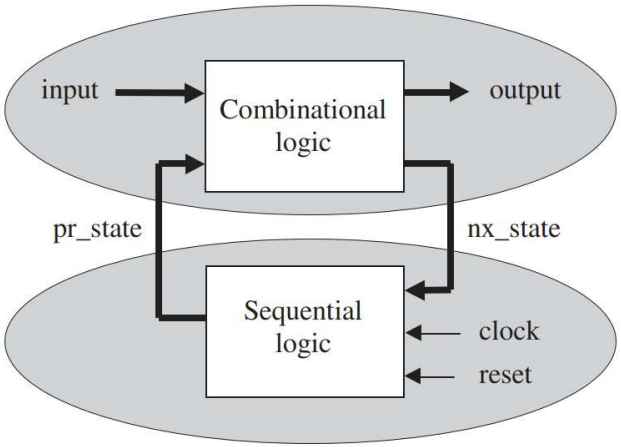
State Machines
Modeling Sequential Logic Circuits

Finite State Machines - FSM

VHDL
VHSIC HARDWARE
DESCRIPTION LANGUAGE

State Machines

Definition Finite State Machines (FSM)



- Figure shows a single-phase State machine.
- Combinational and sequential logic (C.L. and S.L.) are well differentiated.
- C.L. has 2 inputs and 2 output.
- S.L. has 3 inputs and one output.
- S.L. contains the flip-flops and is designed by PROCESS.
- C.L can be designed with or without PROCESS. (designer definition)
- **Clock** and **Reset** are usually in sensitivity list of PROCESS. So, they are in S.L.

- If output of FSM depends on current state and its input -> **Mealy** machine.
- If output of FSM depends only on current state -> **Moore** machine.

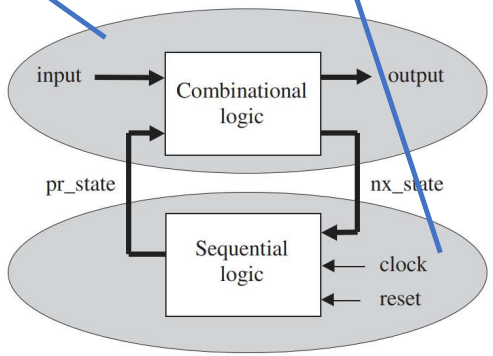
- Although any circuit can be design as FSM, we only use this approach for **systems whose task are clearly a list of different states.** E.g., traffic light.

State Machines

Designing FSM – Template Style 1 (no stored output)

```
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;
```

```
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;
```



S.L Part:

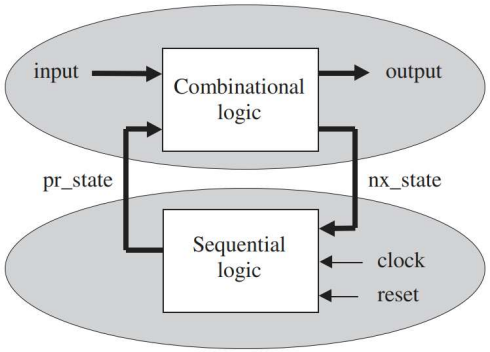
- Is made based on PROCESS
- Reset is asynchronous and set “state 0” to present state (pr_state).
- Next State (nx_state) is assigned synchronously.
- Number of flip-flops (n) is equal to the number of bits (n) used to encode all states (S). 2^n=S; **n=log2(S).**

C.L Part:

- Sequential code is selected using CASE statement.
- The code assign output value
- It stablishes next state.
- Rules of design of C.L. circuits are fulfilled:
  - Rule1: All input/output combinations are specified.
  - Rule2: Sequential statements are present for all signals of sensitivity list.

State Machines

Designing FSM – Template Style 1 (no stored ou



This is how the templete for a full code of FSM looks:

