The Abdus Salam International Centre for Theoretical Physics
IAEA
unesco
Secretaría Nacional de Ciencia y Tecnología
IUPAP INTERNATIONAL UNION OF PURE AND APPLIED PHYSICS
USAC TRICENTENARIA Universidad de San Carlos de Guatemala

1st Mesoamerican Workshop on Reconfigurable X-ray Scientific Instrumentation for Cultural Heritage

Hardware Description Language (HDL) for Reconfigurable Instrumentation

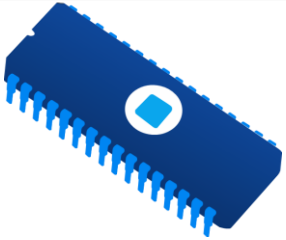
Cristian Sisterna

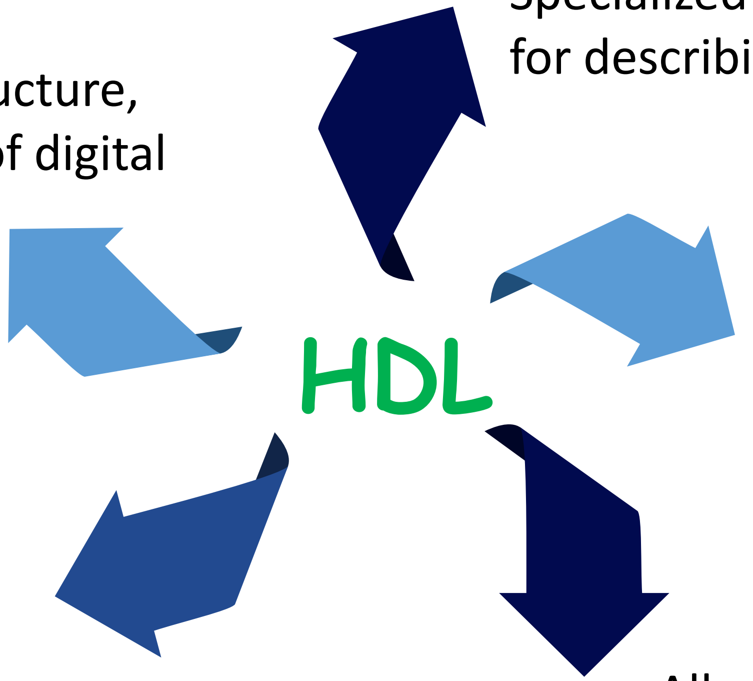
Senior Associate, ICTP-MLAB

Universidad Nacional San Juan- Argentina

Hardware Description Languages

What are them ?





HDL

Specialized computer languages for describing electronic circuits

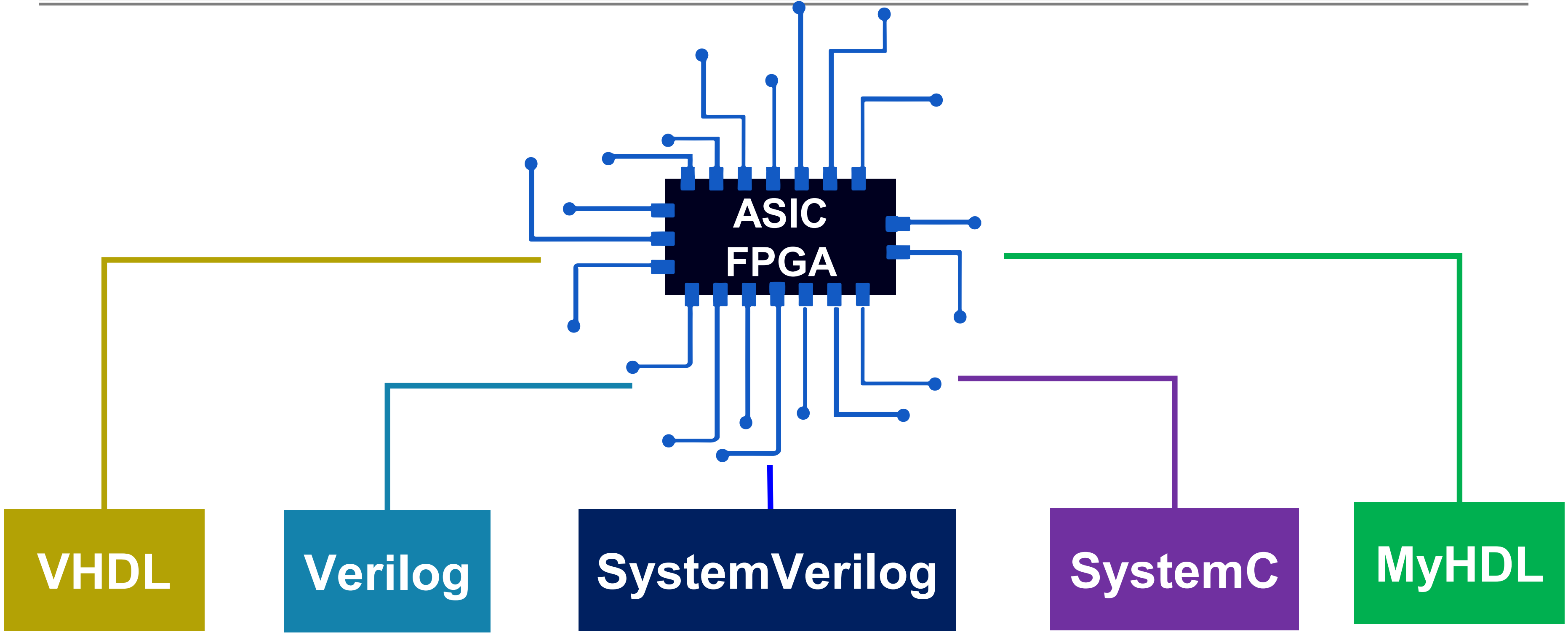
They are fundamental to modern digital design, especially for **creating** ASICs and configuring FPGAs.

Allow designers to work at a higher level of abstraction than schematic capture

VHDL
Verilog
SystemVerilog
Chisel
MyHDL

Used to **describe** structure, behavior, and timing of digital hardware

Hardware Description Languages



ASIC
FPGA

VHDL

Verilog

SystemVerilog

SystemC

MyHDL

- Developed under the U.S. DoD's VHSIC program.
- Strongly typed
- More verbose and formal (resembles Ada)

- Started as a proprietary language.
- Weakly typed.
- Syntax similar to C programming language

Superset of Verilog.

Adds OOP, improved data types, assertions, interfaces.

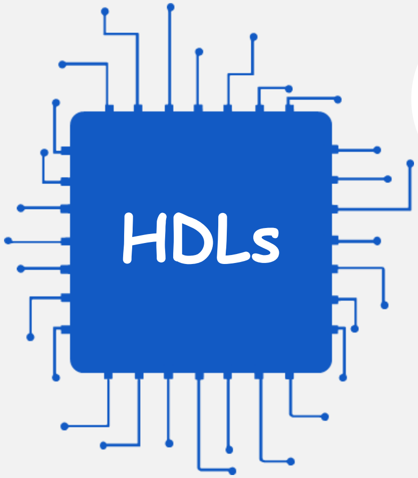
Powerful for system-level verification.

C++ libraries for system-level modeling.

Often used for Transaction-Level Modeling (TLM).

Python based HDL

Why HDLs are keys in Digital Design?



Abstraction

Design complex circuits without low-level details

Verification

Simulate and test designs before fabrication (cost-saving)

Reusability

Create modular and reusable IP (Intellectual Property) blocks.

