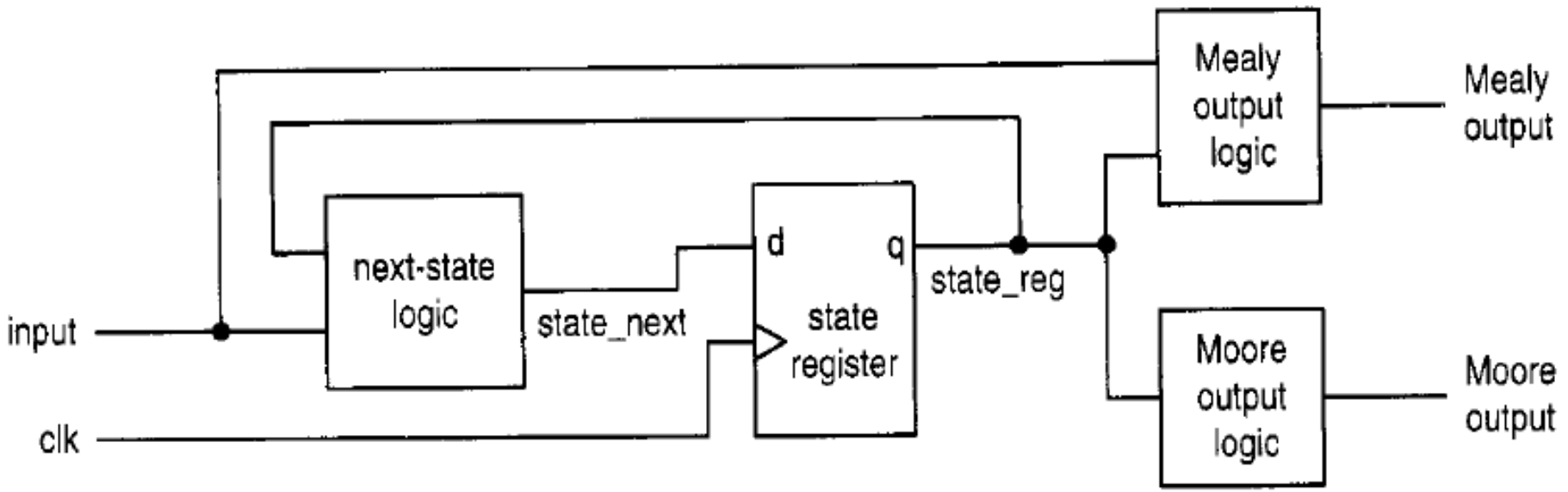
# FSM

# Block Diagram of A Synchronous FSM

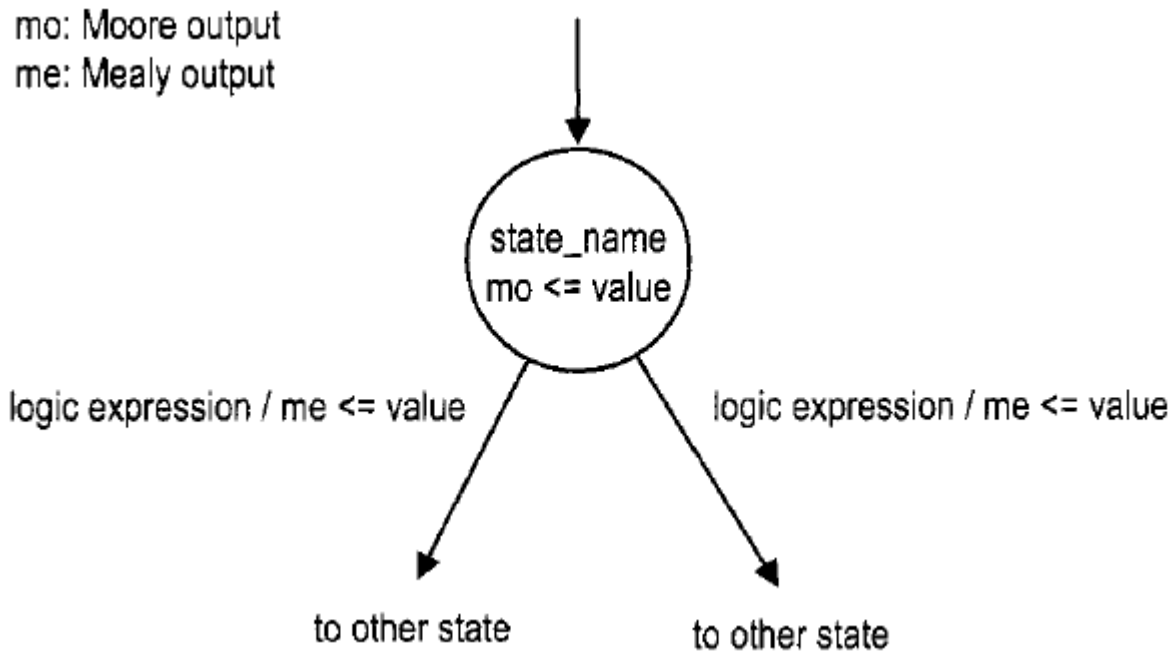


- The basic block diagram of an FSM is the same as that of a regular sequential circuit.
- It consists of a state register, next-state logic and output logic.
- An FSM is known as a *Moore machine* if the output is only a function of state.
- And is known as a *Mealy machine* if the output is a function of state and external input.

# FSM Representation

- An FSM is usually specified by an abstract *state diagram or ASM chart* (algorithmic state machine chart), both capturing the FSM's input, output, states and transitions in a graphical representation.
- The two representations provide the same information.
- The FSM representation is more compact and better for simple applications.
- The ASM chart representation is somewhat like a flow chart and is more descriptive for applications with complex transition conditions and actions.

