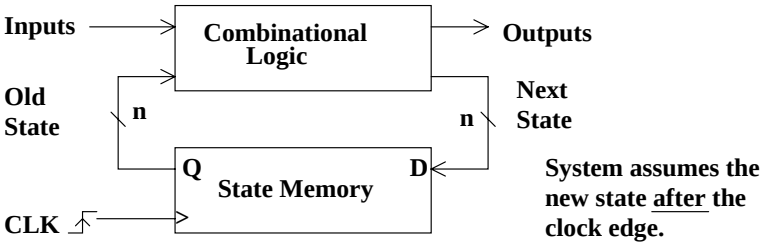



L8: FSMs and Synchronization

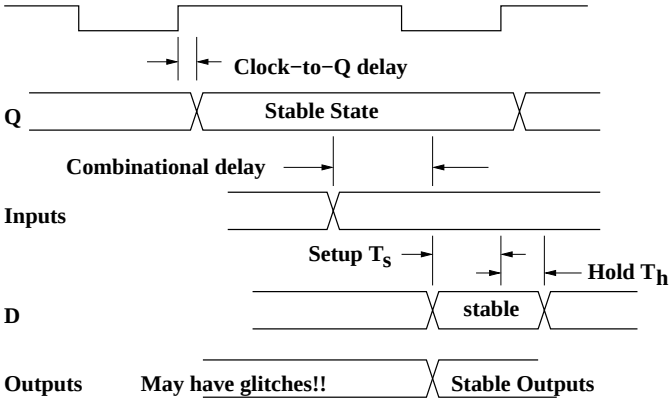
Finite State Machines

- Finite State Machines (FSMs) are clocked sequential systems.
  - We have already seen simple FSMs in flip-flops and counters.
  - But you can do much more complex things with them.
  - After a clock edge, the FSM assumes a state that depends on
    - the state that the FSM WAS in and
    - the inputs just before (and a little after) the clock edge.



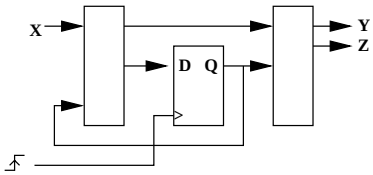
Timing of an FSM

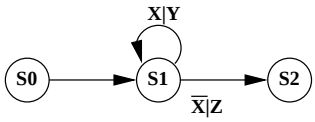
- Clock speed is limited by the sum of the flip-flop delay, combinational delay, setup time, and skew.
- Contamination delay of a flip-flop minus skew must be greater than the hold time of the (succeeding) flip-flop.
- The next state is determined by the inputs and the old state and this combinational function must settle to provide setup time for the next clock edge.



Mealy Model

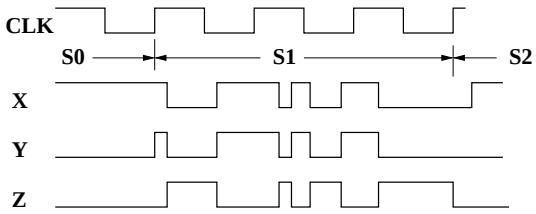
- With this type of FSM, the outputs can change asynchronously in response to changes in the inputs.





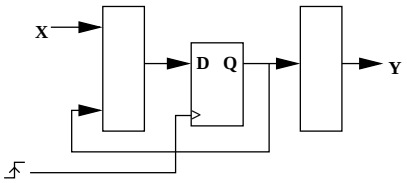
"Mealy Model": Output = F(State, Input)

Arcs between states also note output.

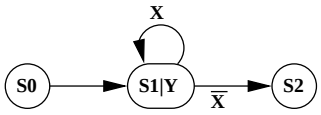


Moore Model

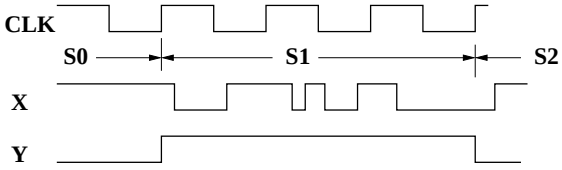
- With this type of FSM, the outputs are fixed for each clock cycle.
- The outputs change only after the clock edges.
  - The inputs do not directly affect the outputs.
  - They determine, with the present state, the next state.