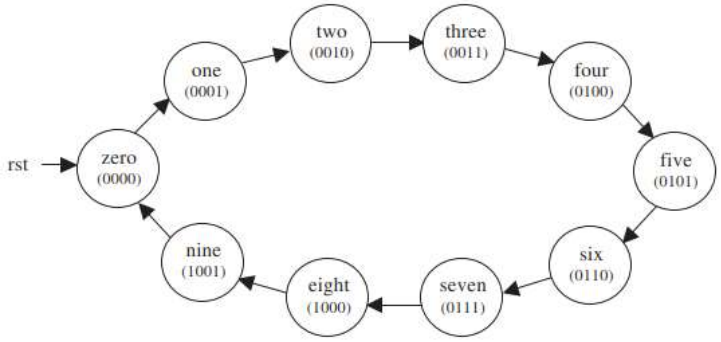
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------------------
ENTITY <entity_name> IS
   PORT ( input: IN <data_type>;
          reset, clock: IN STD_LOGIC;
          output: OUT <data_type>);
END <entity_name>;
-------------------------------------------------------
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>;
```

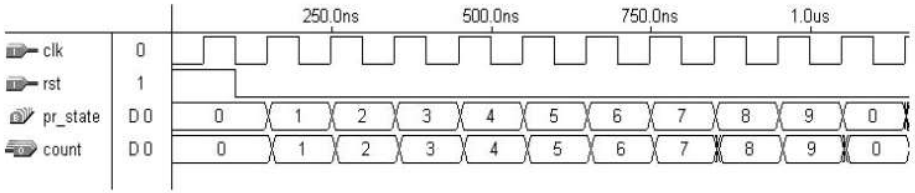
State Machines

Example Moore Machine

A counter is a circuit which next state only depende on its present state. (moore).

It can be implemented as FSM (not Good idea) or conventionally with arithmetic operations as seen before (better idea).



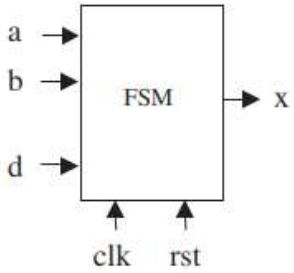


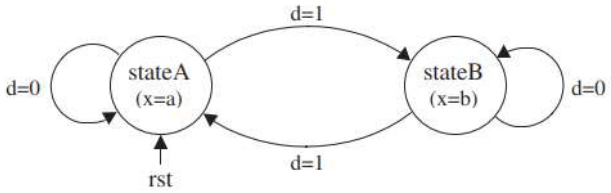

```
-------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------------
ENTITY counter IS

    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, six, seven, eight, nine);
   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: -----------------
   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 <= six;
         WHEN six =>
            count <= "0110";
            nx_state <= seven;
         WHEN seven =>
            count <= "0111";
            nx_state <= eight;
         WHEN eight =>
            count <= "1000";
            nx_state <= nine;
         WHEN nine =>
            count <= "1001";
            nx_state <= zero;
      END CASE;
   END PROCESS;
END state_machine;
```

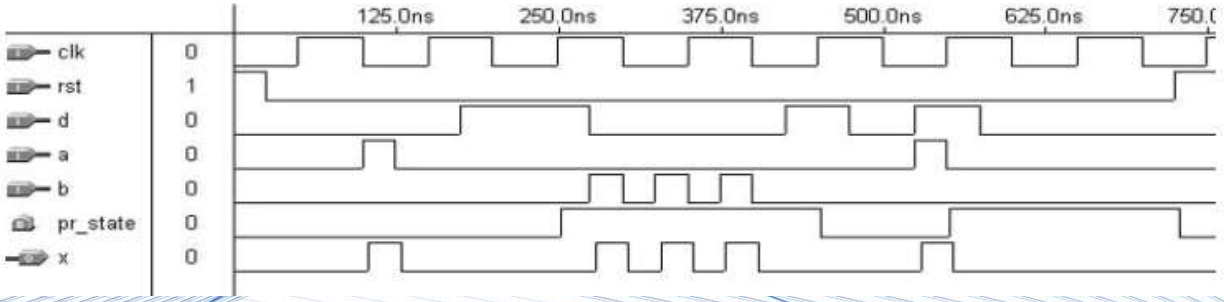
State Machines

Example Mealy Machine

Suppose a circuit that has two states. It changes its state when an input d is equal to 1. Also, has two inputs, a and b, which is reflected to the output according to its state. As showed in this figure.
We can write this FSM using the same structure we have learned.



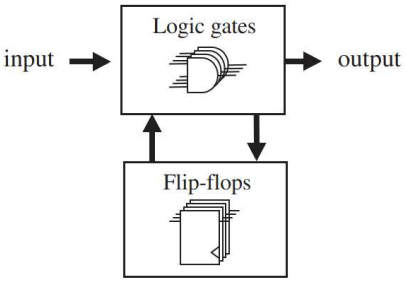





