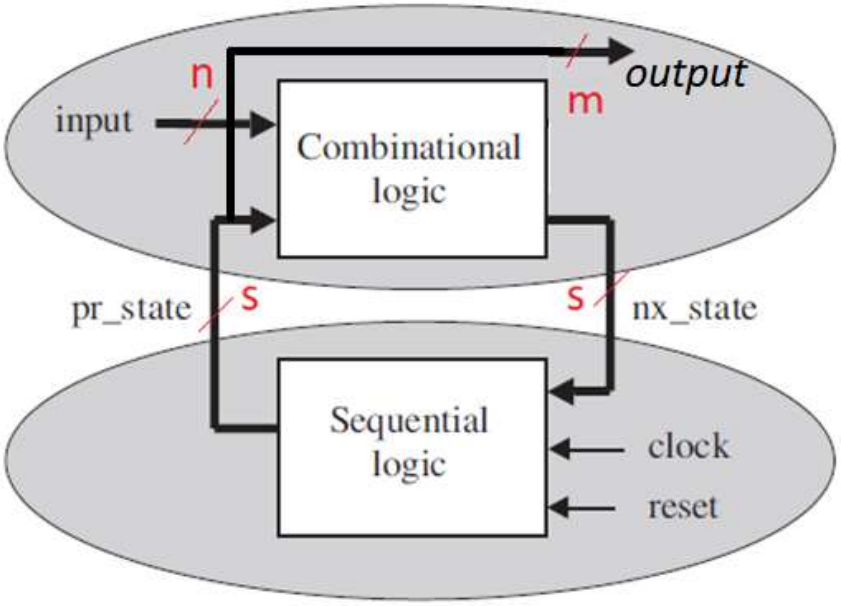
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------------
ENTITY <entity_name> IS
    PORT (  reset, clock: IN STD_LOGIC;
            output: OUT <data_type>);
END <entity_name>;
-------------------------------------------------
ARCHITECTURE <arch_name> OF <entity_name> IS
    TYPE state IS (state0, state1, state2, state3, ...);
    SIGNAL pr_state, nx_state: state;
BEGIN
---------- Lower section: -----------------------

---------- Upper section: -----------------------

END <arch_name>;
```

Direct Moore Machine

If the output is the *current state*, it is called a **Moore direct machine**.



$$output = PS$$
$$NS = g(input\ , PS)$$

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------------
ENTITY <entity_name> IS
    PORT (  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: -----------------------

---------- Upper section: -----------------------

END <arch_name>;
```

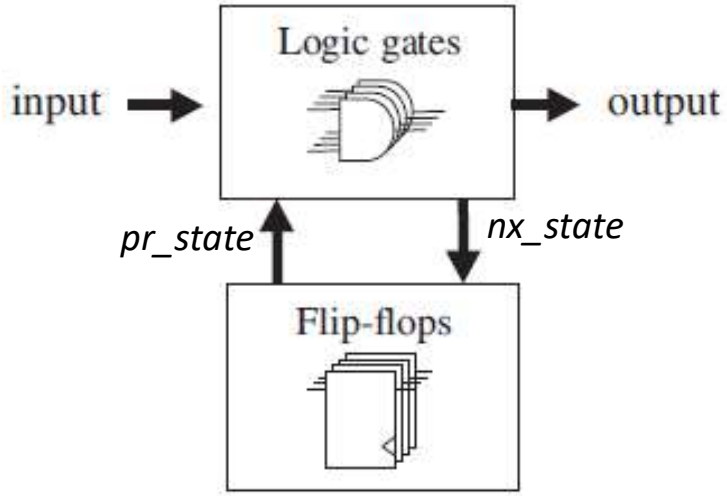
FSM approach - rule of thumb

- 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.
- As a simple rule of thumb, the FSM approach is advisable in systems whose *tasks constitute a well-structured list so all states can be easily enumerated.*
- A typical state machine implementation is used with a user-defined enumerated data type.

```
TYPE state IS (state0, state1, state2, state3, ...);
SIGNAL pr_state, nx_state: state;
```

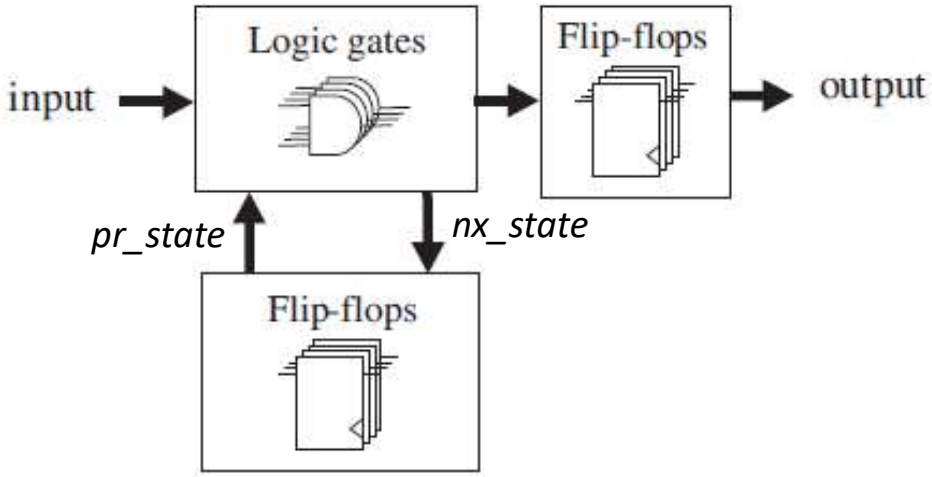
FSM Design Styles

Design Style No.1



In case of a Mealy Machine only pr_state is stored, the output might change when the input changes depending on which state the machine is in regardless of clk (asynchronous output).