"Moore Model": Output = F(State)

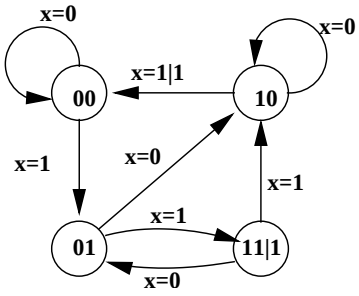


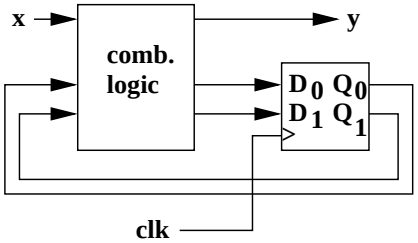
Arcs note transitions only.
State names describe output.



Simple FSM

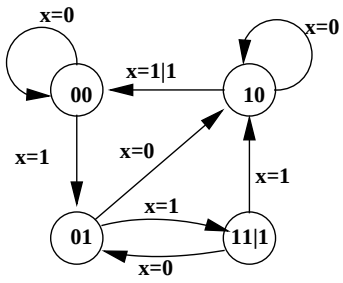
- One way of describing an FSM is in terms of transitions to be made on each clock edge. This is a Mealy machine.
- Here, the state names are numbers in the form Q1 Q0.
- Four states require a minimum of two bits to encode them.
  - Four is the maximum number of bits required.





FSM Logic

- It is straightforward to build a truth table for the next state and the output based on the present state and the input.
  - The equations can be easily derived directly from the truth table or from Karnaugh maps.



Q_1	Q_0	x	D_1	D_0	y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	0	0	1
1	1	0	0	1	1
1	1	1	1	0	1

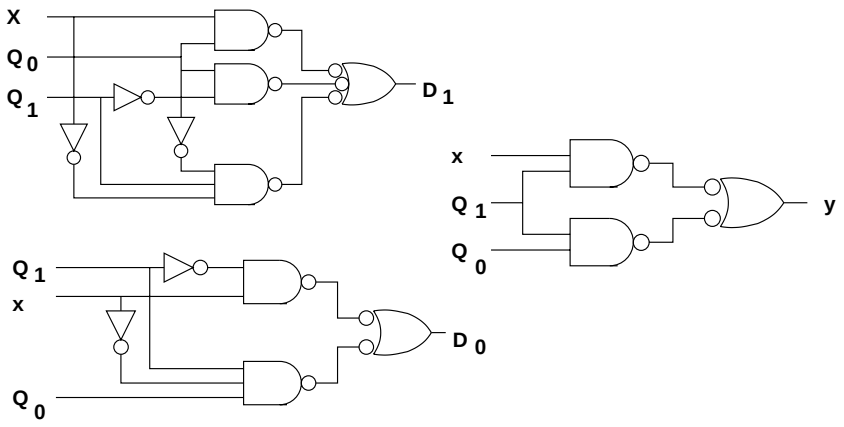
D0 = x*/Q1+/x*Q0*Q1

D1 = x*Q0+/Q1*Q0+/x*Q1*/Q0

y = x*Q1+Q1*Q0

FSM Combinatonal Logic

- This is the logic to implement the FSM if it were described by a schematic of discrete gates.

