OF THE
DEPARTMENT OF
INFORMATION ENGINEERING

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Digital Systems

Basic Sequential Blocks in VHDL: Latch and Flip-flop

Marta Bagatin, marta.bagatin@unipd.it

Degree Course in Information Engineering

Academic Year 2023-2024

Purpose of the Lesson

- Describe in VHDL and simulate the **basic sequential elements**
  - SR Latch with control input
  - D Latch
  - D Flip-flop with asynchronous reset
  - D Flip-flop with synchronous reset

Process Statement: Review

- Within a process, the statements **are executed sequentially**, i.e. one after the other, in the order in which they are written and in an infinitesimal time (in the absence of specific explicit commands, such as «wait»)
- **All processes within an architecture run simultaneously**
- The value of the signals assigned within a process is updated only when the process is suspended (because all statements within the process have been executed or because there is a «wait» instruction)
- A process may have a **sensitivity list**, containing the signals to which the process is sensitive: the process is **activated** when there is a **change** in one or more signals in the sensitivity list
  - To describe a combinational circuit, the sensitivity list must contain all the inputs of the circuit
  - A testbench typically uses processes without a sensitivity list: such processes are activated immediately at the start of the simulation

Process Statement: Review

Syntax for the process statement:

```
[processName:] process [(<sensitivity_list>)]
begin
   <sequential_instruction>
end process;
```

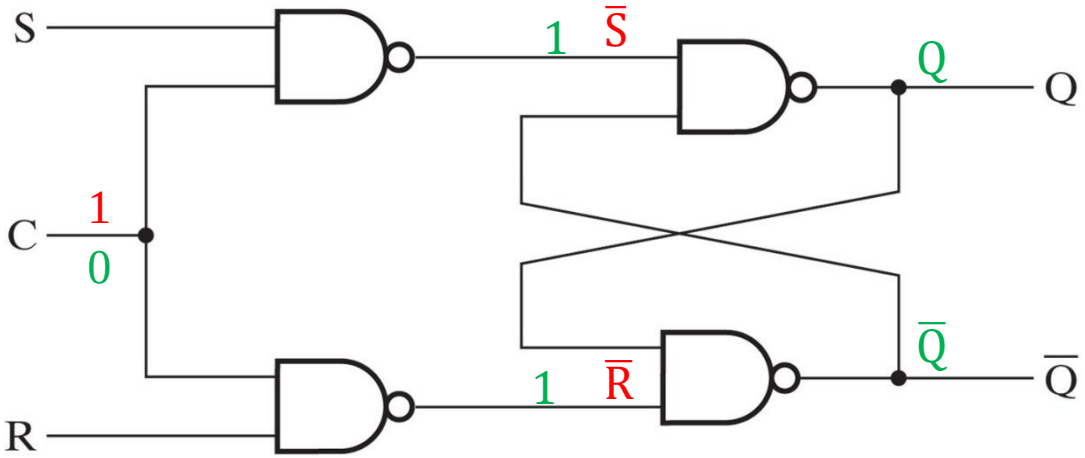
- The **sensitivity list** contains the signals to which change the process is sensitive. If the sensitivity list is empty (e.g. in a testbench), the process runs continuously
- A name can be assigned to the process (before the keyword `process` and it must be followed by ":")

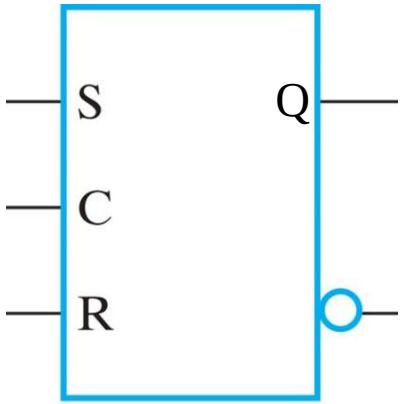
Latch

- SR Latch with control input
- D Latch

SR Latch with Control Input: Review

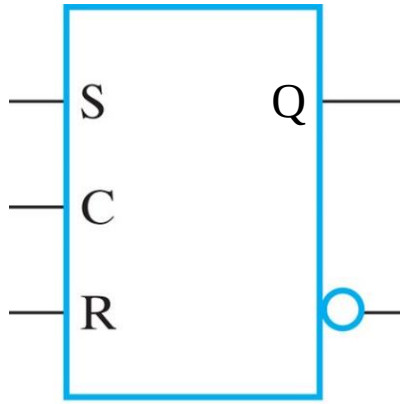
- The **control input C** acts as an enable signal
  - C = 0: latch not enabled, it keeps the previous state
  - C = 1: latch enabled, it works as an SR type latch
    - S = 1, R = 0: set state: Q = 1, $\overline{Q}$= 0
    - S = 0, R = 1: reset state: Q = 0, $\overline{Q}$= 1
    - S = 0, R = 0: memory state
    - S = 1, R = 1: forbidden state: Q = 1, $\overline{Q}$= 1

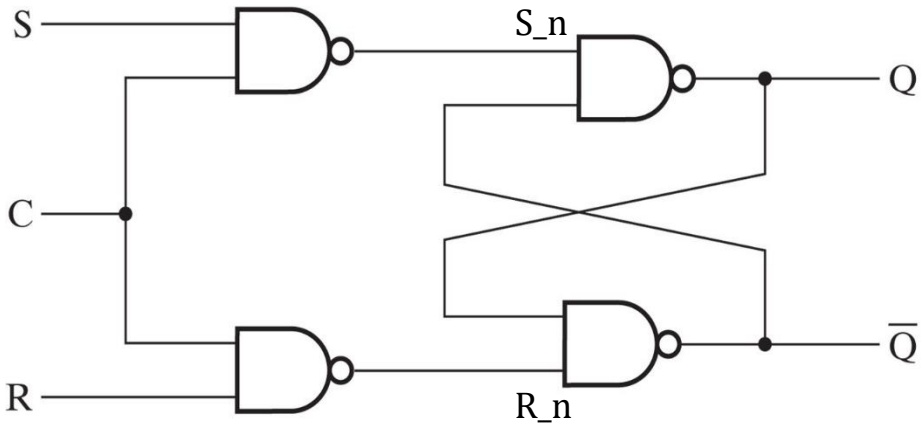




SR Latch with Enable: VHDL (Dataflow)

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

entity SRlatch is
   port(S, R, C : in std_logic;
        Q, Q_n : out std_logic);
end SRlatch;

-- Dataflow architecture
architecture dataflow of SRlatch is
   signal S_n, R_n: std_logic;
   begin
      S_n <= S nand C;
      R_n <= R nand C;
      Q <= S_n nand Q_n;
      Q_n <= R_n nand Q;
end dataflow;
```

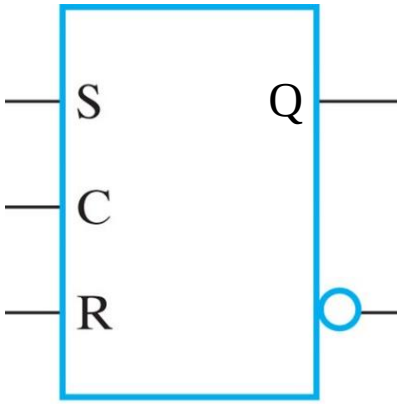




SR Latch with Enable: VHDL (Behavioral)