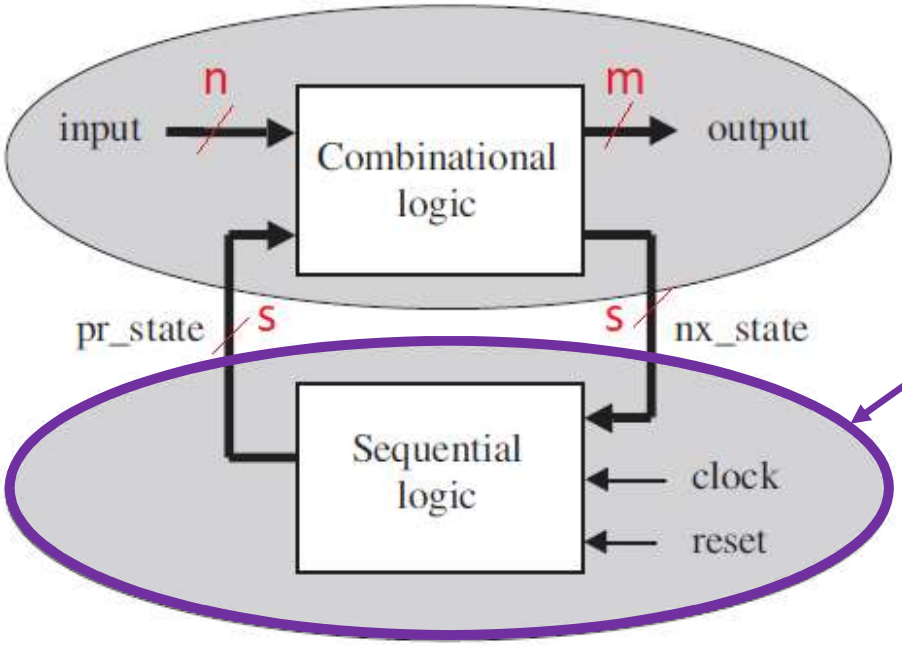
Design Style No.2



In many applications, the signals are required to be synchronous, so the output should be updated only when the proper clock edge occurs (the output must be stored as well).

FSM Design Style No.1 – Sequential Part

In this design style the design of the lower section is completely separated from that of the upper section. All states of the machine are always explicitly declared using an enumerated data type.



Design of the Lower (Sequential) Section

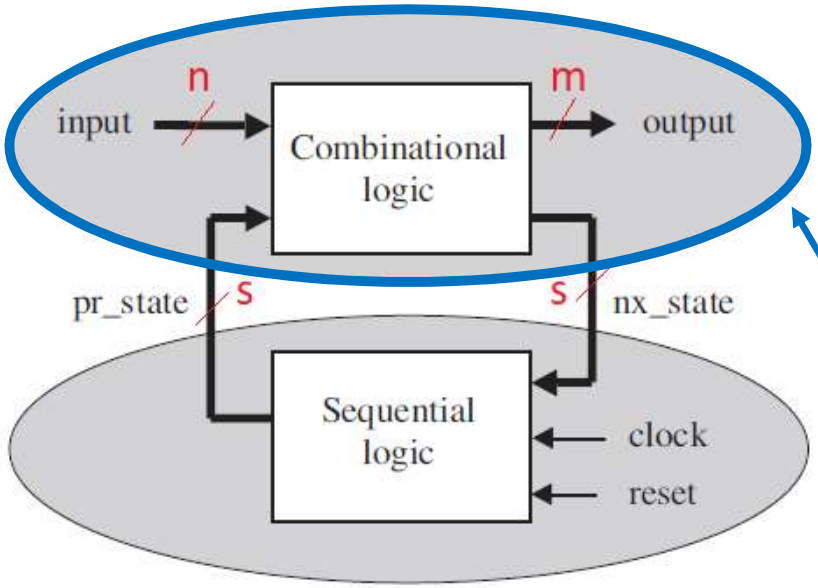
```
PROCESS (reset, clock)
BEGIN
    IF (reset='1') THEN
        pr_state <= state0;
    ELSIF (clock'EVENT AND clock='1') THEN
        pr_state <= nx_state;
    END IF;
END PROCESS;
```

In this approach the lower section is basically standard. The number of flip-flops inferred from this code section is $\lceil \boldsymbol{log_2\, n} \rceil$ *FFs*

FSM Design Style No.1 – Combinational Part

This code does two things: *it assigns the output value* and *establishes the next state.*

The design of the Upper Section is Concurrent.



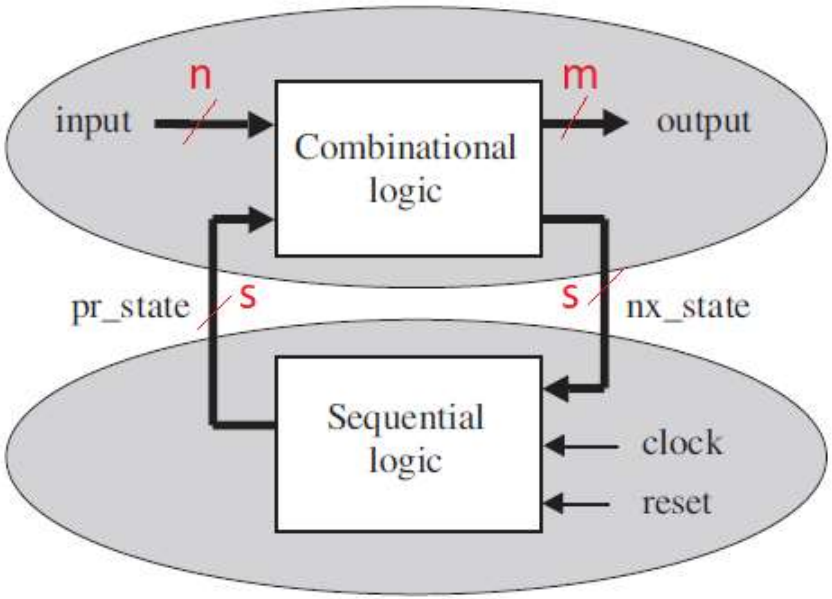

```
PROCESS (input, pr_state)
BEGIN
   CASE pr_state IS
        WHEN state0 =>
             IF (input = ...) THEN
                  output <= <value>;
                  nx_state <= state1;
             ELSE ...
             END IF;
        WHEN state1 =>
             IF (input = ...) THEN
                  output <= <value>;
                  nx_state <= state2;
             ELSE ...
             END IF;
        WHEN state2 =>
             IF (input = ...) THEN
                  output <= <value>;
                  nx_state <= state2;
             ELSE ...
             END IF;
        ...
   END CASE;
END PROCESS;
```