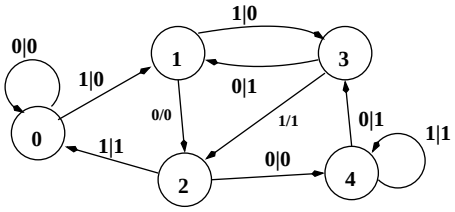


Another FSM - Divide by Five

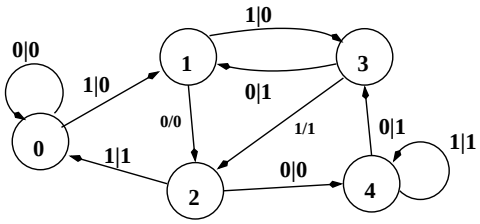
- There is a single input which represents a number.
  - The LSB comes first, another with each clock pulse.
- The output is also serial, with the LSB first.
- The state of the FSM is the remainder of the division of the input number (so far) by five.

$$101 \overline{)\,1011111}\quad 0010011 \qquad 5 \overline{)\,95}\quad 19$$

NS = (2 * PS + input) mod 5
Output = 1 if (2 * PS + input) >= 5



Divide by Five Implementation



One can (if you wish) derive these equations from the truth table assuming the extra states result in "don't cares".

/D2 = /Q2 * x + /Q2 * Q0 + /Q1 */ x

D1 = Q0 * x + /Q1 * Q0 + Q2 * /x

Q_2	Q_1	Q_0	x	D_2	D_1	D_0	y
0	0	0	0	0	0	0	0
0	0	0	1	0	0	1	0
0	0	1	0	0	1	0	0
0	0	1	1	0	1	1	0
0	1	0	0	1	0	0	0
0	1	0	1	0	0	0	1
0	1	1	0	0	0	1	1
0	1	1	1	0	1	0	1
1	0	0	0	0	1	1	1
1	0	0	1	1	0	0	1

D0 = Q2 * /x + Q1 * Q0 * /x
+ /Q2 * /Q1 * x

y = Q2 + Q1 * Q0 + Q1 * x

Implementing FSMs in VHDL

- The hard part is to figure out what you want to do.
  - Implementation in VHDL is straightforward provided you copy from something that works, i.e., get the syntax right!
- The entity is straightforward and easy.
  - Don't forget that the semicolon goes AFTER the last parenthesis.
- The architecture has three goals.
  1. Implement the state register.
  2. Implement the combinational logic for the next state.
  3. Implement the combinational logic for the outputs.
- You must use a process for goal 1.
  - You may use concurrent statements for goals 2 and 3.
  - Use a process for 1, 2, and 3 (no other concurrent statements).
  - Use a process for 1 and 2 and concurrent statements for 3.
  - Use a process for 1 and 3 and concurrent statements for 2.
    - Remember that a process is a wrapper for sequential statements and is concurrent with other processes and/or concurrent statements.

State Assignments

- A key decision is how to encode the outputs.
  - Function of state only
  - Function of input and state
  - Registered – glitch free
  - State flip-flop – glitch free
    - Usually, it is more efficient to have an output flip-flop also be a state flip-flop.
- Let the VHDL compiler make the assignments.
  - This is the easiest thing to do.
    - Most of the time you don't care what the state assignments are –
    - Just that your state machine "works".
    - Almost always, states are assigned in counting order starting with 0.
    - Some compilers will do one-hot, zero-hot, gray code, etc.
- Make the state assignments manually.
  - Do this if you are looking for glitch-free outputs that are a function of state alone.
  - Assign the names of the state vectors as constants.
  - Use enumerated types to make state assignments.
    - This is often vendor specific.

```
attribute enum_encoding : string;
attribute enum_encoding of StateType : type is "111 010 110 011 100"
```

