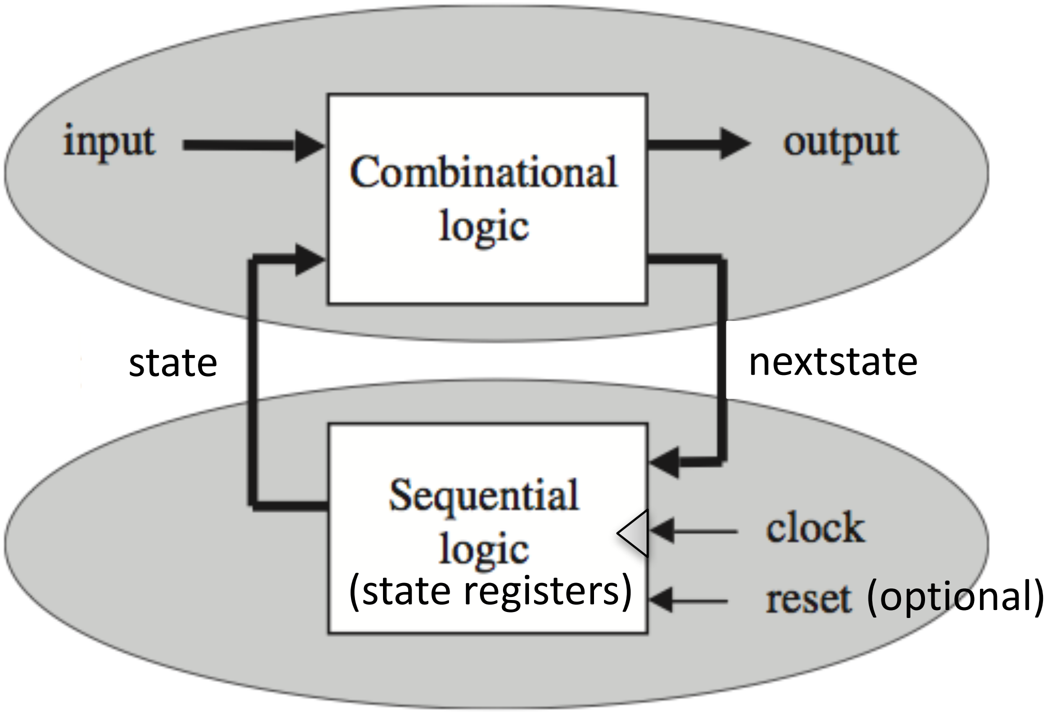
Finite State Machines

FSM



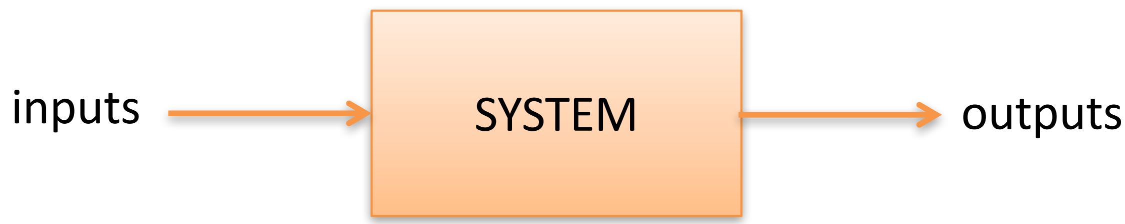
Generic State Machine Model

Guidelines for coding FSMs in VHDL:

* Use separate processes for sequential logic and combinational logic
* Use enumerated data type to list all possible states

State Machine: key idea

- State machines are an effective mathematical MoC that allow to characterize in an un-ambiguous and formal way the behavior of a system



- Goal: given a set of external stimuli we want to design a system that exhibit a desired behavior, i.e. the system must be able to process the stimuli provided at its inputs to produce certain actions at its outputs

State Machine: key idea

System Behavior:

STATE, ACTIONS

↓

check stimuli

↓

If necessary change
STATE and ACTIONS

stimuli ⟷ inputs

actions ⟷ outputs

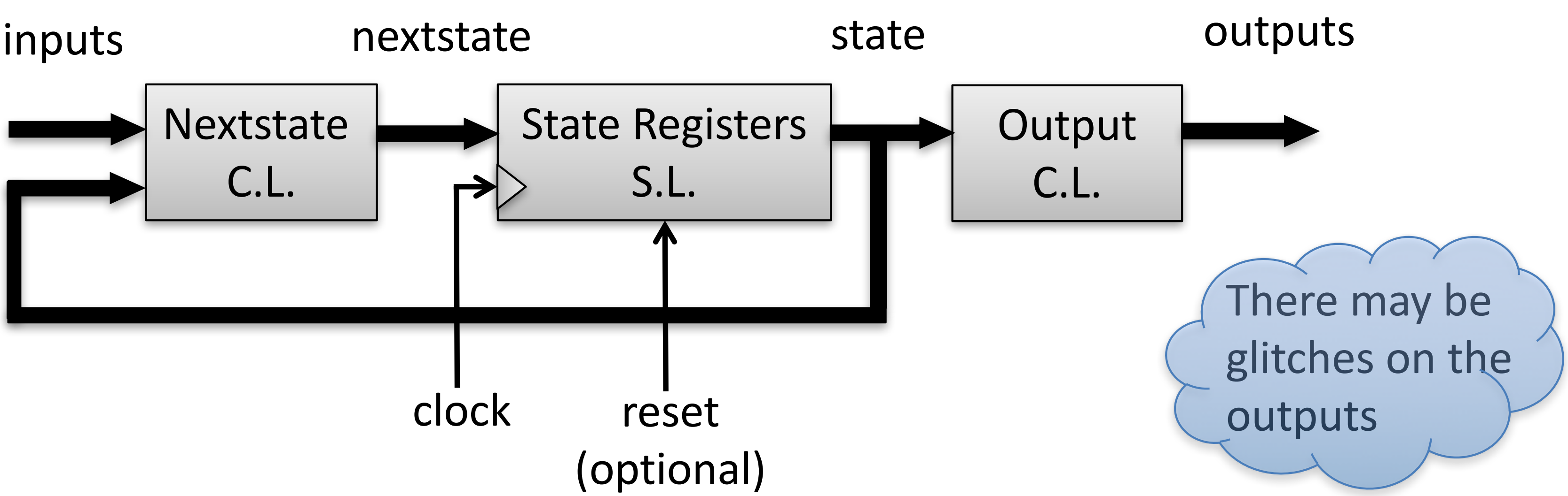
actions depends on stimuli & state

In summary:

