DVA494

Programming of Reliable Embedded Systems

Obed Mogaka | Hamid Mousavi | Masoud Daneshtalab

IDT, Malardalens University

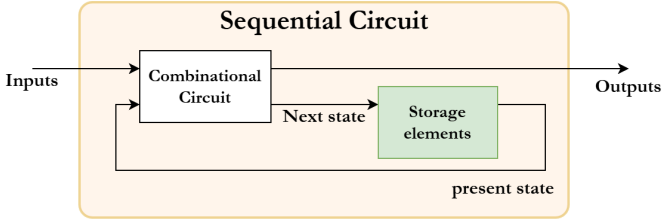
January 25, 2026

Today's Agenda

- Review of Sequential Circuits
  - Latches
  - Flip Flops
  - Registers
  - Counters
- Asynchronous and Synchronous Design
- Mealy and Moore Sequential Circuits
- VHDL Sequential Statements
  - if statement
  - case statement
  - Loops

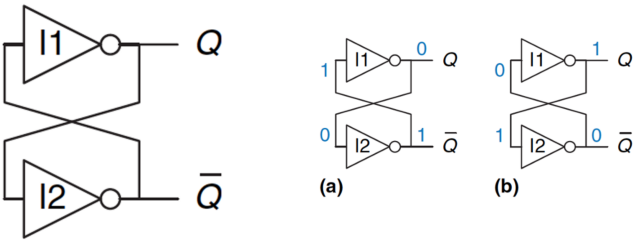
Sequential Circuits

- Combinational circuit output depends only on current input.
- We want circuits that produce output depending on current and past input values – **circuits with memory**
- How can we design a circuit that stores information?



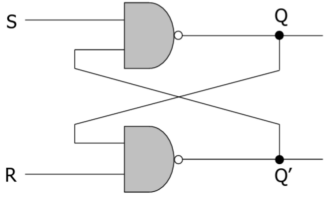
Storage Elements

Basic Element: Cross-Coupled Inverters



- Has two stable states: $Q = 1$ or $Q = 0$.
- Has a third possible **"metastable"** state with both outputs oscillating between 0 and 1 (we will see this later)
- **Not useful without a control mechanism for setting Q**

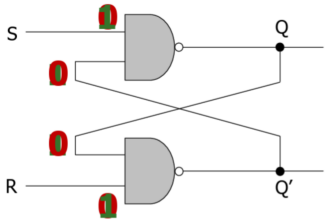
The R-S (Reset-Set) Latch



Input		Output
R	S	Q
1	1	Q_{prev}
1	0	1
0	1	0
0	0	Forbidden

- Cross-coupled **NAND** gates
  - **S** and **R** are control inputs
  - In quiescent (idle) state, both **S and R are held at 1**
  - **S (set)**: drive S to 0 (keeping R at 1) to change Q to 1
  - **R (reset)**: drive R to 0 (keeping S at 1) to change Q to 0
- S and R should never both be 0 at the same time

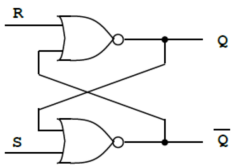
Why not R=S=0?



Input		Output
R	S	Q
1	1	Q_{prev}
1	0	1
0	1	0
0	0	Forbidden

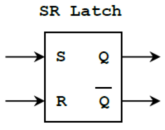
- If **R=S=0**, Q and Q' will both settle to 1, which breaks our invariant that **Q = !Q'**
- If S and R transition back to 1 at the same time, Q and Q' begin to oscillate between 1 and 0 because their final values depend on each other **(metastability)**

SR Latch (NOR Version)

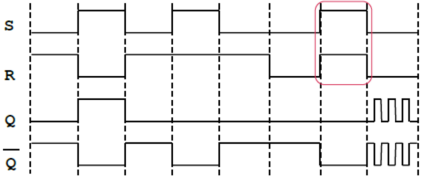


S	R	Q_{t+1}	$\overline{Q_{t+1}}$
0	0	Q_t	$\overline{Q_t}$
0	1	0	1
1	0	1	0
1	1	0	0

restricted



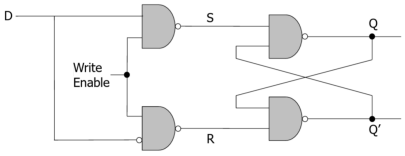
$Q_{t+1} \leftarrow \bar{R}S + \bar{R}Q_t$



- **So then how do we guarantee correct operation of an R-S Latch?**

The Gated D Latch

- **We add two more NAND gates!**

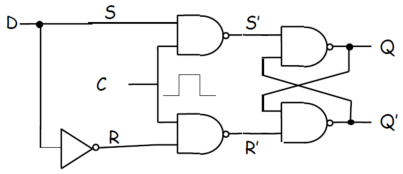


Input		Output
WE	D	Q
0	0	Q_{prev}
0	1	Q_{prev}
1	0	0
1	1	1

- This is essentially an SR Latch with enable, where $R = not(D),\ S = D$.
- **Q** takes the value of **D**, when **write enable (WE)** is set to 1
- S and R can never be 0 at the same time!
- The D Latch always has an enable input.

Characteristic Equation

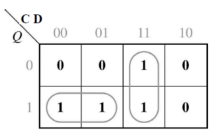
- A **characteristic equation** expresses the **next state** (Q^+) of a memory element in terms of: **Q (present state)** and **Inputs** (such as D, T, J, K, etc.)
- We can derive the characteristic equation using the truth tables and simplify using K-Maps.