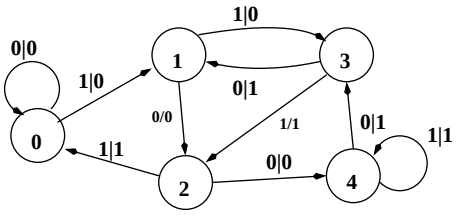
I1by5one.vhd

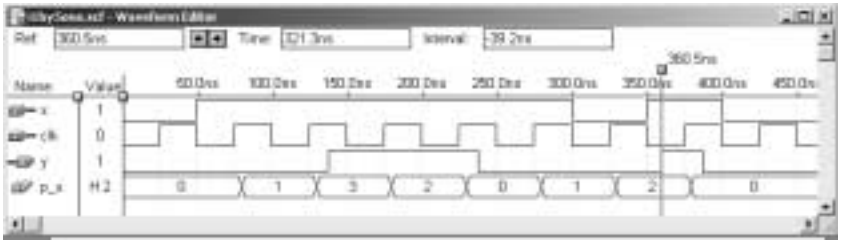
```
library ieee;
use ieee.std_logic_1164.all;
entity i1by5one is port (
  x, clk  : in std_logic;
  y       : out std_logic);
end i1by5one;
architecture state_machine of i1by5one is
  type StateType is (state0, state1,
        state2, state3, state4);
  signal p_s : StateType;
begin
  fsm:process(clk, p_s, x)
  begin
    if rising_edge(clk) then
      case p_s is
        when state0 => if x = '1'
                  then p_s <= state1;
                  else p_s <= state0;
                  end if;
        when state1 => if x = '1'
                  then p_s <= state3;
                  else p_s <= state2;
                  end if;
        when state2 => if x = '1'
                  then p_s <= state0;
                  else p_s <= state4;
                  end if;
        when state3 => if x = '1'
                  then p_s <= state2;
                  else p_s <= state1;
                  end if;
        when state4 => if x = '1'
                  then p_s <= state4;
                  else p_s <= state3;
                  end if;
        when others => p_s <= state0;
      end case;
    end if;
    if (p_s = state4) then y <= '1';
    elsif (p_s = state3) then y <= '1';
    elsif (p_s = state2 and x = '1') then
      y <= '1';
    else y <= '0';
    end if;
  end process fsm;
end architecture state_machine;
```


I1by5.vhd Simulation

$$101 \overline{)\ 1011111}\quad 0010011 \qquad 5 \overline{)\ 95}\quad 19$$

NS = (2 * PS + input) mod 5
Output = 1 if (2 * PS + input) >= 5





I1by5sv.vhd

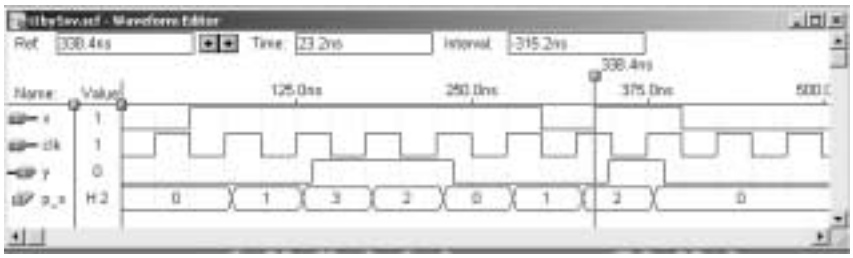
```
entity i1by5sv is port (
  x, clk  : in std_logic;
  y       : out std_logic);
end i1by5sv;
architecture state_machine of i1by5sv is
  signal p_s, n_s : std_logic_vector(2 downto 0);
  signal p_sx : std_logic_vector(3 downto 0);
  constant state0 : std_logic_vector(2 downto 0)
     := "000";
  constant state1 : std_logic_vector(2 downto 0)
     := "001";
  constant state2 : std_logic_vector(2 downto 0)
     := "010";
  constant state3 : std_logic_vector(2 downto 0)
     := "011";
  constant state4 : std_logic_vector(2 downto 0)
     := "100";
begin
  state_clocked:process(clk) -- register
  begin
    if rising_edge(clk) then p_s <= n_s;
    end if;
  end process state_clocked;
```

```
-- combinational output specification
  y <= '1' when ((p_s = state4) or
                 (p_s = state3) or
                 ((p_s = state2) and (x = '1')))
          else '0';
-- combinational next state specification
  p_sx <= p_s & x;
  with p_sx select
  n_s <= state0 when "0000",
         state1 when "0001",
         state2 when "0010",
         state3 when "0011",
         state4 when "0100",
         state0 when "0101",
         state1 when "0110",
         state2 when "0111",
         state3 when "1000",
         state4 when "1001",
         state0 when others;
end architecture state_machine;
```