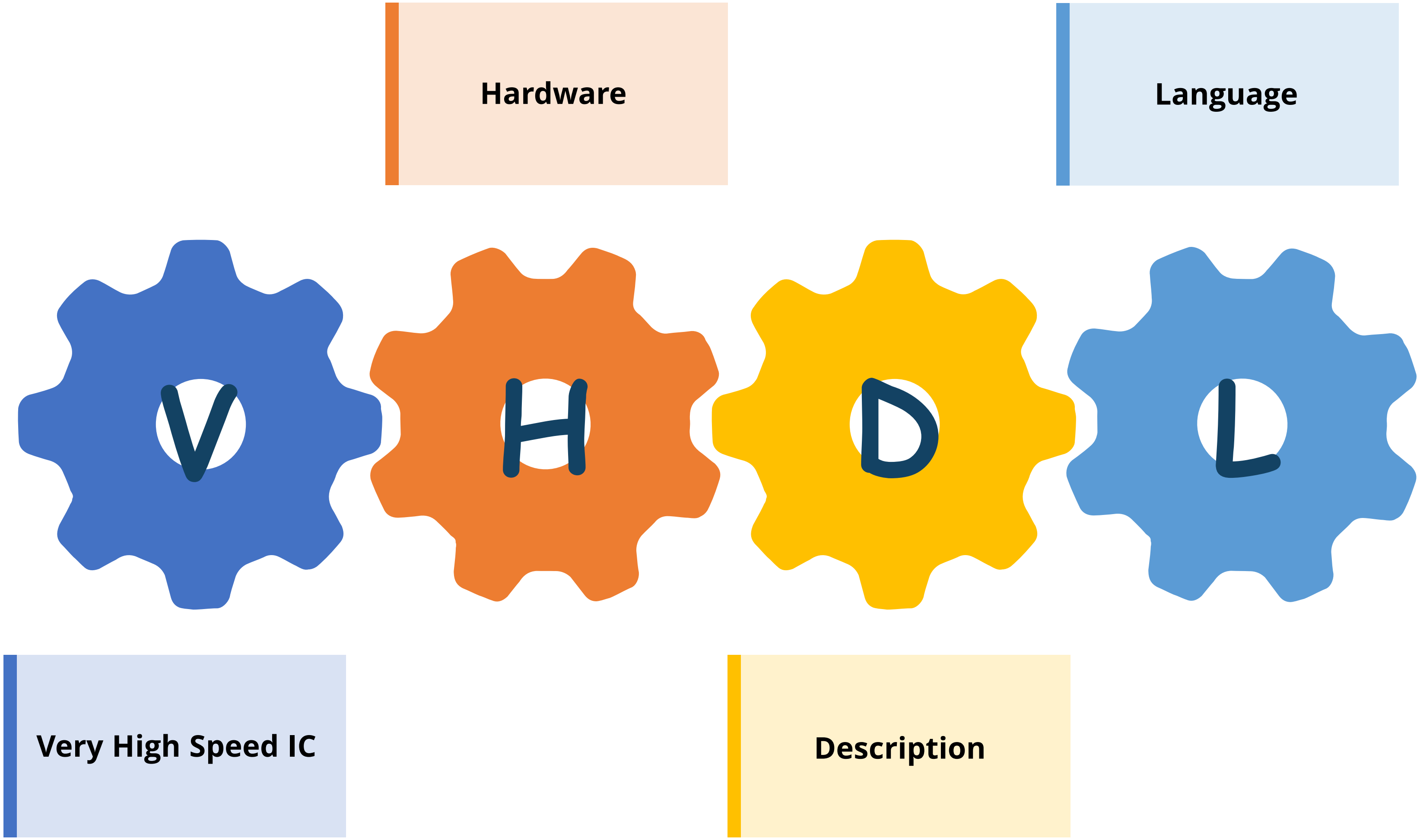
Synthesis

Automatically transform HDL into ASIC/FPGA’s hardware components

Scalability

Enable design of very large and complex systems

Introduction to VHDL



Hardware

Language

Very High Speed IC

Description

Introduction to VHDL

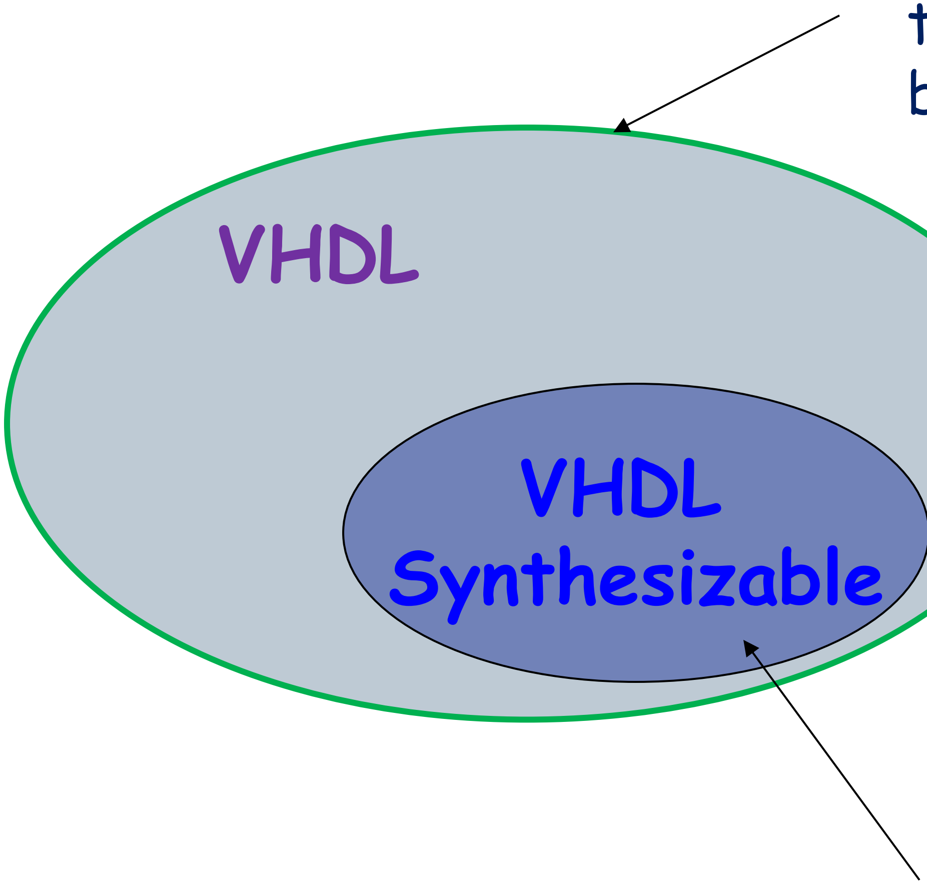
- ☑ High level of abstraction

```
if(reset=‘1’) then
      count <= 0;
elsif(rising_edge(clk)) then
      count <= count+1;
end if;
```

- ☑ Easy to debug
- ☑ Parameterized designs
- ☑ Re-uso
- ☑ IP Cores (free) available

VHDL Synthesis & Simulation

Used to write code to simulate the behavior of a design



Used to implement the design into hardware (for instance in FPGA)

Synthesis versus Simulation

It’s important to understand that VHLD is both, a ***Synthesis*** language and a ***Simulation*** language.

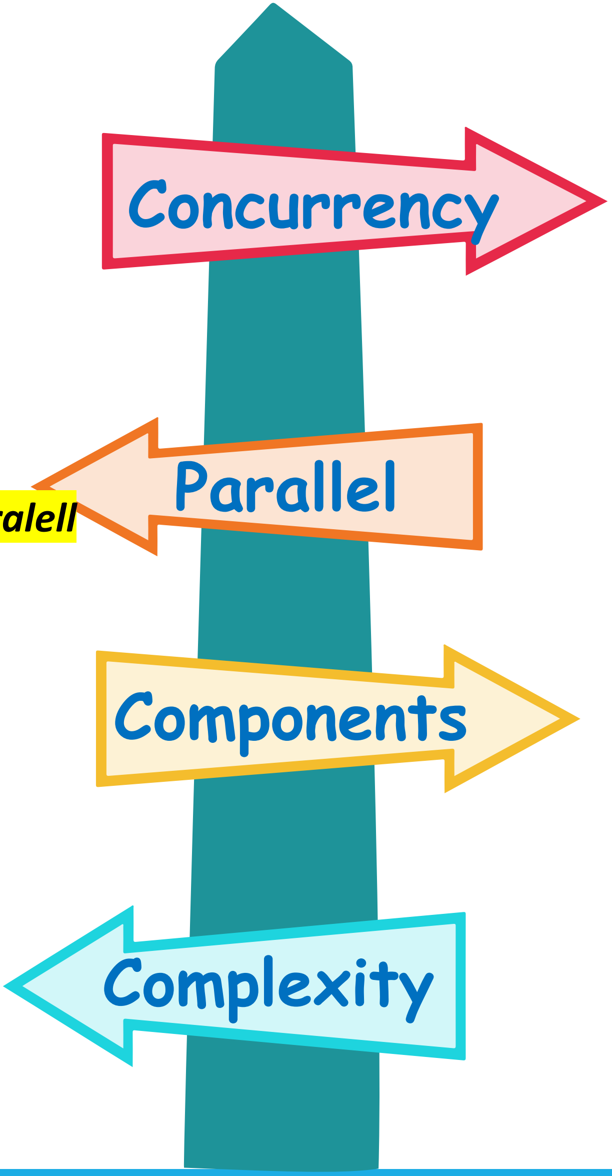
Synthesis

- Small subset of the language is ‘***synthesizable***’, *meaning that it can be translated into logic gates, flip-flops, and other ‘hardware’ components.*
- Every line of VHDL code must have a direct translation into hardware.

Simulation

Another subset of the language include many features for ‘***simulation***’ or ‘***verification***’, features that have **NO meaning in hardware.**

VHDL -> Hardware Description



VHDL is used to **DESCRIBE** the **behavior** and/or structure of a Digital System.

The operations in real systems are executed *concurrently*.

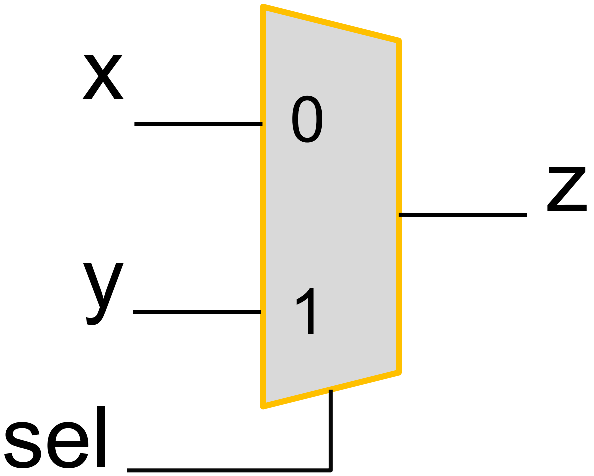
Describing Hardware

Concurrent Code -> Executed in Paralell

The VHDL language describes real systems as a set of **components** that operate *concurrently*.
Each of these **components** is described with concurrent statements.