# State Diagram



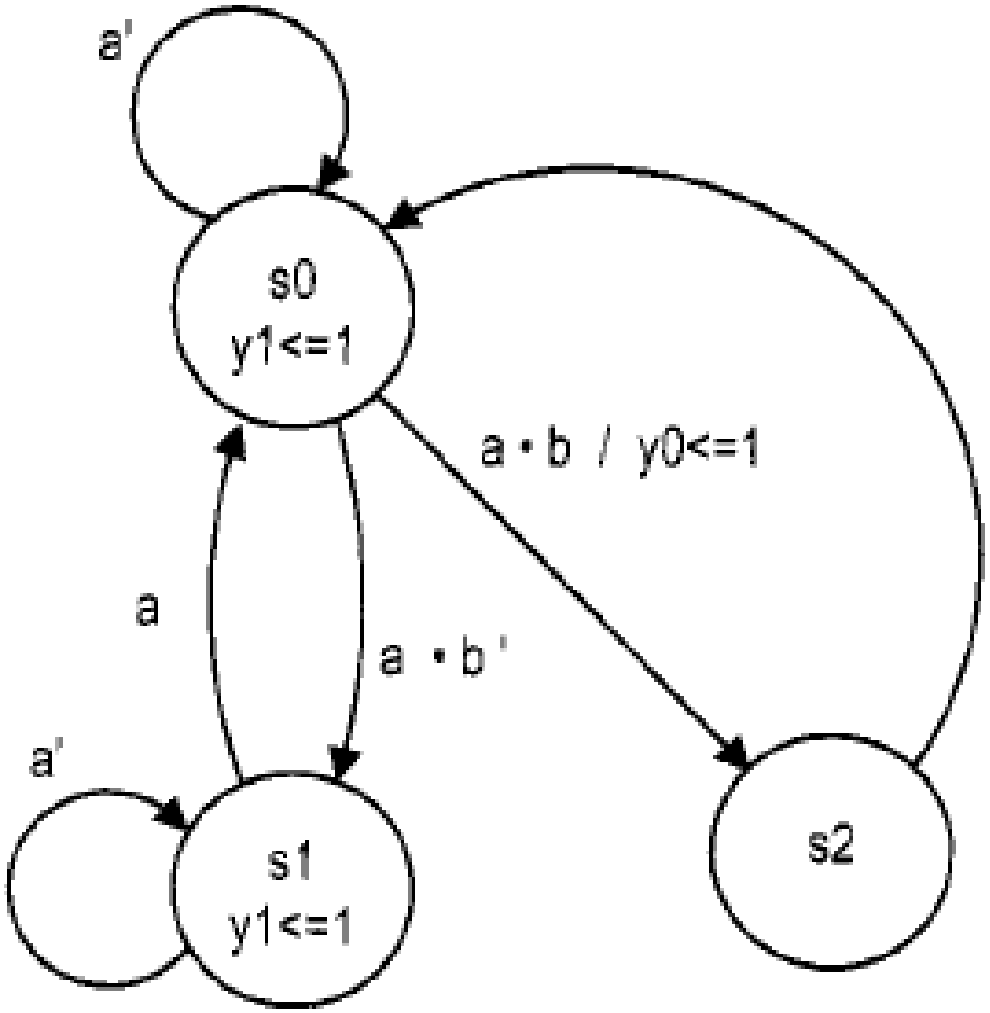

- A state diagram is composed of *nodes*, which represent states and are drawn as circles, and annotated *transitional arcs*.

- A logic expression expressed in terms of input signals is associated with each transition arc and represents a specific condition.
- The arc is taken when the corresponding expression is evaluated true.
- The Moore output values are placed inside the circle since they depend only on the current state.
- The Mealy output values are associated with the conditions of transition arcs since they depend on the current state and external input.

# FSM Code Development

- The procedure of developing code for an FSM is similar to that of a regular sequential circuit.
- We first separate the state register and then derive the code for combinational next-state logic and output logic.
- The main difference is the next-state logic.
- For an FSM, the code for the next-state logic follows the flow of a state diagram.
- For clarity and flexibility, we use the VHDL's *enumerated data type* to represent the FSM's states.

# Example of an FSM



- Consider the FSM of the previous slide, which has three states: s0, s1, and s2.
- We can introduced a user-defined enumerated data type as follows:

  **type** eg_state_type **is** (s0, s1, s2);
- The data type simply lists (i.e., *enumerates*) all symbolic values.
- Once the data type is defined, it can be used for the signals, as in

  **signal** state_reg, state_next: eg_state_type;
- During synthesis, software automatically maps the values in an enumerated data type to binary representations, a process known as *state assignment.*

- It consists of segments for the state register, next-state logic, Moore output logic, and Mealy output logic.