```
architecture behavioral of SRlatch is
signal SR: std_logic_vector (1 downto 0);
begin
   SR <= S & R;
   process (SR, C) begin
      if (C = '1') then
         case (SR) is
            when "00" => -- S = R = 0: MEMORY
               null;

            when "01" => -- S = 0, R = 1: RESET
               Q <= '0';
               Q_n <= '1';

            when "10" => -- S = 1, R = 0: SET
               Q <= '1';
               Q_n <= '0';

            when "11" => -- S = R = 1: FORBIDDEN
               Q <= '1';
               Q_n <= '1';

            when others =>
               Q <= 'X';
               Q_n <= 'X';
         end case;
      end if;
end process;
end behavioral;
```

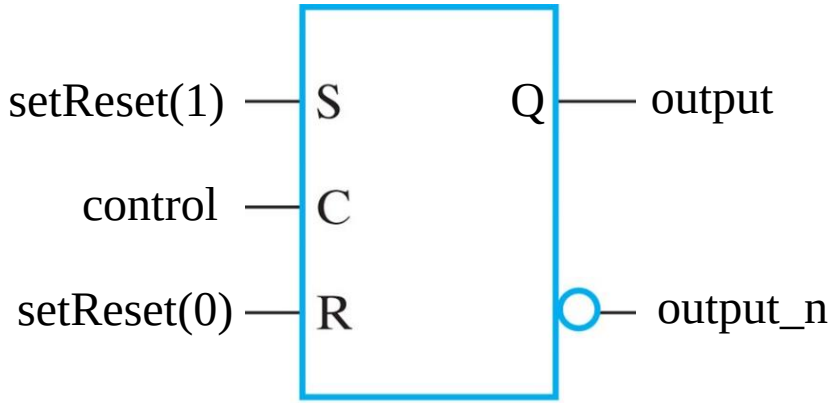
«process» with S, R, and C in the sensitivity list



«if-else» statement without specifying all cases (C = '0') generates memory, i.e. sequential logic!

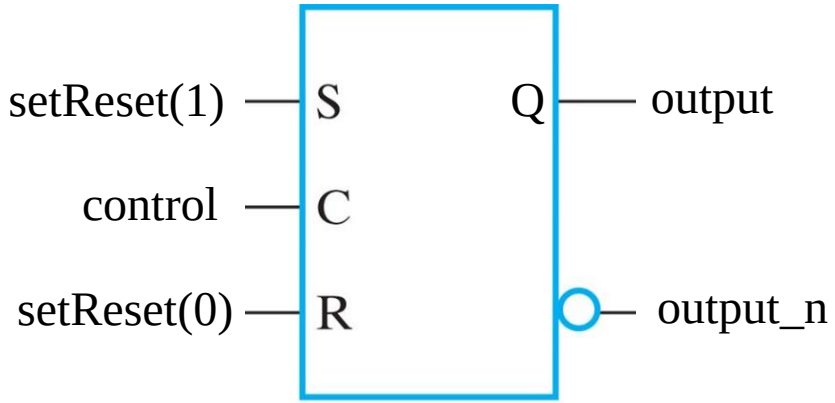
SR Latch with Enable: VHDL Testbench (1/2)

```
library IEEE;
use IEEE.std_logic_1164.all;

entity testbench is
end testbench;

architecture test of testbench is
   signal setReset: std_logic_vector(1 downto 0);
   signal control, output, output_n: std_logic;
begin
DUT: entity work.Srlatch (behavioral) port map( setReset(1), setReset(0), control, output, output_n);
   process begin
      setReset <= "10";
      control<='1';
      wait for 2 ns;
      setReset <= "00";
      wait for 5 ns;
      setReset <= "01";
      wait for 6 ns;
      setReset <= "00";
      wait for 4 ns;
      setReset <= "10";
      wait for 6 ns;
      setReset <= "01";
      wait for 5 ns;
      setReset <= "00";
```

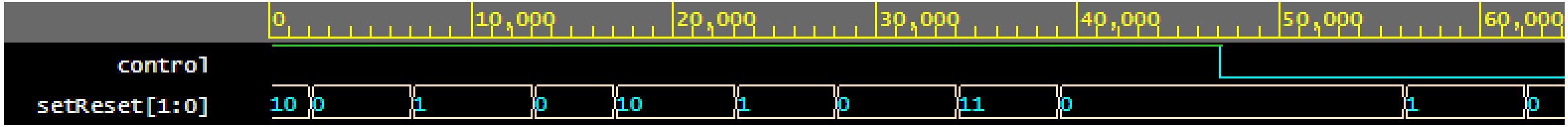


SR Latch with Enable: VHDL Testbench (2/2)