D	C	Q	Q'
0	1	0	1
1	1	1	0
X	0	Q_0	Q_0'

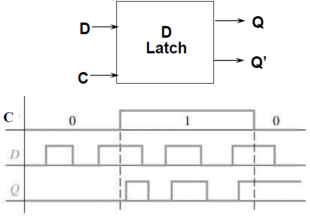
S	R	C	Q	Q'	
0	0	1	Q_0	Q_0'	Store
0	1	1	0	1	Reset
1	0	1	1	0	Set
1	1	1	1	1	Disallowed
X	X	0	Q_0	Q_0'	Store

C	D	Q	Q+
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



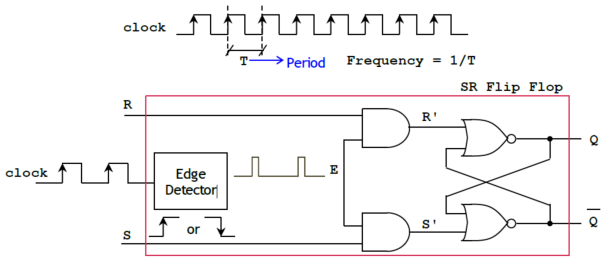
Characteristic equation :

$$Q^+ = \bar{C}.Q + C.D$$



Synchronous Circuits: Flip Flops

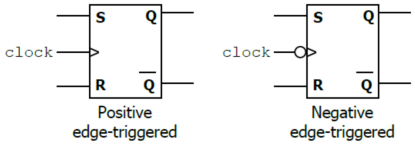
- **Flip flops** are made out of:
  - A Latch with an enable input
  - An Edge detector circuit
- The input to the Edge Detector is a signal called '**clock**'. A clock signal is a square wave with a fixed frequency.



- The edge detector circuit generates short-duration pulses during **rising (or falling)** edges. These pulses act as short-time enable of the Latch.
- Latches are "transparent" (= any change on the inputs is seen at the outputs immediately).

Flip Flops

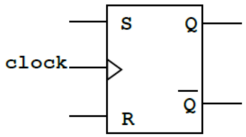
- Depending on what type of edge we are detecting, flip flops can be classified as:
  - **Positive-edge triggered** flip flop: The edge detector circuit generates pulses during rising edges.
  - **Negative-edge triggered** flip flop: The edge detector circuit generates pulses during falling edges.
  - **Dual-edge triggered** flip flop: The edge detector circuit generates pulses during both rising and falling edges.



- Depending on the Latch used, we have diffent types of FFs:
  - RS Flip-Flop
  - D Flip-Flop
  - JK Flip-Flop
  - T Flip-Flop

SR Flip Flop

clock	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



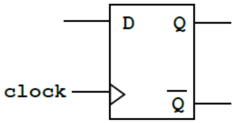
Excitation Equation :

$$Q_{t+1} = \bar{R}S + \bar{R}Q_t$$

Note that when there are no rising edges, $Q_{t+1} = Q_t$

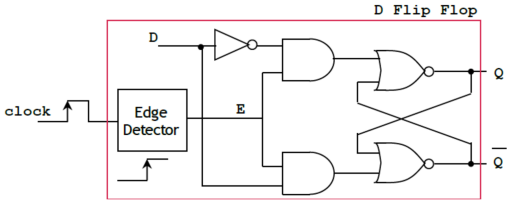
D Flip Flop

clock	D	Q_{t+1}
↑	0	0
↑	1	1

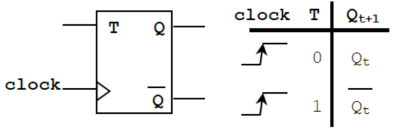


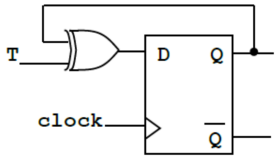
Excitation Equation :

$$Q_{t+1} \leftarrow D$$



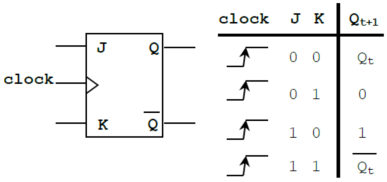
T and JK Flip Flops

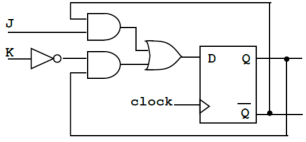




Excitation Equation :

$$Q_{t+1} \leftarrow D = T \oplus Q_t$$





Excitation Equation :

$$Q_{t+1} \leftarrow J\bar{Q}_t + \bar{K}Q_t$$

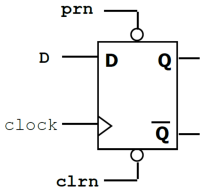
Synchronous and Asynchronous Inputs

Synchronous Inputs

- Typically, flip flops only change their outputs on the rising (or falling edge).
- Usually, a change on the inputs forces a change on the outputs.
- These inputs are known as **synchronous inputs**, as the inputs' state is only checked on the rising (or falling) edges.
- Example: Input D of a D flip flop, Inputs J, K of a JK flip flop.

Asynchronous Inputs

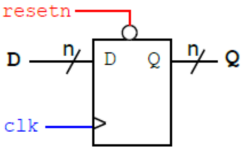
- However, in many instances, it is useful to have inputs that force the outputs to a value immediately, disregarding the rising (or falling edges).
- These inputs are known as **asynchronous inputs**.
- Common asynchronous inputs are preset (prn) and reset (clrn, resetn) (they can be active-low or active high)
- A Flip flop can have more than one asynchronous inputs, or none.

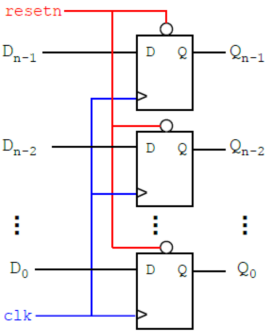


Registers

n-BIT REGISTER

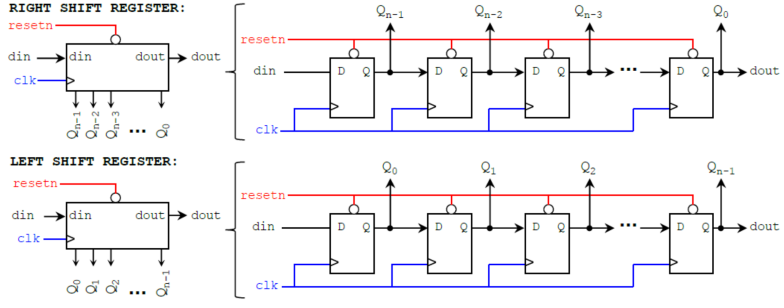
- This is a collection of 'n' D-type flip flops
- Each flip flop independently stores one bit.
- The flip flops are connected in parallel.
- They also share the same resetn and clock signals.