```
library ieee;
use ieee.std_logic_1164.all;
entity fsm_eg is
   port(
      clk, reset: in std_logic;
      a, b: in std_logic;
      y0, y1: out std_logic
   );
end fsm_eg;

architecture mult_seg_arch of fsm_eg is
   type eg_state_type is (s0, s1, s2);
   signal state_reg, state_next: eg_state_type;
begin
   -- state register
   process(clk,reset)
   begin
      if (reset='1') then
         state_reg <= s0;
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
      end if;
   end process;
```

```
    -- next-state logic
25  process(state_reg,a,b)
    begin
       case state_reg is
          when s0 =>
             if a='1' then
30              if b='1' then
                   state_next <= s2;
                else
                   state_next <= s1;
                end if;
35           else
                state_next <= s0;
             end if;
          when s1 =>
             if (a='1') then
40              state_next <= s0;
             else
                state_next <= s1;
             end if;
          when s2 =>
45           state_next <= s0;
       end case;
    end process;
```

```
    -- Moore output logic
    process(state_reg)
    begin
       case state_reg is
          when s0|s2 =>
             y1 <= '0';
          when s1 =>
             y1 <= '1';
       end case;
    end process;
    -- Mealy output logic
    process(state_reg,a,b)
    begin
       case state_reg is
          when s0 =>
             if (a='1') and (b='1') then
               y0 <= '1';
             else
               y0 <= '0';
             end if;
          when s1 | s2 =>
             y0 <= '0';
       end case;
    end process;
 end mult_seg_arch;
```

