RECRLab
Reconfigurable Computing Research Laboratory

DIGITAL LOGIC DESIGN
VHDL Coding for FPGAs
Unit 5

✓ *SEQUENTIAL CIRCUITS*

- Asynchronous sequential circuits: Latches
- Synchronous circuits: flip flops, counters, registers.
- Testbench: Generating clock stimulus

Daniel Llamocca

OAKLAND UNIVERSITY

✓ COMBINATIONAL CIRCUITS

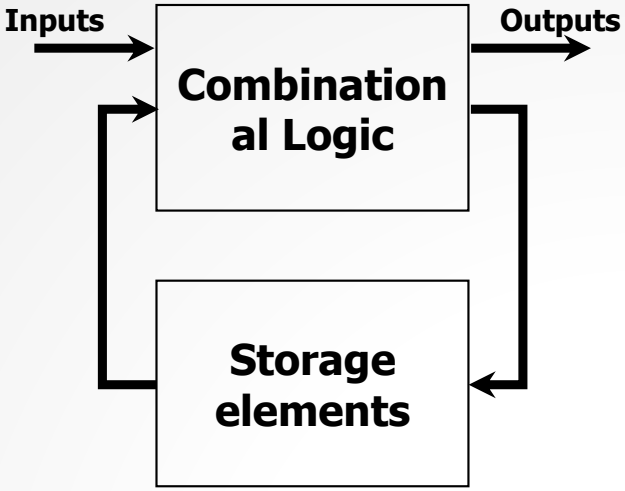
- In combinational circuits, the output only depends upon the present input values.

- There exist another class of logic circuits whose outputs not only depend on the present input values but also on the past values of inputs, outputs, and/or internal signal. These circuits include storage elements to store those previous values.

- The content of those storage elements represents the *circuit state*. When the circuit inputs change, it can be that the circuit stays in certain state or changes to a different one. Over time, the circuit goes through a sequence of states as a result of a change in the inputs. The circuits with this behavior are called *sequential circuits*.

COMBINATIONAL CIRCUIT

SEQUENTIAL CIRCUIT



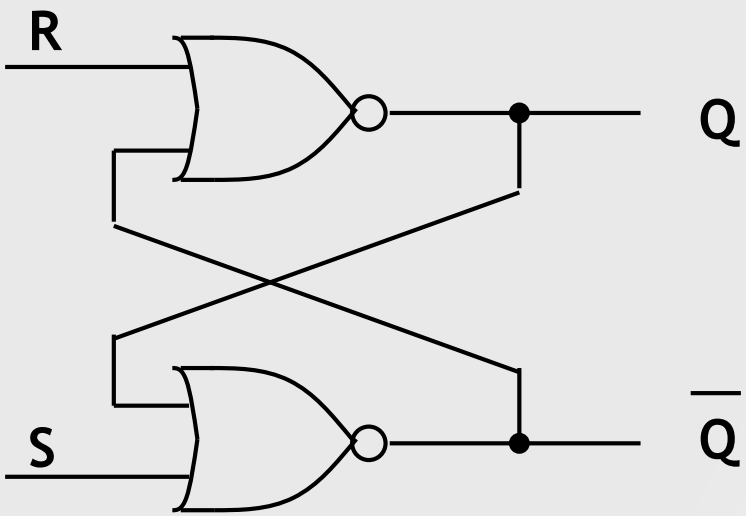
✓SEQUENTIAL CIRCUITS

- Combinational circuits can be described with concurrent statements or behavioral statements.
- Sequential circuits are best described with sequential statements.
- Sequential circuits can either be asynchronous or synchronous. In VHDL, they are described with asynchronous/synchronous processes.
  - ✓ Basic asynchronous sequential circuits: Latches
  - ✓ Basic synchronous sequential circuits: flip flops, counters, and registers.

- Here, we cover the VHDL description of typical asynchronous and synchronous sequential circuits.

✓ ASYNCHRONOUS PROCESS:

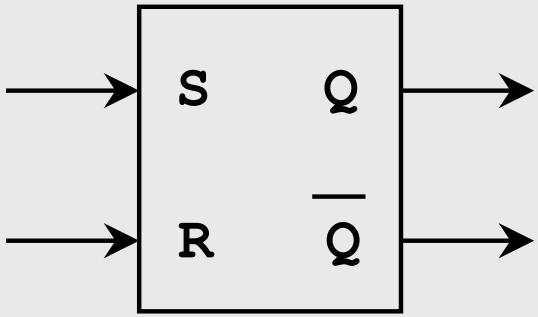
- **SR Latch**
- An SR Latch based on NOR gates:

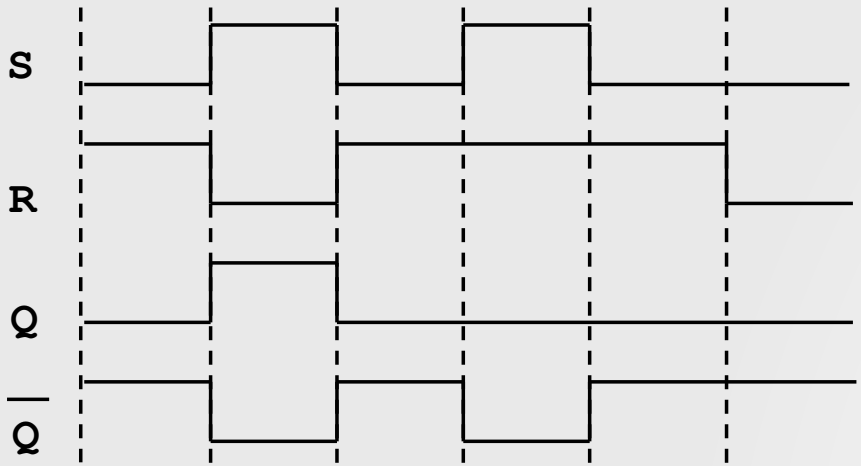


S	R	Q_{t+1}	$\overline{Q}_{t+1}$	
0	0	Q_t	$\overline{Q_t}$	
0	1	0	1	
1	0	1	0	
1	1	0	0	restricted

- According to its truth table, the output can be assigned to either '0' or '1'. This circuit state ('0' or '1') is stored in the circuit when S=R='0'.
- FPGAs usually have trouble implementing these circuits as FPGAs are synchronous circuits.

▪ SR Latch: VHDL code

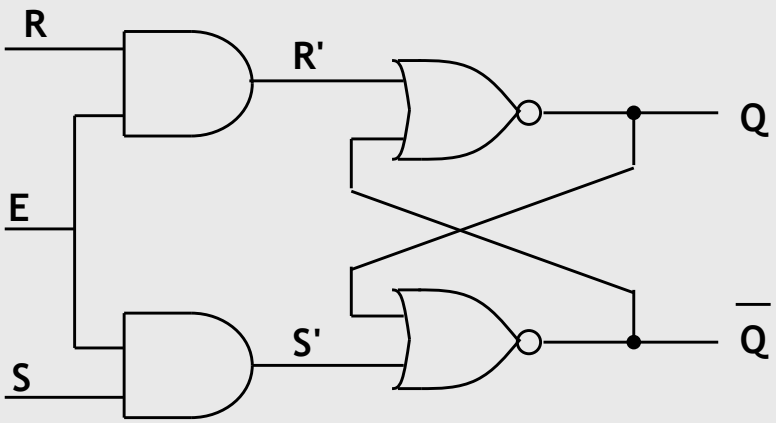





```
library ieee;
use ieee.std_logic_1164.all;

entity latch_sr is
 port ( s,r: in std_logic;
        q, qn: out std_logic);
end latch_sr;

architecture bhv of latch_sr is
  signal qt,qnt: std_logic;
begin
  process (s,r)
  begin
    if s='1' and r='0' then
       qt<='1'; qnt<='0';
    elsif s='0' and r='1' then
       qt<='0'; qnt <='1';
    elsif s='1' and r='1' then
       qt<='0'; qnt <= '0';
    end if;
  end process;
  -- we don't specify what happens
  -- if s=r='0' --> q, qn kept their
  -- previous values
  q <= qt; qn <= qnt;
end bhv;
```