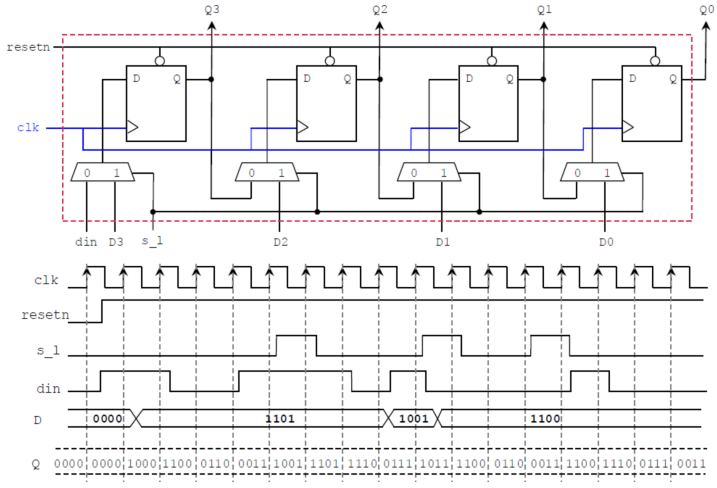
n-Bit SHIFT REGISTER

- This is a collection of 'n' D-type flip flops, connected serially.
- The flip flops share the same resetn and clock signals. The serial input is called 'din', and the serial output is called 'dout'.
- The flip flop outputs (also called the parallel output) are called $Q = Q_{n-1}Q_{n-2}...Q_0$.
- Depending on how we label the bits, we can have:
  - **Right shift register:** The input bit moves from the MSB to the LSB, and
  - **Left shift register:** The input bit moves from the LSB to the MSB.



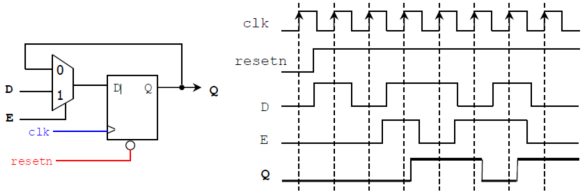
Parallel Access Shift Register

- This is a shift register in which we can write data on the flip flops in parallel. $s\_l = 0 \rightarrow$ shifting operation, $s\_l = 1 \rightarrow$ parallel load. The figure below shows a 4-bit parallel access right shift register.



Adding an Enable Input to Flip Flops

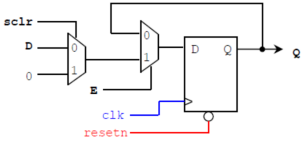
- In many instances, it is very useful to have a signal that controls whether the value of the flip flop is kept.
- The following circuit represent a flip flop with synchronous enable.
  - **E = '0'**: the flip flop keeps its value.
  - **E = '1'**: the flip flop captures the value of the input D
- We can thus create n-bit registers and n-bit shift registers with enable. Here, all the flip flops share the same enable input.
- Excitation equation: $Q_{t+1} \leftarrow \bar{E}Q_t + ED$

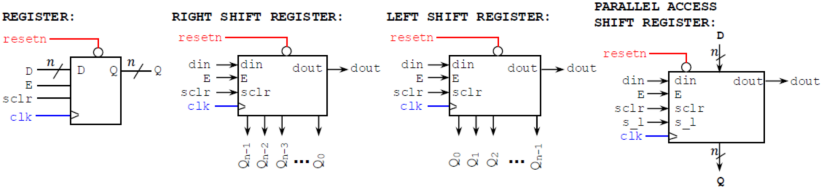


Adding a Synchronous clear input to Flip Flops

- In many instances, it is very useful to have a signal that can clear the value of the flip flop but only on the rising (or falling edges): synchronous clear (sclr).
- Typically, all synchronous signals are validated by enable. For example, for a D flip flop, the table show how the output state Q changes (on the rising edge):

E	*sclr*	*Q* (output state)
0	X	$Q \leftarrow Q$
1	0	$Q \leftarrow D$
1	1	$Q \leftarrow 0$



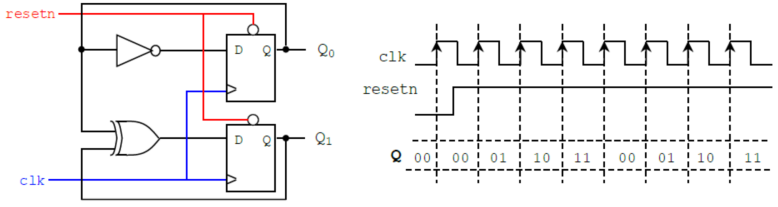


Synchronous Counters

Counters

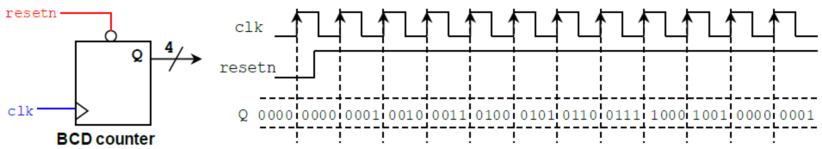
- Counters are useful for: counting the number of occurrences of a certain event, generate time intervals for task control, track elapsed time between two events, etc. Counters are made of flip flops and combinatorial logic.
- Synchronous counters change their output on the clock edge (rising or falling). Each flip flop shares the same clock input signal. If the initial count is zero, each flip flop shares the resetn input signal.

Binary counter: An n-bit counter counts from 0 to $2^n - 1$.



Counters

Modulus counter: A counter *modulo — N* counts from 0 to N-1.
Special case: BCD (or decade) counter: Counts from 0 to 9.

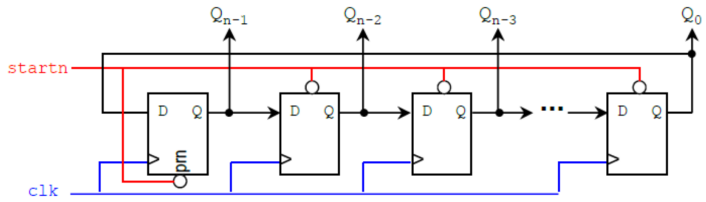


- **Up/down counter:** Counts both up and down, under command of a control input.
- **Parallel load counter:** The count can be given an arbitrary value.
- **Counter with enable:** If enable = 0, the count stops. If enable = 1, the counter counts. This is usually done by connecting the enable inputs of the flip flops to a single enable.