```
        wait for 6 ns;
        setReset <= "11";
        wait for 5 ns;
        setReset <= "01";
        wait for 0.1 ns;
        setReset <= "00";
        wait for 8 ns;
        control<='0';
        wait for 4 ns;
        setReset <= "00";
        wait for 5 ns;
        setReset <= "01";
        wait for 6 ns;
        setReset <= "00";
        wait for 2 ns;
        wait;
    end process;
end test;
```



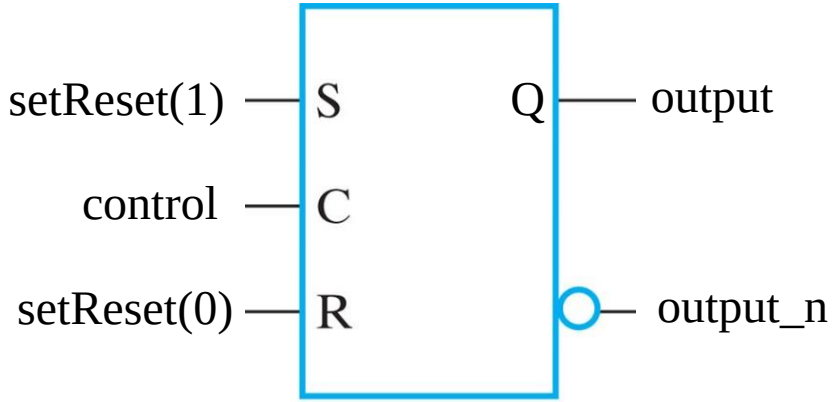
SR Latch with Enable Simulation: Time Diagram



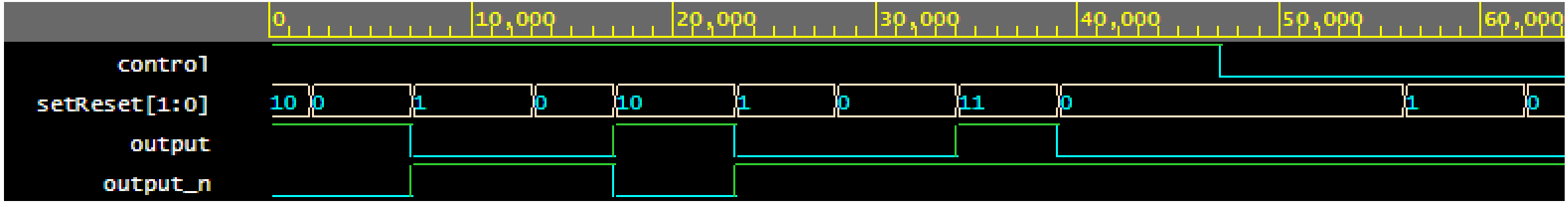
C = 1: latch enabled

- S = 1, R = 0: set
- S = 0, R = 1: reset
- S = 0, R = 0: memory
- S = 1, R = 1: forbidden

C = 0: latch disabled



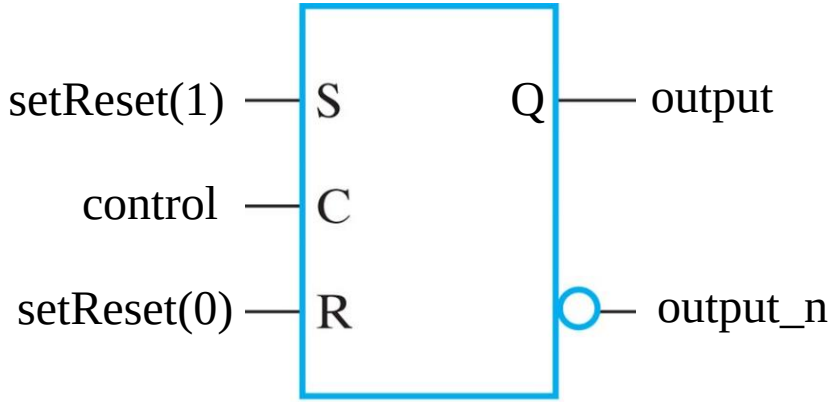
SR Latch with Enable Simulation: Time Diagram



C = 1: latch enabled

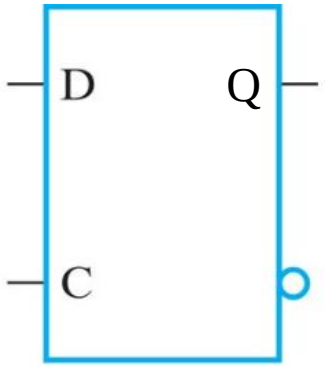
- S = 1, R = 0: set
- S = 0, R = 1: reset
- S = 0, R = 0: memory
- S = 1, R = 1: forbidden

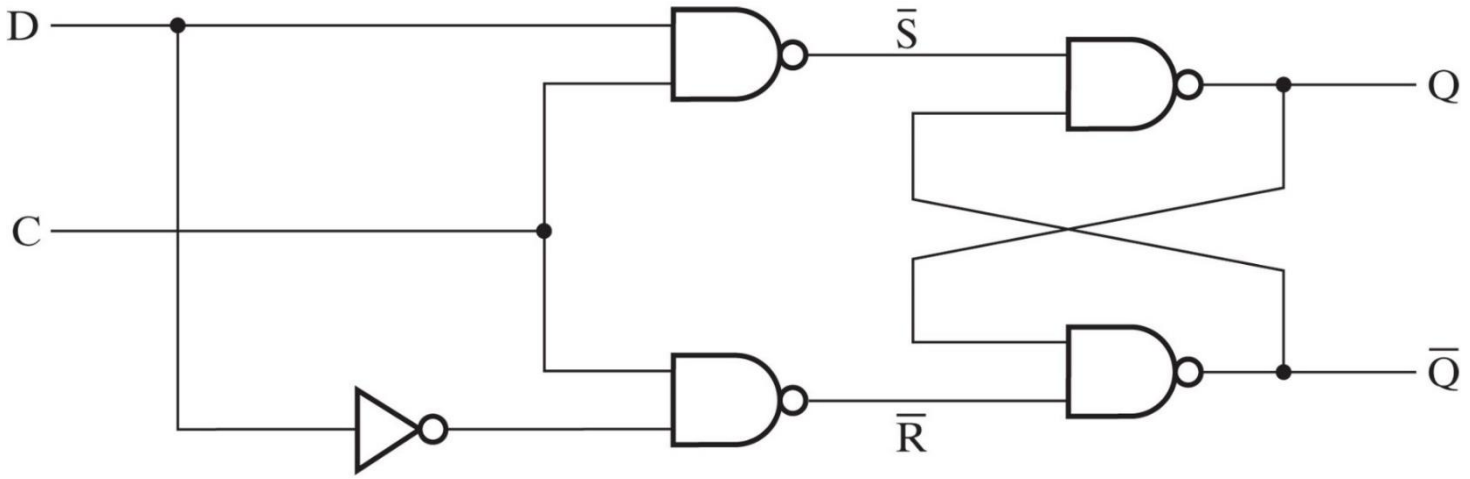
C = 0: latch disabled



D Latch: Review

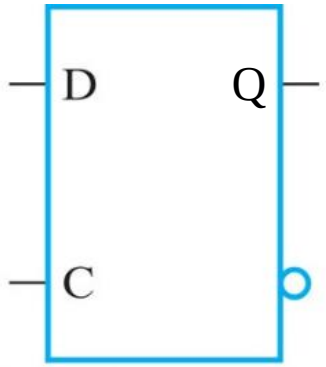
- The D latch **avoids the problem of the forbidden state**, having a single data input
  - Control input C = 0: the circuit maintains the previous state (memory)
  - Control input C = 1: D is transferred to output Q
    - C = 1, D = 0: reset state => Q = 0
    - C = 1, D = 1: set state => Q = 1





D Latch: VHDL Design (Behavioral)

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Dlatch is
   port (D, C : in std_logic;
         Q, Q_n : out std_logic);
end Dlatch;

-- behavioral architecture
architecture Dlatch_arch of Dlatch is
begin
   process (D, C)
      begin
      if (C = '1') then
         Q <= D;
         Q_n <= not D;
      end if;
   end process;
end Dlatch_arch;
```