I1by5sv.vhd Simulation

```
  state_clocked:process(clk) -- register
  begin
    if rising_edge(clk) then p_s <= n_s;
    end if;
  end process state_clocked;
-- combinational output specification
  y <= '1' when ((p_s = state4) or
                 (p_s = state3) or
                 ((p_s = state2) and (x = '1')))
          else '0';
```

```
-- combinational next state specification
  p_sx <= p_s & x;
  with p_sx select
  n_s <= state0 when "0000",
         state1 when "0001",
         state2 when "0010",
         state3 when "0011",
         state4 when "0100",
         state0 when "0101",
         state1 when "0110",
         state2 when "0111",
         state3 when "1000",
         state4 when "1001",
         state0 when others;
```



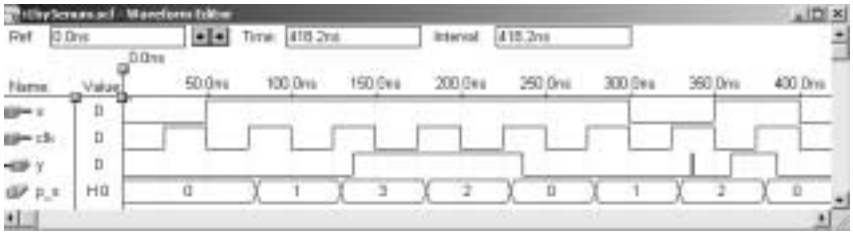
I1by5enum.vhd

```
library ieee;
use ieee.std_logic_1164.all;
entity i1by5enum is port (
 x, clk  : in std_logic;
 y       : out std_logic);
end i1by5enum;
architecture state_machine of i1by5enum is
 type StateType is (state0, state1,
        state2, state3, state4);
 attribute enum_encoding : string;
 attribute enum_encoding of StateType :
       type is "000 001 010 011 100";
 signal p_s, n_s : StateType;
begin
 fsm:process(p_s, x) -- combinational
 begin  -- case
  case p_s is
    when state0 => if x = '1' then
                n_s <= state1;
                y <= '0';
               else
                n_s <= state0;
                y <= '0';
               end if;
    when state1 => if (x = '1') then
                n_s <= state3;
                y <= '0';
               else
                n_s <= state2;
                y <= '0';
               end if;
    when state2 => if (x = '1') then
                n_s <= state0;
                y <= '1';
               else  n_s <= state4;
                y <= '0';
               end if;
    when state3 => if (x = '1') then
                n_s <= state2;
                y <= '1';
               else n_s <= state1;
                y <= '1';
               end if;
    when state4 => if (x = '1') then
                n_s <= state4;
                y <= '1';
               else
                n_s <= state3;
                y <= '1';
               end if;
    when others => n_s <= state0;
  end case;
 end process fsm;
 state_clocked:process(clk)
 begin
  if rising_edge(clk) then p_s <= n_s;
  end if;
 end process state_clocked;
end architecture state_machine;
```


Simulation of i1by5enum.vhd

attribute enum_encoding : string;

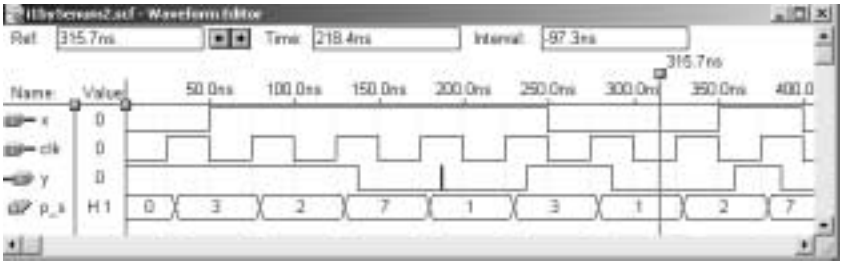
attribute enum_encoding of StateType : type is “000 001 010 011 100";