The complexity of each component may vary from a simple logic gate to a processor

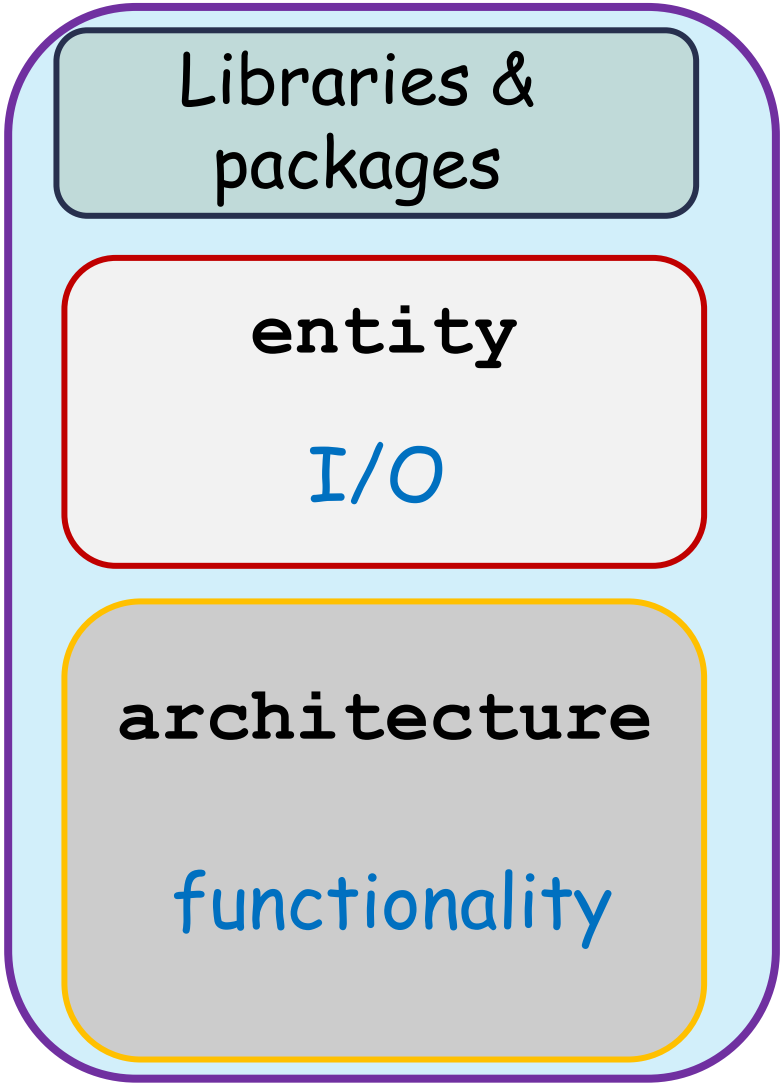
VHDL 'Description' Examples



```
if(sel='1') then
        z <= y;
else
        z <= x;
end if;
```

```
z <= y when sel='1' else x;
```

VHDL - General Component Structure



Libraries and **packages** provides the incorporation of external functions, data types and components to the component to be described

The **entity** defines the I/O ports as well as the name of the component.

Some times a constant(s) is defined (generic) to write parameterized VHDL code

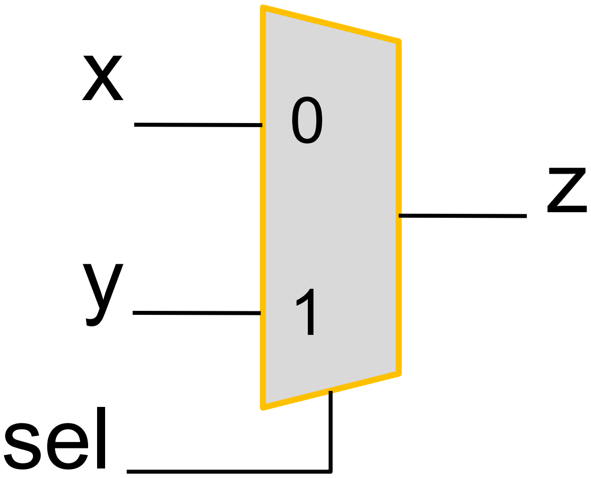
The **architecture** it's where the hardware **behavior** and/or **structure** is described.

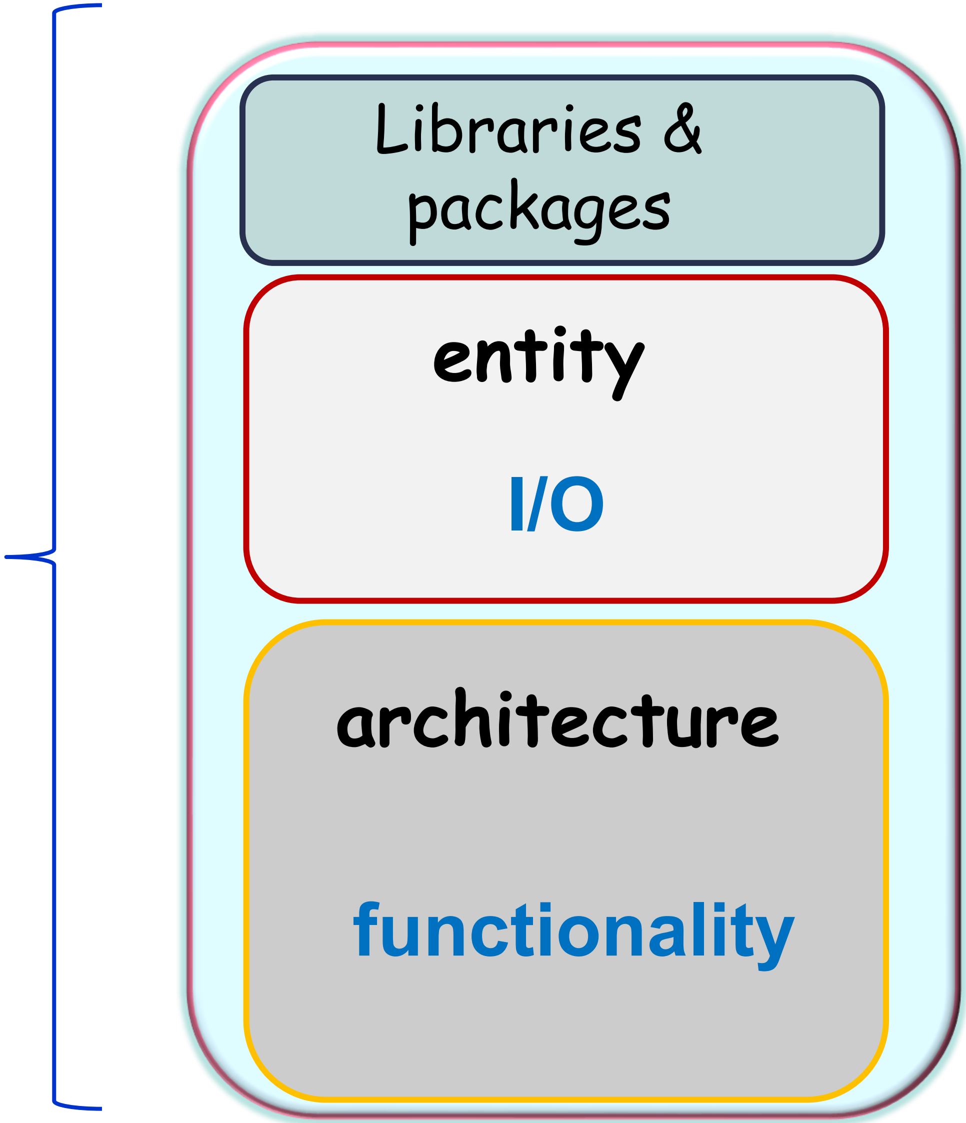
It can have from a couple of lines to thousands lines of VHDL code.

ALL CONCURRENTs !