Counters

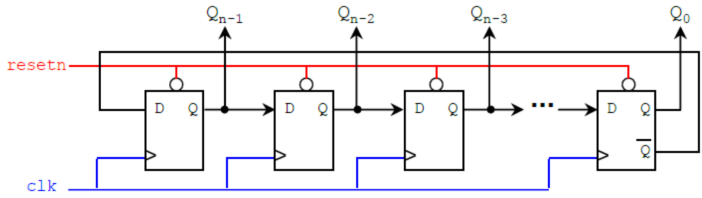
Ring counter: Also called one-hot counter (only one bit is 1 at a time).

- It can be constructed using a shift register. The output of the last stage is fed back to the input to the first stage, which creates a ring-like structure.
- The asynchronous signal startn sets the initial count to 100…000 (first bit set to 1).
- Example (4-bits): 1000, 0100, 0010, 0001, 1000, …



Johnson counter: Also called twisted ring counter.

- It can be constructed using a shift register, where the $\bar{Q}$ output of the last flip flop is fed back to the first stage.
- The result is a counter where only a single bit has a different value for two consecutive counts.
- All the flip flops share the asynchronous signal 'resetn', which sets the initial count to 000…000.
- Example (4 bits): 0000, 1000, 1100, 1110, 1111, 0111, 0011, 0001, 0000, …



Synchronous versus asynchronous circuits

- **Globally synchronous circuit (or simply synchronous circuit)**.
  - Uses FFs as memory elements, and all FFs are controlled (i.e., synchronized) by a single global clock signal.
  - Synchronous design is the most important methodology used to design and develop large, complex digital systems.
  - It not only facilitates the synthesis but also simplifies the verification, testing, and prototyping process. Our discussion is focused mainly on this type of circuit.

- **Globally asynchronous locally synchronous circuit.**.
  - Sometimes physical constraints, such as the distance between components, prevent the distribution of a single clock signal.
  - In this case, a system may be divided into several smaller subsystems.
  - The subsystems are synchronous internally utilizing its own clock, operation between the subsystems is asynchronous.
  - We need special interface circuits between the subsystems to ensure correct operation.

- **Globally asynchronous circuit.**.
  - Does not use a clock signal to coordinate the memory operation.
  - The state of a memory element changes independently.

Types of synchronous circuits

- **Regular sequential circuit,**.
  - The state representation and state transitions have a simple, regular pattern, as in a counter and a shift register. *(What we covered so far)*

- **Random sequential circuit.**.
  - The state transitions are more complicated and there is no special relation between the states and their binary representations.
  - The next-state logic must be constructed from scratch (i.e., by random logic).
  - This kind of circuit is known as a **finite stare machine (FSM)**.

- **Combined sequential circuit.**.
  - Consists of both a regular sequential circuit and an FSM.
  - The FSM is used to control operation of the regular sequential circuit.
  - This kind of circuit is based on the register transfer methodology and is sometimes known as **finite state machine with data path (FSMD)**.

VHDL Code

"Talk is cheap. Show me the code."

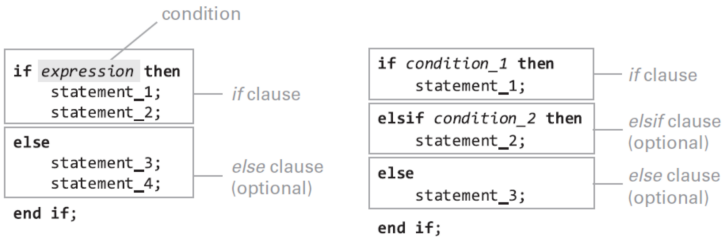
— Linus Torvalds

VHDL Sequential Statements

Control Structures

- Idealy, sequential statements are executed from **top to bottom in the order** in which they appear in the source code.
- However, most interesting algorithms require variations in this order of execution.
- These are called **control flow statement or control structures.**
- These include:
  - `if`
  - `case`
  - `loop`

The *if* Statement

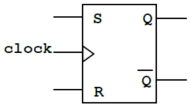


(a) A simple *if-else* statement.

(b) An *if-elsif-else* statement.

- The `if` statement can only be used in regions where there is an explicit order of execution — in sequential code.

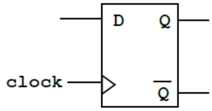
VHDL Code: SR Flip Flop



clock	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

```vhdl
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 Behavioral of ff_sr is
    signal qt, qnt: std_logic;

begin
    process (s, r, clock)
    begin
        if (clock'event and clock='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 Behavioral;
```

VHDL Code: D Flip Flop



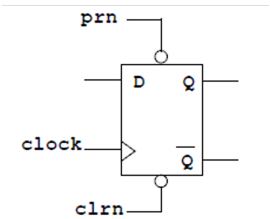
clock	D	Q_{t+1}
↑	0	0
↑	1	1

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

entity d_ff is
    port(
        D, clock : in std_logic;
        Q        : out std_logic
    );
end d_ff;

architecture Behavioral of d_ff is
begin
    dff: process (clock)
    begin
        if (rising_edge(clock)) then
-- or   if (clock'event and clock='1') then
            Q <= D;
        end if;
    end process;
end Behavioral;
```

D Flip Flop with asynchronous inputs: `clrn`, `prn`

- 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.

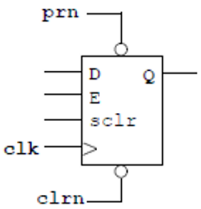
clrn = '0' ➔ q = '0'
prn = '0' ➔ q = '1'