C = 0: keeps the previous state

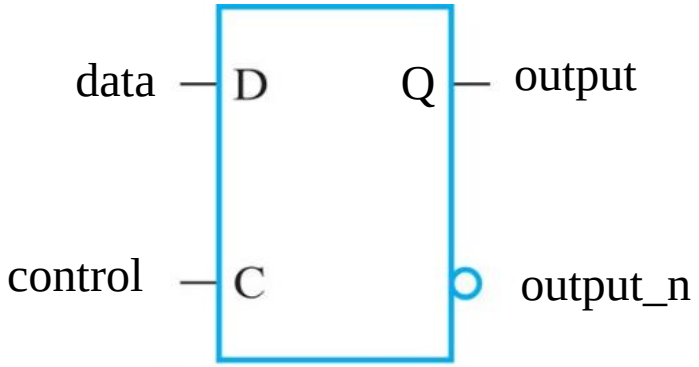
C = 1: D is transferred to output Q

C = 1, D = 0: reset state => Q = 0

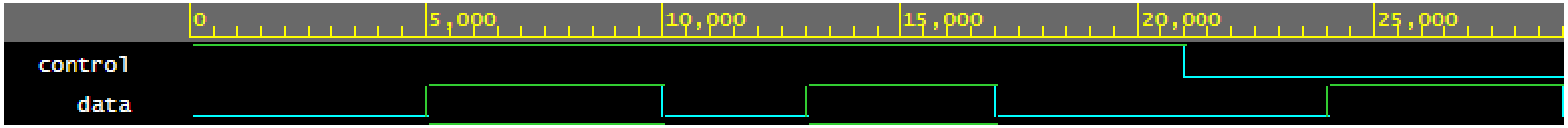
C = 1, D = 1: set state => Q = 1

D Latch: VHDL Testbench

```
library IEEE;
use IEEE.std_logic_1164.all;

entity testbench is
end testbench;

architecture test of testbench is
   signal data, control, output, output_n : std_logic;
   begin
   DUT: entity work.Dlatch port map (data, control, output, output_n);
   process begin
      data <= '0';
      control <= '1';
      wait for 5 ns;
      data <= '1';
      wait for 5 ns;
      data <= '0';
      wait for 3 ns;
      data <= '1';
      wait for 4 ns;
      data <= '0';
      wait for 4 ns;
      control <= '0';
      wait for 3 ns;
      data <= '1';
      wait for 5 ns;
      data <= '0';
      wait;
   end process;
end test;
```



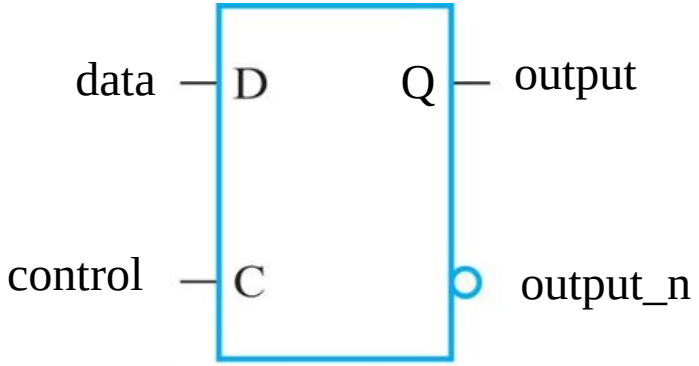
D Latch Simulation: Time Diagram



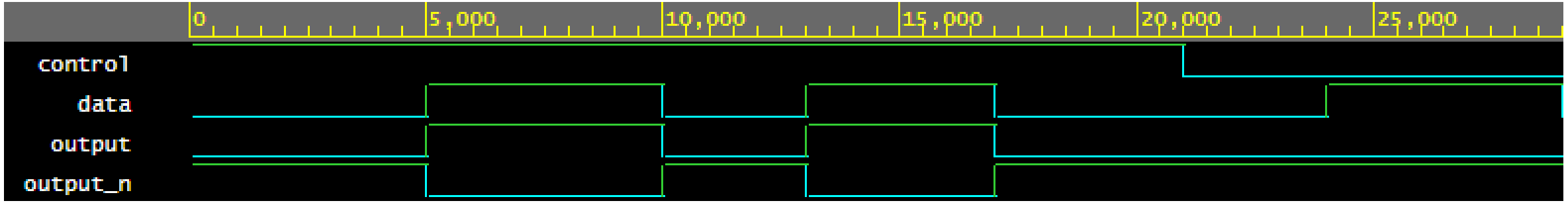
C = 1: D is transferred to output Q

- C = 1, D = 0: reset state => Q = 0
- C = 1, D = 1: set state => Q = 1

C = 0: keeps the previous state



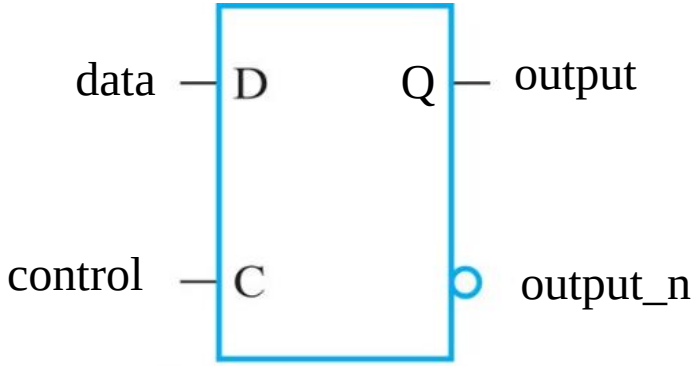
D Latch Simulation: Time Diagram



C = 1: D is transferred to output Q

- C = 1, D = 0: reset state => Q = 0
- C = 1, D = 1: set state => Q = 1

C = 0: keeps the previous state

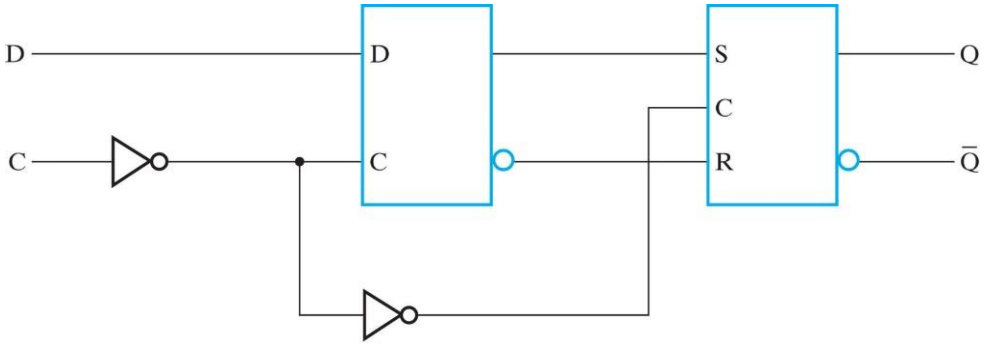


D Flip-flops

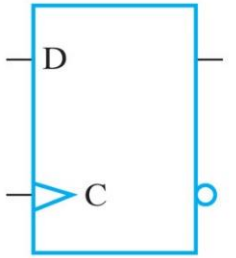
- With asynchronous reset
- With synchronous reset

