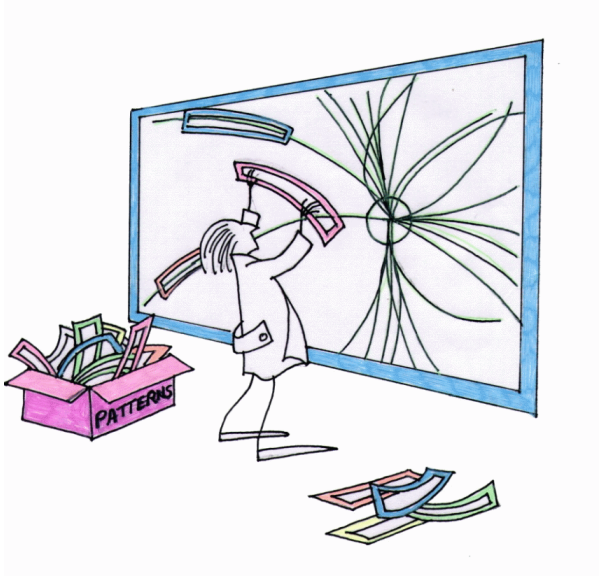
MARIE CURIE IAPP: FAST TRACKER FOR HADRON COLLIDER EXPERIMENTS

1ST SUMMER SCHOOL: VHDL BOOTCAMP

PISA, JULY 2013

Finite State Machines (FSMs)

Calliope-Louisa Sotiropoulou
PhD Candidate/Researcher
Aristotle University of Thessaloniki