```
1  architecture bhv of ff_dp is
2      signal qt,qnt: std_logic;
3
4  begin
5      process (clrn,prn,clock)
6      begin
7          if clrn = '0' then
8              qt <= '0';
9          elsif prn = '0' then
10             qt <= '1';
11         elsif (clock'event and clock='1') then
12             qt <= d;
13         end if;
14     end process;
15
16     q <= qt;
17     qn <= not(qt);
18 end;
```

D Flip Flop with enable and synchronous clear

- This is a design that includes asynchronous inputs `(prn, clrn)` and synchronous inputs `(E, sclr, D)` .



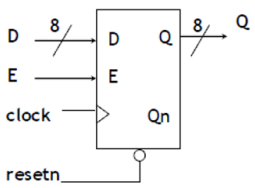

```vhdl
architecture bhv of dffes is
begin
    process (clrn, prn, clock)
    begin
        -- 1. Asynchronous active-low set/reset
        if clrn = '0' then
            q <= '0';
        elsif prn = '0' then
            q <= '1';

        -- 2. Synchronous logic on rising edge
        elsif (clock'event and clock = '1') then
            if sclr = '1' then
                q <= '0';      -- Sync clear (priority)
            elsif E = '1' then
                q <= d;        -- Sync enable
            end if;
            -- if sclr='0' and E='0', q holds its value
        end if;
    end process;
end architecture bhv;
```

NOTE: Key points about the sensitivity list

- **Rule 1:** The list must include any signal that causes an immediate, asynchronous change like `(clrn, prn)` and the `clock` itself.
- **Rule 2:** The list must not include synchronous signals `(sclr, E, d)` , because they should only be checked when the clock edge occurs.

8-bit Register with Enable & Asynchronous Reset

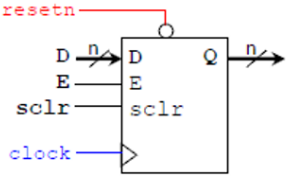
- This is an 8-bit register design with parallel load/output.
- Features: Asynchronous active-low reset (`resetn`), synchronous enable (`E`).




```vhdl

entity reg8_en_areset is
    port (
        D       : in  std_logic_vector(7 downto 0);
        E       : in  std_logic;
        clock   : in  std_logic;
        resetn  : in  std_logic;
        Q       : out std_logic_vector(7 downto 0)
    );
end entity reg8_en_areset;

architecture rtl of reg8_en_areset is
    signal q_reg : std_logic_vector(7 downto 0);
begin
    process (clock, resetn)
    begin
        -- Asynchronous reset (highest priority)
        if resetn = '0' then
            q_reg <= (others => '0');
        -- Rising edge of the clock
        elsif rising_edge(clock) then
            -- Synchronous load with enable
            if E = '1' then
                q_reg <= D;
            end if;
        end if;
    end process;

    Q <= q_reg; -- Output assignment
end architecture rtl;
```

N-bit Register with Enable, Sync & Async Clear

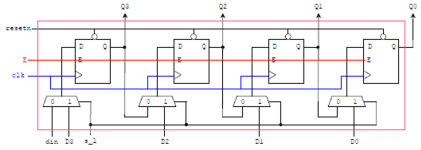
- A generic N-bit register, configurable via a `generic` parameter.
- Asynchronous Reset: `resetn` (active-low). and Synchronous Controls: `sclr` (active-high) and `E` (active-high).

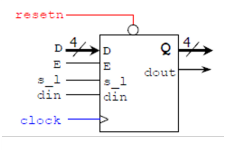



```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity regN_en_sclr_areset is
    generic ( N : positive := 8 );
    port (
        resetn, clock  : in  std_logic;
        D       : in  std_logic_vector(N-1 downto 0);
        E, sclr : in  std_logic;
        Q       : out std_logic_vector(N-1 downto 0)
    );
end entity regN_en_sclr_areset;

architecture rtl of regN_en_sclr_areset is
    signal q_reg : std_logic_vector(N-1 downto 0);
begin
    process (clock, resetn)
    begin
        if resetn = '0' then          -- Async reset
            q_reg <= (others => '0');
        elsif rising_edge(clock) then
            if sclr = '1' then        -- Sync clear
                q_reg <= (others => '0');
            elsif E = '1' then        -- Sync enable
                q_reg <= D;
            end if;
        end if;
    end process;
    Q <= q_reg; -- Concurrent output assignment
end architecture rtl;
```

Parallel Access Shift Register: 4-bit Right with Enable

- This design implements a 4-bit PISO (Parallel-In, Serial-Out) right shift register.
- Key controls: Asynchronous reset (`resetn`), synchronous enable (`E`), and shift/load select (`s_l`).



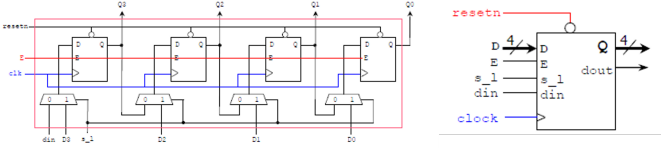


Version 1: Explicit Bit Assignments

```
1  architecture bhv of pashreg4_right is
2      signal Qt: std_logic_vector (3 downto 0);
3  begin
4      process (resetn, clock, s_l, E) -- Note: Sensitivity
           issue
5      begin
6          if resetn = '0' then Qt <= "0000";
7          elsif (clock'event and clock = '1') then
8              if E = '1' then
9                  if s_l = '1' then Qt <= D;
10                 else -- Shift Right
11                     Qt(0) <= Qt(1); Qt(1) <= Qt(2);
12                     Qt(2) <= Qt(3); Qt(3) <= din;
13                 end if;
14             end if;
15         end if;
16     end process;
17     Q <= Qt; dout <= Qt(0);
18 end architecture bhv;
```

Parallel Access Shift Register: 4-bit Right with Enable

- This design implements a 4-bit PISO (Parallel-In, Serial-Out) right shift register.
- Key controls: Asynchronous reset (`resetn`), synchronous enable (`E`), and shift/load select (`s_l`).



Version 1: Explicit Bit Assignments

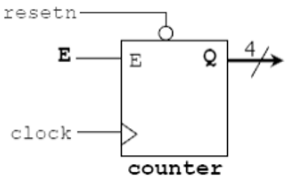
```
1   architecture bhv of pashreg4_right is
2       signal Qt: std_logic_vector (3 downto 0);
3   begin
4       process (resetn, clock, s_l, E) -- Note: Sensitivity
              issue
5       begin
6           if resetn = '0' then Qt <= "0000";
7           elsif (clock'event and clock = '1') then
8               if E = '1' then
9                   if s_l = '1' then Qt <= D;
10                  else -- Shift Right
11                      Qt(0) <= Qt(1); Qt(1) <= Qt(2);
12                      Qt(2) <= Qt(3); Qt(3) <= din;
13                  end if;
14              end if;
15          end if;
16      end process;
17      Q <= Qt; dout <= Qt(0);
18  end architecture bhv;
```

Version 2: With a `for` loop (more scalable)