Daniel Llamocca

▪ SR Latch with enable

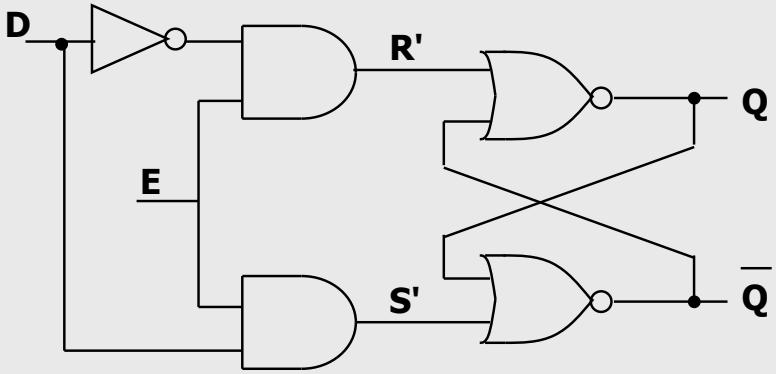


E	S	R	Q_{t+1}	$\overline{Q_{t+1}}$
0	x	x	Q_t	$\overline{Q_t}$
1	0	0	Q_t	$\overline{Q_t}$
1	0	1	0	1
1	1	0	1	0
1	1	1	0	0

- Note: If E = '0', the previous output is kept.

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

entity latch_sr_E is
 port ( s,r, E: in std_logic;
        q, qn: out std_logic);
end latch_sr_E;

architecture bhv of latch_sr_E is
  signal qt,qnt: std_logic;
begin
  process (s,r,E)
  begin
    if E = '1' then
      if s='1' and r='0' then
         qt<='1'; qnt<='0';
      elsif s='0' and r='1' then
         qt<='0'; qnt <='1';
      elsif s='1' and r='1' then
         qt<='0'; qnt <= '0';
      end if;
    end if;
  end process;
  q <= qt; qn <= qnt;
end bhv;
```

▪ D Latch with enable

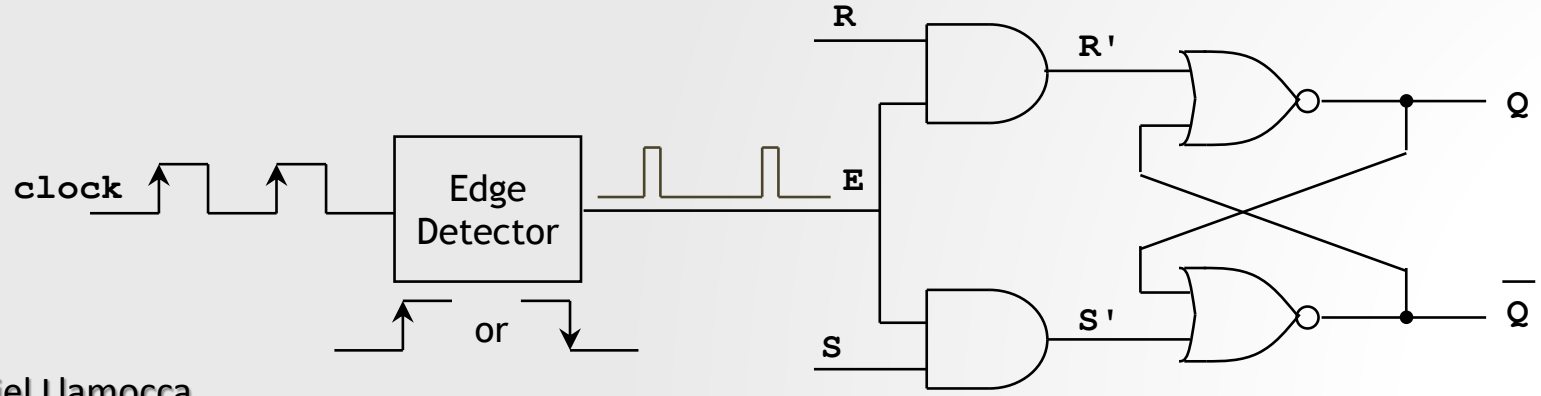


E	D	Q_{t+1}
0	x	Q_t
1	0	0
1	1	1

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

entity latch_D is
 port ( D, E: in std_logic;
        q, qn: out std_logic);
end latch_D;

architecture bhv of latch_D is
  signal qt: std_logic;
begin
  process (D,E)
  begin
    if E = '1' then
       qt <= d;
    end if;
  end process;
  q <= qt; qn <= not(qt);
end bhv;
```

✓ SYNCHRONOUS PROCESSES

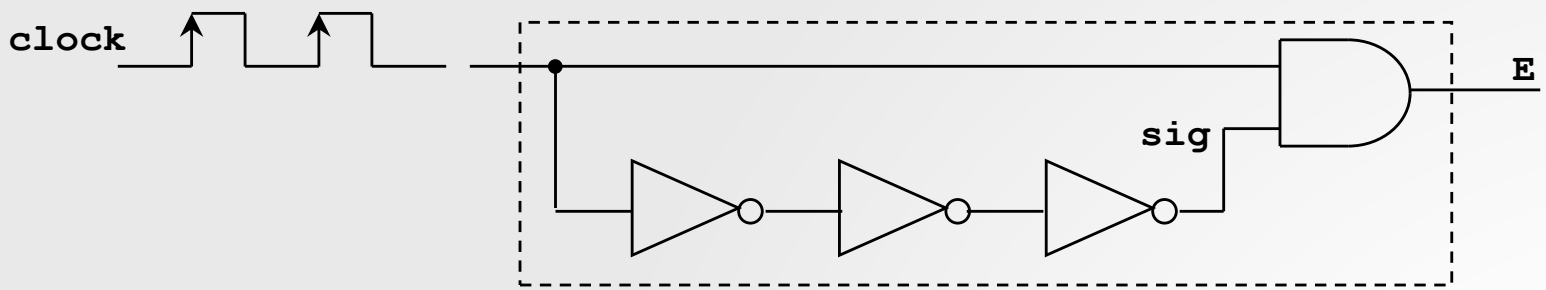
- **Flip Flops**
- Unlike a Latch, a flip flop only changes its outputs on the edge (rising or falling) of a signal called clock. A clock signal is a square wave with a fixed frequency.
- To detect a rising or falling edge, flip flops include an edge detector circuit. Input: a clock signal, Output: short duration pulses during the rising (or falling) clock edges. These pulses are then connected to the enable input in a Latch.
- For example, an SR flip flop is made out of: a SR Latch with an edge detector circuit. The edge detector generates enable signals during the during the rising (or falling) clock edges.

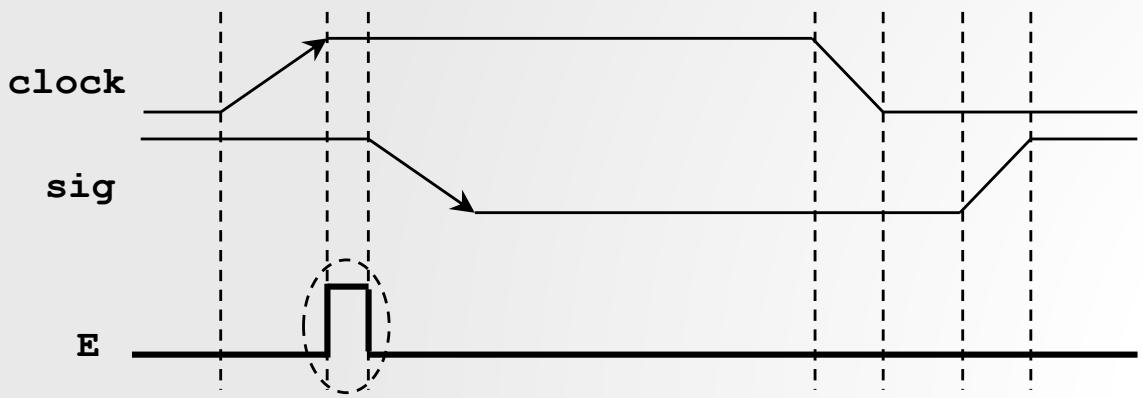


✓ SYNCHRONOUS PROCESSES

Flip Flops:

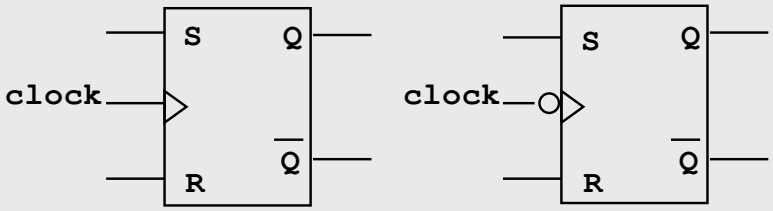
- The edge detector circuit generates E='1' during the edge (rising or falling). We will work with circuits activated by either rising or falling edge. We will not work with circuits activated by both edges.
- An example of a circuit that detects a rising edge is shown below. The redundant NOT gates cause a delay that allows a pulse to be generated during a rising edge (or positive edge).