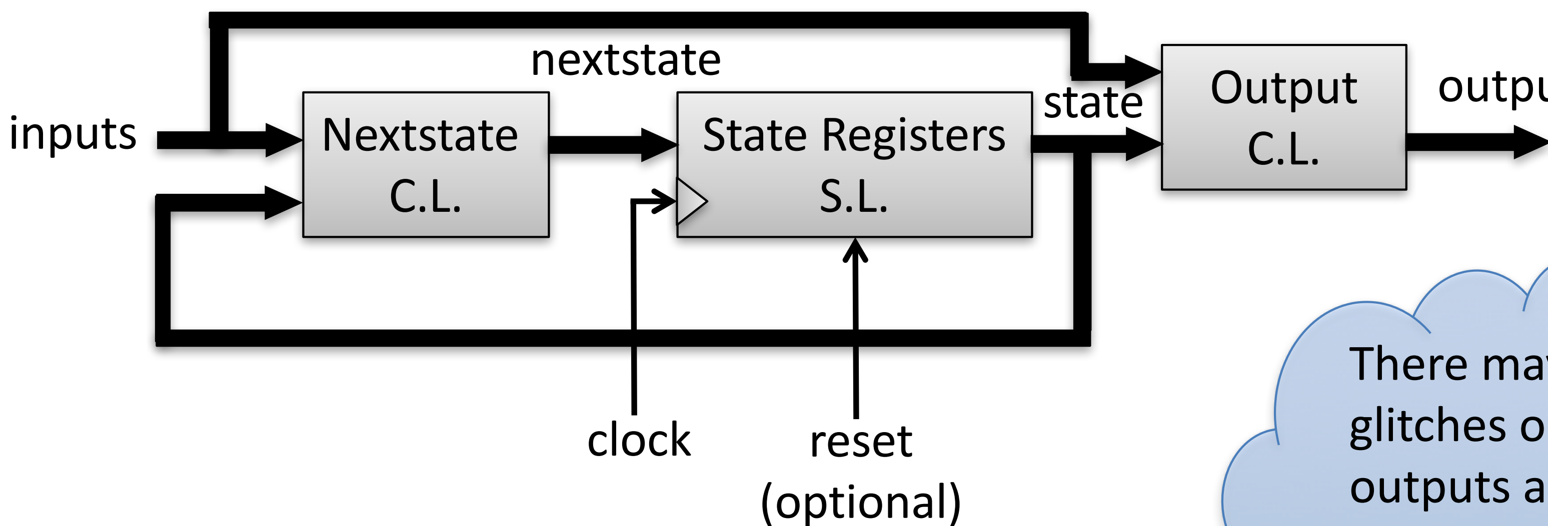
1. at any given time you are in a certain state, and perform certain actions

2. monitor external stimuli and “decide” what next state and actions should be

Moore style FSM



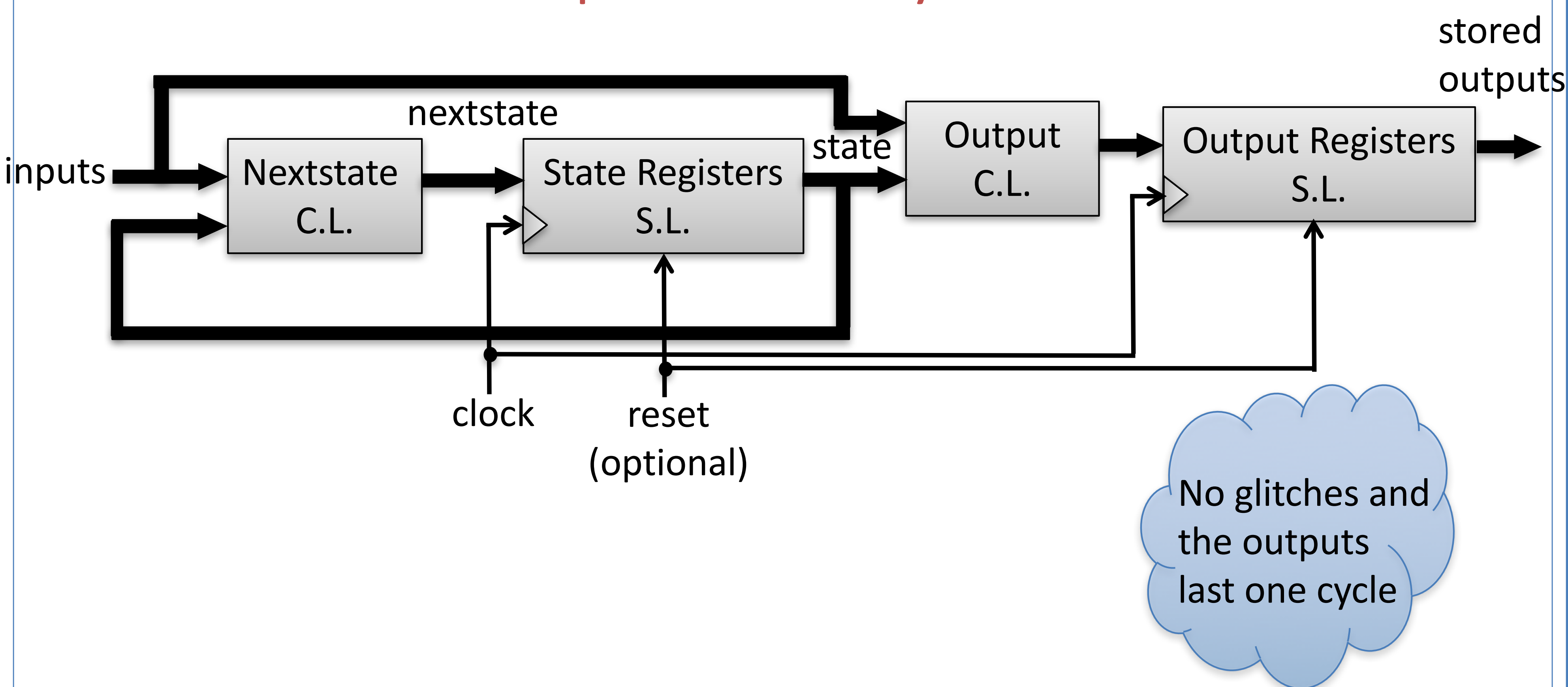
Mealy style FSM



There may be glitches on the outputs and the outputs may last less than one cycle