```
1       process (resetn, clock) -- Corrected sensitivity list
2       begin
3           if resetn = '0' then
4               Qt <= (others => '0');
5           elsif rising_edge(clock) then
6               if E = '1' then
7                   if s_l = '1' then -- Load
8                       Qt <= D;
9                   else -- Shift Right
10                      gg: for i in 0 to 2 loop
11                          Qt(i) <= Qt(i+1);
12                      end loop;
13                      Qt(3) <= din; -- Serial In
14                  end if;
15              end if;
16          end if;
17      end process;
18      Q <= Qt;
19      dout <= Qt(0);
20  end architecture rtl;
```

4-bit Binary Counter with Enable & Async. Reset

- Synchronous enable signal (`E`): Only considered on the rising clock edge.
- This is a **Modulo-16** counter (it has 16 states, 0 to 15). If the counter is at 15 (`1111`), the next enabled count wraps around to 0.




```vhdl
-- Note: Using integer for ports is non-standard.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

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

architecture bhv of my_count4b_E is
    signal Qt: integer range 0 to 15;
begin
    -- Sensitivity list corrected:
    -- Only async reset and clock
    process (resetn, clock)
    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; -- Concurrent output
end architecture bhv;
```

Best Practice: Why Avoid `INTEGER` on Ports?

Why `std_logic_vector` is the Industry Standard

1. **Physical Reality:** Hardware components communicate via wires (bits). An `integer` is abstract; a `std_logic_vector` accurately represents the physical bus.
2. **Synthesis Ambiguity:** Integers are 32-bits by default. Passing them between modules can lead to width mismatches if ranges aren't strictly defined.
3. **Interoperability:** External modules (memories, display drivers) almost always expect `std_logic_vector`. Using integers requires constant conversion functions at boundaries.
4. **Logic States:** Integers cannot represent High-Impedance ('Z') or Undefined ('X') states, which are critical for debugging and tri-state logic.

Design Rule

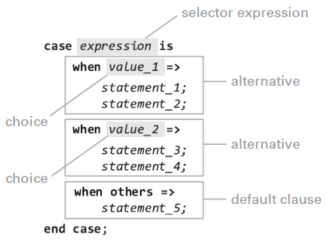
Ports = `std_logic_vector` (Interface) vs. **Internal Signals** = `unsigned/signed` (Arithmetic)

The Professional Approach:

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL; -- Industry Standard

entity my_count4b_E is
    port (
        resetn : in  std_logic;
        clock  : in  std_logic;
        E      : in  std_logic;
        -- Port is a vector of bits (Physical)
        Q      : out std_logic_vector(3 downto 0)
    );
end entity my_count4b_E;

architecture rtl of my_count4b_E is
    -- Internal math uses 'unsigned'
    signal Qt: unsigned(3 downto 0);
begin
    process (resetn, clock)
    begin
        if resetn = '0' then
            Qt <= (others => '0');
        elsif rising_edge(clock) then
            if E = '1' then
                Qt <= Qt + 1; -- Native Math
            end if;
        end if;
    end process;

    -- Cast to vector only at the boundary
    Q <= std_logic_vector(Qt);
end architecture rtl;
```

The `case` Statement

- The `case` statement selects one sequence of statements for execution based on the value of a single **selector expression**.
- It offers a structured and clear alternative to complex `if-elsif` chains.
- The selector expression is evaluated **only once**, then its value is matched against `when` clauses (alternatives).
- All possible values of the selector expression **must be covered** (e.g., using `when others`).



(a) Elements of a *case* statement.

```
case count_value is
    when 0 | 1 | 9 =>
        statement_1;
    when 2 to 5 =>
        statement_2;
    when others =>
        statement_3;
end case;
```

(b) Covering multiple values per alternative.

NOTE: Restrictions

- **Data Types:** The selector expression must produce a discrete type (e.g., `integer`, `std_logic_vector`, enumeration types).
- **Static Choices:** Choices (`value_1`, `2 to 5`) must be locally static (known at analysis time), not variables.

Example: 4-to-1 Multiplexer using `case`

- The `case` statement is ideal for implementing logic defined by a Truth Table.
- Here, a 2-bit signal `sel` selects one of four inputs to drive the output `y`.

Logic Function

Sel (2-bit)	Output (y)
"00"	a
"01"	b
"10"	c
"11"	d
others	'o'

Note: Since `std_logic` has 9 states ('X', 'Z', etc.), "11" is not the last possible value. We must handle the rest.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity mux4 is
    port (
        sel : in  std_logic_vector(1 downto 0);
        a, b, c, d : in  std_logic;
        y   : out std_logic
    );
end entity mux4;

architecture beh of mux4 is
begin
    -- Case statements must be inside a process
    process (sel, a, b, c, d)
    begin
        case sel is
            when "00" => y <= a;
            when "01" => y <= b;
            when "10" => y <= c;
            when "11" => y <= d;

            -- Mandatory coverage for std_logic
            when others => y <= '0';
        end case;
    end process;
end architecture beh;
```

Loops

- **Loop:** A control structure that allows repeated execution of a sequence of statements.
- Useful in both **behavioral** (testbenches, algorithms) and **synthesizable** (hardware replication) code.
- VHDL has one loop statement with three configurations:
  1. **Simple Loop:** Infinite loop (unless exited manually).
  2. **While Loop:** Condition-controlled (executes while condition is true).
  3. **For Loop:** Count-controlled (executes a fixed number of times).

```
variable i: integer := 0;
...
loop
    statements;
    i := i + 1;
    exit when i = 10;
end loop;
```

(a) A simple loop.