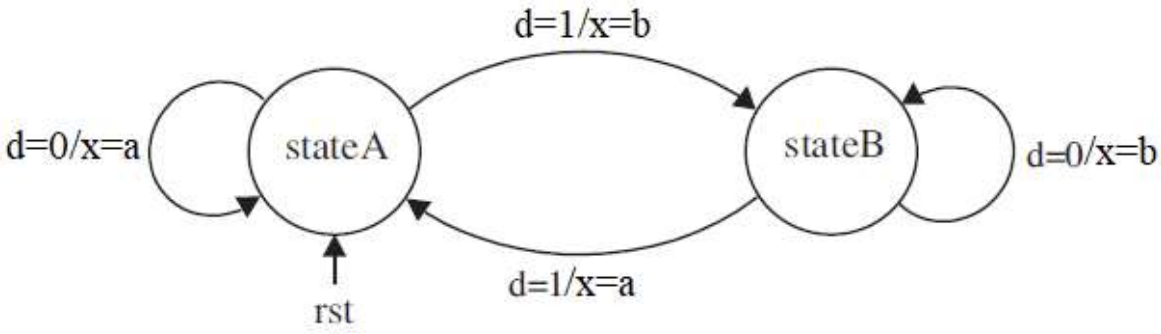
In order to cover all ***pr_state*** signal permutations (Latch infer avoiding) in case of complicated and big FSMs, we can write default assignments to all outputs (before the **CASE** line).

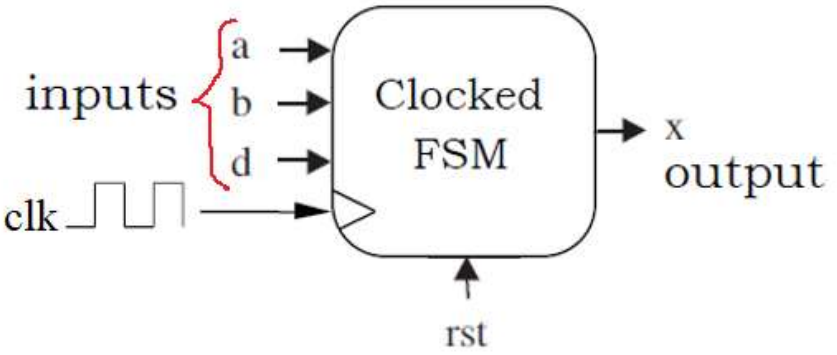
In order to drain unplanned states we can use (before END CASE)

WHEN OTHERS => nx_state <= idle_state ;

Mealy Machine – Design Style No.1 Example





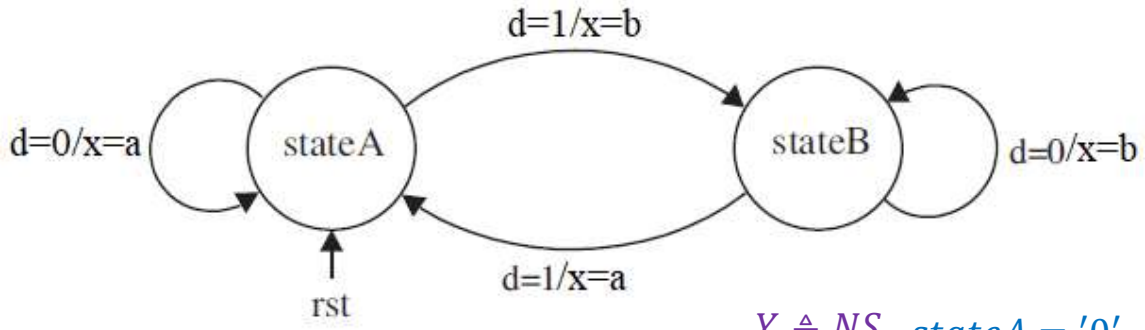


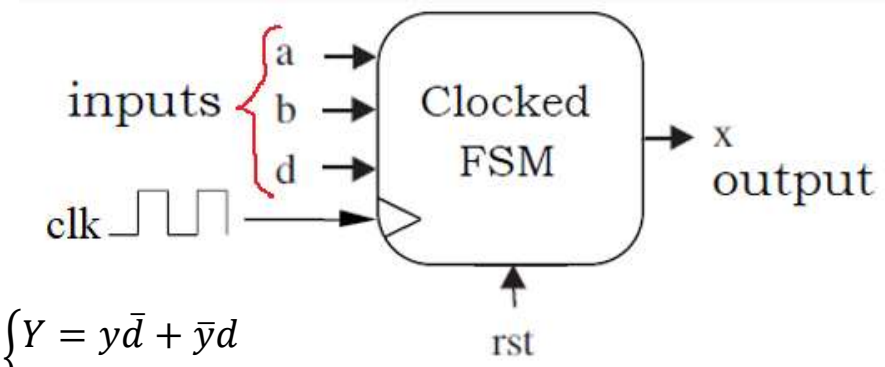
Mealy Machine – simple Example

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------------
ENTITY simpleFSM IS
    PORT (  a, b, d, clk, rst: IN STD_LOGIC;
            x: OUT STD_LOGIC_VECTOR (3 DOWNTO 0));
END simpleFSM;
-------------------------------------------------
ARCHITECTURE state_machine OF simpleFSM IS
    TYPE state IS (stateA, stateB);
    SIGNAL pr_state, nx_state: state;
BEGIN
---------- Lower section: -----------------------
  PROCESS (rst, clk)
  BEGIN
    IF (rst='1') THEN
        pr_state <= stateA;
    ELSIF (clk'EVENT AND clk='1') THEN
        pr_state <= nx_state;
    END IF;
  END PROCESS;
---------- Upper section: -----------------------
```