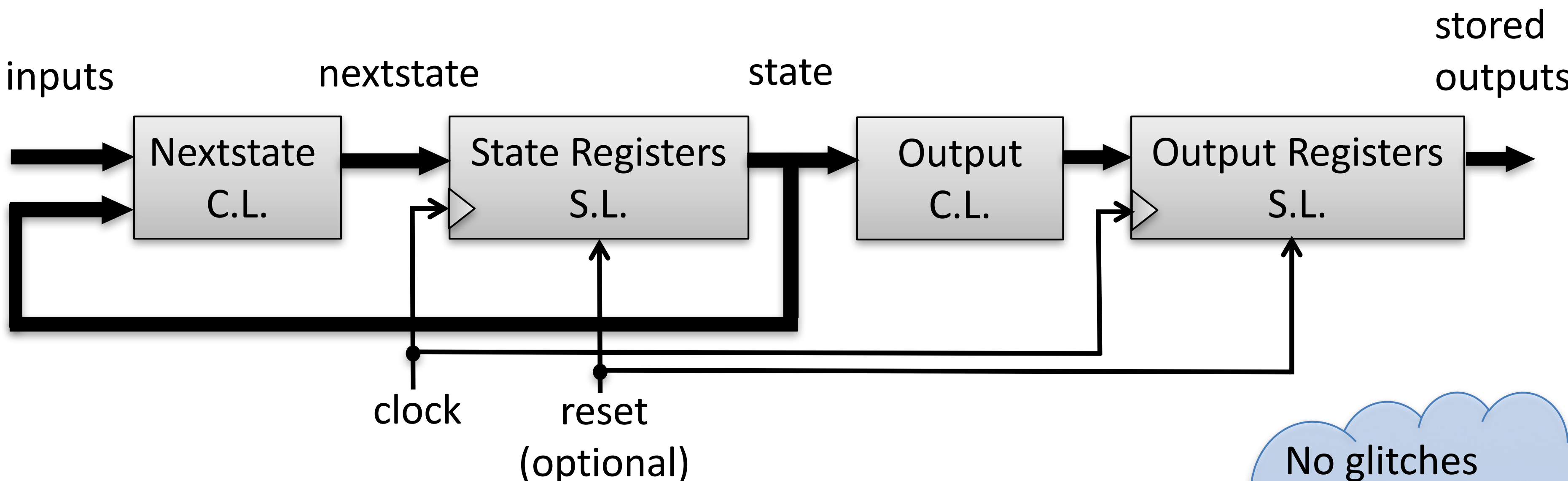
Registered Outputs FSMs

Pipelined Mealy



Registered Outputs FSMs

If we want, we can definitely pipeline "Moore" …

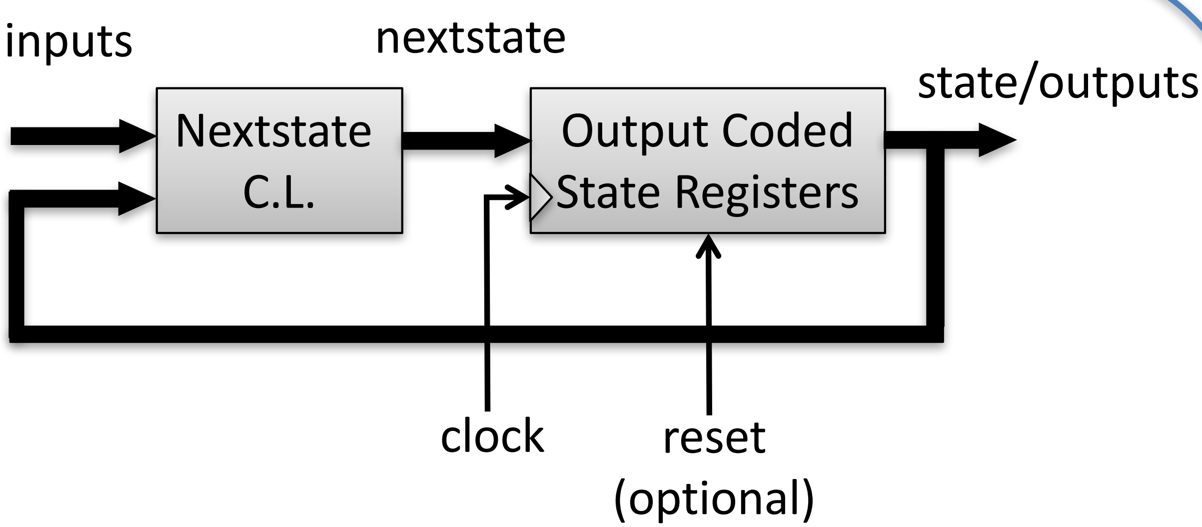


No glitches and the outputs last one clock cycle

Registered (a.k.a stored) outputs FSMs

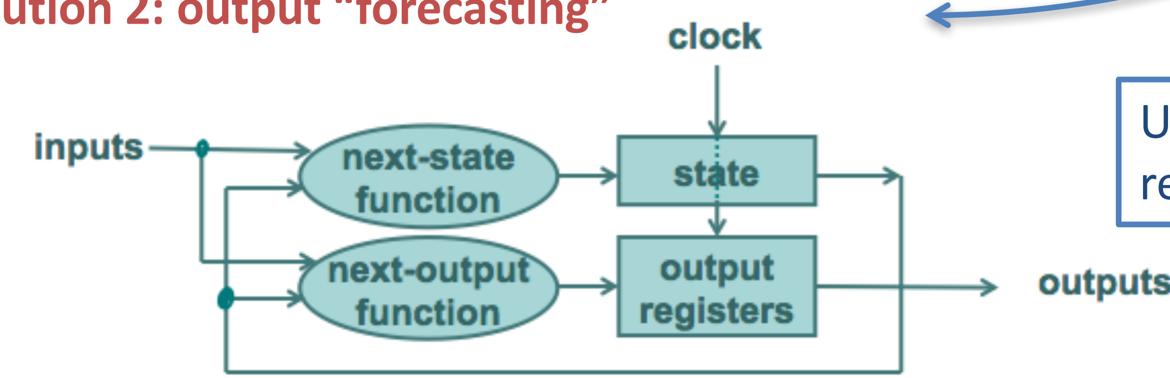
NOTE: with Moore FSMs is possible to get registered outputs without having to add a pipeline stage → outputs get speed up by one clock cycle !!!

Solution 1: output coded Moore



No glitches + outputs are faster

Solution 2: output "forecasting"