✓ SYNCHRONOUS PROCESSES

- **SR Flip Flop**



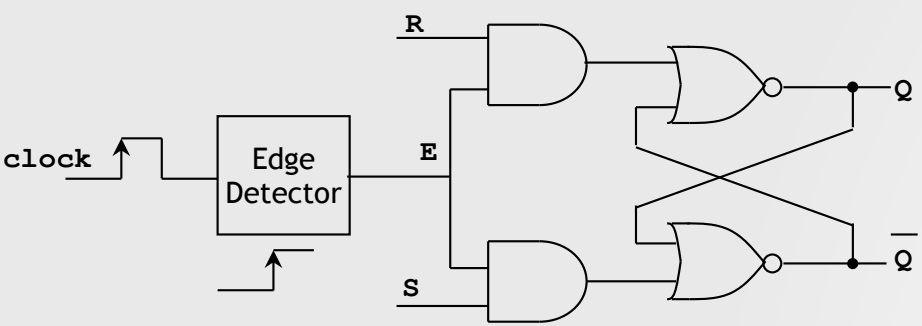
Positive edge-triggered

Negative edge-triggered

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

entity ff_sr is
 port ( s,r, clock: in std_logic;
        q, qn: out std_logic);
end ff_sr;

architecture bhv of ff_sr is
  signal qt,qnt: std_logic;
begin
  process (s,r,clock)
  begin
    if (clock'event and clock='1') then
    --if (clock'event and clock='0') then
      if s='1' and r='0' then
        qt<='1'; qnt<='0';
      elsif s='0' and r='1' then
        qt<='0'; qnt <='1';
      elsif s='1' and r='1' then
        qt<='0'; qnt <= '0';
      end if;
    end if;
  end process;
  q <= qt; qn <= qnt;
end bhv;
```

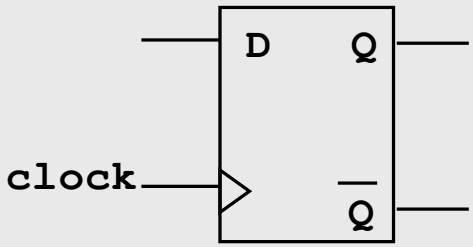
Positive-edge triggered

Negative-edge triggered



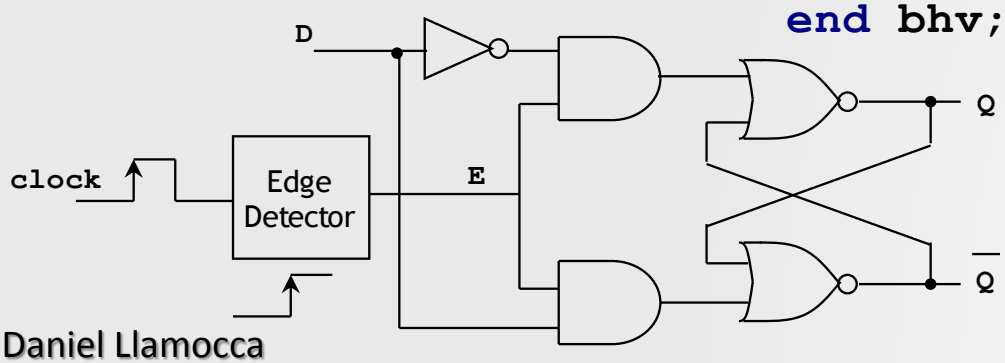
✓ SYNCHRONOUS PROCESSES

- **D Flip Flop**



clock	D	Q_{t+1}
↑	0	0
↑	1	1

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

entity ff_d is
 port ( d, clock: in std_logic;
        q, qn: out std_logic);
end ff_d;

architecture bhv of ff_d is
  signal qt,qnt: std_logic;
begin
  process (d,clock)
  begin
    if (clock'event and clock='1') then
       qt<=d;
    end if;
  end process;
  q <= qt; qn <= not(qt);
end bhv;
```

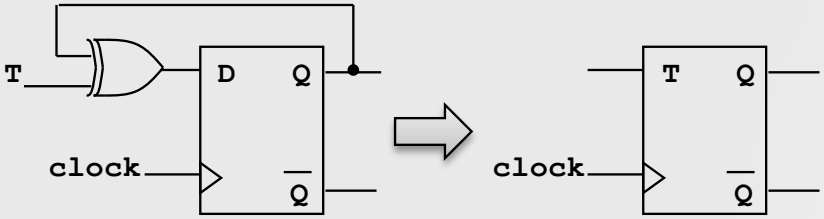


* Note: Signal `qt` is needed. In VHDL, we cannot feedback an output (in this case `q`) as an input of the circuit

✓ SYNCHRONOUS PROCESSES

- **T Flip Flop**

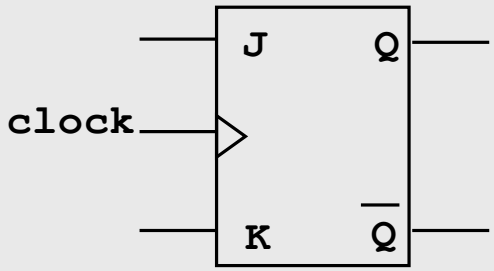
clock	T	Q_{t+1}
↑	0	Q_t
↑	1	$\overline{Q_t}$




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

entity ff_t is
 port ( t, clock: in std_logic;
        q, qn: out std_logic);
end ff_t;

architecture bhv of ff_t is
  signal qt,qnt: std_logic;
begin
  process (t,clock)
  begin
    if (clock'event and clock='1') then
      if t = '1' then
        qt <= not(qt);
      end if;
    end if;
  end process;
  q <= qt; qn <= not(qt);
end bhv;
```

✓ SYNCHRONOUS PROCESSES

- **JK Flip Flop**



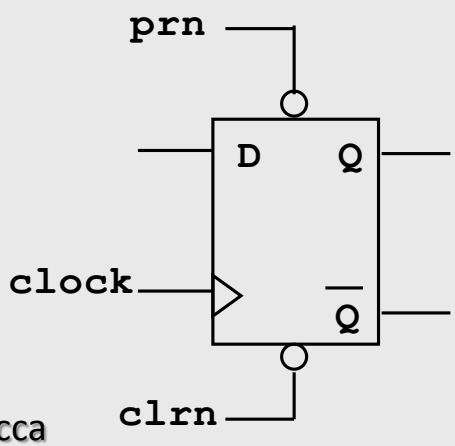
clock	J	K	Q_{t+1}
↑	0	0	Q_t
↑	0	1	0
↑	1	0	1
↑	1	1	$\overline{Q_t}$

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

entity ff_jk is
 port ( s,r, clock: in std_logic;
        q, qn: out std_logic);
end ff_jk;

architecture bhv of ff_jk is
  signal qt,qnt: std_logic;
begin
  process (j,k,clock)
  begin
    if (clock'event and clock='1') then
      if j='1' and k='1' then
         qt<= not(qt);
      elsif j='1' and k='0' then
         qt<='0';
      elsif j='0' and k='1' then
         qt<='1';
      end if;
    end if;
  end process;
  q <= qt; qn <= qnt;
end bhv;
```