# FSM with merged combinational logic

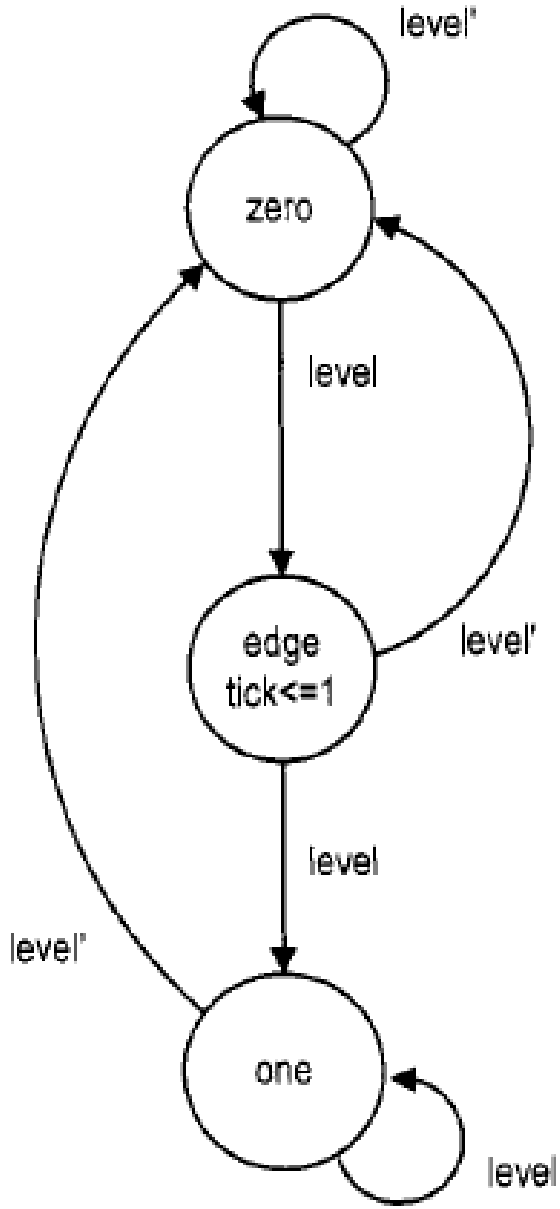
An alternative code is to merge next-state logic and output logic into single combinational block.

```
architecture two_seg_arch of fsm_eg is
   type eg_state_type is (s0, s1, s2);
   signal state_reg, state_next: eg_state_type;
begin
   -- state register
   process(clk,reset)
   begin
      if (reset='1') then
         state_reg <= s0;
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
      end if;
   end process;
   -- next-state/output logic
   process(state_reg,a,b)
   begin
      state_next <= state_reg;  -- default back to same state
      y0 <= '0';   -- default 0
      y1 <= '0';   -- default 0
```

```
20          case state_reg is
               when s0 =>
                  if a='1' then
                     if b='1' then
                        state_next <= s2;
25                      y0 <= '1';
                     else
                        state_next <= s1;
                     end if;
                  -- no else branch
30                end if;
               when s1 =>
                  y1 <= '1';
                  if (a='1') then
                     state_next <= s0;
35                -- no else branch
                  end if;
               when s2 =>
                  state_next <= s0;
            end case;
40       end process;
     end two_seg_arch;
```

# Example: Rising Edge Detector

- The rising-edge detector is a circuit that generates a short, one-clock-cycle pulse *(tick)* when the input signal changes from ‘0’ to ‘1’.
- **Moore-based design**
  - The zero and one state indicate that the input signal has been ‘0’ and ‘1’ for awhile.
  - The rising edge occurs when the input changes to ‘1’ in the zero state.
  - The FSM moves to the edge state and the output, tick, is asserted in this state.