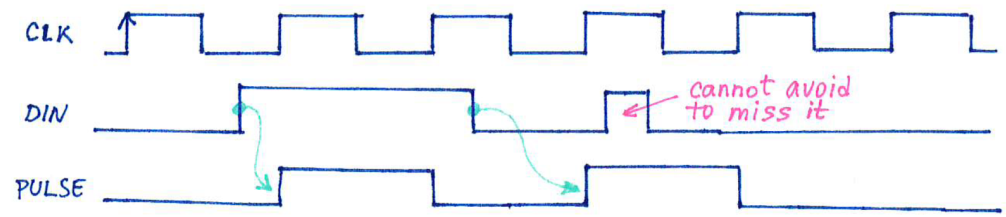


Usually solution 2 requires less thinking

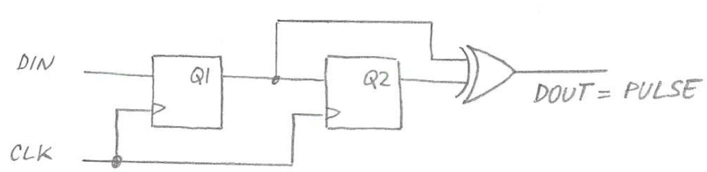
Design Example: Edge Detector

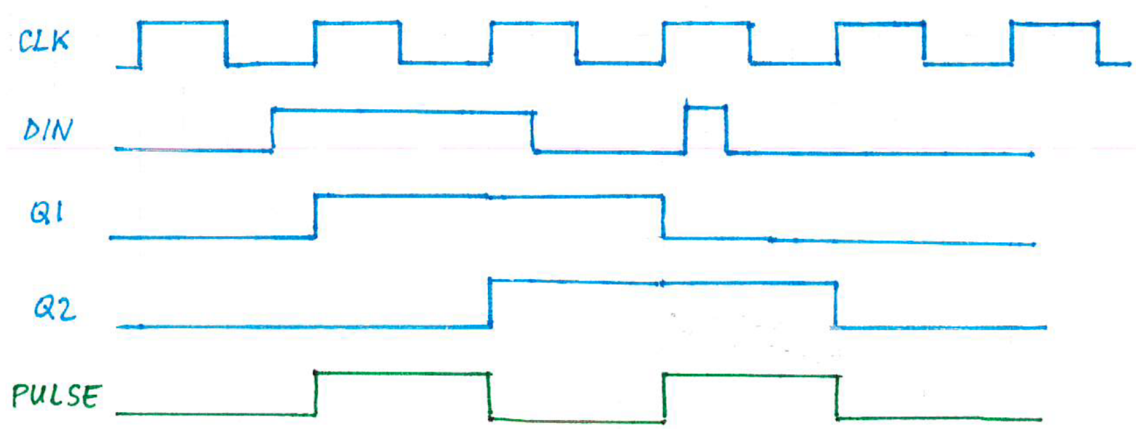
Edge Detector

- Design a state machine to implement an edge detector

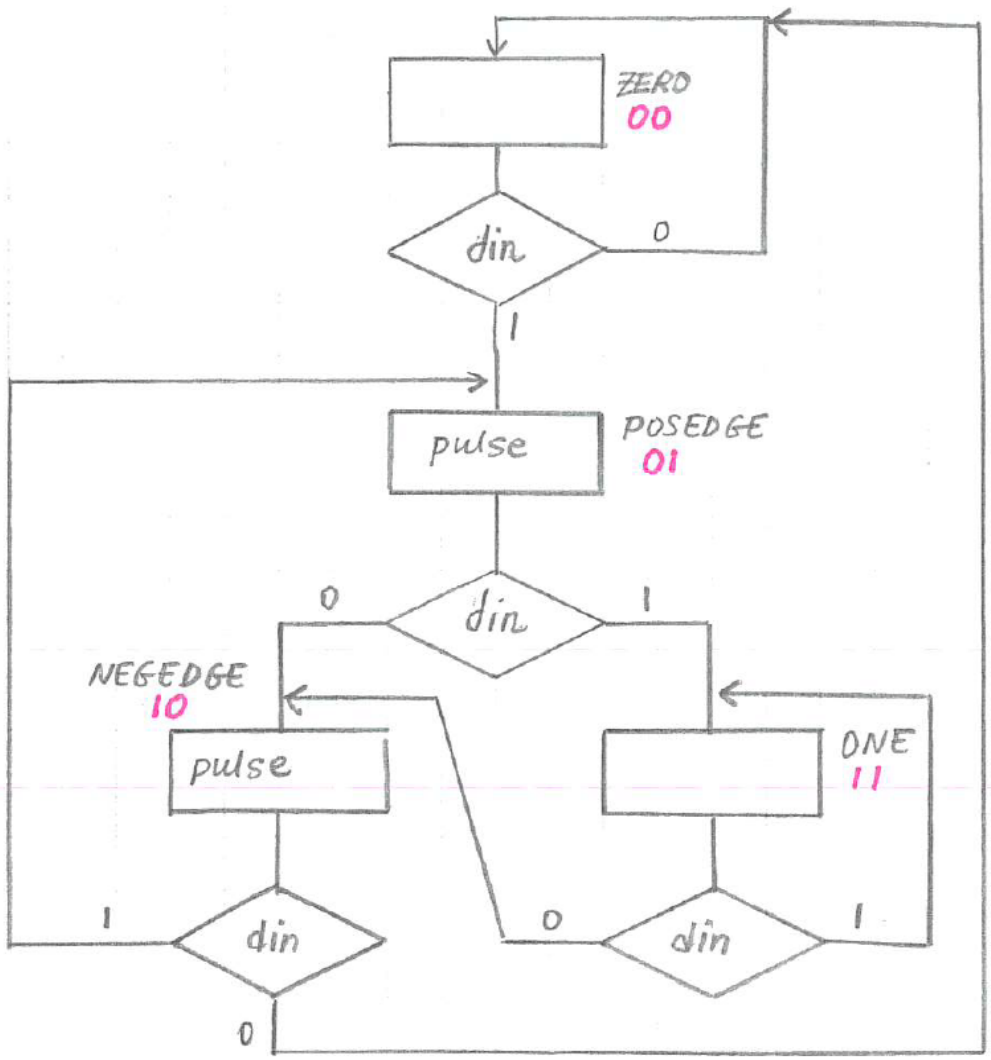


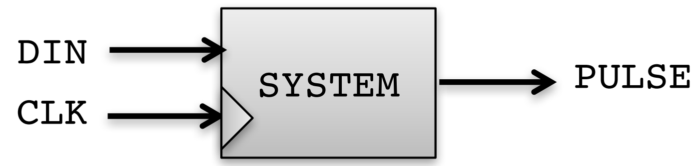
Intuitive solution:

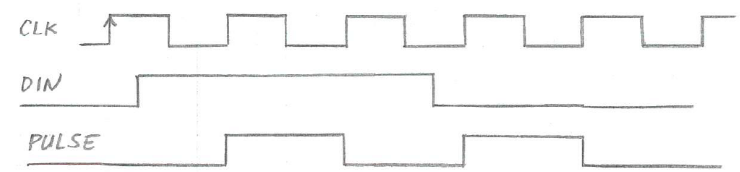


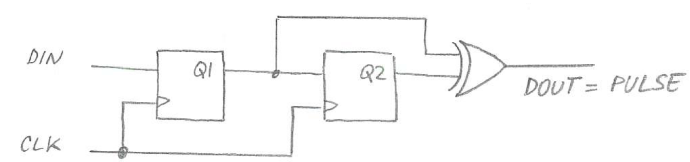


Moore style ASM









K. Maps for Moore's FSM

din	state = Q_1Q_0	next state = $Q_1^+Q_0^+$
0	zero = 00	zero = 00
1	zero = 00	posedge = 01
0	posedge = 01	negedge = 10
1	posedge = 01	one = 11
0	negedge = 10	zero = 00
1	negedge = 10	posedge = 01
0	one = 11	negedge = 10
1	one = 11	one = 11

din \ Q_1Q_0	00	01	11	10
0	0	1	1	0
1	0	1	1	0

$Q_1^+ = Q_0$

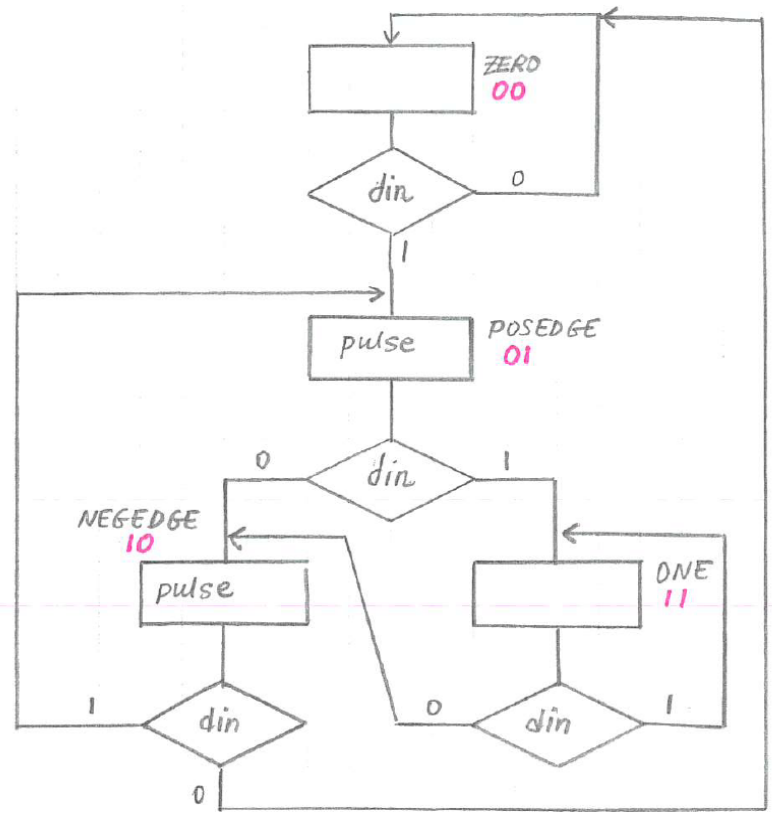
din \ Q_1Q_0	00	01	11	10
0	0	0	0	0
1	1	1	1	1