✓ SYNCHRONOUS PROCESSES

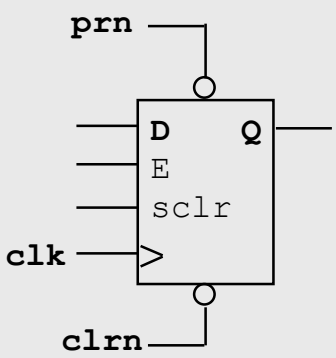
- **D Flip Flop with asynchronous inputs: `clrn, prn`**
- clrn = '0' ➔ q = '0'
  prn = '0' ➔ q = '1'
- These inputs force the outputs to a value immediately.
- This is a useful feature if we want to initialize the circuit with no regards to the rising (or falling) clock edge

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

entity ff_dp is
 port ( d,clrn,prn,clock: in std_logic;
        q, qn: out std_logic);
end ff_dp;

architecture bhv of ff_dp is
  signal qt,qnt: std_logic;
begin
  process (d,clrn,prn,clock)
  begin
    if clrn = '0' then
       qt <= '0';
    elsif prn = '0' then
       qt <= '1';
    elsif (clock'event and clock='1') then
       qt <= d;
    end if;
  end process;
  q <= qt; qn <= not(qt);
end bhv;
```



✓ SYNCHRONOUS PROCESSES

- **D Flip Flop with enable and synchronous clear**
- This is a complete design that includes asynchronous inputs (`prn, clrn`) and synchronous inputs (`E, sclr, D`).




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

entity dffes is
 port ( d,clrn,prn,clk,E,sclr: in std_logic;
        q: out std_logic);
end dffes;

architecture bhv of dffes is
begin
  process (d,clrn,prn,E,clock)
  begin
    if clrn = '0' then     q <= '0';
    elsif prn = '0' then   q <= '1';
    elsif (clock'event and clock='1') then
       if E = '1' then
          if sclr = '1' then
              q <= '0';
          else
              q <= d;
          end if;
       end if;
    end if;
  end process;
end bhv;
```

✓ SYNCHRONOUS PROCESSES

- **Registers**
- These are sequential circuits that store the values of signals. There exist many register types: registers to handle interruptions in a PC, microprocessor registers, pipelining registers, etc.

- n-bit Register: Storage element that can hold 'n' bits. It is a collection of 'n' D-type flip flops

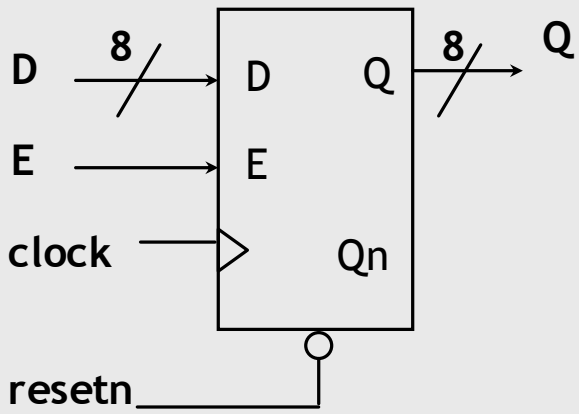
- Register types:
  - Simple Register (with/without enable)
  - Shift register (with/without enable)

    Serial input, parallel output

    Serial input, serial output
  - Parallel access shift register (parallel/serial input, parallel/serial output).

✓ PARALLEL LOAD, PARALLEL OUTPUT

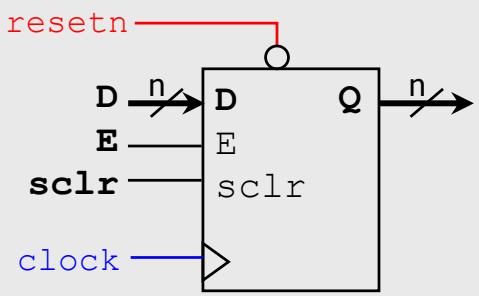
- **8-bit register with enable and asynchronous reset**




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

entity reg8 is
 port (clock, resetn, E: in std_logic;
  D: in std_logic_vector (7 downto 0);
  Q: out std_logic_vector (7 downto 0));
end reg8;

architecture bhv of reg8 is
begin
 process (resetn,E,clock)
 begin
  if resetn = '0' then
   Q <= (others => '0');
  elsif (clock'event and clock = '1') then
   if E = '1' then
     Q <= D;
   end if;
  end if;
 end process;
end bhv;
```

✓ PARALLEL LOAD, PARALLEL OUTPUT

- **n-bit register with enable, sclr and asynchronous reset**
- sclr: only considered if E = '1'




```
library ieee;
use ieee.std_logic_1164.all;
entity my_rege is
   generic (N: INTEGER:= 4);
   port ( clock, resetn: in std_logic;
          E, sclr: in std_logic;
          D: in std_logic_vector (N-1 downto 0);
          Q: out std_logic_vector (N-1 downto 0));
end my_rege;

architecture Behavioral of my_rege is
   signal Qt: std_logic_vector (N-1 downto 0);
begin
   process (resetn, clock)
   begin
      if resetn = '0' then Qt <= (others => '0');
      elsif (clock'event and clock = '1') then
         if E = '1' then
            if sclr='1' then Qt <= (others =>'0');
            else Qt <= D;
            end if;
         end if;
      end if;
   end process;
   Q <= Qt;
end Behavioral;
```

PARAMETRIC CODE:

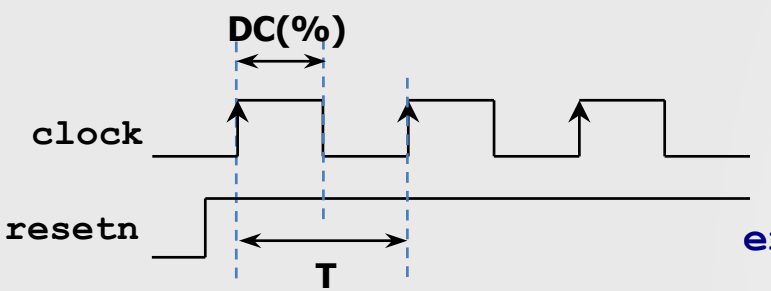
- **my_rege.zip:** my_rege.vhd, tb_my_rege.vhd

Daniel Llamocca

✓ TESTBENCH

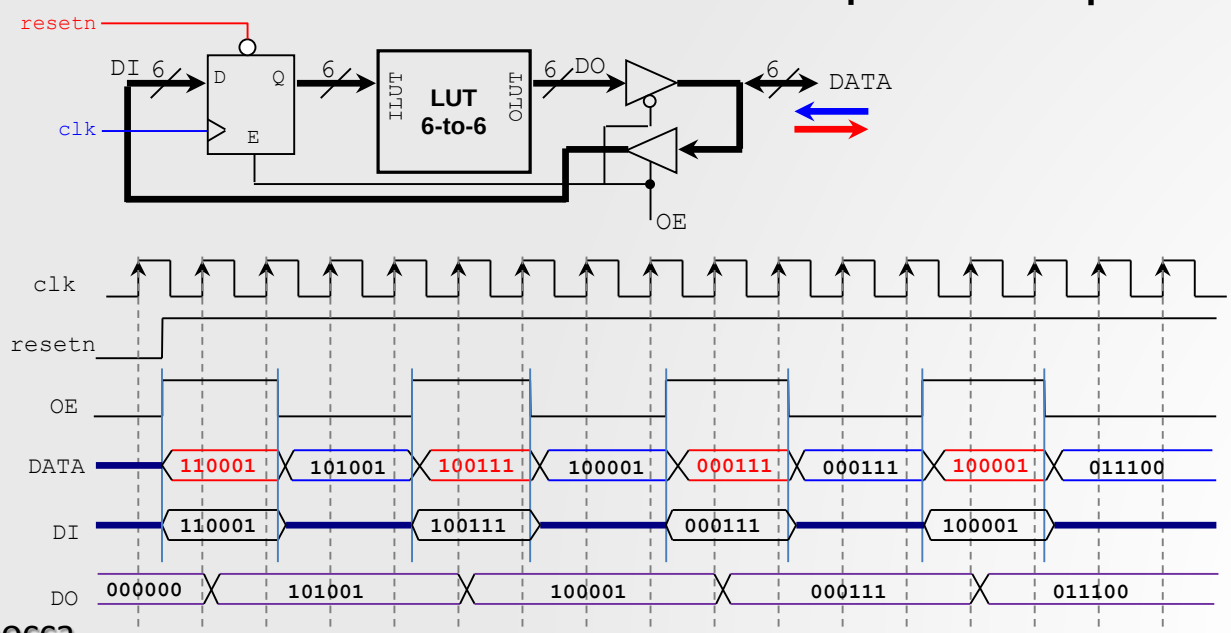
- **Generating clock stimulus**
- A clock signal is an square wave with a fixed frequency. The Duty Cycle is usually 50%.
- The example shows a code snippet of the testbench for my_reg3.vhd: An independent process is needed just to create the clock signal

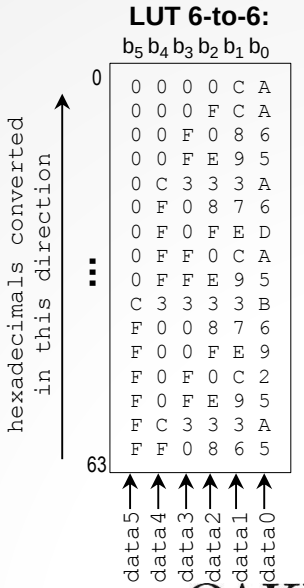
```
...
architecture bhv of tb_my_rege is
  ...
  ...
  constant T: time:= 10 ns;
  constant DC: real:= 0.5;