Flip-flop D: Review

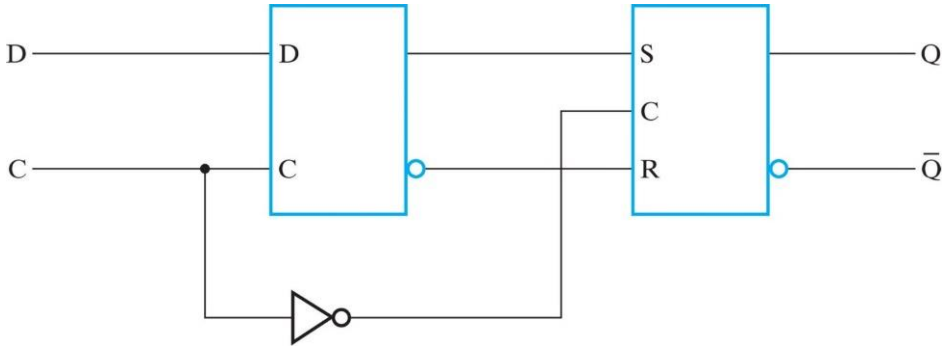
- A positive-edge-triggered (or negative-edge-triggered) D flip-flop consists of two latches connected in cascade, driven by complementary control signals
  - At each rising (or falling) edge of the clock, the last value of D sampled by the master is transferred to the slave, e.g. to the flip-flop output Q
  - At all other instants of time, the previous state is maintained



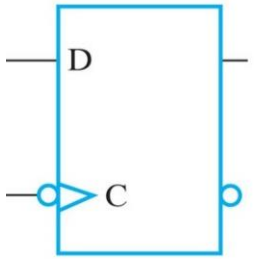
PET-D-Flip-flop
Logic diagram
and symbol



Triggered D



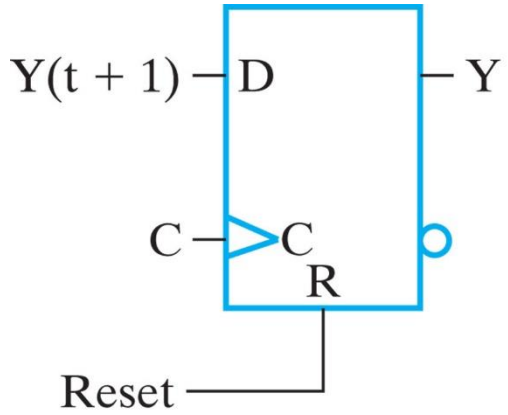
NET-D-Flip-flop
Logic diagram
and symbol



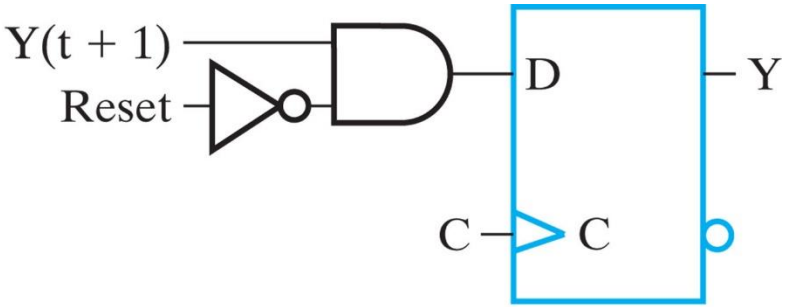
Triggered D

D Flip-flop with Reset

- **Asynchronous reset**: the reset signal is asynchronous with respect to the clock, i.e. reset can arrive at any time

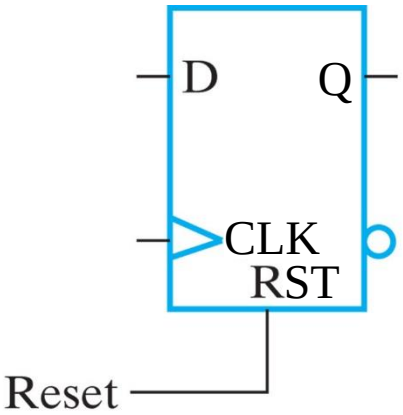


- **Synchronous reset**: the reset signal is synchronized with the clock, i.e. reset can only arrive at the clock edges (positive or negative depending on whether it is positive- or negative-edge-triggered FF). Therefore, synchronous reset **requires a clock trigger to be effective**



D-Flip flop with Asynchronous Reset: VHDL Design

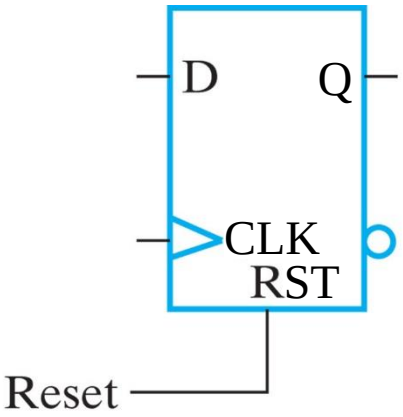
```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

entity dff is
   port (CLK, RST, D: in std_logic;
                   Q: out std_logic);
end dff;
-- behavioral architecture
architecture dff_as of dff is
begin
   process (RST, CLK)
   begin
      if (RST = '1') then
         Q <= '0';
      elsif (CLK'event and CLK = '1') then
         Q <= D;
      end if;
   end process;
end dff_as;
```



Reset is not synchronized with the clock: it can arrive at any time

D-Flip flop with Asynchronous Reset: VHDL Design

```
library IEEE;
use IEEE.std_logic_1164.all;

entity dff is
   port (CLK, RST, D: in std_logic;
                   Q: out std_logic);
end dff;
-- behavioral architecture
architecture dff_as of dff is
begin
   process (RST, CLK)
   begin
      if (RST = '1') then
         Q <= '0';
      elsif (CLK'event and CLK = '1') then
         Q <= D;
      end if;
   end process;
end dff_as;
```



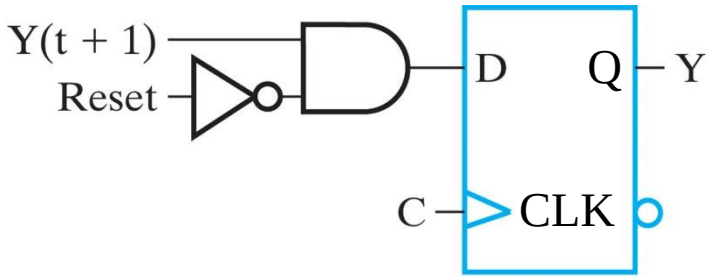
The process is activated at the change of at least one between **CLK** and **RESET**

CLK'event is 1 whenever CLK changes. This boolean variable ANDed with «CLK = '1'» indicates a clock positive edge. "event" is an attribute of the clock signal

AT RESET: output becomes 0 (no matter the clock value)
AT CLK POSITIVE EDGE: output becomes D