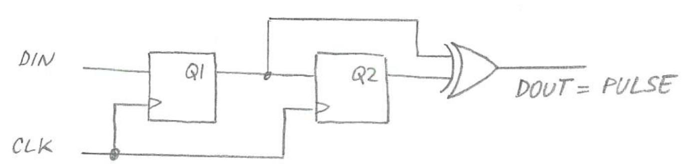
$Q_0^+ = din$

state = Q_1Q_0	dout = pulse
zero = 00	0
posedge = 01	1
negedge = 10	1
one = 11	0

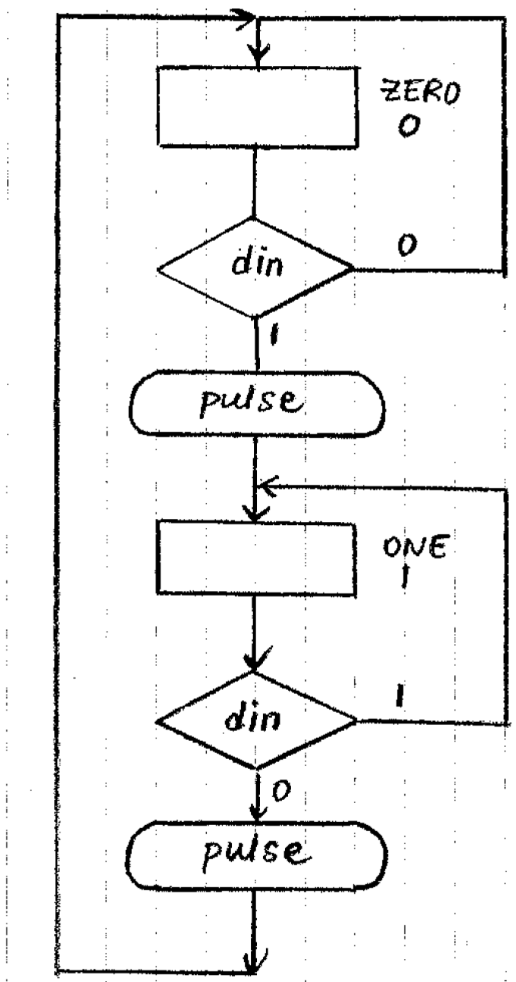
Q_0 \ Q_1	0	1
0	0	1
1	1	0

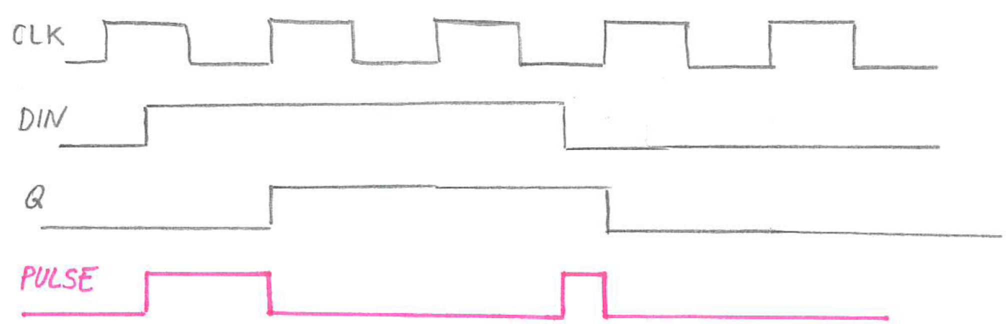
$pulse = Q_0 \oplus Q_1$

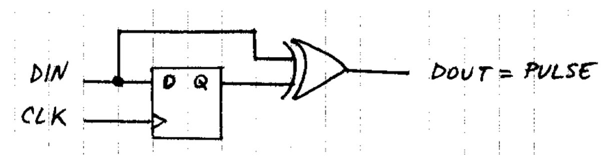




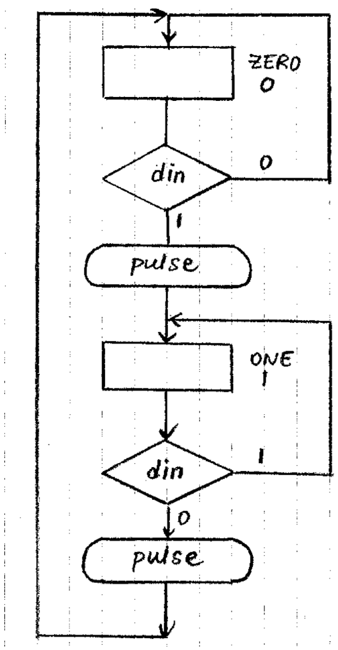
Mealy style ASM







K. Maps for Mealy's FSM



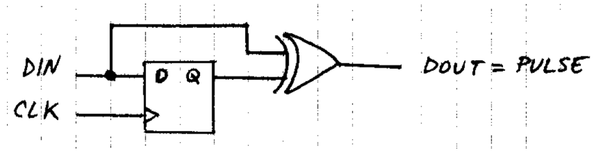
din	state = Q	nextstate = Q^+	pulse
0	zero = 0	zero = 0	0
1	zero = 0	one = 1	1
0	one = 1	zero = 0	1
1	one = 1	one = 1	0