begin
  uut: my_rege port map (clock=>clock,E=>E,
       resetn=>resetn,sclr=>sclr,D=>D,Q=>Q);

  clock_process: process
  begin
    clock <='0'; wait for (T - T*DC);
    clock <='1'; wait for T*DC;
  end process;

  stim_process: process
  begin
    wait for 100 ns;
    resetn <= '1'; wait for 2*T;
    --
  end process;
end bhv;
```



✓ REGISTER: Example

- **3-state buffers and 6-to-6 LUT**
- LUT6-to-6: built by grouping six LUT6-to-1 in parallel. LUT6-to-1: made out of LUT4-to-1.
- Note that the port DATA can be input or output at different times. In VHDL, we use the **INOUT** data type to specify this.
- LUT6-to-6 contents: Any function of 6 input bits and 6 output bits can be pre-computed and stored in the LUT6-to-6. In the example, the function is $OLUT = \lceil ILUT^{0.95} \rceil$





✓ REGISTER: Example

- **3-state buffers and 6-to-6 LUT**
- Data: 64 rows of 6 bits. Or 6 columns of 64 bits. The figure shows the entity VHDL portion of the system.

```
entity sysLUT6to6 is
  generic( data5: std_logic_vector(63 downto 0):=x"ffffffc000000000";
           data4: std_logic_vector(63 downto 0):=x"fc00003ffffc0000";
           data3: std_logic_vector(63 downto 0):=x"03ff003ff003ff00";
           data2: std_logic_vector(63 downto 0):=x"83e0f83e0f83e0f0";
           data1: std_logic_vector(63 downto 0):=x"639ce739ce7398cc";
           data0: std_logic_vector(63 downto 0):=x"5a5296b5ad6a56aa");
   port (clk, resetn, OE: in std_logic;
         data: inout std_logic_vector (5 downto 0));
end sysLUT6to6;
```

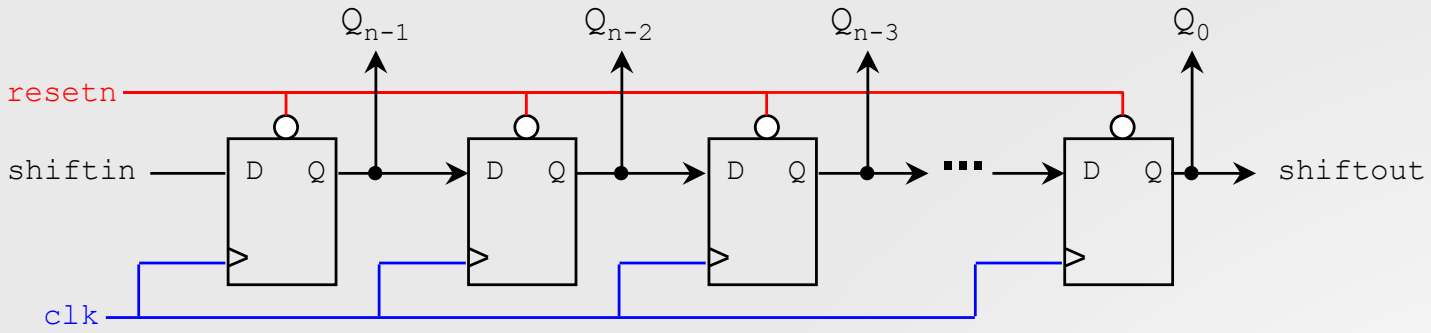
- Testbench: The code shows that when DATA is output (OE=0), it MUST be assigned the value 'Z'.