AUTH e-LAB
Aristotle University of Thessaloniki-Electronics Laboratory

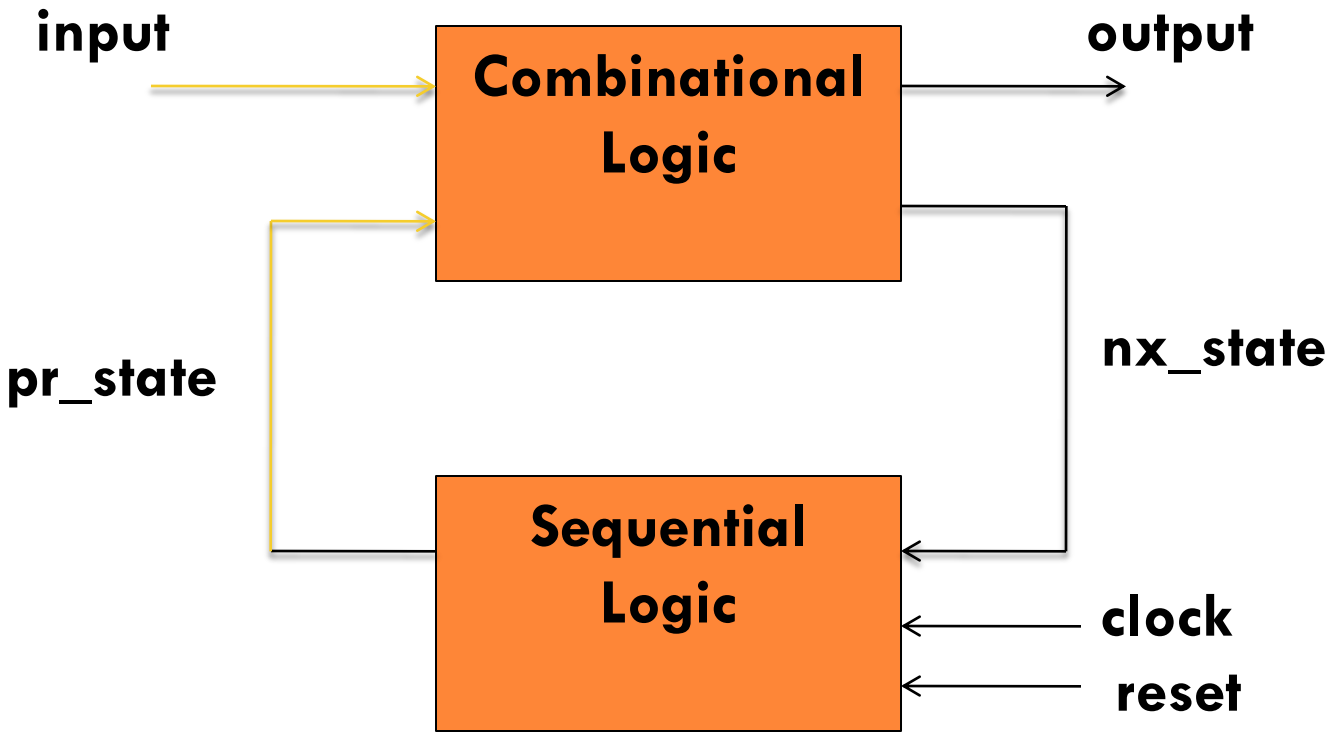
UNIVERSITÀ DI PISA

Finite State Machines

Finite State Machines

- A **finite state machine (FSM)** is a sequential logic circuit which moves between a finite set of states, dependent upon the values of the inputs and the previous state. The state transitions are synchronized on a clock.
- FSMs are used for designs who require a defined sequence of events (e.g. **control modules**)
- They consist of two parts:
  - Combinational Logic
  - Memory elements

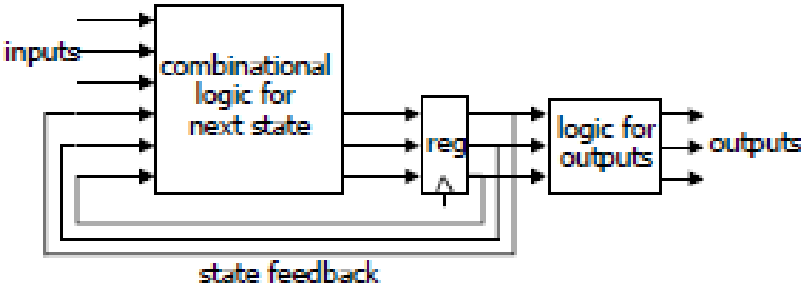
Finite State Machines



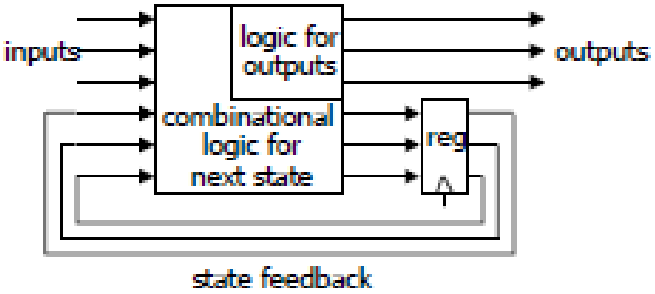
Finite State Machines

- There are two different types of FSMs **Mealy** vs **Moore**
- **Moore Machines:**
  Outputs are a function of current state
  Outputs change synchronously with state changes
- **Mealy machine:**
  Outputs depend on state and on inputs
  Input changes can cause immediate output changes (asynchronous)

Moore vs Mealy Machines



Moore Machine



Mealy Machine

How to write an FSM – Style 1

- Using two processes: One for the combinational part and one for the FSM state
- For the different states a new enumerated type of signal is usually defined

```
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;
```