din \ Q	0	1
0	0	0
1	1	1

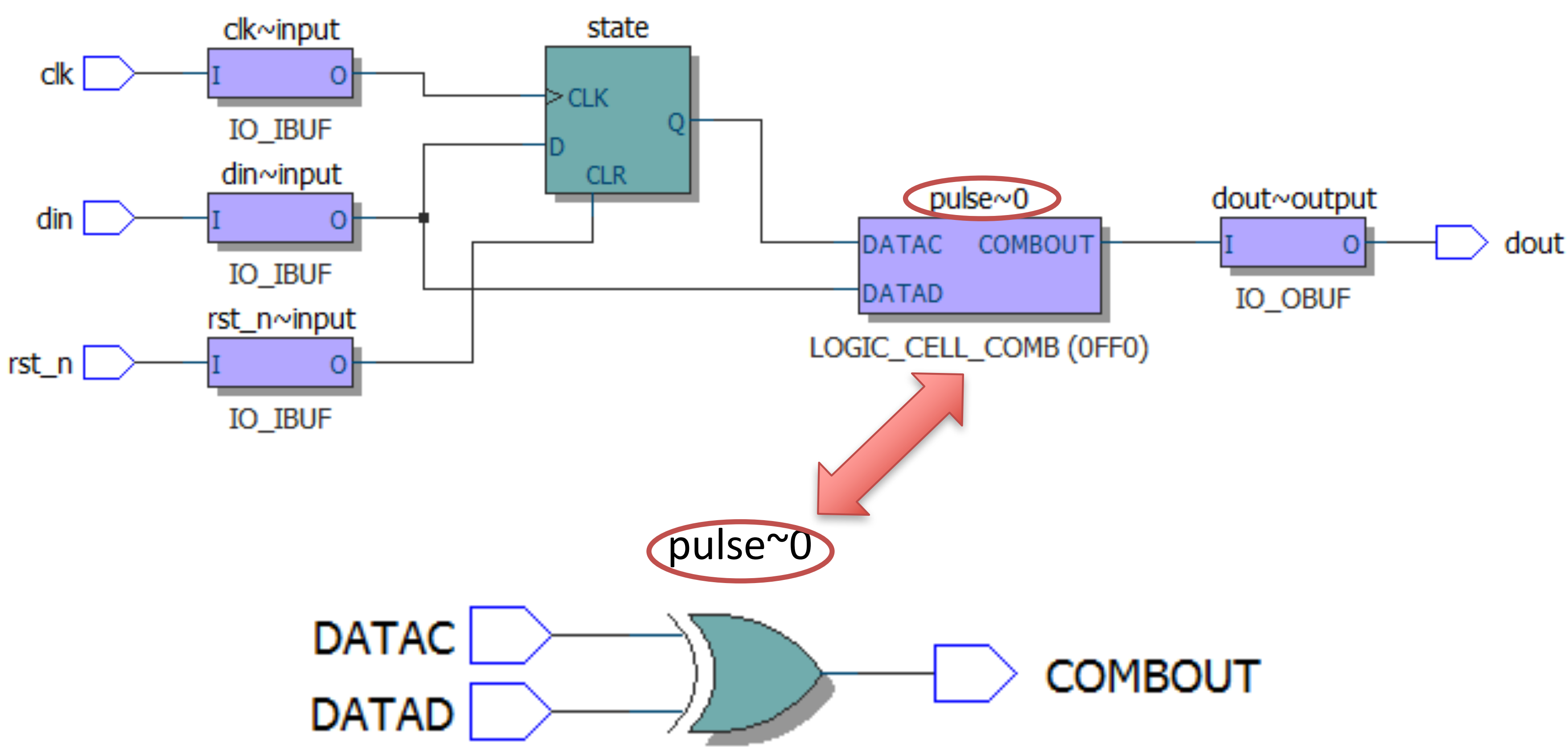
$Q^+ = din$

din \ Q	0	1
0	0	1
1	1	0

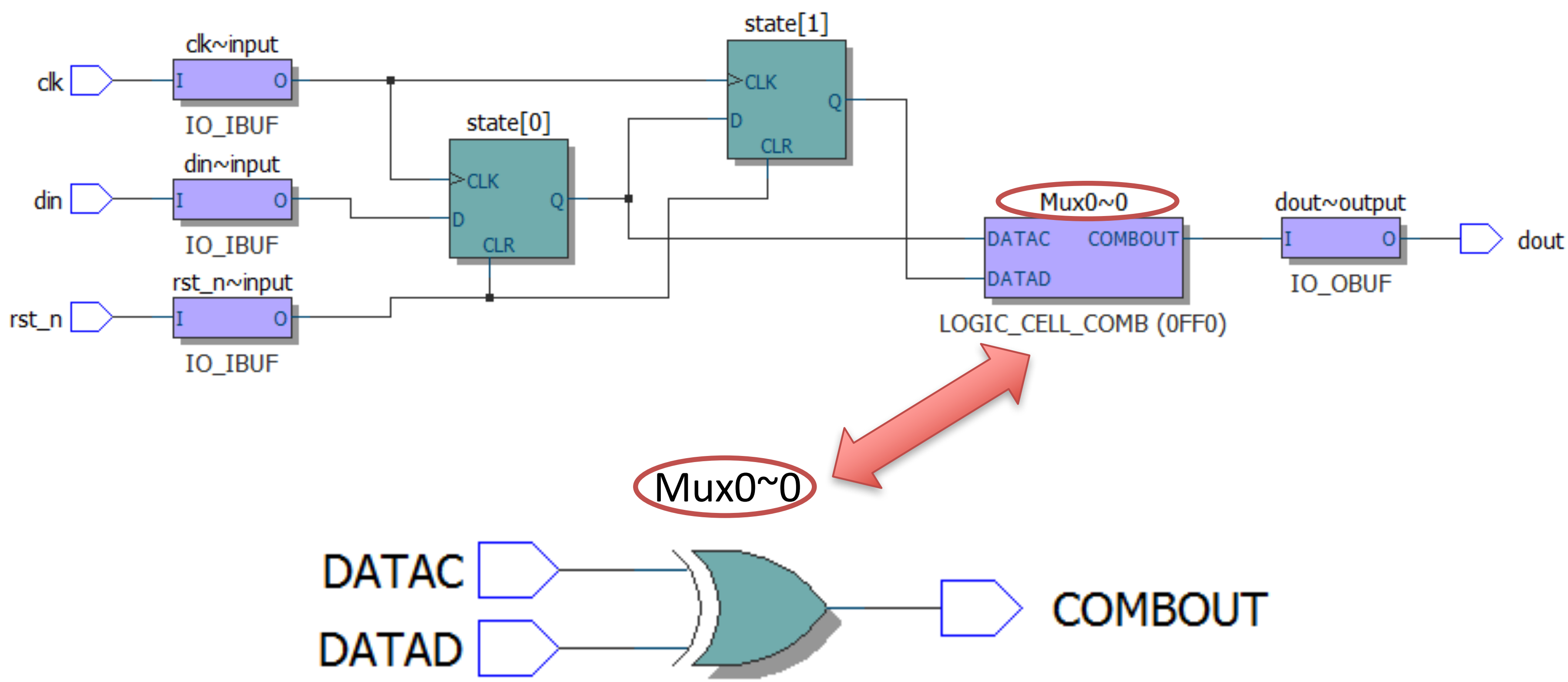
$pulse = din \oplus Q$



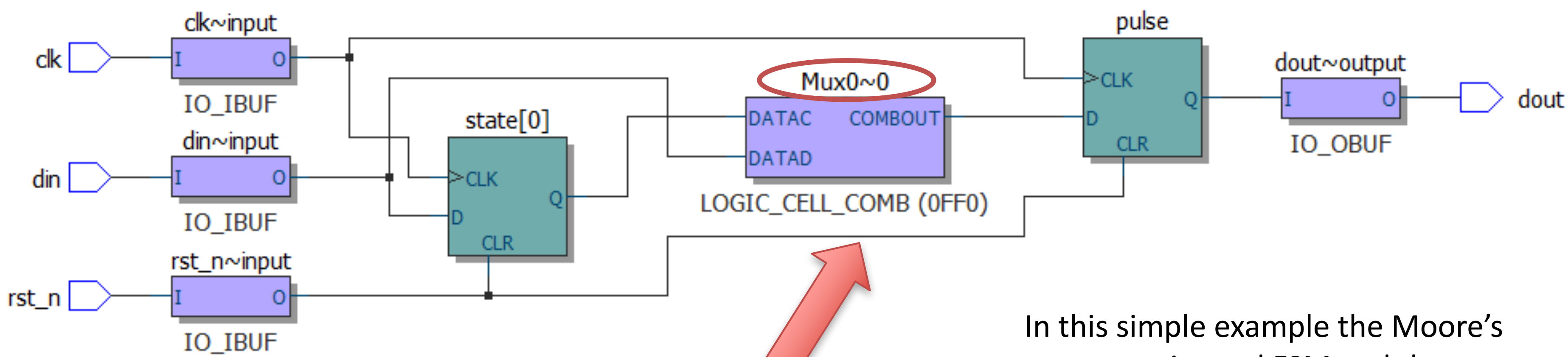
Mealy Implementation

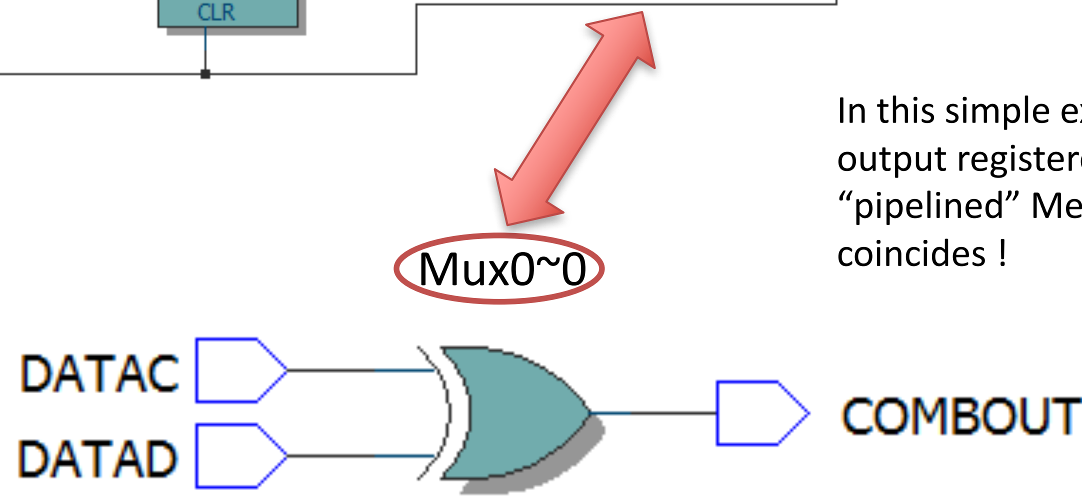


Moore implementation



Moore with registered output

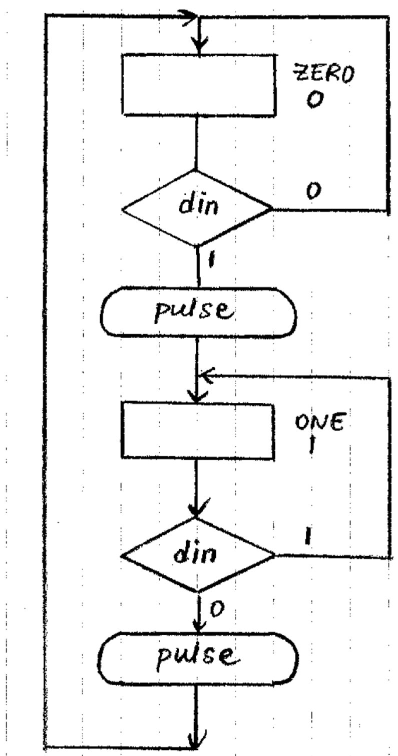




In this simple example the Moore's output registered FSM and the "pipelined" Mealy FSM happen to coincides !

Design Example:
Edge Detector VHDL coding

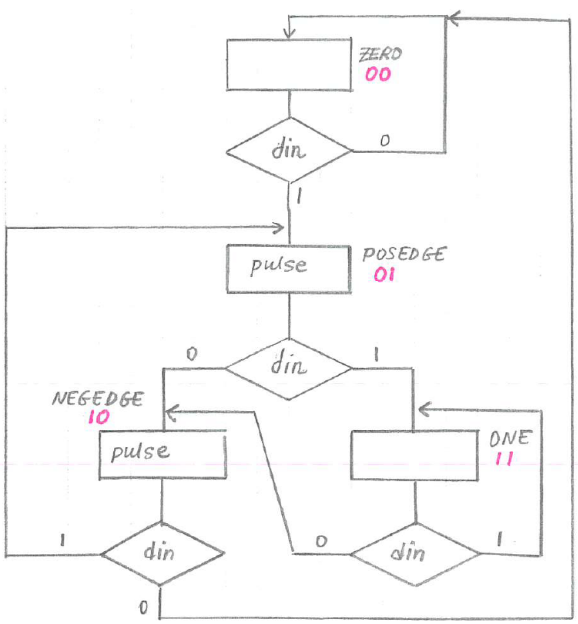
Mealy's FSM




```
--
-- author:   Claudio Talarico
-- file:     ed-mealy-rtl.vhd
-- comments: edge detector (Mealy FSM)
--

library ieee;
use ieee.std_logic_1164.all;

entity edge_detector is
port ( din    : in  std_logic;
       clk    : in  std_logic;
       rst_n  : in  std_logic;
       dout   : out std_logic
     );

end edge_detector;
```

```
architecture rtl of edge_detector is
type state_t is (zero, one);
signal state, next_state : state_t;
signal pulse : std_logic;

begin

  the_machine: process(din,state)
  begin

    -- defaults
    next_state <= zero;
    pulse      <= '0';

    case state is
      when zero =>
        if (din = '0') then
          next_state <= zero;
        else
          next_state <= one;
          pulse      <= '1';
        end if;
      when one =>
        if (din = '0') then
          next_state <= zero;
          pulse      <= '1';
        else
          next_state <= one;
        end if;
     when others =>
       -- do nothing
   end case;
  end process the_machine;

  the_registers: process(clk, rst_n)
  begin
    if (rst_n = '0') then
      state <= zero;
    elsif (clk='1' and clk'event) then
      state <= next_state;
    end if;
  end process the_registers;

  --dummy assignment
  dout <= pulse;
end rtl;
```