VHDL - General Component Structure

mux2x1.vhd





Libraries & packages

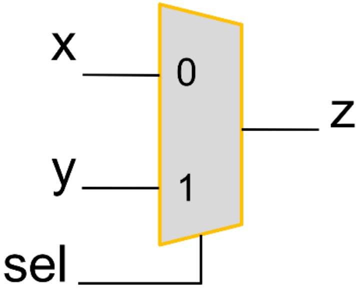
entity

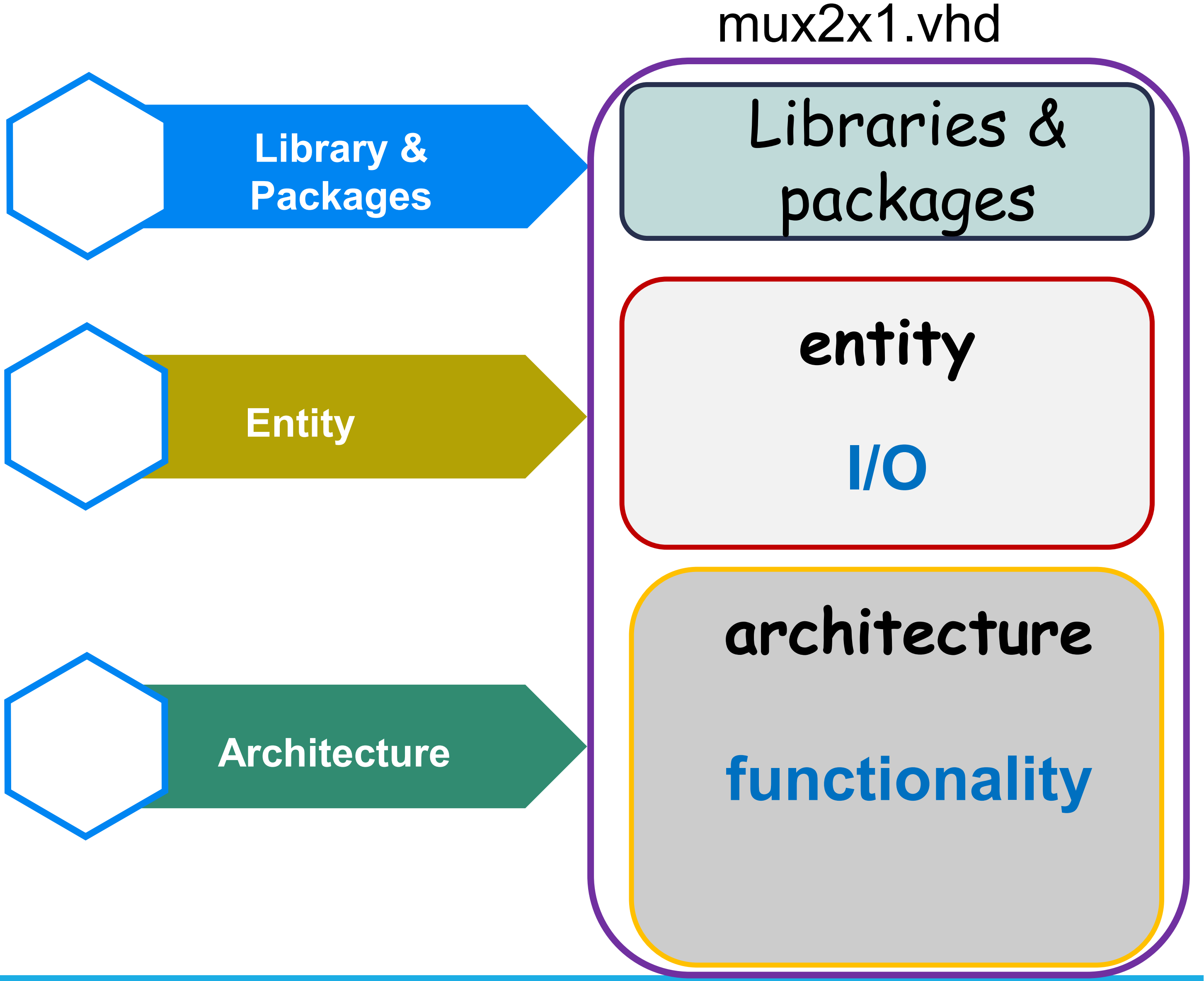
I/O

architecture

functionality

VHDL – General Component Structure





mux2x1.vhd

Library & Packages

Entity

Architecture

Libraries & packages

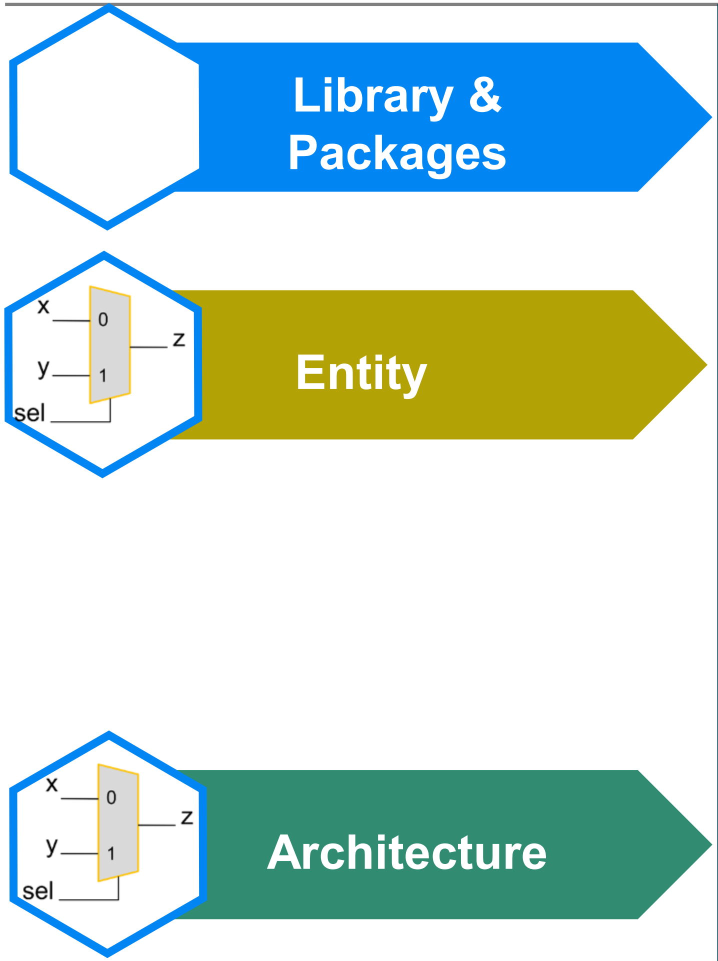
entity

I/O

architecture

functionality

VHDL - General Component Structure



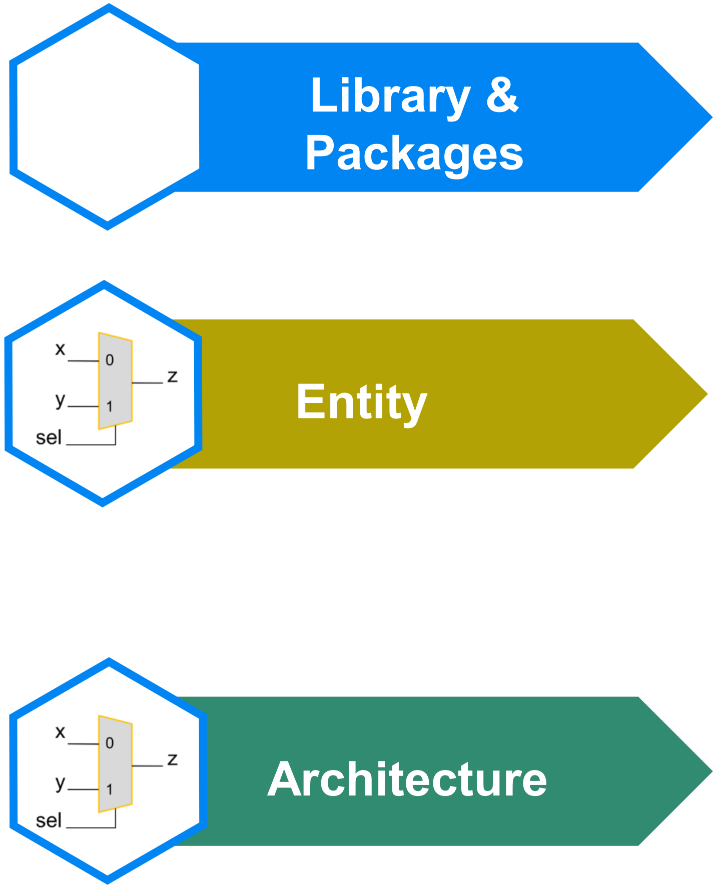
Library & Packages

Entity

Architecture

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

entity mux2x1 is
port(
   x,y,sel: in  std_logic;
    z     : out std_logic);
end mux2x1;

architecture test of mux2x1 is
begin
 process(x,y,sel)
 begin
  if(sel=‘1’) then
        z <= y;
  else
        z <= x;
  end if;
 end process;
end test;
```

VHDL - General Component Structure


Library & Packages

Entity

Architecture

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

entity mux2x1 is
port(
   x,y,sel: in  std_logic;
    z     : out std_logic);
end mux2x1;

architecture test of mux2x1 is
begin

 z <= y when sel=‘1’ else x;

end test;
```

Test Bench

VHDL Code - Is it really Works?

A **test bench** is a crucial part of the hardware design and verification process.

It's a separate VHDL entity that is used to **simulate and verify the functionality of a VHDL design** (known as the "Device Under Test" or "DUT") by providing it with input stimuli and observing its outputs

Purposes of a test bench

Stimuli Generation

It applies a sequence of input values (stimuli) to the DUT's input ports. These stimuli are designed to stimulate all the different operational modes and corner cases of the DUT.

Output Monitoring

It observes the DUT's output ports and compares them against expected values.

Verification

It determines whether the DUT behaves as intended:

Assertions: Checking if certain conditions are met during simulation.

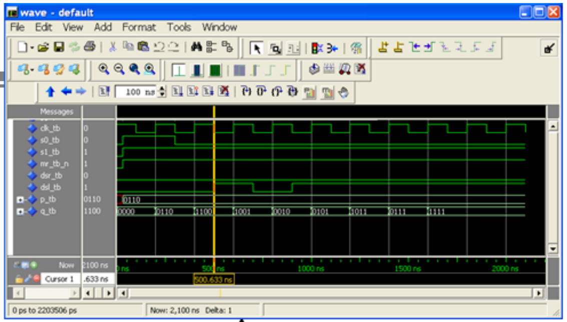
Expected Value Comparison: Directly comparing actual outputs with pre-calculated expected outputs.

Self-checking Test Benches: More advanced test benches that automatically report pass/fail status.

Debugging

When the DUT doesn't behave as expected, the test bench provides a controlled environment to isolate and debug issues.