```
---------- Upper section: -------------
  PROCESS (a, b, d, pr_state)
  BEGIN
    CASE pr_state IS
        WHEN stateA =>
            x <= a;
            IF (d='1') THEN
                nx_state <= stateB;
            ELSE
                nx_state <= stateA;
            END IF;
        WHEN stateB =>
            x <= b;
            IF (d='1') THEN
                nx_state <= stateA;
            ELSE
                nx_state <= stateB;
            END IF;
    END CASE;
  END PROCESS;
END state_machine;
```

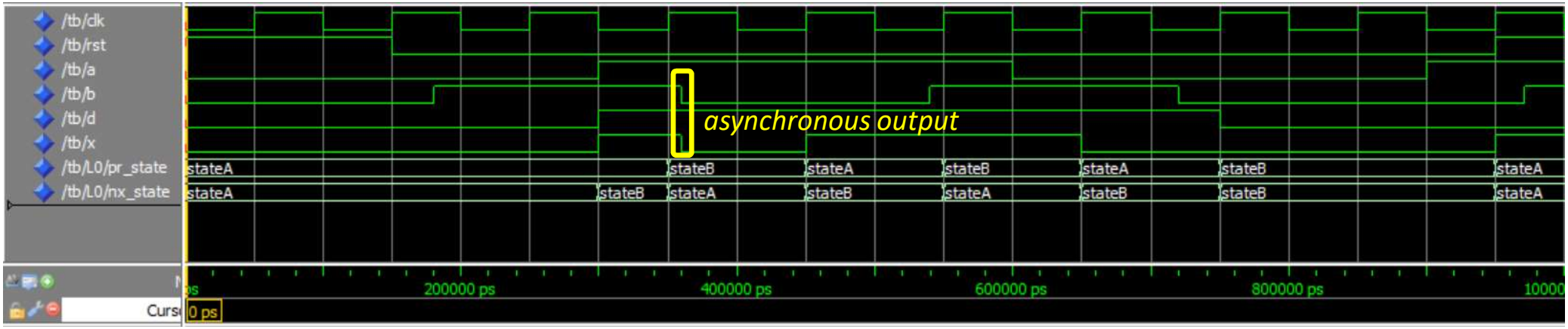
Mealy Machine – simple Example



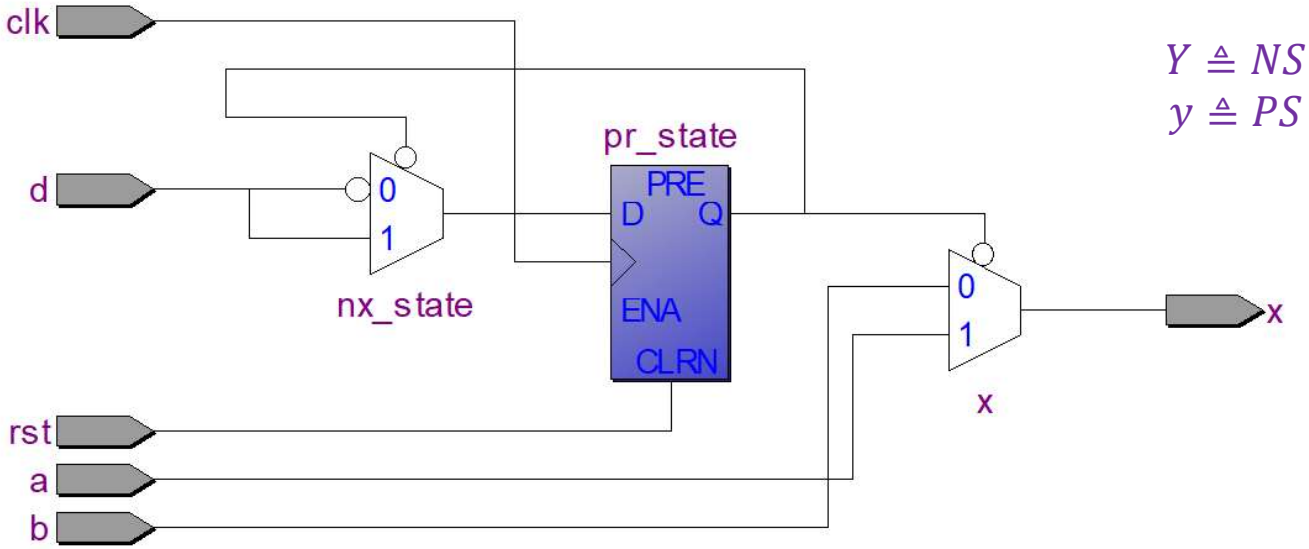


$Y \triangleq NS \quad stateA = '0' \quad \begin{cases} Y = y\bar{d} + \bar{y}d \\ x = yb + \bar{y}a \end{cases}$

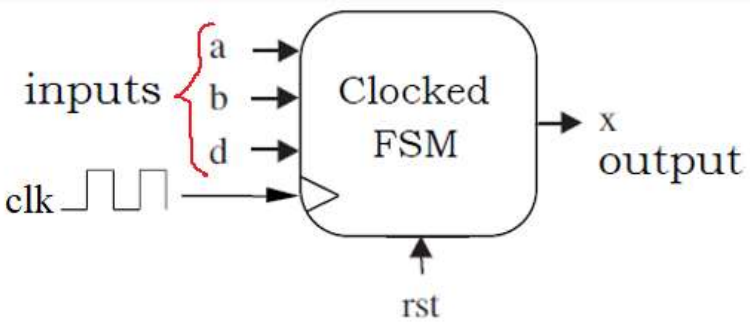
$y \triangleq PS \quad stateB =' 1'$



Mealy Machine – simple Example

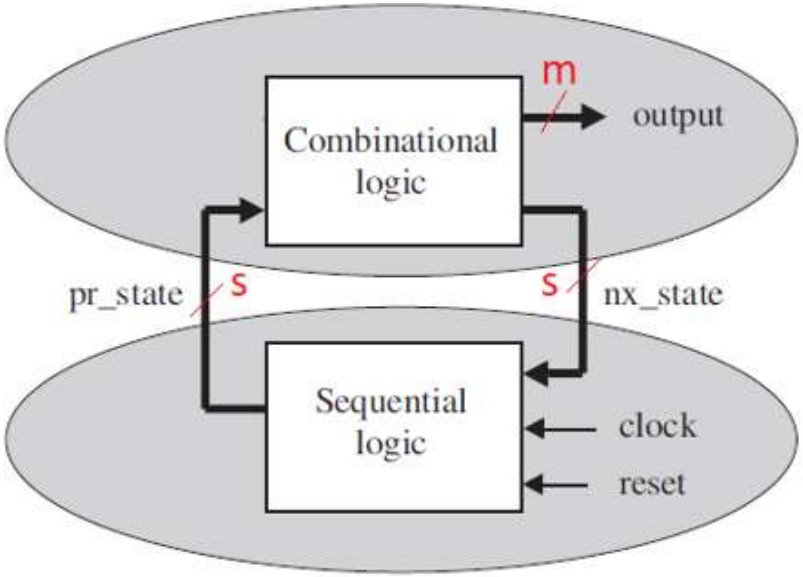


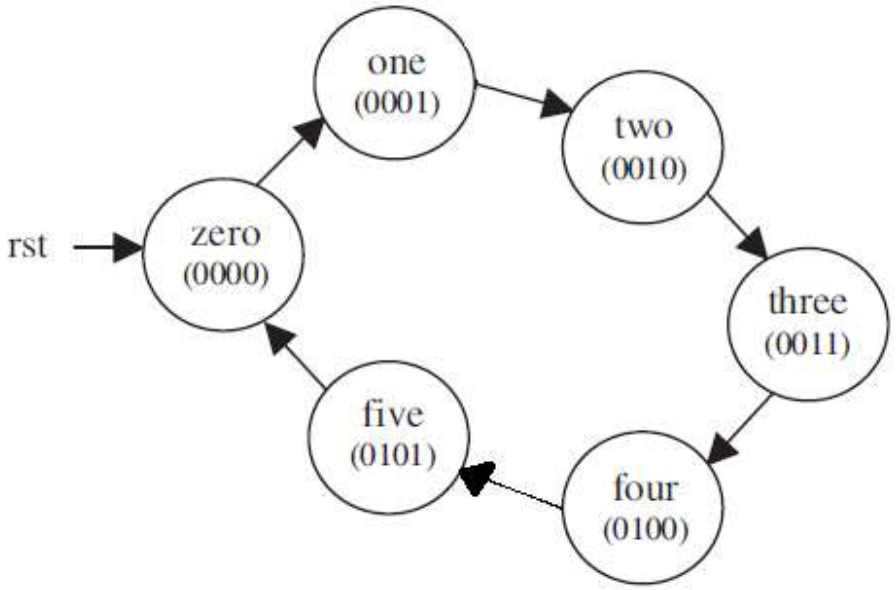
$$Y \triangleq NS \quad stateA = '0' \quad \begin{cases} Y = y\bar{d} + \bar{y}d \\ x = yb + \bar{y}a \end{cases}$$
$$y \triangleq PS \quad stateB =' 1'$$



Moore Machine Example – modulo five

- A counter is an example of Moore machine, for the output depends only on the stored (present) state.
- *As a simple registered circuit and as a sequencer, it can be easily implemented in conventional approach or using FSM approach* (when the number of states is large it becomes cumbersome to enumerate them all).



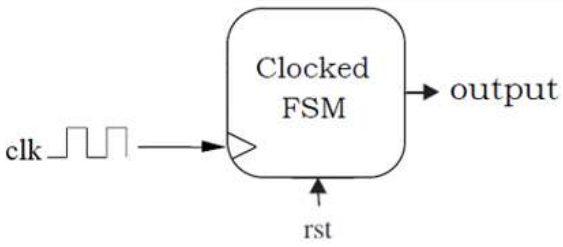


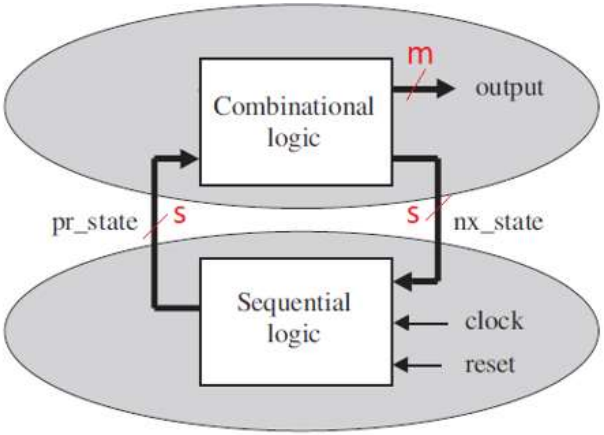
Moore Machine Example – modulo five

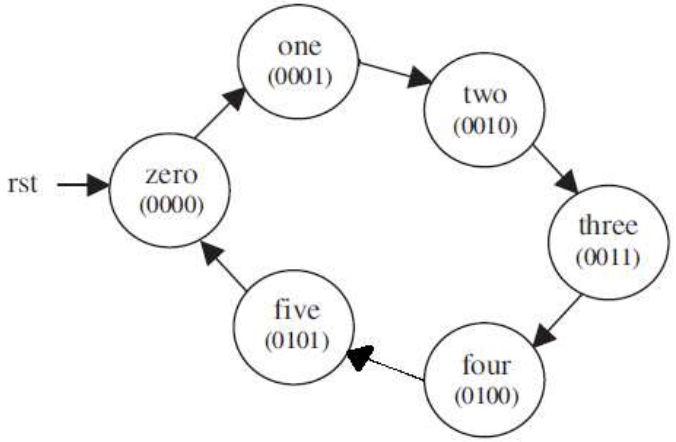
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------------
ENTITY counter IS
    PORT (  clk, rst: IN STD_LOGIC;
            count: OUT STD_LOGIC_VECTOR (3 DOWNTO 0));
END counter;
-------------------------------------------------
ARCHITECTURE state_machine OF counter IS
    TYPE state IS (zero, one, two, three, four,five);
    SIGNAL pr_state, nx_state: state;
BEGIN
---------- Lower section: -----------------------
  PROCESS (rst, clk)
  BEGIN
    IF (rst='1') THEN
        pr_state <= zero;
    ELSIF (clk'EVENT AND clk='1') THEN
        pr_state <= nx_state;
    END IF;
  END PROCESS;
---------- Upper section: -----------------------
```