```
...
resetn<='0'; DATA<="ZZZZZZ";wait for 100 ns;
resetn<='1'; wait for T;
OE<='1'; DATA<="110001";wait for 2*T;
OE<='0'; DATA<="ZZZZZZ";wait for 2*T;
OE<='1'; DATA<="100111";wait for 2*T;
OE<='0'; DATA<="ZZZZZZ";wait for 2*T;
```

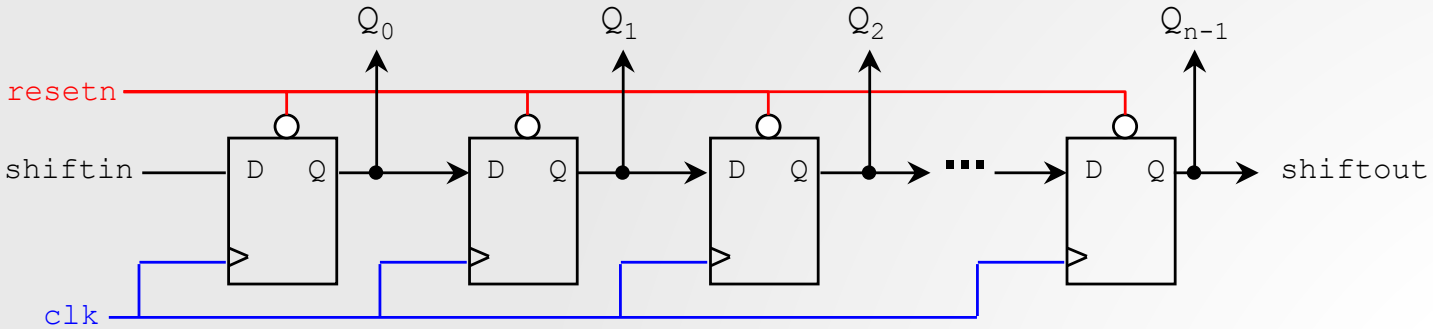
- **sysLUT6to6.zip:**
  sysLUT6to6.vhd,
  my6to6LUT.vhd,
  my6to1LUT.vhd,
  my5to1LUT.vhd,
  my4to1LUT.vhd,
  my_rege.vhd,
  tb_sysLUT6to6.vhd

✓ SHIFT REGISTER: Serial Input, Serial/ Parallel Output

- **n-bit right shift register:**



- **n-bit left shift register:**

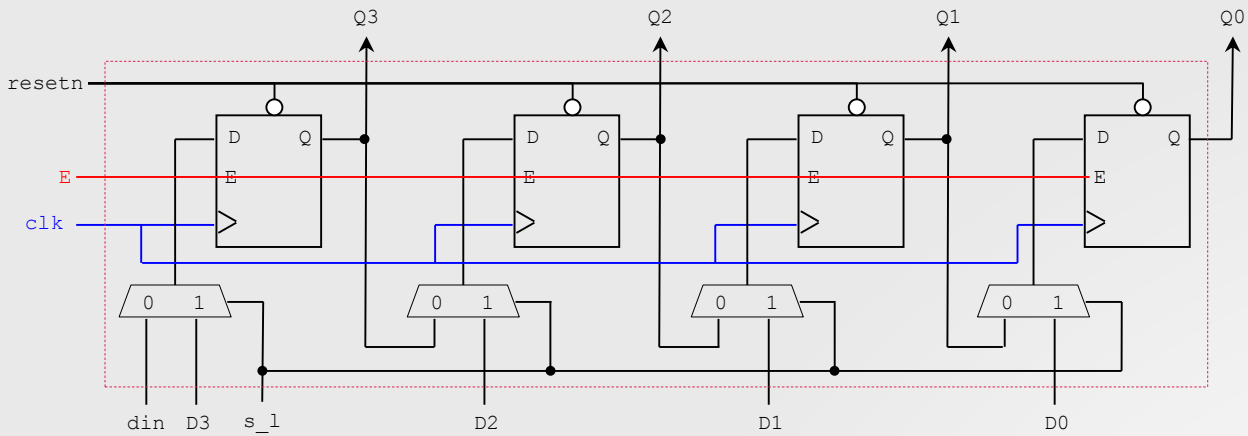


PARAMETRIC CODE:

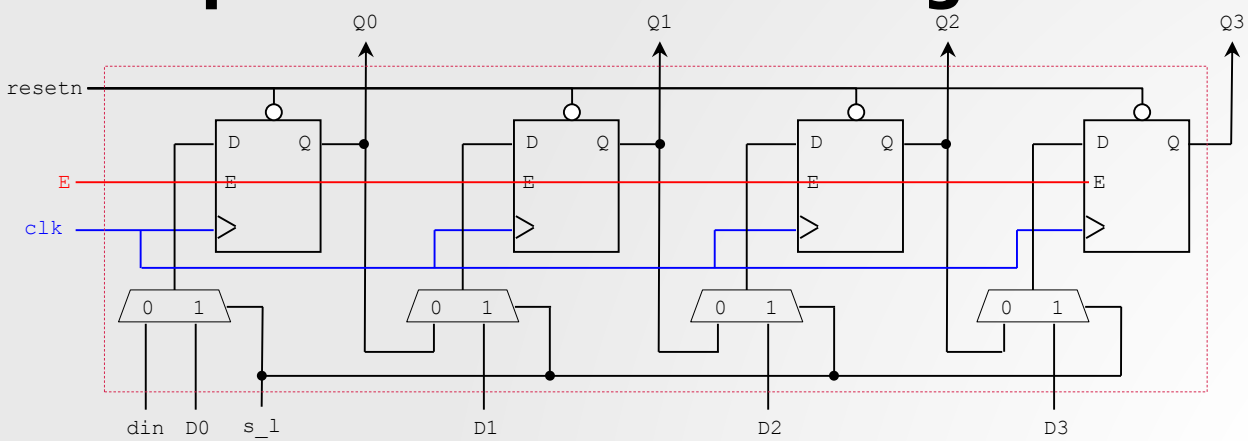
➢ **`my_shiftreg.zip`:** Generic `n`-bit left/right Shift Register
`my_shiftreg.vhd`,
`tb_my_shiftreg.vhd`

✓ PARALLEL ACCESS SHIFT REGISTER

- **4-bit right parallel access shift register with enable:**



- **4-bit left parallel Access shift register with enable:**

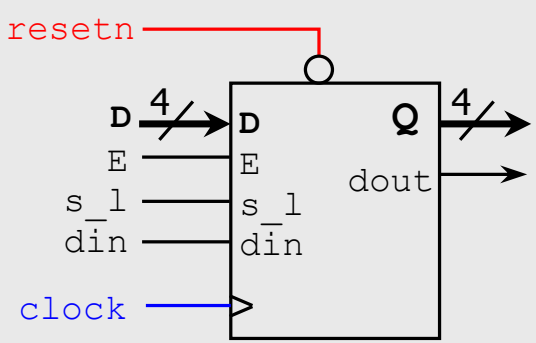


PARAMETRIC CODE:

➢ **`my_pashiftreg.zip`:** Generic `n`-bit left/right Parallel Access Shift Register
`my_pashiftreg.vhd`, `tb_my_pashiftreg.vhd`

✓ PARALLEL ACCESS SHIFT REGISTER

- **Parallel/serial load**
  Parallel/serial output
  Shift to the **right**, 4 bits
- `s_l=1` -> Paralled load
  `s_l=0` -> Serial load
- 'din': serial input
- 'D': parallel input
- 'dout': serial output
- 'Q': parallel output

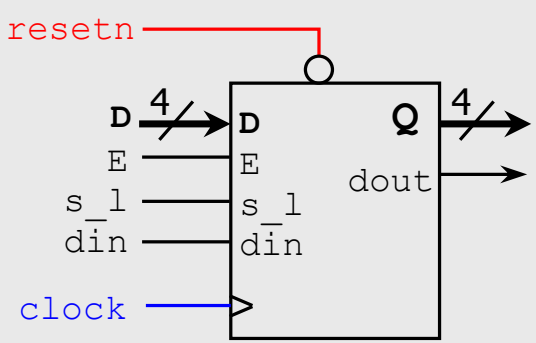



```
library ieee;
use ieee.std_logic_1164.all;
entity pashreg4_right is
   port (clock, resetn: in std_logic;
         E, s_l, din: in std_logic;
         dout: out std_logic;
         D: in std_logic_vector (3 downto 0);
         Q: out std_logic_vector (3 downto 0));
end pashreg4_right;

architecture bhv of pashreg4_right is
   signal Qt: std_logic_vector (3 downto 0);
begin
   process (resetn, clock, s_l, E)
   begin
      if resetn = '0' then Qt <= "0000";
      elsif (clock'event and clock = '1') then
         if E = '1' then
            if s_l='1' then Qt <= D;
            else
               Qt(0) <= Qt(1); Qt(1) <= Qt(2);
               Qt(2) <= Qt(3); Qt(3) <= din;
            end if;
         end if;
      end if;
   end process;
   Q <= Qt; dout <= Qt(0);
end bhv;
```

Daniel Llamocca

✓ PARALLEL ACCESS SHIFT REGISTER

- **Parallel/serial load**
  Parallel/serial output
  Shift to the **right**, 4 bits
- **Use of VHDL for loop**
- `s_l=1` -> Parallel load
  `s_l=0` -> Serial load
- 'din': serial input
- 'D': parallel input
- 'dout': serial output
- 'Q': parallel output



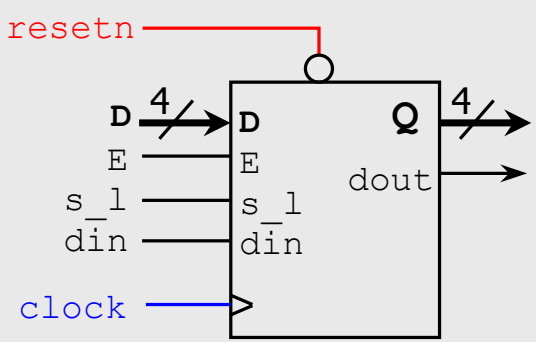