D-Flip flop with Synchronous Reset: VHDL Design

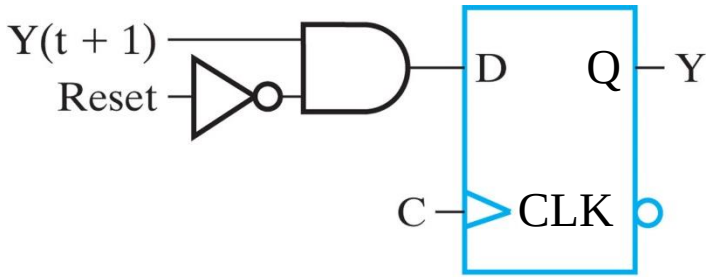
```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity dff is
   port (CLK, RST, D: in std_logic;
                   Q: out std_logic);
end dff;
-- behavioral architecture
architecture dff_sync of dff is
begin
   process (CLK)
   begin
      if (CLK'event and CLK = '1') then
         if (RST = '1') then
            Q <= '0';
         else
            Q <= D;
         end if;
      end if;
   end process;
end dff_sync;
```



Reset is synchronized with the clock: it can arrive only at the active clock edges

D-Flip flop with Synchronous Reset: VHDL Design

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

entity dff is
   port (CLK, RST, D: in std_logic;
                   Q: out std_logic);
end dff;
-- behavioral architecture
architecture dff_sync of dff is
begin
   process (CLK)
   begin
      if (CLK'event and CLK = '1') then
         if (RST = '1') then
            Q <= '0';
         else
            Q <= D;
         end if;
      end if;
   end process;
end dff_sync;
```



Only **CLK** signal is present in the sensitivity list

The **RESET** does no longer dominate the clock, it can only occur in correspondence of the rising edge

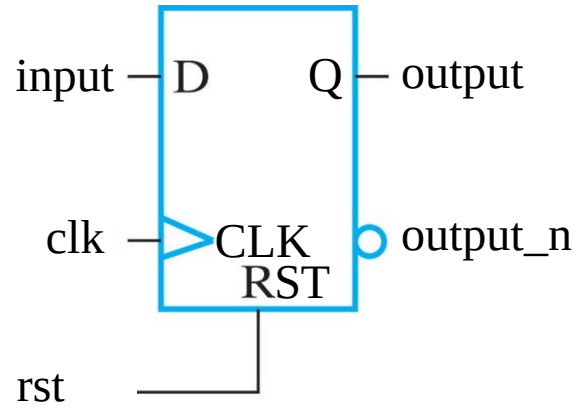
Simulation of Synchronous Sequential Logic

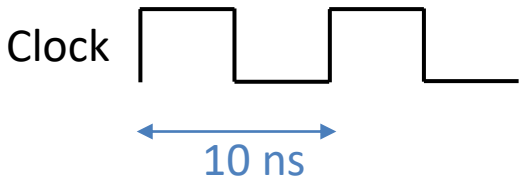
- It is necessary that **changes in the inputs occur far enough from the active clock edge** (e.g. during the falling edge in case of PET D-FF). In this way, combinational logic has the time to calculate the next state and the output before the next active clock edge
- We will implement the testbench with **two processes**, working in parallel:

1) The first process **generates the clock**. This process is never explicitly stopped and runs continuously, ensuring that the clock keeps oscillating
2) The second process **generates the sequence of input signals**, including the reset, with proper synchronization, for example at the falling edges of the clock. This process is stopped with the command "`std.env.stop`", which ends the simulation

D-Flip-flop with Asynchronous Reset: Testbench (1/2)

```
library IEEE;
use IEEE.std_logic_1164.all;

entity test_dff is
end test_dff;