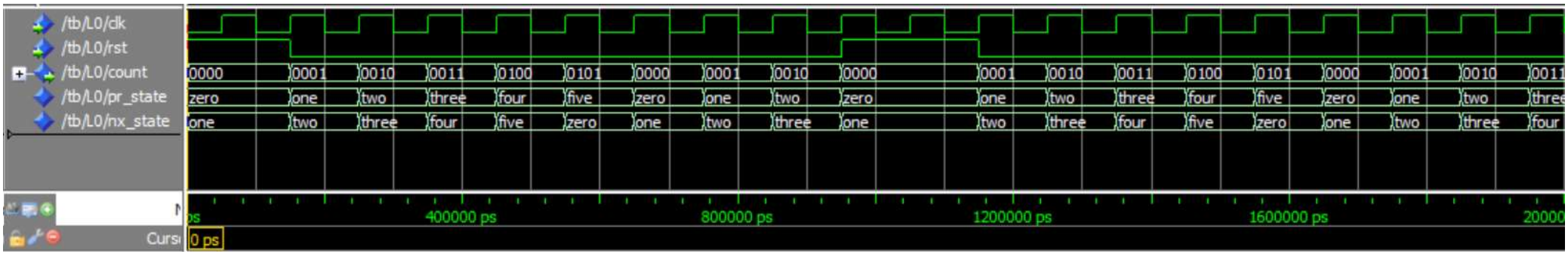
```
---------- Upper section: -----
  PROCESS (pr_state)
  BEGIN
    CASE pr_state IS
        WHEN zero =>
            count <= "0000";
            nx_state <= one;
        WHEN one =>
            count <= "0001";
            nx_state <= two;
        WHEN two =>
            count <= "0010";
            nx_state <= three;
        WHEN three =>
            count <= "0011";
            nx_state <= four;
        WHEN four =>
            count <= "0100";
            nx_state <= five;
        WHEN five =>
            count <= "0101";
            nx_state <= zero;
    END CASE;
  END PROCESS;
END state_machine;
```