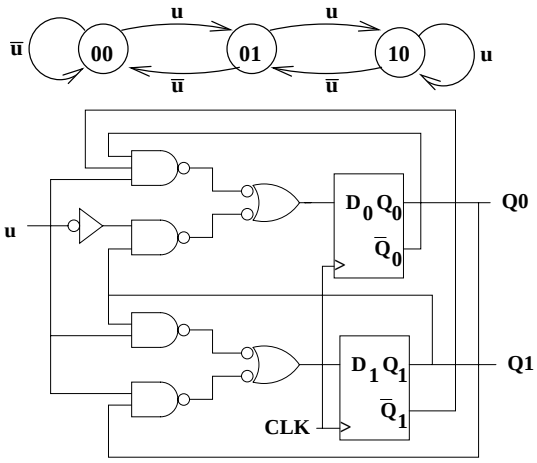
Simulation of i1by5enum2.vhd

attribute enum_encoding : string;

attribute enum_encoding of StateType : type is “111 001 010 011 100";



A Simple FSM



u \ Q1 Q0	00	01	11	10
0	0	0	X	0
1	0	1	X	1

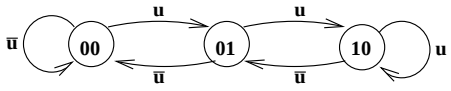
D1 = u*Q0 + u*Q1

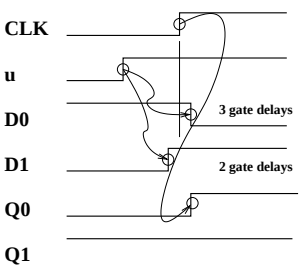
u \ Q1 Q0	00	01	11	10
0	0	0	X	1
1	1	0	X	0

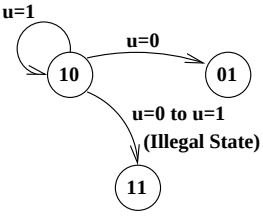
D0 = u*/Q0*/Q1 + /u*Q1

Problem Transition

Consider the transition from state 10 (2) to state 01 (1), if u changes from 0 to 1 close to the clock edge:







Q1 Q0

u \ Q1Q0	00	01	11	10
0	0	0	X	0
1	0	1	X	1

D1 = u*Q0 + u*Q1

Q1 Q0

u \ Q1Q0	00	01	11	10
0	0	0	X	1
1	1	0	X	0

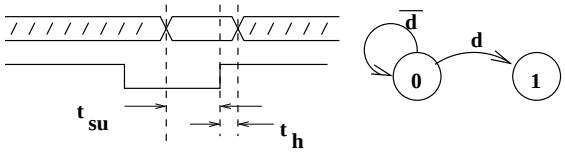
D0 = u*/Q0*/Q1 + /u*Q1

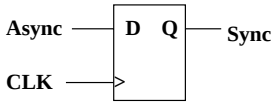
Most of the time, this circuit will work just fine. It makes a mistake and enters an illegal state ONLY when the input transition is close to a clock edge.

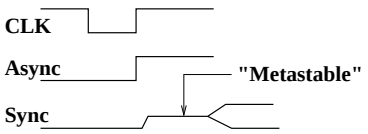
Important Design Rule

DESIGN RULE:

1. **Synchronize ALL external signals.**
2. **Any asynchronous input must affect ONLY ONE flip−flop.**



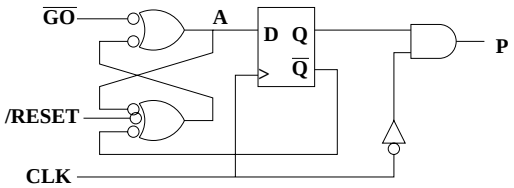




Any combinational logic with "Sync" as an input will be "glitchy" until after the metastable state has expired. In particular, do NOT use "Sync" as a CLK input.