```
variable i: integer := 0;
...
while i < 10 loop
    statements;
    i := i + 1;
end loop;
```

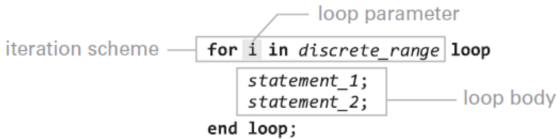
(b) A *while* loop.

```
for i in 1 to 10 loop
    statements;
end loop;
```

(c) A *for* loop.

The `for` Loop

- Repeats a sequence of statements for a pre-determined number of iterations.
- **Syntax:** `for <parameter> in <range> loop ... end loop;`



NOTE: Loop Parameter Rules

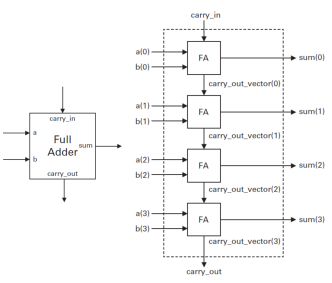
- **Implicit Declaration:** The loop parameter (e.g., `i`) is automatically declared. It exists *only* inside the loop.
- **Constant:** Inside the loop, it is read-only. You cannot modify it (e.g., `i := i + 1` is illegal).
- **Range:** Can be numeric (`0 to 3`) or enumerated (e.g., `for state in state_type`).

Synthesizable Hardware Use:

`for` loops are powerful for generating replicated hardware structures (e.g., shift registers, arrays of components).

Example: Behavioral Ripple Carry Adder

- **Goal:** Describe a generic N-bit adder without manually instantiating N full adders.
- **Key Concept:** Use a **variable** for the Carry bit. Variables update *immediately*, allowing the carry to "ripple" from bit 0 to bit 3 within a single execution of the process.



Hardware Inference:
The synthesis tool **"unrolls"** this loop to create a physical chain of logic gates (Full Adders).

```vhdl
entity ripple_adder is
    port ( A, B : in  std_logic_vector(3 downto 0);
           Cin  : in  std_logic;
           S    : out std_logic_vector(3 downto 0);
           Cout : out std_logic );
end entity ripple_adder;

architecture beh of ripple_adder is
begin
    process(A, B, Cin)
        -- Variable for immediate carry propagation
        variable v_carry : std_logic;
    begin
        v_carry := Cin; -- Initialize

        for i in 0 to 3 loop
            -- Calculate Sum for bit i
            S(i) <= A(i) xor B(i) xor v_carry;

            -- Update Carry for next iteration (i+1)
            v_carry := (A(i) and B(i)) or
                       (v_carry and (A(i) xor B(i)));
        end loop;

        Cout <= v_carry; -- Final Carry Out
    end process;
end architecture beh;
```

The `wait` Statement

- The `wait` statement suspends the execution of a process until a specific condition is met.
- **Important:** A process with a sensitivity list (e.g., `process(clk)`) cannot contain `wait` statements. They are mutually exclusive.
- The condition may include three optional clauses:
  1. **Sensitivity List (`on`):** Resumes when an event occurs on a signal.
  2. **Condition (`until`):** Resumes when an expression evaluates to `true`.
  3. **Timeout (`for`):** Resumes after a specified time elapses.

General Syntax:

```
wait on <signal_list> until <boolean_expr> for <time_expr>;
```

The Unconditional Wait:

- `wait;` suspends the process indefinitely (forever).
- Often used in testbenches to stop stimulus generation.

The `on`, `until`, and `for` Clauses

1. `wait on` **(Sensitivity)**
   - Explicitly defines signals that trigger the process to resume.
   - Example: `wait on clock, reset;`
   - Resumes whenever `clock` or `reset` changes value.
2. `wait for` **(Timeout)**
   - Suspends for a specific time interval.
   - Example: `wait for 10 ns;`
   - **Note:** Meaningless for synthesis; ignored by some tools or generates errors.
3. `wait until` **(Condition)**
   - Suspends until the expression becomes `true`.
   - **Implicit Sensitivity:** If no `on` clause is present, the sensitivity list is implicitly composed of all signals in the condition.
   - The condition is evaluated *only* when a signal in the sensitivity list changes.

```
1  -- Equivalent statements:
2  wait on clk until clk='1';
3  wait until clk='1';
```

The `wait` Statement: Synthesis vs. Simulation

1. Simulation (Testbenches)

- `wait for` is widely used to model delays and generate stimuli.
- Multiple `wait` statements allow complex, sequential test sequences.

```
1  -- Testbench Stimulus
2  process begin
3      rst <= '1';
4      wait for 10 ns; -- OK in Sim
5      rst <= '0';
6      wait;
7  end process;
```

2. Synthesis (Hardware)

- **Restriction:** `wait for` is **NOT** synthesizable. Hardware cannot produce arbitrary delays without counters/FIFOs.
- **Restriction:** Multiple `wait` statements imply an FSM. If used, they *must* all check the **same clock edge** to keep the FSM synchronous.

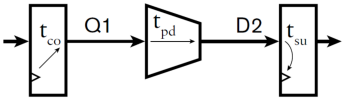
NOTE: Best Practice for Synthesis

- **Single Wait:** Your best chance for correct synthesis is using a **single** `wait` statement per process.
- **Location:** Place it at the **bottom** of the process. This ensures signal values are calculated at least once before suspending, behaving exactly like a sensitivity list.