Moore Machine Example – modulo five



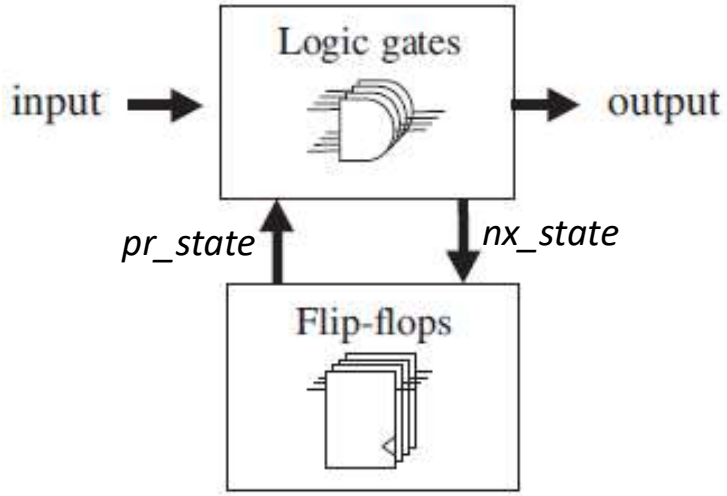






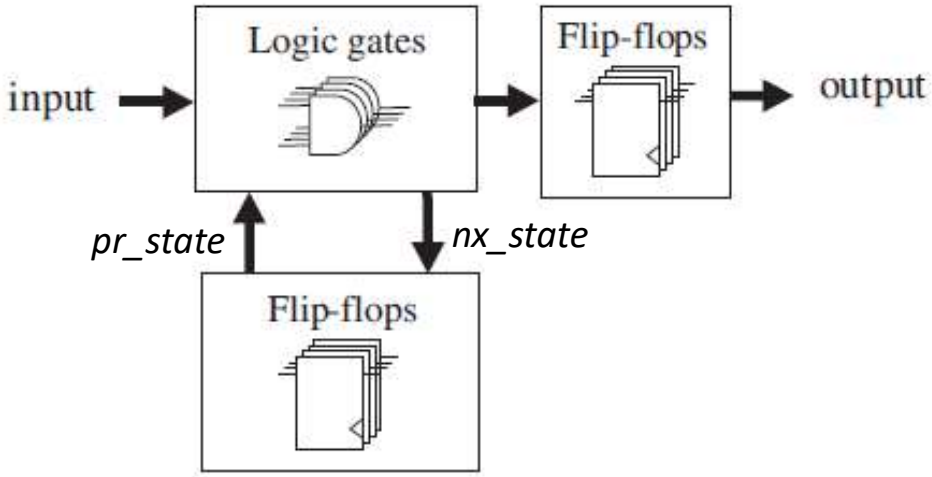
Reminder - FSM Design Styles

Design Style No.1



In case of a Mealy Machine only pr_state is stored, the output might change when the input changes depending on which state the machine is in regardless of clk (asynchronous output).