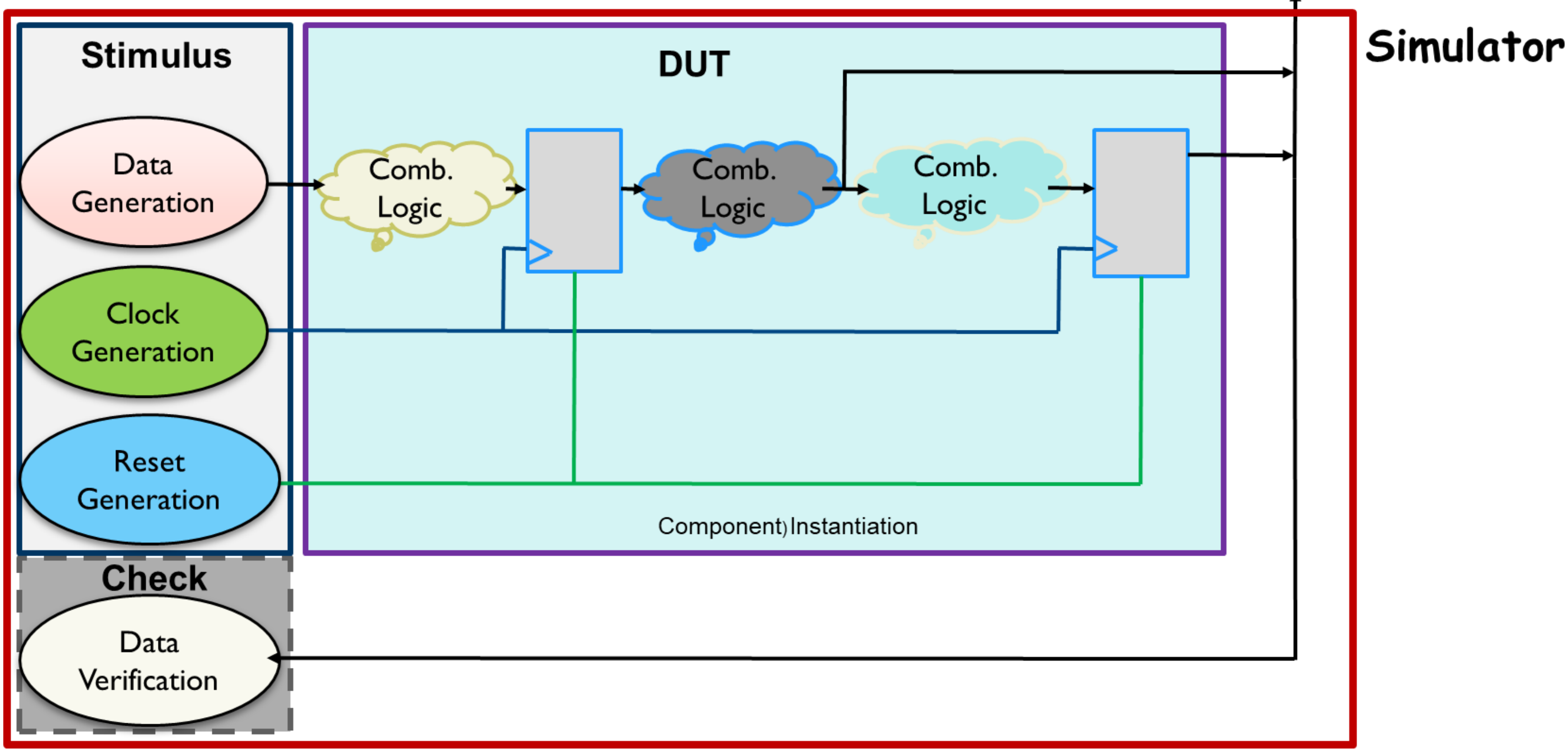
Test Bench - Verification



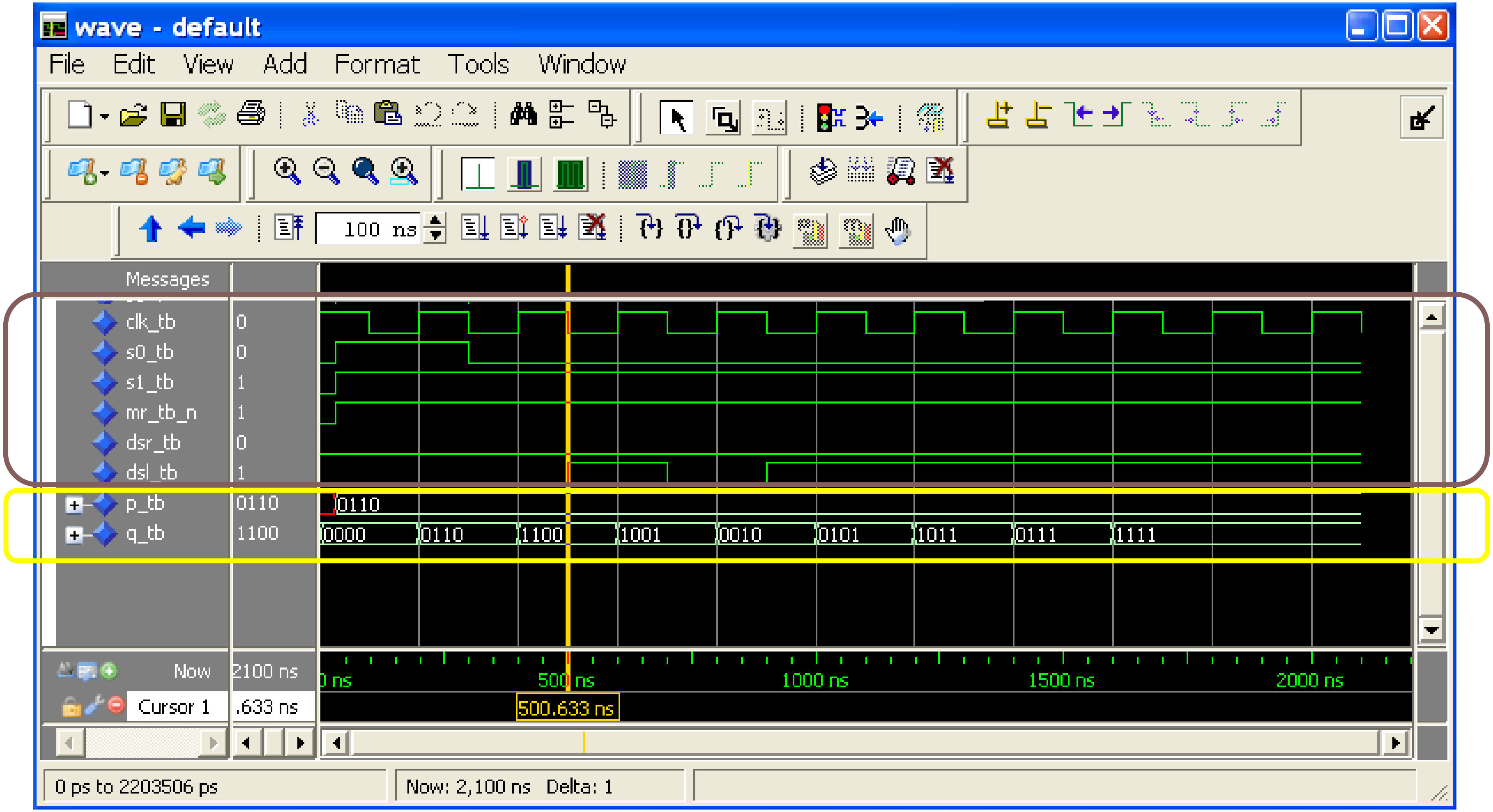
clk_tb 0
s0_tb 0
s1_tb 1
mr_tb_n 1
dsr_tb 0
dsl_tb 1
p_tb 0110
q_tb 1100

0 ps to 2203506 ps
Now: 2,100 ns Delta: 1

Test Bench



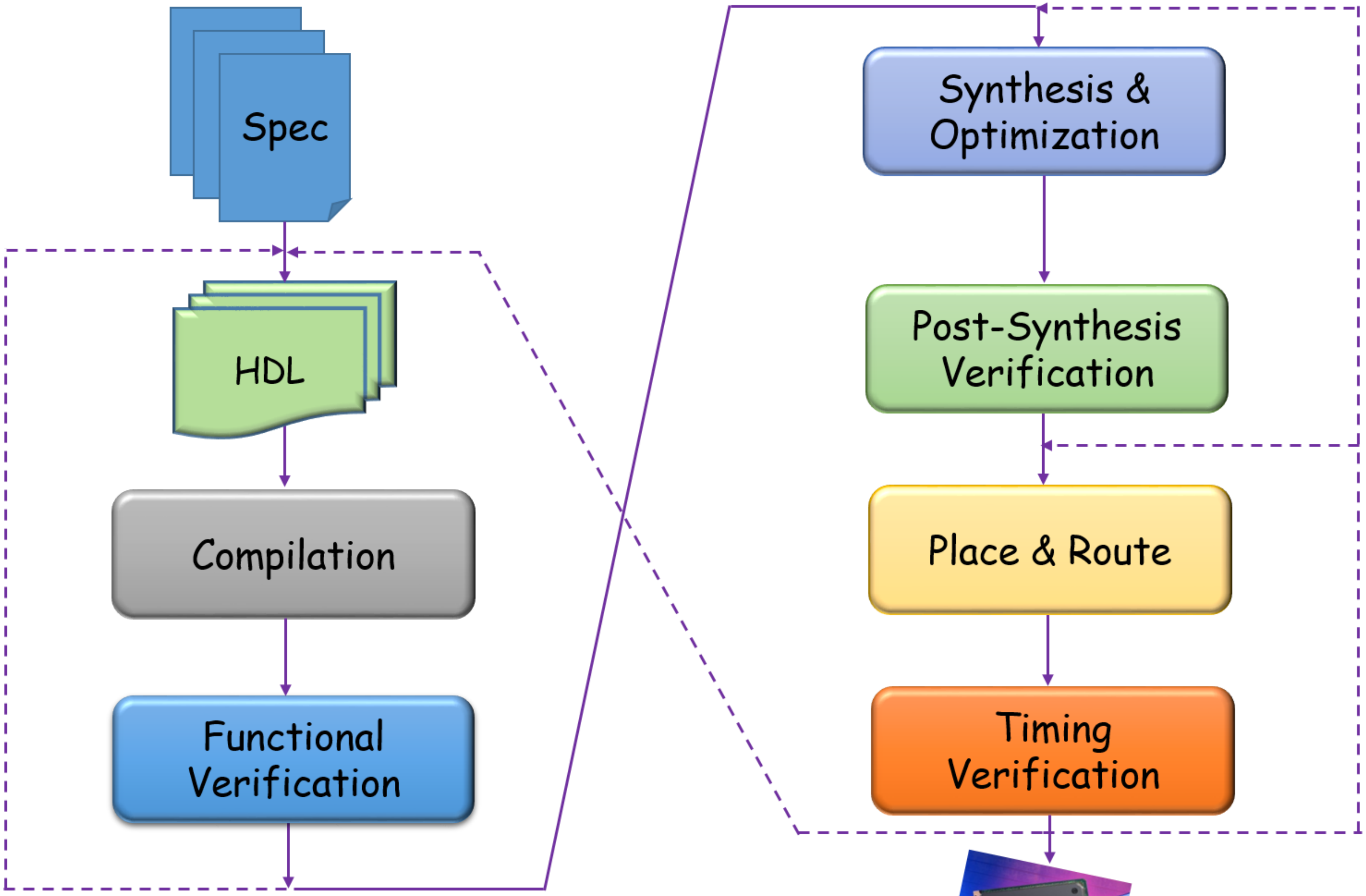
VHDL - Simulation / Verification



```
clk_tb     0
s0_tb      0
s1_tb      1
mr_tb_n    1
dsr_tb     0
dsl_tb     1
p_tb       0110    0110
q_tb       1100    0000  0110  1100  1001  0010  0101  1011  0111  1111

Now        2100 ns     0 ns   500 ns   1000 ns   1500 ns   2000 ns
Cursor 1   .633 ns     500.633 ns

0 ps to 2203506 ps     Now: 2,100 ns  Delta: 1
```

VHDL-FPGA Design Flow

VHDL - FPGA Design Flow



FGPA - Hardware Design Flow

VHDL Code

```
process (clk,rst)
 if (rst = '1')then
   dbus <= (others => '0');
 elsif( rising_edge(clk)) then
   dbus <= data;
 endif;
end process;
```