```
1  process -- No sensitivity list allowed here!
2  begin
3      -- ... combinational logic ...
4      wait until clock = '1'; -- Implicit sensitivity to clock
5  end process;
```

Synchronous Timing Analysis: The Model

- In synchronous design, we treat the system as a collection of registers connected by combinational logic.
- **Goal:** Determine the maximum speed (clock frequency) the circuit can operate reliably.
- **The Path:** We analyze the path from a source register (`Q1`) through combinational logic to a destination register (`D2`).

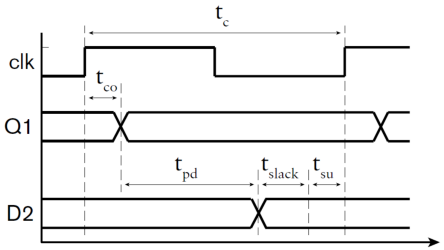


Key Timing Parameters:

- T_{co} **(Clock-to-output):** Delay from clock edge to data appearing at Q output.
- T_{comb} **or** T_{pd} **(Propagation Delay):** Delay through the combinational logic.
- T_{setup}**:** Data must be stable *before* the clock edge.
- T_{hold}**:** Data must be stable *after* the clock edge.

Constraint 1: Setup Time (T_{setup})

- **The Rule:** The new data must arrive at the destination register (`D2`) *before* the setup window begins.
- If the data arrives too late, the register samples unstable values (timing violation).



Max Frequency Calculation

The clock period (T_c) must be long enough to accommodate all delays:

$$T_c \geq T_{co} + T_{comb(max)} + T_{setup}$$

$$f_{max} = \frac{1}{T_{min}} = \frac{1}{T_{cq} + T_{comb(max)} + T_{setup}}$$

Constraint 2: Hold Time (T_{hold})

- **The Rule:** The data at `D2` must remains stable for a short time *after* the clock edge.
- **The Risk:** If the logic is too fast (short path), the *new* data from `Q1` might race through and overwrite the *old* data at `D2` while it is still trying to latch it.

Hold Violation Condition:

$$T_{co} + T_{comb(min)} < T_{hold}$$

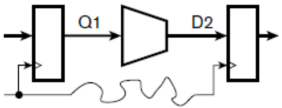
Interpretation: If the fastest path is quicker than the hold requirement, the circuit fails.

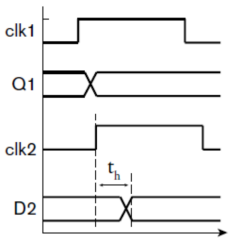
NOTE: Crucial Difference

- **Setup** violations are fixed by **slowing down** the clock.
- **Hold** violations **CANNOT** be fixed by the clock speed. They are fixed by adding delay (buffers) to the data path.

Reality Check: Clock Skew

- Ideally, the clock arrives at all registers simultaneously.
- In reality, wires have delays. The clock might arrive at `Reg1` earlier or later than `Reg2`. This difference is **Clock Skew**.





Impact of Skew:

- **Negative Skew:** Clock arrives at destination *earlier*. Makes **Setup** harder to meet (reduces effective cycle time).
- **Positive Skew:** Clock arrives at destination *later*. Makes **Hold** harder to meet (race condition is more likely).

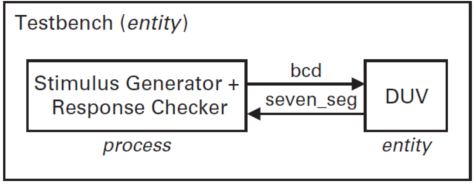
Solution: FPGA/ASIC tools use "Clock Trees" and buffers to minimize skew automatically.

Testbenches for Sequential Circuits

- **Challenge:** Sequential circuits (Registers, Counters, FSMs) require a continuous **Clock** signal to operate.
- A simple linear process (like we used for combinational logic) cannot easily generate a continuous clock while simultaneously applying stimuli.
- **Solution:** We need **Multiple Processes** running in parallel.

The Standard Architecture:

1. **Clock Generator Process:** dedicated to creating the periodic clock signal.
2. **Stimulus Process:** applies reset and inputs, waits for clock edges, and checks outputs.



Clock Generation Strategies

Method 1: The Constant Half-Period Loop (Recommended)

- Simple, robust, and easy to adjust frequency via a constant.

```
constant T_CLK : time := 10 ns; -- 100 MHz Clock
signal clk : std_logic := '0';  -- Initial value is CRITICAL!

-- Clock Generation Process
clk_gen_proc: process
begin
    wait for T_CLK / 2;
    clk <= not clk;
end process clk_gen_proc;
```

Method 2: The `while` Loop (Conditional)

- Useful if you want to stop the clock automatically after simulation ends.

```
clk_gen_proc: process
begin
    while not stop_simulation loop -- boolean flag
        clk <= '0'; wait for T_CLK / 2;
        clk <= '1'; wait for T_CLK / 2;
    end loop;
    wait; -- Suspend forever when loop ends
end process;
```

Applying Stimuli: Reset & Synchronization

1. The Reset Sequence

- Almost every sequential test starts with a Reset Pulse.
- It must be long enough to be recognized (usually $>$ **1** clock cycle).

2. Synchronizing with the Clock

- **Best Practice:** Apply inputs *just after* the active clock edge.
- This avoids race conditions in simulation and mimics realistic Setup/Hold behavior.
- Use `wait until rising_edge(clk);` before assigning new values.

```
stim_proc: process
begin
    -- 1. Apply Reset
    resetn <= '0';
    wait for 20 ns;
    resetn <= '1';

    -- 2. Synchronized Stimulus
    wait until rising_edge(clk); -- Wait for edge
    D <= "1010";                 -- Apply input

    wait until rising_edge(clk); -- Wait for next cycle
    D <= "0011";

    wait; -- Stop stimulus
end process;
```

Template: Complete Sequential Testbench

```vhdl
architecture tb of tb_counter is
    constant T_CLK : time := 10 ns;
    signal clk    : std_logic := '0';
    signal resetn : std_logic := '0';
    signal D      : std_logic_vector(3 downto 0) := (others => '0');
begin
    -- 1. Instantiate DUV
    uut: entity work.my_counter port map (clk, resetn, D);

    -- 2. Clock Generation (Runs forever)
    process begin
        wait for T_CLK / 2;
        clk <= not clk;
    end process;

    -- 3. Stimulus Process (Runs once)
    process begin
        report "Starting Simulation";
        resetn <= '0';  -- Assert Reset
        wait for 2 * T_CLK;
        resetn <= '1';  -- Release Reset

        wait until rising_edge(clk); -- Sync with clock
        D <= "1100";

        wait for 5 * T_CLK; -- Let it run for 5 cycles

        assert (output = expected) report "Error!" severity failure;
        report "Simulation Finished";
        std.env.finish; -- Stop simulation
    end process;
end architecture tb;
```

See you next time !!