Design Style No.2



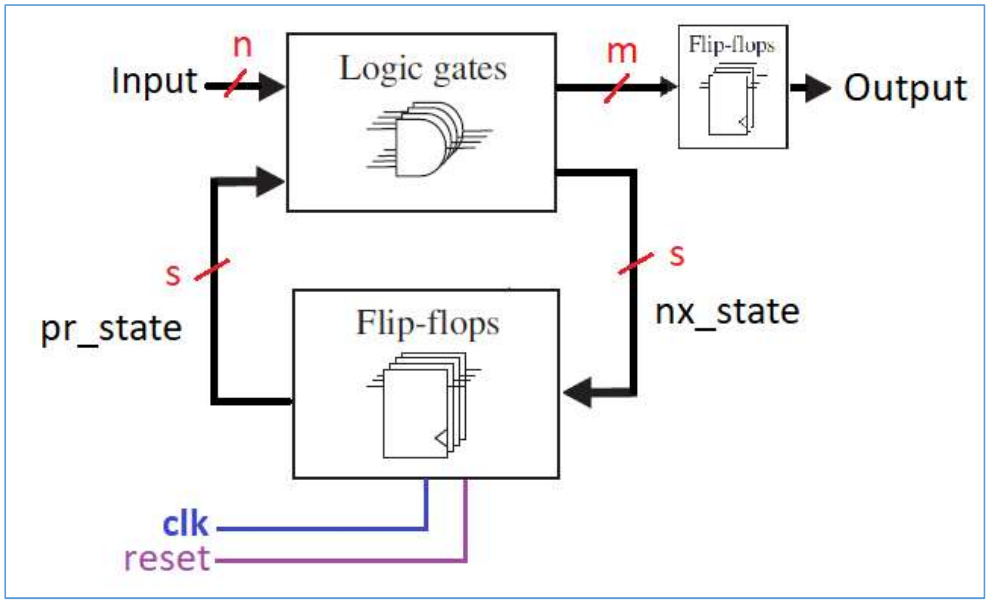
In many applications, the signals are required to be synchronous, so the output should be updated only when the proper clock edge occurs (the output must be stored as well).

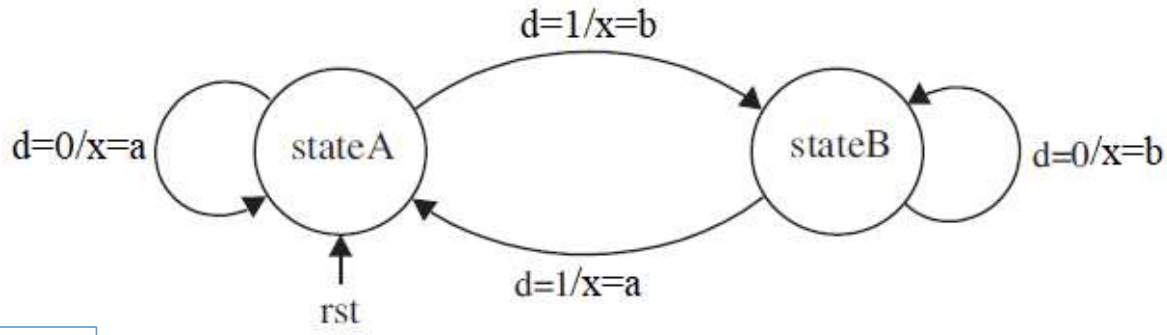
FSM Design Style No.2 (Stored Output)