VHDL for a Short Pulse Catcher

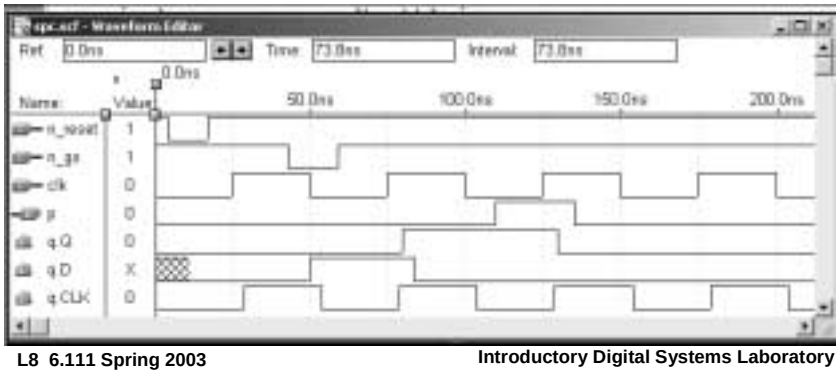
```
library ieee;
use ieee.std_logic_1164.all;
entity spc is
   port(n_go, clk, n_reset : in std_logic;
        p : out std_logic);
end spc;
-- purpose: catch a short pulse
architecture behavioral of spc is
   signal a, n_a, q, n_q, n_clk: std_logic;
  begin  -- behavioral
   a <= (not n_go) or (not n_a);
   n_a <= (not a) or (not n_q) or (not n_reset);
   n_q <= (not q);
   p <= q and (not clk);
ff: process(clk)
   begin
      if rising_edge(clk) then
         q <= a;
      end if;
   end process ff;
end behavioral;
```



Simulation of a Short Pulse Catcher

```
architecture behavioral of spc is
   signal a, n_a, q, n_q, n_clk: std_logic;
  begin  -- behavioral
   a <= (not n_go) or (not n_a);
   n_a <= (not a) or (not n_q) or (not n_reset);
   n_q <= (not q);
   p <= q and (not clk);
```

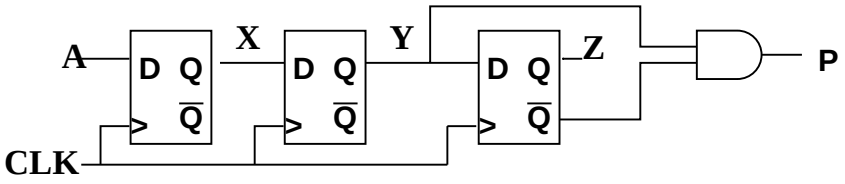
```
ff: process(clk)
   begin
      if rising_edge(clk) then
         q <= a;
      end if;
   end process ff;
end behavioral;
```



Level to Pulse

```
library ieee;
use ieee.std_logic_1164.all;
entity pform is
  port(A, CLK: in std_logic;
       X, Y, Z: buffer std_logic;
       P: out std_logic);
end pform;
-- purpose: turn a level into a
-- finite width pulse
```

```
architecture behavioral of pform is
begin  -- behavioral
ff: process(CLK)
begin
  if rising_edge(CLK) then
    X <= A;
    Y <= X;
    Z <= Y;
  end if;
end process ff;
P <= (Y AND (not Z));
end behavioral;
```



Simulation of Level to Pulse

```
architecture behavioral of pform is
begin  -- behavioral
ff: process(CLK)
begin
  if rising_edge(CLK) then
    X <= A;
    Y <= X;
    Z <= Y;
  end if;
end process ff;
P <= (Y AND (not Z));
end behavioral;
```