```
-----------------------------------------------
ENTITY simple_fsm IS
   PORT ( a, b, d, clk, rst: IN BIT;
          x: OUT BIT);
END simple_fsm;
-----------------------------------------------
ARCHITECTURE simple_fsm OF simple_fsm IS
   TYPE state IS (stateA, stateB);
   SIGNAL pr_state, nx_state: 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: -----------------
   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 simple_fsm;
-----------------------------------------------
```

State Machines

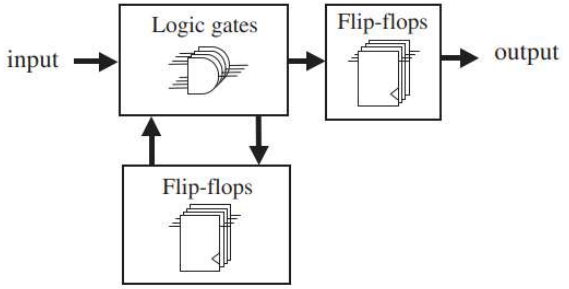
Designing FSM with Synchronous Output



So far, the output of our FSM changes asynchronously. Therefore, flip-flops are only on S.L. circuits.

If we need to **update the output synchronously** with a change in the input (synchronous Mealy machine), **output must be stored** as well.

Now **temp is a signal, and the output is updated synchronously**.



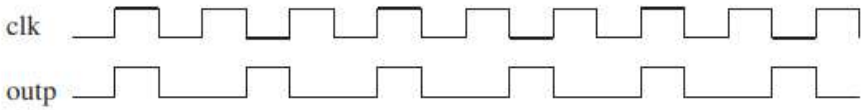

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------------------
ENTITY <ent_name> IS
   PORT (input: IN <data_type>;
         reset, clock: IN STD_LOGIC;
         output: OUT <data_type>);
END <ent_name>;
-------------------------------------------------------
ARCHITECTURE <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>;
```

State Machines

Exercise

Generate a circuit that can output a signal as showed in the figure. You must use rising and falling edges of the clock to implement this clock divider (Hint: you can use two state machines , one for each edge and then AND them). The circuit has only clk as input. (output is obviously synchronous)




```
-----------------------------------------
ENTITY signal_gen IS
   PORT ( clk: IN BIT;
          outp: OUT BIT);
END signal_gen;
-----------------------------------------
ARCHITECTURE fsm OF signal_gen IS
   TYPE state IS (one, two, three);
   SIGNAL pr_state1, nx_state1: state;
   SIGNAL pr_state2, nx_state2: state;
   SIGNAL out1, out2: BIT;
BEGIN

----- Lower section of machine #1: ---
PROCESS(clk)
BEGIN
   IF (clk'EVENT AND clk='1') THEN
      pr_state1 <= nx_state1;
   END IF;
END PROCESS;
----- Lower section of machine #2: ---
PROCESS(clk)
BEGIN
   IF (clk'EVENT AND clk='0') THEN
      pr_state2 <= nx_state2;
   END IF;
END PROCESS;
---- Upper section of machine #1: -----
PROCESS (pr_state1)
BEGIN
   CASE pr_state1 IS
      WHEN one =>
         out1 <= '0';
         nx_state1 <= two;
      WHEN two =>
         out1 <= '1';
         nx_state1 <= three;
      WHEN three =>
         out1 <= '1';
         nx_state1 <= one;
   END CASE;
END PROCESS;
---- Upper section of machine #2: -----
PROCESS (pr_state2)
BEGIN
   CASE pr_state2 IS
      WHEN one =>
         out2 <= '1';
         nx_state2 <= two;
      WHEN two =>
         out2 <= '0';
         nx_state2 <= three;
      WHEN three =>
         out2 <= '1';
         nx_state2 <= one;
   END CASE;
END PROCESS;
outp <= out1 AND out2;
END fsm;
-----------------------------------------
```

Fixed Point Division

Exercise

Generate a circuit can make a generic division of two aleatory numbers a and b (y=a/b).

Remember that division operation "/" represent only a shift operation (equivalent to say that is a division by 2).

See this table to understand the algorithm required for division 11/3=3 con resto 2:

Index (i)	a-related input (a_inp)	Comparison	b-related input (b_inp)	y (quotient)	Operation on 1st column
3	1011	<	0011000	0	none
2	1011	<	0001100	0	none
1	1011	>	0000110	1	a_inp(i)-b_inp(i)
0	0101	>	0000011	1	a_inp(i)-b_inp(i)
	0010 (rem)				

a_inp
11
11
11
11-6=5
5-3=2

b_inp
24
12
6
3

a and **b** are of n+1 bits (e.g 4bits)
Extended **b** is of 2n+1 bits (e.g. 7bits)
b is shifted in **i=n=3** positions.

Y=0011=3

Remainder: 0010=2.

END-State Machines
Modeling Sequential Logic Circuits

Finite State Machines - FSM

VHDL
VHSIC HARDWARE
DESCRIPTION LANGUAGE