Moore’s FSM




```
--
-- author:   Claudio Talarico
-- file:     ed-moore-rtl.vhd
-- comments: edge detector
--           Moore FSM
--

library ieee;
use ieee.std_logic_1164.all;

entity edge_detector is
port ( din   : in  std_logic;
       clk   : in  std_logic;
       rst_n : in  std_logic;
       dout  : out std_logic
     );

end edge_detector;
```

```
architecture rtl of edge_detector is

type state_t is (zero, posedge, negedge, one);
signal state, next_state : state_t;
signal pulse : std_logic;

-----------------------------------------------------------------------
-- altera's specific attributes
attribute enum_encoding : string;
attribute enum_encoding of state_t : type is "sequential";
-- attribute enum_encoding of state_t : type is "gray";
-- attribute enum_encoding of state_t : type is "johnson";
-- attribute enum_encooding of state_t : type is "one-hot"; -- dafault
-- end of altera's specific attributes
-----------------------------------------------------------------------

begin

  the_machine: process(din,state)
  begin

    -- defaults
    next_state <= zero;
    pulse      <= '0';

    case state is
      when zero => -- stable zero
        if (din = '0') then
          next_state <= zero;
        else
          next_state <= posedge;
        end if;
      when posedge => -- positive edge
        pulse <= '1';
        if (din = '0') then
          next_state <= negedge;
        else
          next_state <= one;
        end if;
      when negedge => -- negative edge
        pulse <= '1';
        if (din = '0') then
          next_state <= zero;
        else
          next_state <= posedge;
        end if;
      when one => -- stable one
        if (din = '0') then
          next_state <= negedge;
        else
          next_state <= one;
        end if;
     when others =>
       -- do nothing
   end case;

  end process the_machine;

  the_registers: process(clk, rst_n)
  begin
    if (rst_n = '0') then
      state <= zero;
    elsif (clk='1' and clk'event) then
      state <= next_state;
    end if;
  end process the_registers;

  --dummy assignment
  dout <= pulse;

end rtl;
```

Moore's stored output FSM

```
--
-- author:   Claudio Talarico
-- file:     ed-moore-so-rtl.vhd
-- comments: edge detector
--           Moore FSM with stored output
--

library ieee;
use ieee.std_logic_1164.all;

entity edge_detector is
port ( din   : in  std_logic;
       clk   : in  std_logic;
       rst_n : in  std_logic;
       dout  : out std_logic
     );

end edge_detector;
```

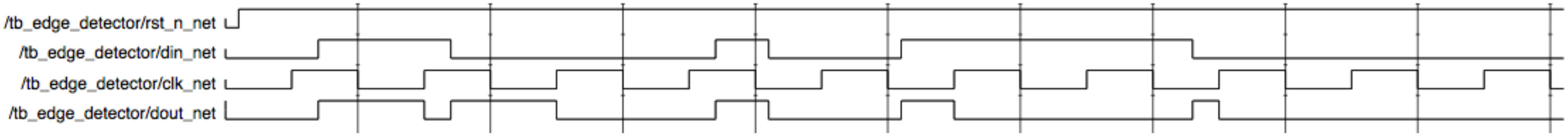
```
architecture rtl of edge_detector is

type state_t is (zero, posedge, negedge, one);
signal state, next_state : state_t;
signal next_pulse, pulse : std_logic;

------------------------------------------------------------------------
-- altera's specific attributes
attribute enum_encoding : string;
attribute enum_encoding of state_t : type is "sequential";
-- attribute enum_encoding of state_t : type is "gray";
-- attribute enum_encoding of state_t : type is "johnson";
-- attribute enum_encooding of state_t : type is "one-hot"; -- dafault
-- end of altera's specific attributes
------------------------------------------------------------------------

begin

  the_machine: process(din,state)
  begin

    -- defaults
    next_state <= zero;
    next_pulse <= '0';

    case state is
      when zero => -- stable zero
        if (din = '0') then
          next_state <= zero;
          next_pulse <= '0';
        else
          next_state <= posedge;
          next_pulse <= '1';
        end if;
      when posedge => -- positive edge
        if (din = '0') then
          next_state <= negedge;
          next_pulse <= '1';
        else
          next_state <= one;
          next_pulse <= '0';
        end if;
      when negedge => -- negative edge
        if (din = '0') then
          next_state <= zero;
          next_pulse <= '0';
        else
          next_state <= posedge;
          next_pulse <= '1';
        end if;
      when one => -- stable one
        if (din = '0') then
          next_state <= negedge;
          next_pulse <= '1';
        else
          next_state <= one;
          next_pulse <= '0';
        end if;
     when others =>
       -- do nothing
    end case;

  end process the_machine;
```

```
  the_registers: process(clk, rst_n)
  begin
    if (rst_n = '0') then
      state <= zero;
      pulse <= '0';
    elsif (clk='1' and clk'event) then
      state <= next_state;
      pulse <= next_pulse;
    end if;
  end process the_registers;

  --dummy assignment
  dout <= pulse;

end rtl;
```

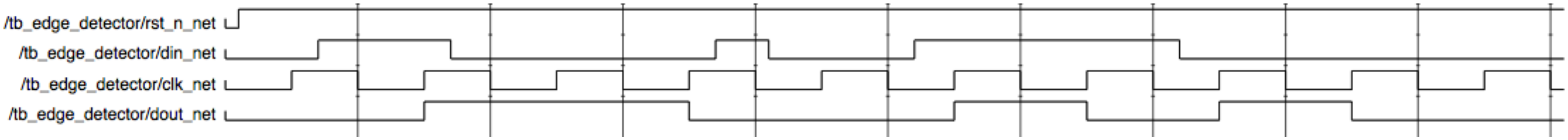
Edge Detector: Functional Simulation

Functional Simulation

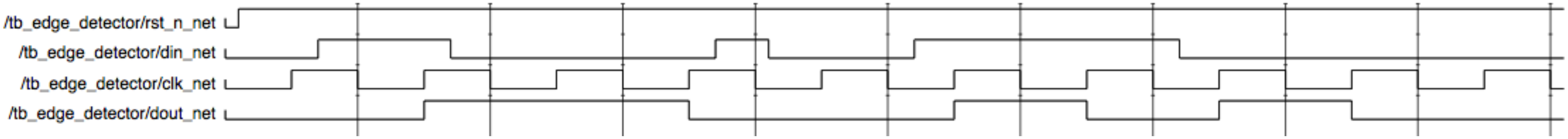
Mealy:



Moore:



Moore with stored output:



Edge Detector: Testbench VHDL code

Testbench

```
library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;

entity tb_edge_detector is
  --empty
end tb_edge_detector;


architecture beh of tb_edge_detector is

  component edge_detector
    port(
      din    :  in   std_logic;
      clk    :  in   std_logic;
      rst_n  :  in   std_logic;
      dout   :  out  std_logic
    );
  end component edge_detector;


  --signal declaration

     signal clk_net          : std_logic;
     signal rst_n_net        : std_logic;
     signal din_net          : std_logic;
     signal dout_net         : std_logic;

 begin
   inst_1: edge_detector
     port map(
       din    =>  din_net,
       clk    =>  clk_net,
       rst_n  =>  rst_n_net,
       dout   =>  dout_net
     );

clk_p : process
begin
  clk_net <= '0';
  wait for 5 ns;
  clk_net <= '1';
  wait for 5 ns;
end process clk_p;

input_data : process
  begin
    din_net <= '0';
    wait for 7 ns;
    din_net <= '1';
    wait for 10 ns;
    din_net <= '0';
    wait for 20 ns;
    din_net <= '1';
    wait for 4 ns;
    din_net <= '0';
    wait for 11 ns;
    din_net <= '1';
    wait for 20 ns;
    din_net <= '0';
    wait for 100 ns;
  end process input_data;

test_bench : process
  begin

    rst_n_net <= '0';
    wait for 1 ns;
    rst_n_net <= '1';
    wait for 100 ns;

    assert false
    report "End of Simulation"
    severity failure;

  end process test_bench;

end beh;
```