architecture test of test_dff is
signal clk, rst, input, output: std_logic;
begin
    DUT: entity work.dff (dff_as) port map (clk, rst, input, output);
    generate_clock: process
    begin
        clk <= '1';
        wait for 5 ns;
        clk <= '0';
        wait for 5 ns;
    end process;
```



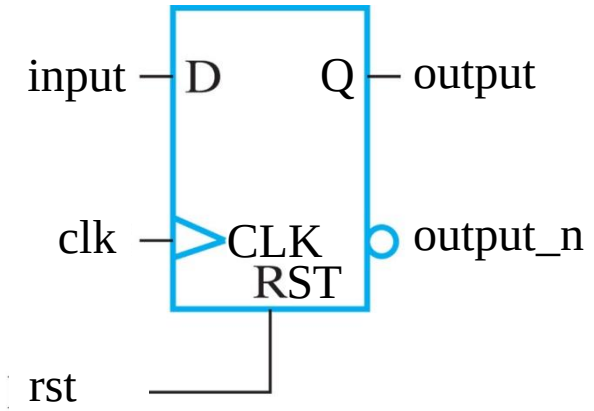


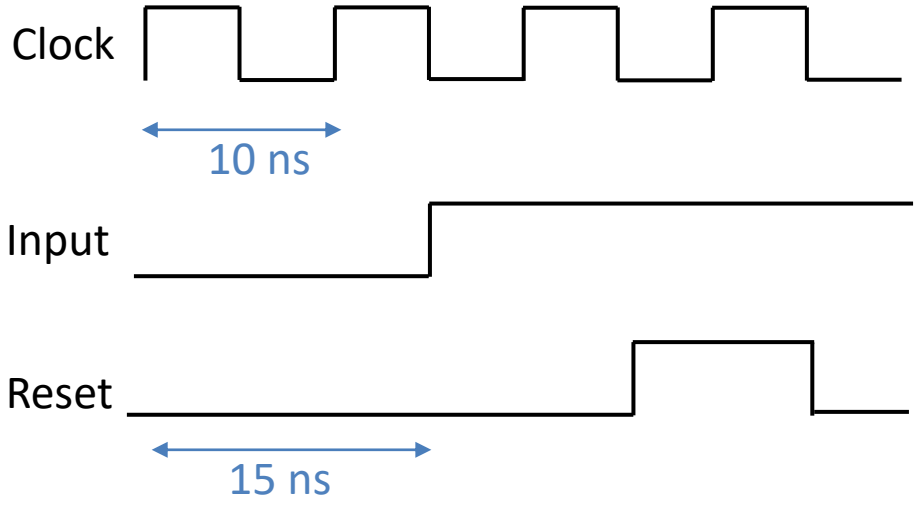
Process to **generate the clock** (square wave with 10 ns period)

D-Flip-flop with Asynchronous Reset: Testbench (2/2)

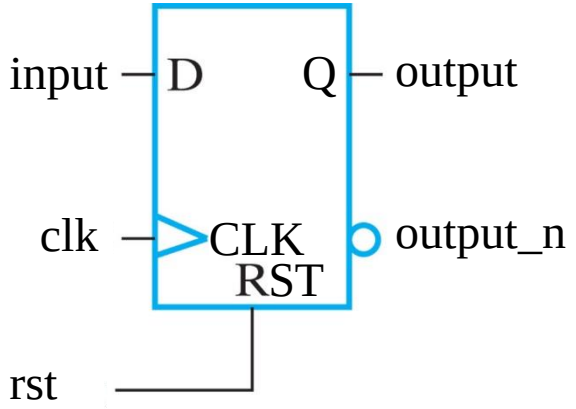
```
    apply_inputs: process
    begin
        rst <= '0';
        input <= '0';
        wait for 15 ns;
        input <= '1';
        wait for 10 ns;
        rst <= '1';
        wait for 20 ns;
        rst <= '0';
        wait for 10 ns;
        input <= '0';
        wait for 10 ns;
        input <= '1';
        std.env.stop;
    end process;
endtest;
```

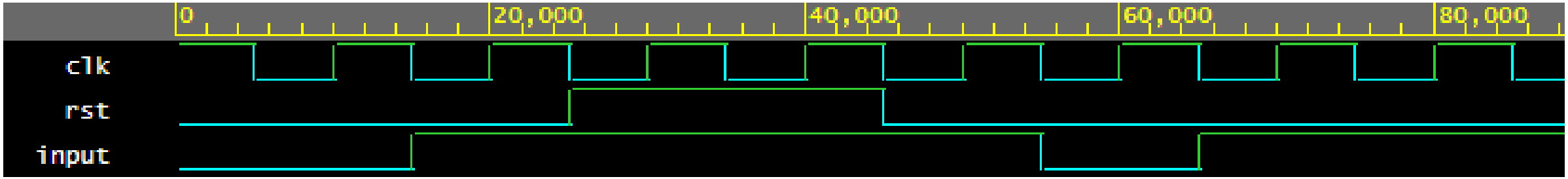
Process to **change the inputs** at the falling edge of the clock



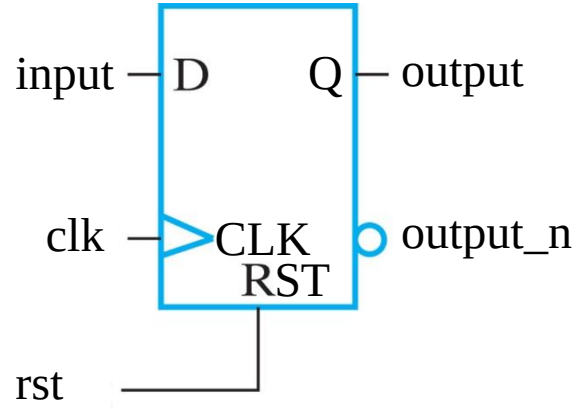


D-Flip-flop with Asynchronous Reset: Simulation

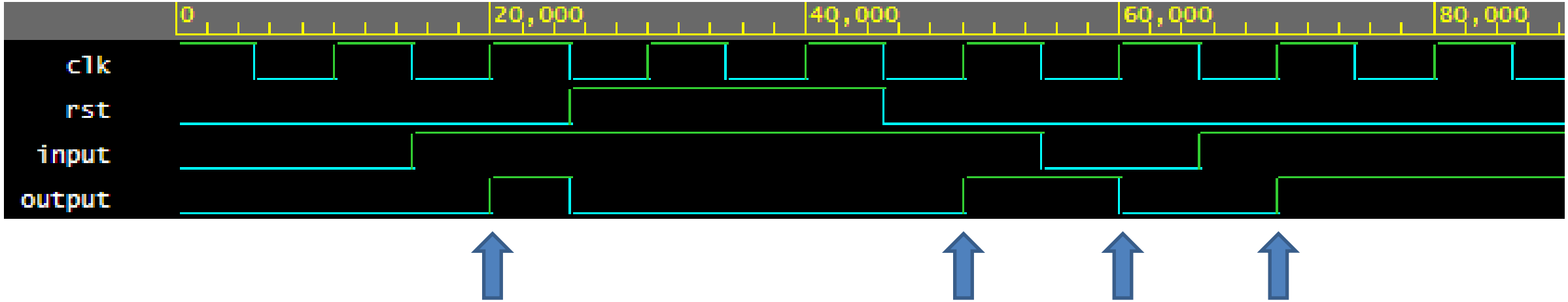




D-Flip-flop with Asynchronous Reset: Simulation



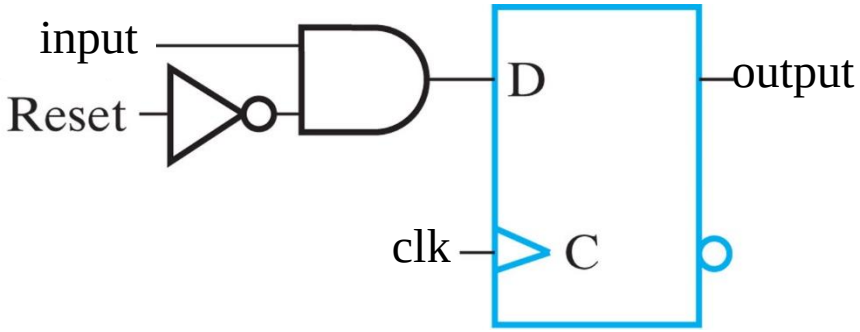
When reset =1 (at any time, not necessarily at clock edges): the output becomes 0



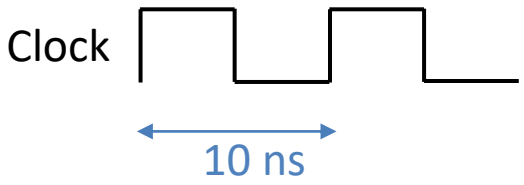
When reset is zero: at the clock rising edges, the output follows the input

D-Flip-flop with Synchronous Reset: Testbench (1/2)

```
library IEEE;
use IEEE.std_logic_1164.all;

entity test_dff is
end test_dff;