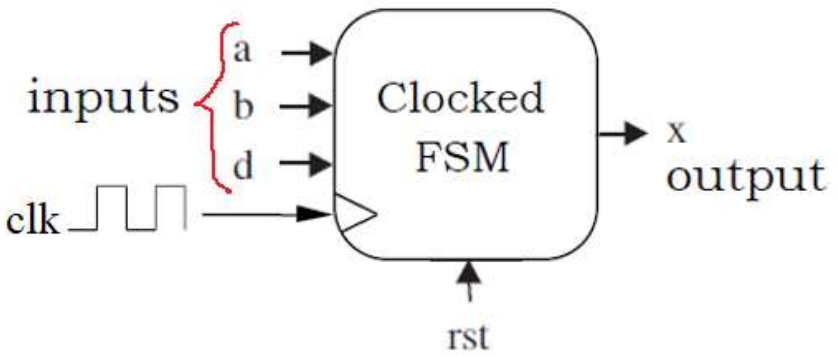
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------------
ENTITY <entity_name> IS
    PORT (  input: IN <data_type>;
            reset, clock: IN STD_LOGIC;
            output: OUT <data_type>);
END <entity_name>;
-------------------------------------------------
ARCHITECTURE <arch_name> OF <entity_name> IS
    TYPE state IS (state0, state1, state2, state3, ...);
    SIGNAL pr_state, nx_state: state;
    SIGNAL temp: <data_type>;    -- first addition
BEGIN
---------- Lower section: -----------------------
  PROCESS (reset, clock)
  BEGIN
    IF (reset='1') THEN
        pr_state <= state0;
    ELSIF (clock'EVENT AND clock='1') THEN
        output <= temp;          -- second addition
        pr_state <= nx_state;
    END IF;
  END PROCESS;
---------- Upper section: -----------------------
```

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

Mealy Machine – Design Style No.2 Example





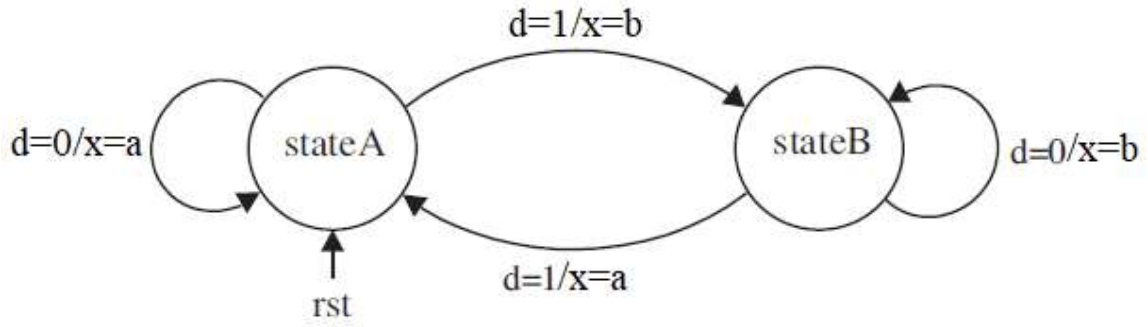


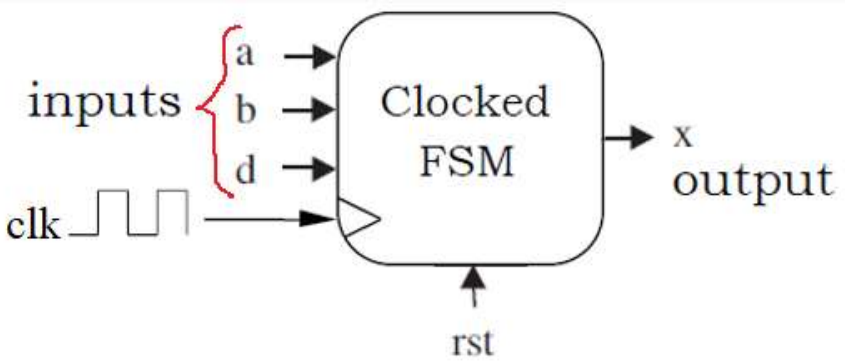
Mealy Machine – Design Style No.2 Example

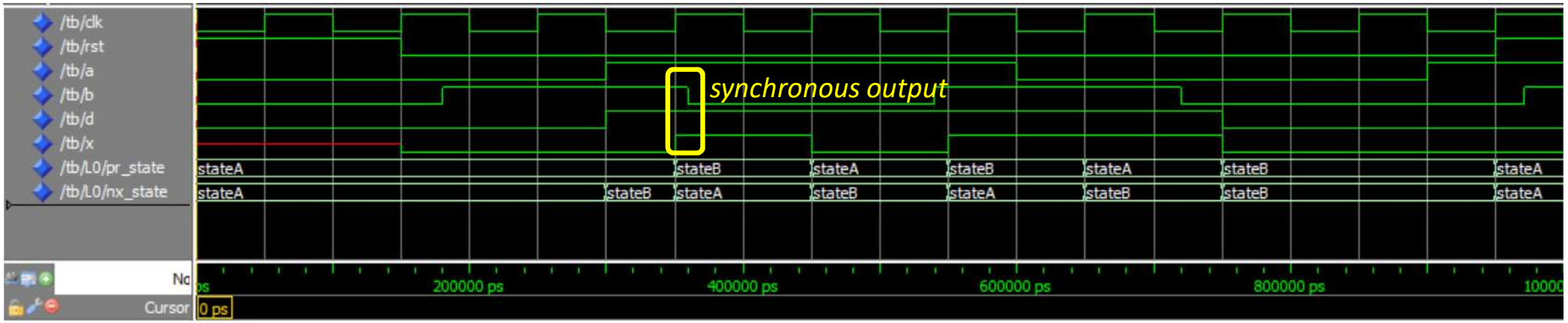
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------------------
ENTITY simpleFSM IS
    PORT (  a, b, d, clk, rst: IN STD_LOGIC;
            x: OUT STD_LOGIC);
END simpleFSM;
-----------------------------------------
ARCHITECTURE state_machine OF simpleFSM IS
    TYPE state IS (stateA, stateB);
    SIGNAL pr_state, nx_state: state;
    SIGNAL temp: STD_LOGIC;
BEGIN
---------- Lower section: ----------------
  PROCESS (rst, clk)
  BEGIN
    IF (rst='1') THEN
        pr_state <= stateA;
    ELSIF (clk'EVENT AND clk='1') THEN
        x <= temp;
        pr_state <= nx_state;
    END IF;
  END PROCESS;
---------- Upper section: ----------------
```

```
---------- Upper section: ----------
  PROCESS (a, b, d, pr_state)
  BEGIN
    CASE pr_state IS
        WHEN stateA =>
            temp <= a;
            IF (d='1') THEN
                nx_state <= stateB;
            ELSE
                nx_state <= stateA;
            END IF;
        WHEN stateB =>
            temp <= b;
            IF (d='1') THEN
                nx_state <= stateA;
            ELSE
                nx_state <= stateB;
            END IF;
    END CASE;
  END PROCESS;
END state_machine;
```

Stored Output Mealy Machine – simple Example







Stored Output Mealy Machine – simple Example

$Y \triangleq NS \quad stateA = '0' \quad \begin{cases} Y = y\bar{d} + \bar{y}d \\ x = yb + \bar{y}a \end{cases}$

$y \triangleq PS \quad stateB =' 1'$