Timing & Placement Constraints

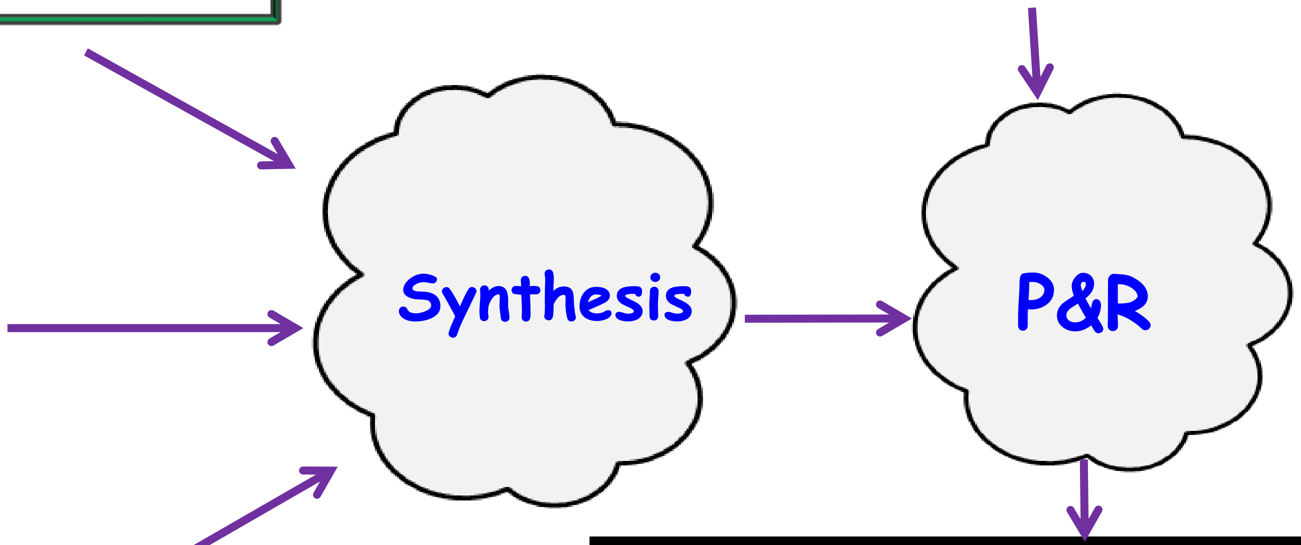
```
Net "CLK" LOC=V10 | IOSTANDARD=LVCMOS33;
Net "CLK" TNM_NET = sys_clk_pin;
TIMESPEC TS_sys_clk_pin = PERIOD sys_clk_pin 100000 kHz;
Net "CS" LOC = T12 | IOSTANDARD = LVCMOS33;
```

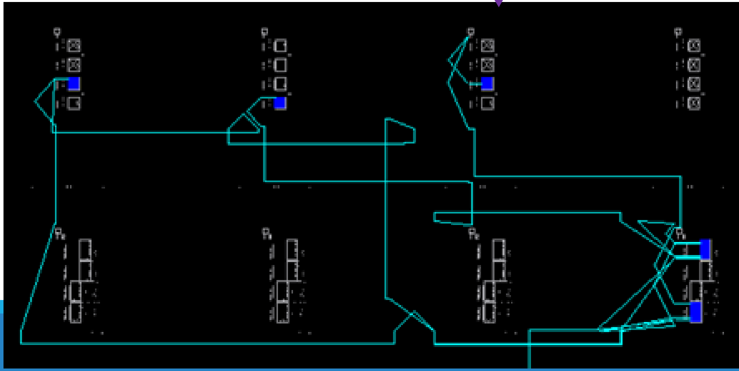
FPGA Library of Components

Virtex, Spartan,
Arria, Artix, Zynq

Synthesis Attributes

```
attribute syn_encoding of my_fsm: type is
   "one-hot";
```



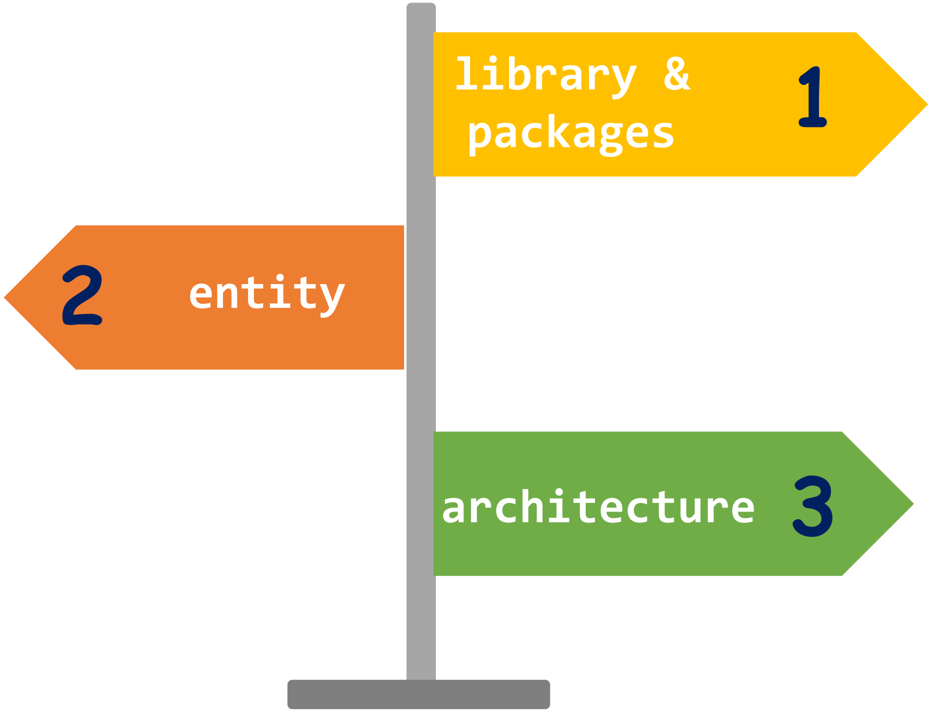


VHDL Simple Example

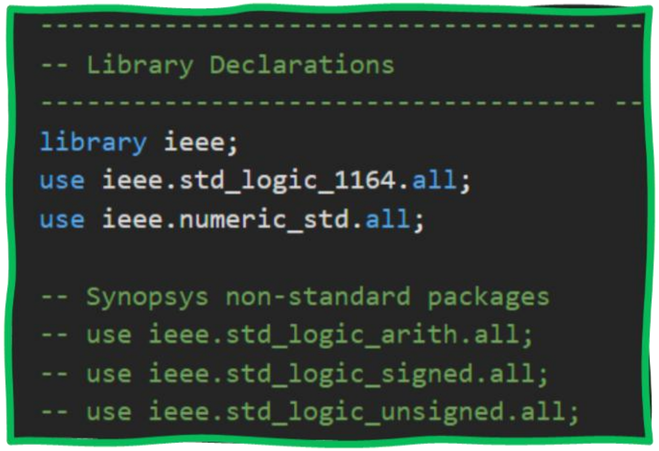
Simple Example - VHDL

Design a BCD up-down counter. The count should be displayed in a 7-segment display.

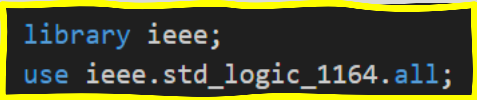
The system has a high frequency clock and system reset as inputs.



Libraries & Packages

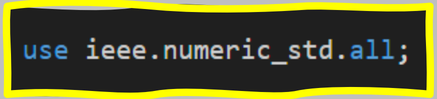


```
-------------------------------------- --
-- Library Declarations
-------------------------------------- --
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

-- Synopsys non-standard packages
-- use ieee.std_logic_arith.all;
-- use ieee.std_logic_signed.all;
-- use ieee.std_logic_unsigned.all;
```



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

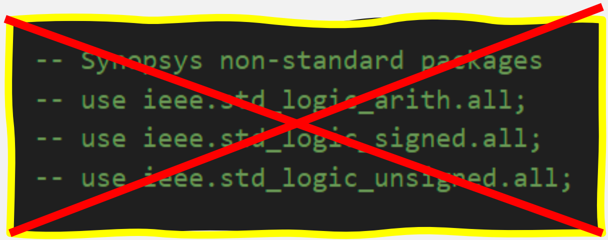
Must be present to use ***std_logic*** type. That is, for ALL synthesisable designs.



```
use ieee.numeric_std.all;
```

Must be present to add arithmetic functions for ***signed*** and ***unsigned*** types.

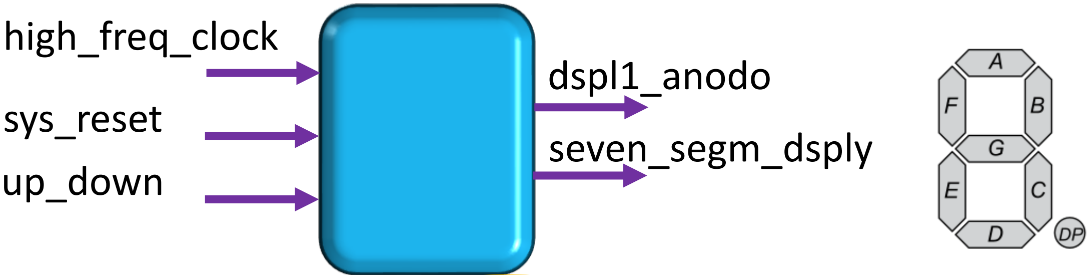
Note: do not do arithmetic operations with std_logic/std_logic_vector