How to write an FSM – Style 1

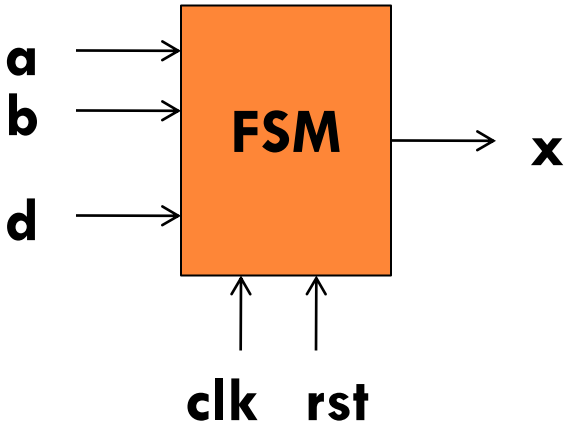
```
Process (input, pr_state)
Begin
        case pr_state is
                when state0 =>
                                if (input = …) then
                                                output <= <value>;
                                                nx_state <= state1;
                                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 <= state1;
                                else …
                                end if;
…
        end case;
End process;
```

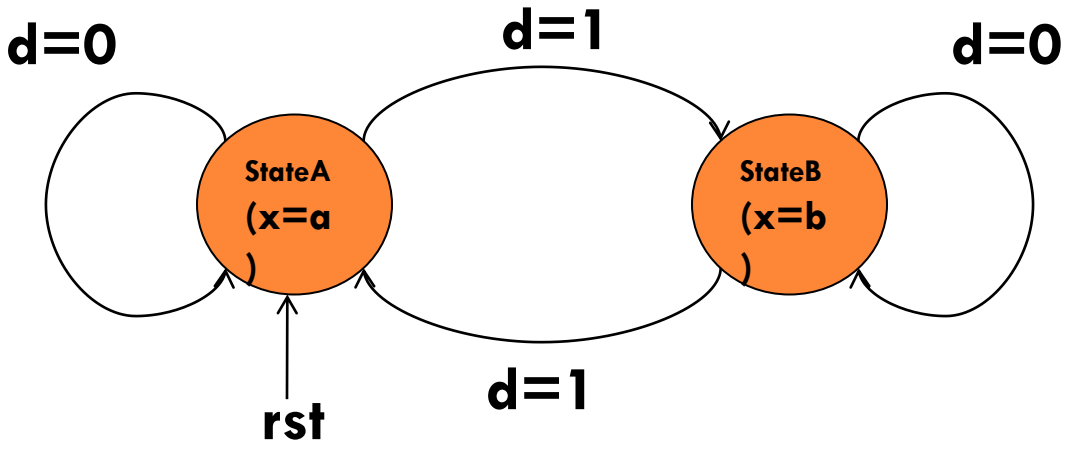
The use of **CASE** is very common

All the input signals must be in the sensitivity list

All the possible outputs must be defined

How to write an FSM – Style 1






```
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
```

How to write an FSM – Style 1

```
------------  sequential part ---------------------
Process (rst, clk)
Begin
      if (rst='1') then
            pr_state <= stateA;
      elsif (clock'event and clk='1')  then
            pr_state <= nx_state;
      end if;
End process;
```

```
-------------- combinational part----------------------
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;
```

How to write an FSM – Style 2

- The output is assigned on clock change
- Only one process is used

```
Architecture <arch_name> of <ent_name>
    type states is (state0, state1, state2, state3, …);
    signal pr_state: states;
    signal temp: <data_type>
Begin
    Process (reset, clock, pr_state)
    Begin
        if (reset='1') then
            pr_state <= state0;
        elsif (clock'event and clock='1') then
            case pr_state is
                when state0 =>
                    output<= <value>;
                    if (condition) then  pr_state <= state1;
                        …
                    end if;
                when state1 =>
                        output<= <value>;
                        if (condition) then  pr_state <= state2;
                            …
                        end if;
                ……
            end case;
        end if;
    end process;
End <arch_name>;
```

Example – BCD counter

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

Example – BCD counter

```
------------ ---------------------
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 eight =>
                                                count <= "1000";
                                                nx_state <= nine;
                                 when nine =>
                                                count <= "1001";
                                                nx_state <= zero;
                end case;
        end process;
end state_machine;
```

FSM

- So…

What does our FSM do?