```
library ieee;
use ieee.std_logic_1164.all;
entity pashreg4_right is
   port (clock, resetn: in std_logic;
         E, s_l, din: in std_logic;
         dout: out std_logic;
         D: in std_logic_vector (3 downto 0);
         Q: out std_logic_vector (3 downto 0));
end pashreg4_right;

architecture bhv of pashreg4_right is
   signal Qt: std_logic_vector (3 downto 0);
begin
   process (resetn, clock, s_l, E)
   begin
      if resetn = '0' then Qt <= "0000";
      elsif (clock'event and clock = '1') then
         if E = '1' then
            if s_l='1' then Qt <= D;
            else
               gg: for i in 0 to 2 loop
                      Qt(i) <= Qt(i+1);
                   end loop;
                   Qt(3) <= din;
            end if;
         end if;
      end if;
   end process;
   Q <= Qt; dout <= Qt(0);
end bhv;
```

Daniel Llamocca

✓ PARALLEL ACCESS SHIFT REGISTER

- **Parallel/serial load**
  Parallel/serial output
  Shift to the **left**, 4 bits

- `s_l=1` -> Paralled load
  `s_l=0` -> Serial load
- 'din': serial input
- 'D': parallel input
- 'dout': serial output
- 'Q': parallel output

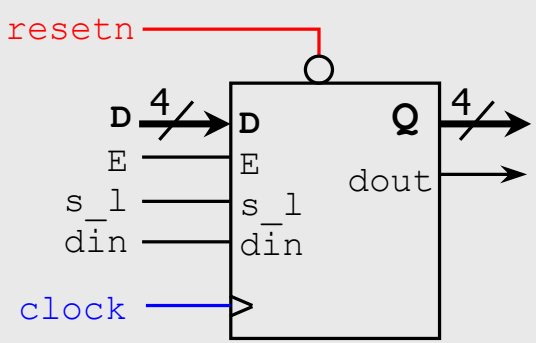



```
library ieee;
use ieee.std_logic_1164.all;
entity pashreg4_left is
   port (clock, resetn: in std_logic;
         E, s_l, din: in std_logic;
         dout: out std_logic;
         D: in std_logic_vector (3 downto 0);
         Q: out std_logic_vector (3 downto 0));
end pashreg4_left;

architecture bhv of pashreg4_left is
   signal Qt: std_logic_vector (3 downto 0);
begin
   process (resetn, clock, s_l, E)
   begin
      if resetn = '0' then Qt <= "0000";
      elsif (clock'event and clock = '1') then
         if E = '1' then
            if s_l='1' then Qt <= D;
            else
               Qt(3) <= Qt(2); Qt(2) <= Qt(1);
               Qt(1) <= Qt(0); Qt(0) <= din;
            end if;
         end if;
      end if;
   end process;
   Q <= Qt; dout <= Qt(3);
end bhv;
```

Daniel Llamocca

✓ PARALLEL ACCESS SHIFT REGISTER

- **Parallel/serial load**
  Parallel/serial output
  Shift to the **left**, 4 bits
- **Use of VHDL for loop**
- `s_l=1` -> Parallel load
  `s_l=0` -> Serial load
- 'din': serial input
- 'D': parallel input
- 'dout': serial output
- 'Q': parallel output

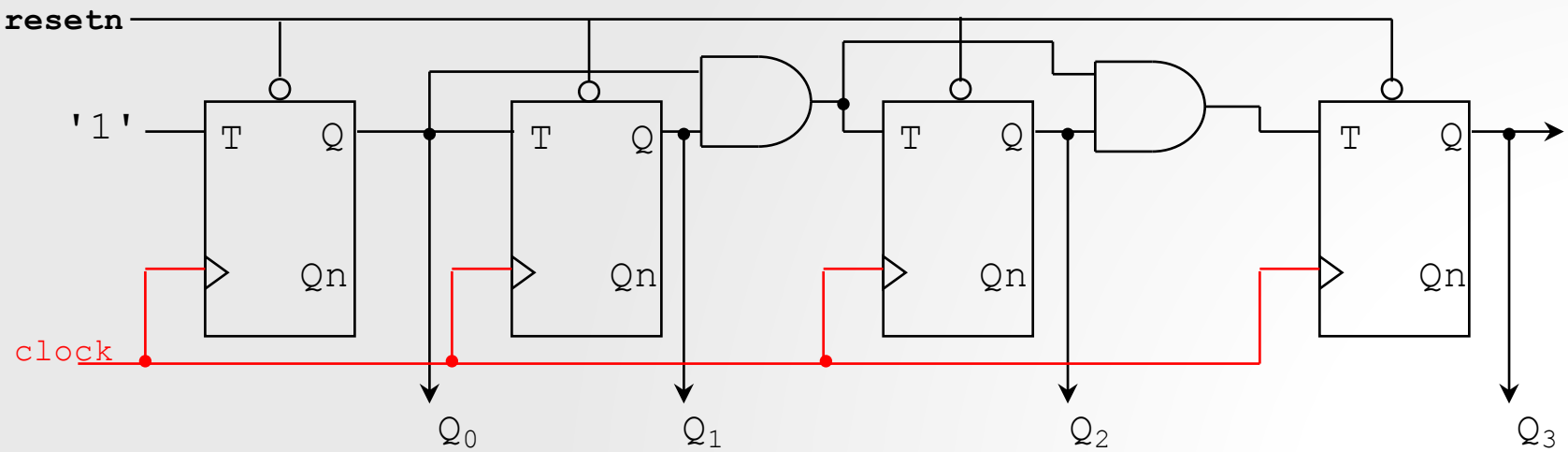



```
library ieee;
use ieee.std_logic_1164.all;
entity pashreg4_left is
   port (clock, resetn: in std_logic;
         E, s_l, din: in std_logic;
         dout: out std_logic;
         D: in std_logic_vector (3 downto 0);
         Q: out std_logic_vector (3 downto 0));
end pashreg4_left;

architecture bhv of pashreg4_left is
   signal Qt: std_logic_vector (3 downto 0);
begin
   process (resetn, clock, s_l, E)
   begin
      if resetn = '0' then Qt <= "0000";
      elsif (clock'event and clock = '1') then
         if E = '1' then
            if s_l='1' then Qt <= D;
            else
               gg: for i in 1 to 3 loop
                      Qt(i) <= Qt(i-1);
                   end loop;
                   Qt(0) <= din;
            end if;
         end if;
      end if;
   end process;
   Q <= Qt; dout <= Qt(3);
end bhv;
```

Daniel Llamocca

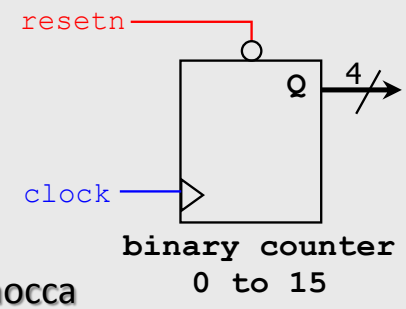
✓ SYNCHRONOUS PROCESSES

- **Synchronous Counters**
- Counters are very useful in digital systems. They can count the number of occurrences of a certain event, generate time intervals for task control, track elapsed time between two events, etc.
- Synchronous counters change their output on the clock edge (rising or falling). Counters are made of flip flops and combinatorial logic. Every flip flop in a synchronous counter shares the same clock signal. The figure shows a 4-bit synchronous binary counter (`0000` to `1111`). A `resetn` signal is also included to initialize the count.



✓ 4-bit binary counter with asynchronous active-low reset

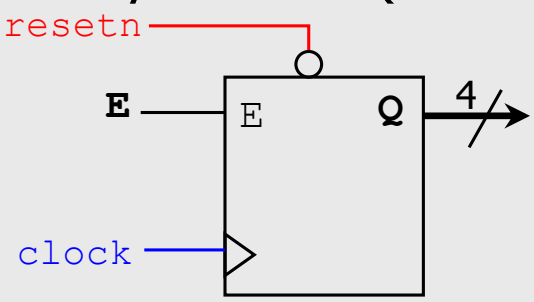
- Count: `0 to 2⁴-1` (once it gets to `2⁴-1`, it goes back to 0)
- `resetn`: active-low signal that sets Q to `0` as soon as it is 0, with no regards to the clock
- VHDL code: The behavioral style is preferred for counters (instead of the structural description).
- VHDL code: integer is used instead of std_logic_vector. The number of bits (4) is automatically computed.




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

entity my_count4b is
 port ( clock, resetn: in std_logic;
        Q: out integer range 0 to 15);
end my_count4b;

architecture bhv of my_count4b is
  signal Qt: integer range 0 to 15;
begin
  process (resetn,clock)
  begin
    if resetn = '0' then
       Qt <= 0;
    elsif (clock'event and clock='1') then
       Qt <= Qt + 1;
    end if;
  end process;
  Q <= Qt;
end bhv;
```