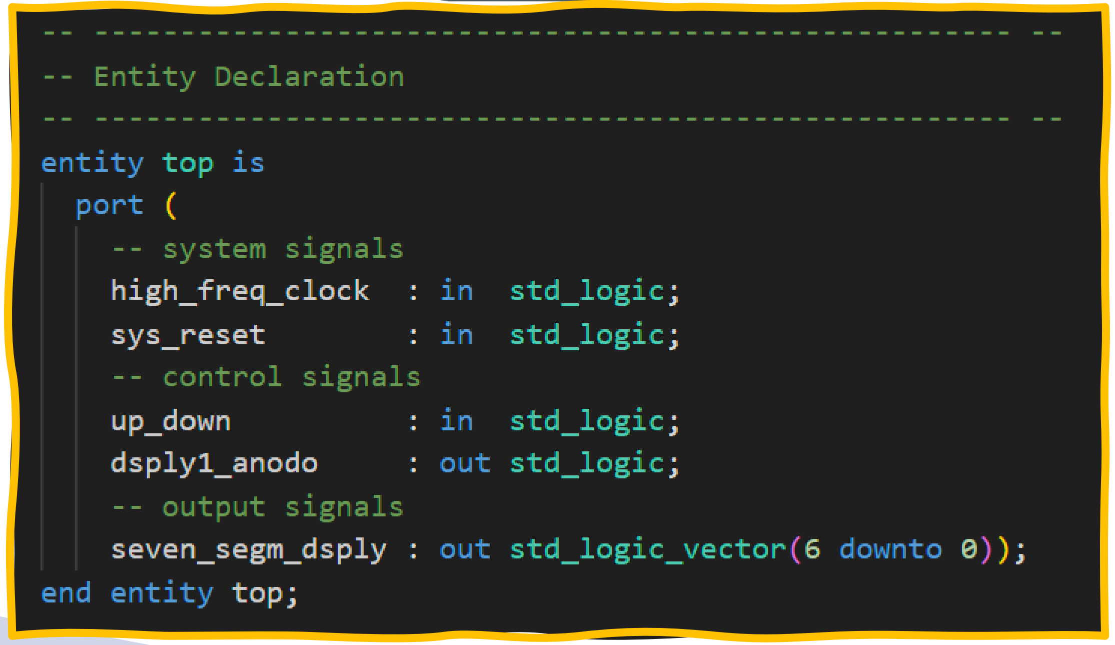


```
-- Synopsys non-standard packages
-- use ieee.std_logic_arith.all;
-- use ieee.std_logic_signed.all;
-- use ieee.std_logic_unsigned.all;
```

DO NOT USE these packages. There do not belong to the VHDL IEEE standard.

Signal/Port Declarations in the Entity





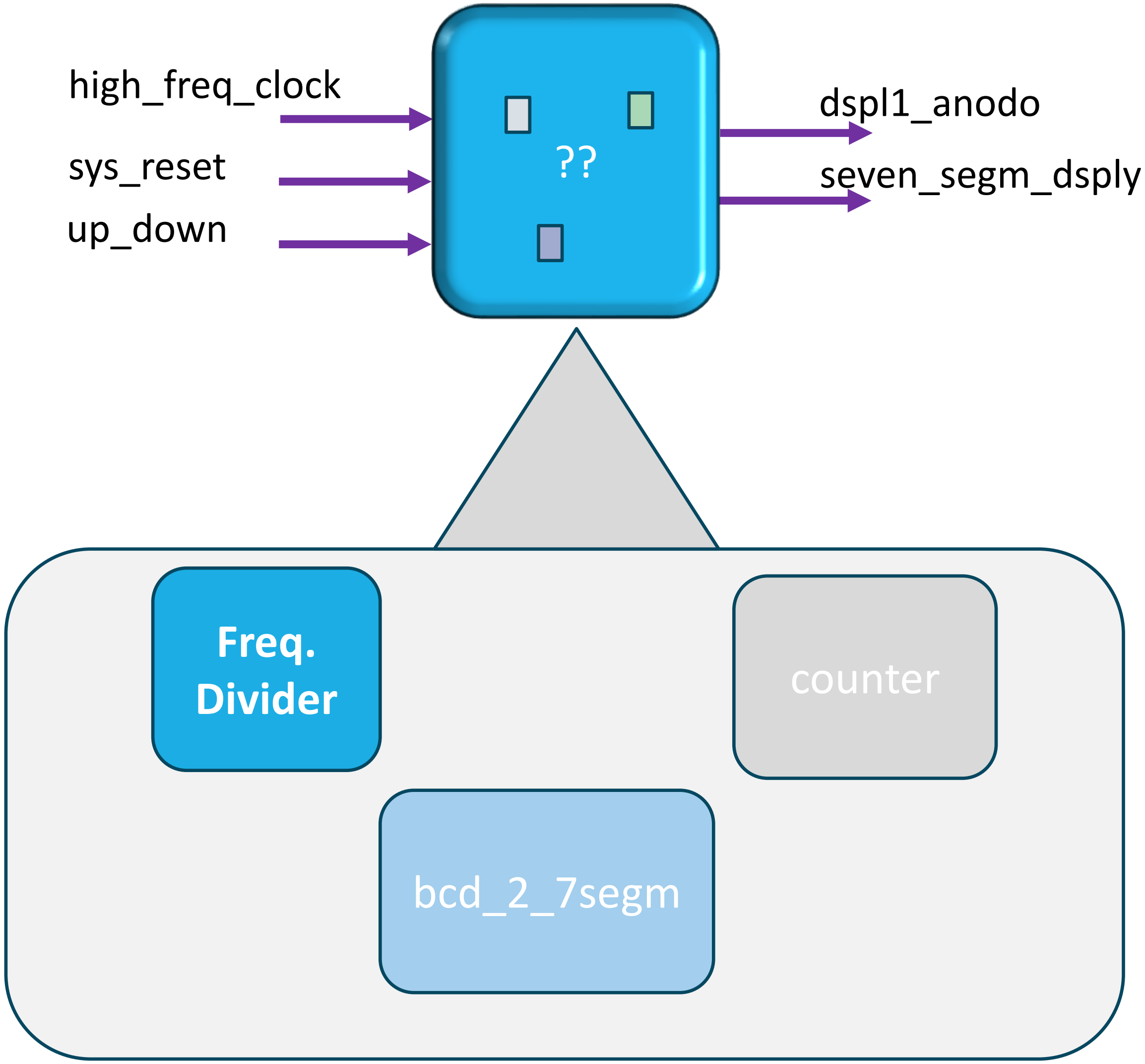
```vhdl
-- ------------------------------------------------------- --
-- Entity Declaration
-- ------------------------------------------------------- --
entity top is
  port (
    -- system signals
    high_freq_clock  : in  std_logic;
    sys_reset        : in  std_logic;
    -- control signals
    up_down          : in  std_logic;
    dsply1_anodo     : out std_logic;
    -- output signals
    seven_segm_dsply : out std_logic_vector(6 downto 0));
end entity top;
```

use only **in/out** modes

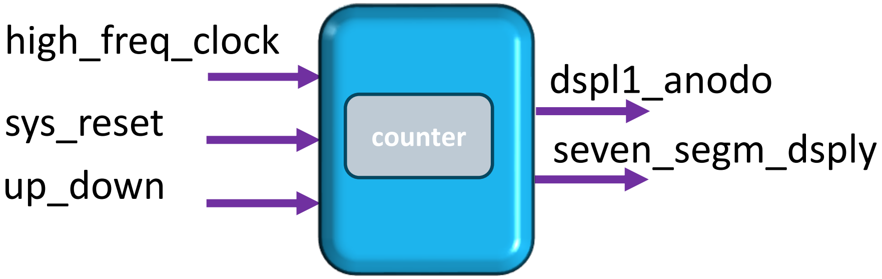
use only **std_logic/std_logic_vector** types

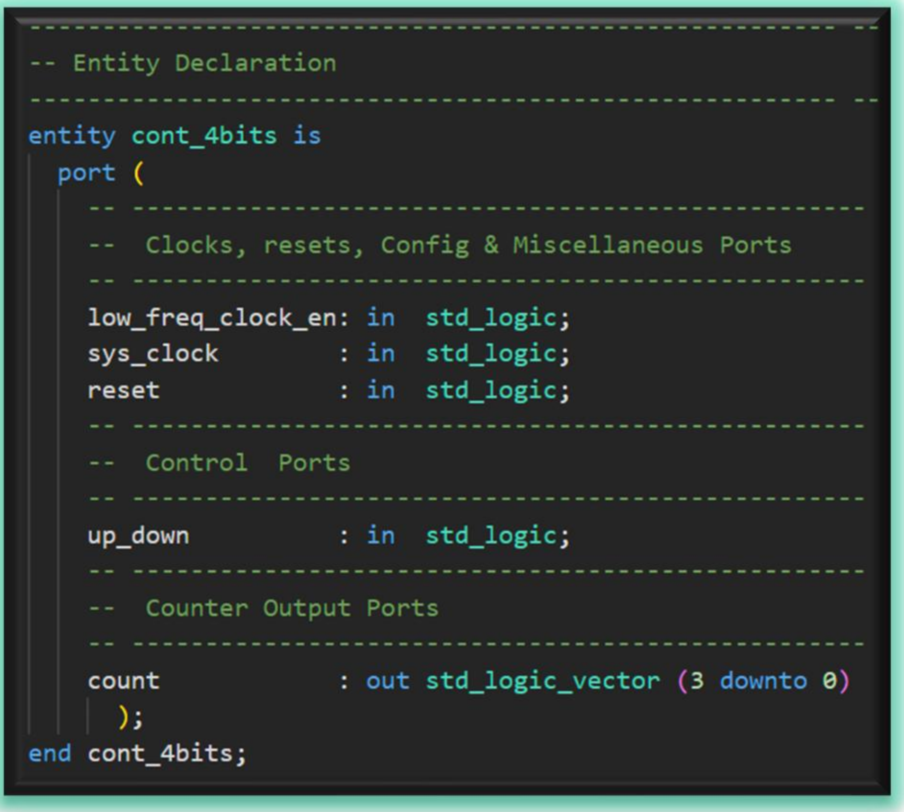
use only **inout** mode in the higher lever top-module

Architecture (top)