level'
zero
level
edge
tick<=1
level'
level
level'
one
level

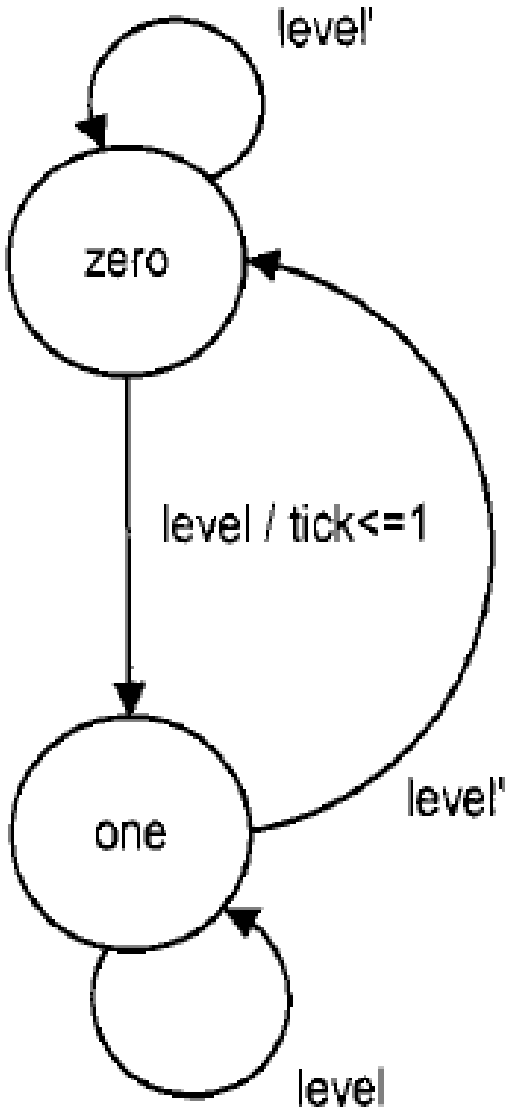
# Moore-machine based edge detector

```
use ieee.std_logic_1164.all;
entity edge_detect is
    port(
        clk, reset: in std_logic;
        level: in std_logic;
        tick: out std_logic
    );
end edge_detect;

architecture moore_arch of edge_detect is
    type state_type is (zero, edge, one);
    signal state_reg, state_next: state_type;
begin
    -- state register
    process(clk,reset)
    begin
        if (reset='1') then
            state_reg <= zero;
        elsif (clk'event and clk='1') then
            state_reg <= state_next;
        end if;
    end process;
    -- next-state/output logic
```

```
25      process(state_reg,level)
        begin
           state_next <= state_reg;
           tick <= '0';
           case state_reg is
30            when zero=>
                 if level= '1' then
                    state_next <= edge;
                 end if;
              when edge =>
35               tick <= '1';
                 if level= '1' then
                    state_next <= one;
                 else
                    state_next <= zero;
40               end if;
              when one =>
                 if level= '0' then
                    state_next <= zero;
                 end if;
45        end case;
     end process;
  end moore_arch;
```

- **Mealy-based design**



```
architecture mealy_arch of edge_detect is
   type state_type is (zero, one);
   signal state_reg, state_next: state_type;
begin
   -- state register
   process(clk,reset)
   begin
      if (reset='1') then
         state_reg <= zero;
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
      end if;
   end process;
   -- next-state/output logic
   process(state_reg,level)
   begin
      state_next <= state_reg;
      tick <= '0';
```

```
        case state_reg is
            when zero=>
                if level= '1' then
                    state_next <= one;
                    tick <= '1';
                end if;
            when one =>
                if level= '0' then
                    state_next <= zero;
                end if;
        end case;
    end process;
end mealy_arch;
```