architecture test of test_dff is
signal clk, rst, input, output: std_logic;
begin
   DUT: entity work.dff (dff_sync) port map (clk, rst, input, output);
   generate_clock: process
      begin
      clk <= '1';
      wait for 5 ns;
      clk <= '0';
      wait for 5 ns;
   end process;
```



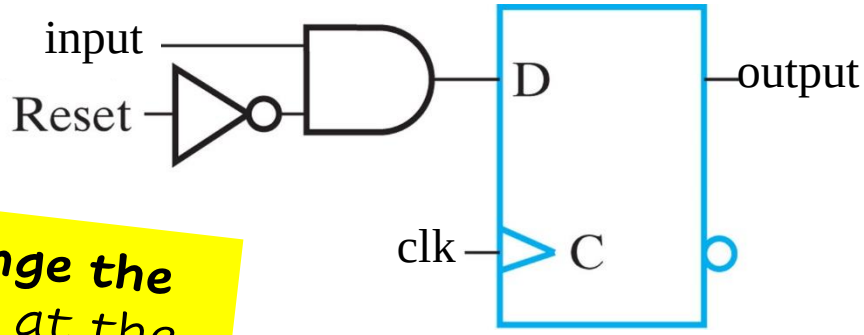
Process to generate a **clock** with period 10 ns

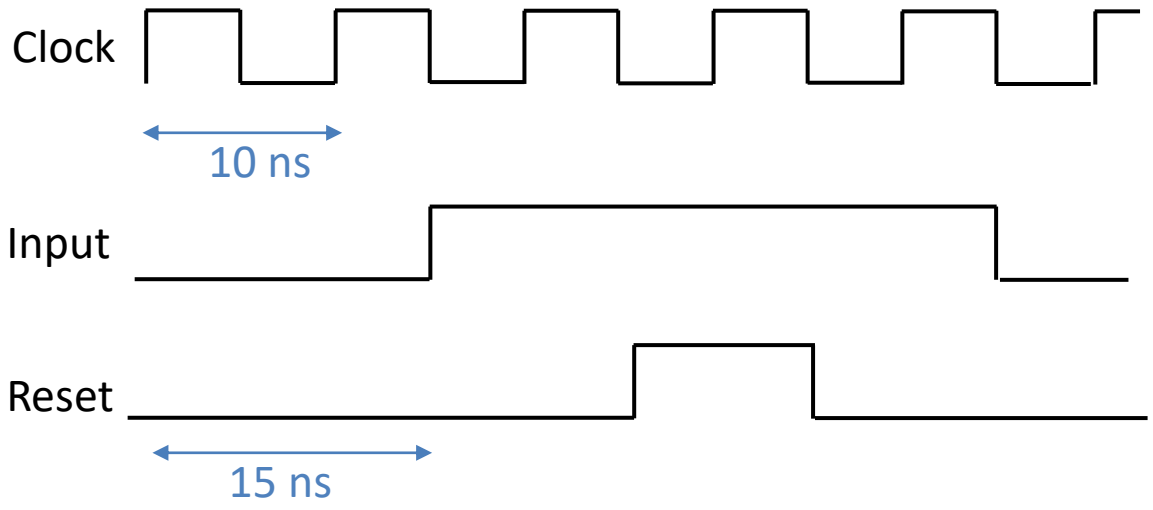


D-Flip-flop with Synchronous Reset: Testbench (2/2)

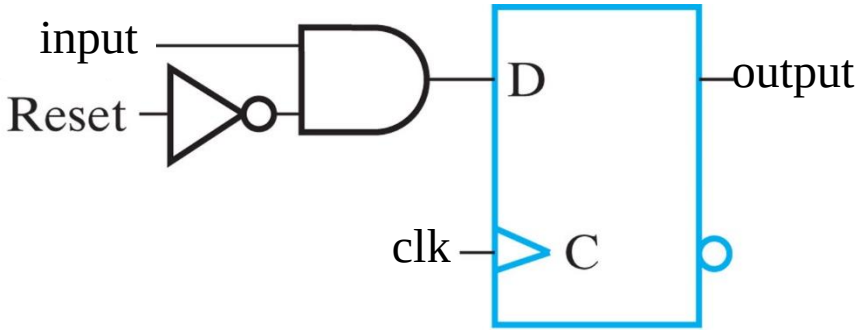
```
    apply_inputs: process
       begin
       rst <= '0';
       input <= '0';
       wait for 15 ns;
       input <= '1';
       wait for 10 ns;
       rst <= '1';
       wait for 10 ns;
       rst <= '0';
       wait for 10 ns;
       input <= '0';
       std.env.stop;
    end process;
end test;
```

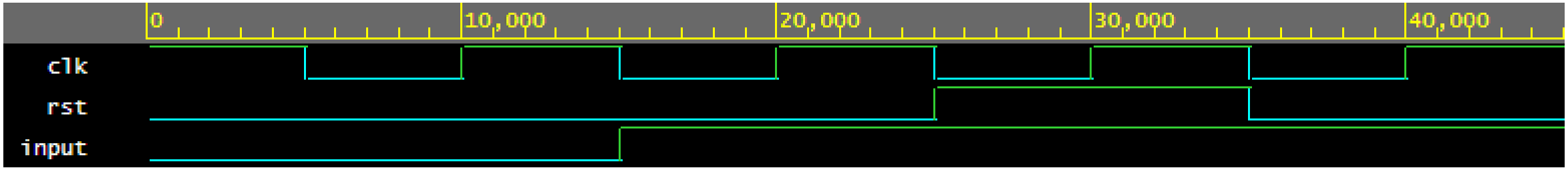
Process to **change the inputs and reset** at the falling edge of the clock



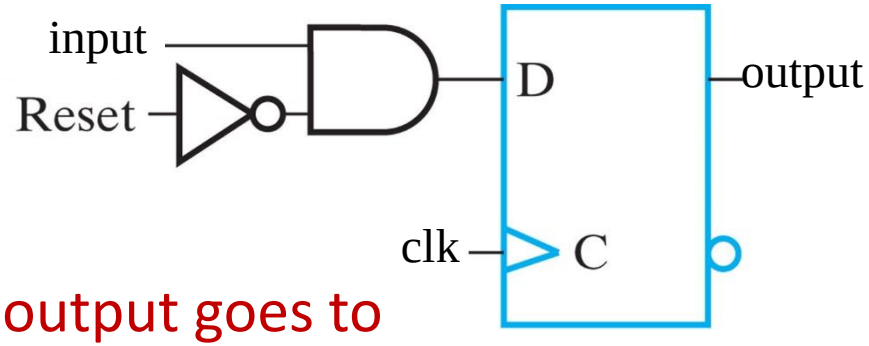


D-Flip-flop with Synchronous Reset: Simulation

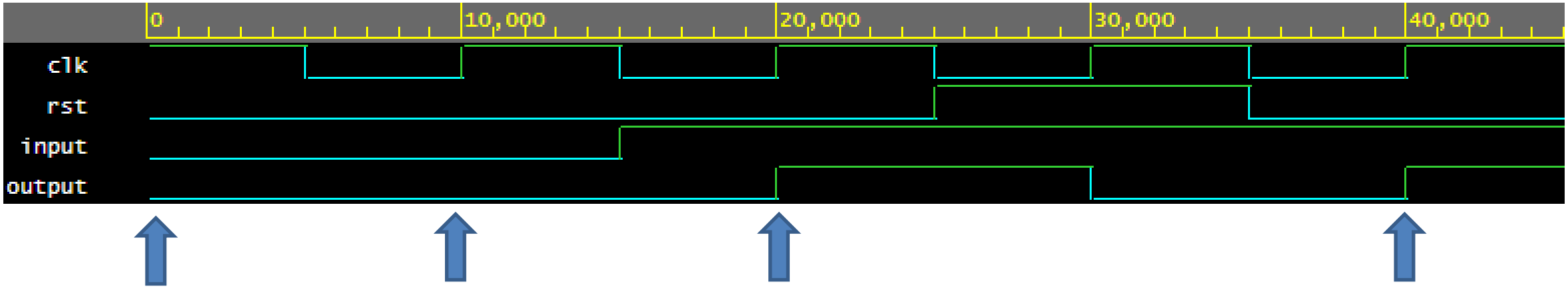




D-Flip-flop with Synchronous Reset: Simulation



Reset is asserted => the output goes to 0 at the next rising edge of the clock



If reset is 0: at the rising edges of the clock, the output follows the input

Disclaimer

Figures from*Logic and Computer Design Fundamentals*, Fifth Edition, GE Mano |Kime| Martin