✓ 4-bit binary counter with enable and asynchronous active-low reset

- Synchronous enable signal ('E'): Only considered on the rising clock edge.
- This is also called a counter modulo-15
- If Qt=1111, then Qt ← Qt+1 results in Qt=0000 (since for 4 bits, 1111+1=0000) This is true of any binary counter (0 to 2^n-1)

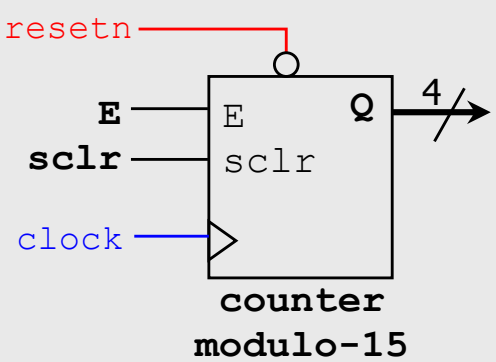


counter modulo-15

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

entity my_count4b_E is
 port ( clock, resetn, E: in std_logic;
        Q: out integer range 0 to 15);
end my_count4b_E;

architecture bhv of my_count4b_E is
  signal Qt: integer range 0 to 15;
begin
  process (resetn,clock, E)
  begin
    if resetn = '0' then
       Qt <= 0;
    elsif (clock'event and clock='1') then
       if E = '1' then
          Qt <= Qt + 1;
       end if;
    end if;
  end process;
  Q <= Qt;
end bhv;
```

✓ 4-bit binary counter with enable, asynchronous active-low reset and synchronous clear

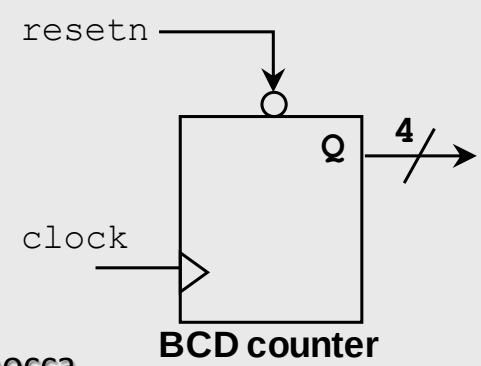
- The signals 'E' and 'sclr' are synchronous, thus they are only considered on the rising clock edge.
- If E=sclr=1 then Qt is set to 0.
- When Qt = 15, then Qt ← Qt+1 will result in Qt = 0.




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

entity my_count4b_E_sclr is
 port ( clock, resetn, E, sclr: in std_logic;
        Q: out integer range 0 to 15);
end my_count4b_E_sclr;

architecture bhv of my_count4b_E_sclr is
  signal Qt: integer range 0 to 15;
begin
  process (resetn,clock, E)
  begin
    if resetn = '0' then
       Qt <= 0;
    elsif (clock'event and clock='1') then
       if E = '1' then
          if sclr = '1' then Qt <= 0;
          else
             Qt <= Qt + 1;
          end if;
       end if;
    end if;
  end process;
  Q <= Qt;
end bhv;
```

✓ BCD (or modulo-10) counter with asynchronous active-low reset

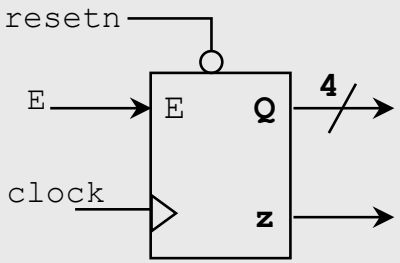
- Count: 0 to 9 (4 bits)
- When Qt = 9, then Qt ← Qt+1 results in Qt = 0.
  Note: This behavior (9 → 0) must be explicitly specified.
- This is different from the 0-to-15 counter, where the behavior (15 → 0) was implicit.




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

entity my_bcd_count is
 port ( clock, resetn: in std_logic;
        Q: out integer range 0 to 15);
end my_bcd_count;

architecture bhv of my_bcd_count is
  signal Qt: integer range 0 to 15;
begin
  process (resetn,clock)
  begin
    if resetn = '0' then
       Qt <= 0;
    elsif (clock'event and clock='1') then
       if Qt = 9 then
          Qt <= 0;
       else
          Qt <= Qt + 1;
       end if;
    end if;
  end process;
  Q <= Qt;
end bhv;
```

✓ modulo-13 counter with enable and asynchronous active-low reset

- Count: 0 to 12 (4 bits)
- Output 'z': Asserted when count is 12.



Counter 0 to 12

0000 - 0001 - 0010 - ... - 1010 - 1011 - 1100 - 0000 - ...

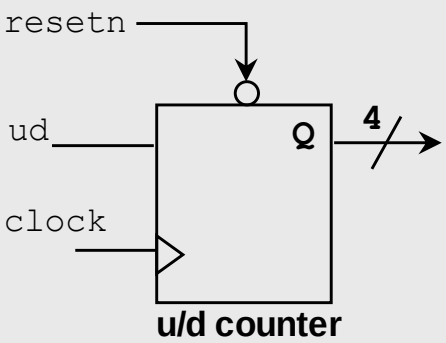
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
entity my_mod13count is
 port ( clock, resetn, E: in std_logic;
        Q: out std_logic_vector(3 downto 0);
        z: out std_logic);
end my_mod13count;

architecture bhv of my_mod13count is
  signal Qt: integer range 0 to 12);
begin
  process (resetn,clock,E)
  begin
    if resetn = '0' then Qt <= 0;
    elsif (clock'event and clock='1') then
       if E = '1' then
          if Qt=12 then Qt <= 0;
          else Qt <= Qt + 1;
          end if;
       end if;
    end if;
  end process;
  Q <= conv_std_logic_vector(Qt,4);
  z <= '1' when Qt = 12 else '0';
end bhv;
```

➢ **my_mod13count.zip:** my_mod13count.vhd, tb_my_mod13count.vhd.

Daniel Llamocca

✓ 4-bit synchronous up/down counter with asynchronous active-low reset

- ud = 0 → down
- ud = 1 → up
- When Qt = 0000, then Qt ← Qt-1 will result in Qt = 1111

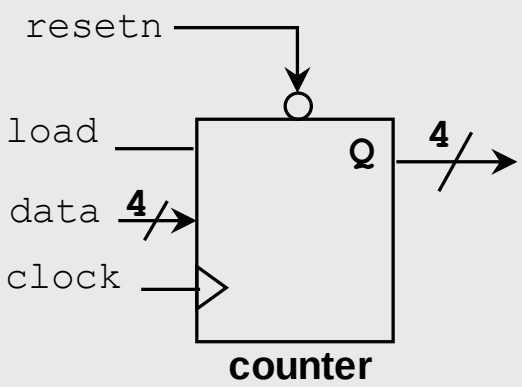



```
library ieee;
use ieee.std_logic_1164.all;

entity my_bcd_ud_count is
 port ( clock, resetn,ud: in std_logic;
        Q: out integer range 0 to 15);
end my_bcd_ud_count;

architecture bhv of my_bcd_ud_count is
  signal Qt: integer range 0 to 15;
begin
  process (resetn,clock,ud)
  begin
    if resetn = '0' then
       Qt <= 0;
    elsif (clock'event and clock='1') then
       if ud = '0' then
          Qt <= Qt - 1;
       else
          Qt <= Qt + 1;
       end if;
    end if;
  end process;
  Q <= Qt;
end bhv;
```

➢ **mybcd_udcount.zip:**
mybcd_udcount.vhd,
tb_mybcd_udcount.vhd
mybcd_udcount.ucf

✓ 4-bit Synchronous counter with parallel load

- Here, we use Q as a std_logic_vector.




```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity my_lcount is
 port ( clock, resetn,load: in std_logic;
        data: in std_logic_vector(3 downto 0);
        Q: out std_logic_vector(3 downto 0));
end my_lcount;

architecture bhv of my_lcount is
  signal Qt: std_logic_vector(3 downto 0);
begin
  process (resetn,clock,load)
  begin
    if resetn = '0' then
       Qt <= "0000";
    elsif (clock'event and clock='1') then
       if load = '1' then
          Qt <= data;
       else
          Qt <= Qt + "0001";
       end if;
    end if;
  end process;
  Q <= Qt;
end bhv;
```