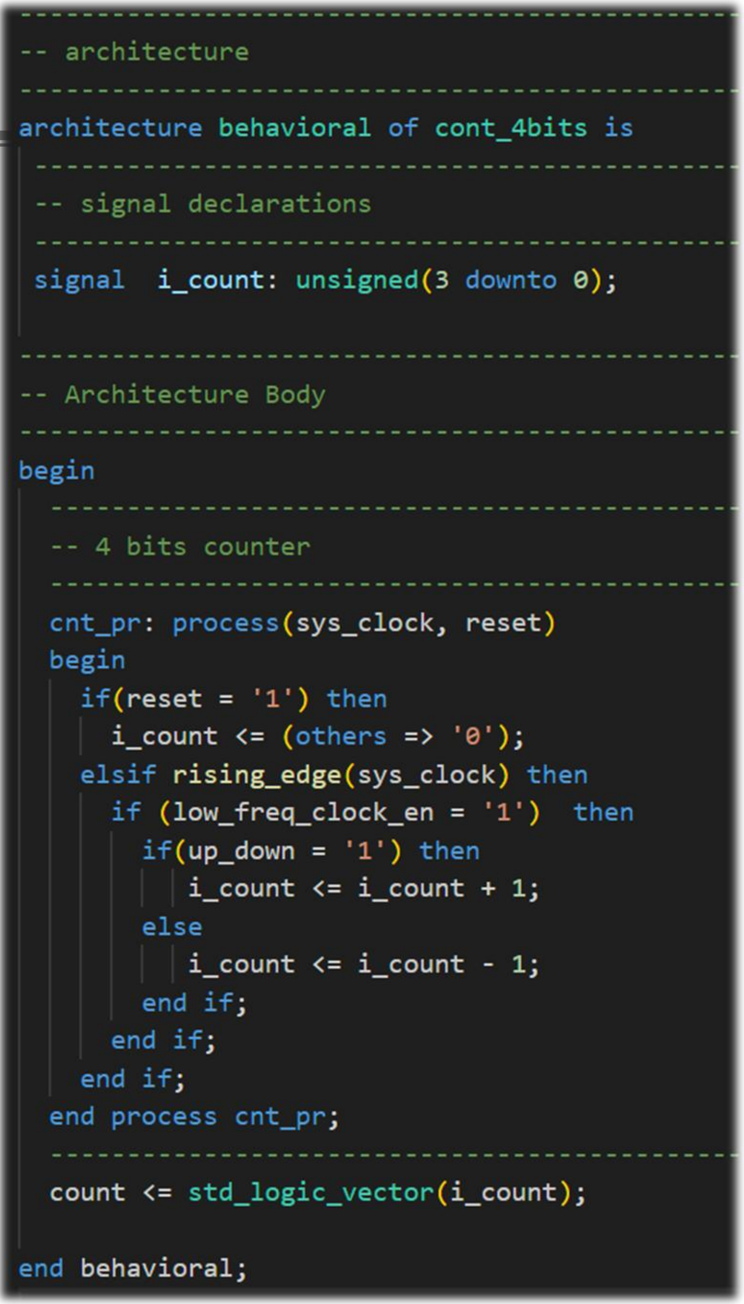


Counter entity/arch.





```
-------------------------------------------------- --
-- Entity Declaration
-------------------------------------------------- --
entity cont_4bits is
  port (
    -- -------------------------------------------------
    --  Clocks, resets, Config & Miscellaneous Ports
    -- -------------------------------------------------
    low_freq_clock_en: in  std_logic;
    sys_clock        : in  std_logic;
    reset            : in  std_logic;
    -- -------------------------------------------------
    --  Control  Ports
    -- -------------------------------------------------
    up_down          : in  std_logic;
    -- -------------------------------------------------
    --  Counter Output Ports
    -- -------------------------------------------------
    count            : out std_logic_vector (3 downto 0)
     );
end cont_4bits;
```



```
------------------------------------------------------
-- architecture
------------------------------------------------------
architecture behavioral of cont_4bits is
 ----------------------------------------------------
 -- signal declarations
 ----------------------------------------------------
 signal  i_count: unsigned(3 downto 0);

------------------------------------------------------
-- Architecture Body
------------------------------------------------------
begin
  ---------------------------------------------------
  -- 4 bits counter
  ---------------------------------------------------
  cnt_pr: process(sys_clock, reset)
  begin
    if(reset = '1') then
      i_count <= (others => '0');
    elsif rising_edge(sys_clock) then
      if (low_freq_clock_en = '1')  then
        if(up_down = '1') then
           i_count <= i_count + 1;
        else
           i_count <= i_count - 1;
        end if;
      end if;
    end if;
  end process cnt_pr;
  ---------------------------------------------------
  count <= std_logic_vector(i_count);

end behavioral;
```

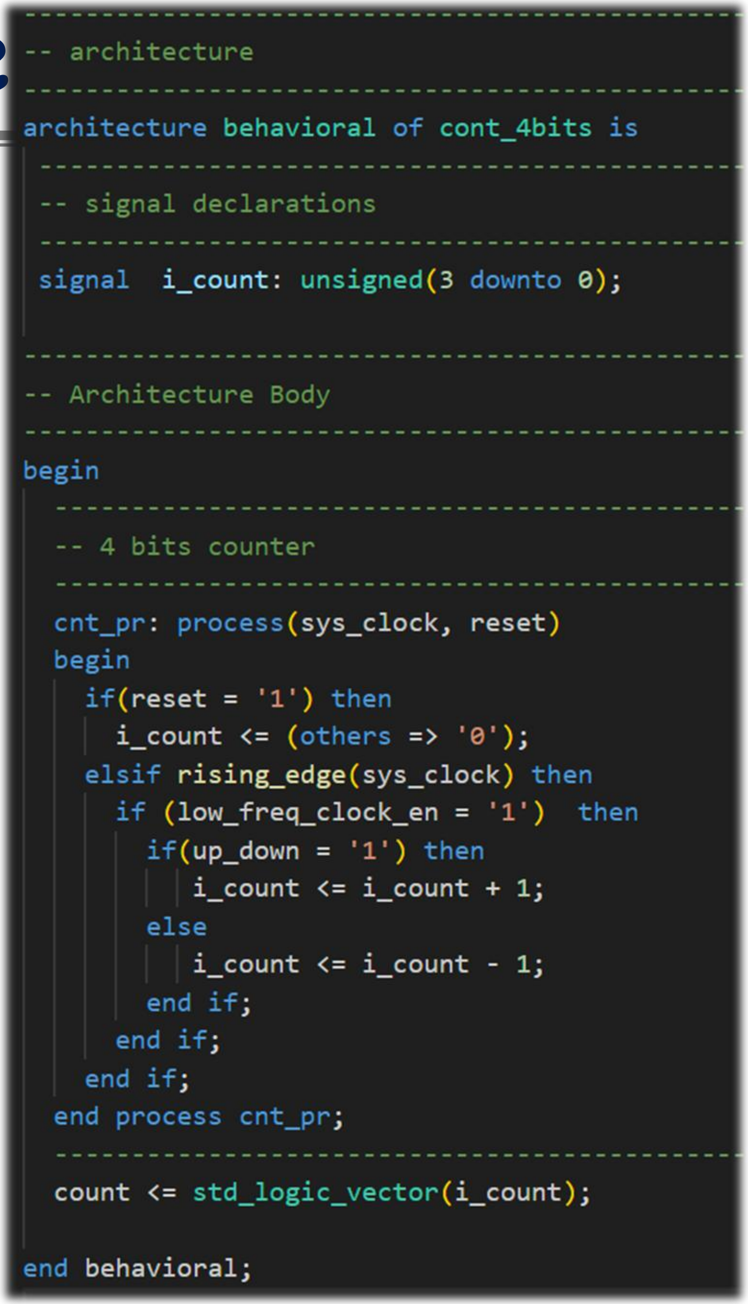
Counter Architecture

Declarative part

Descriptive part (concurrent)

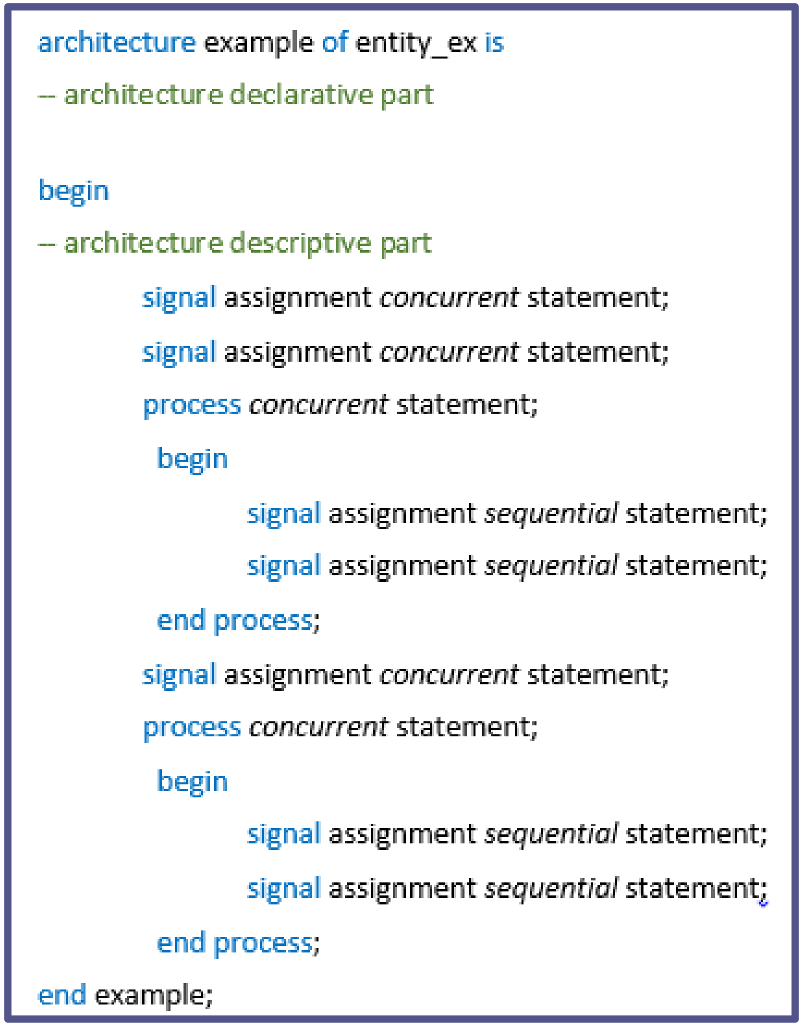
Sequential statements (inside a process)

Concurrent statement

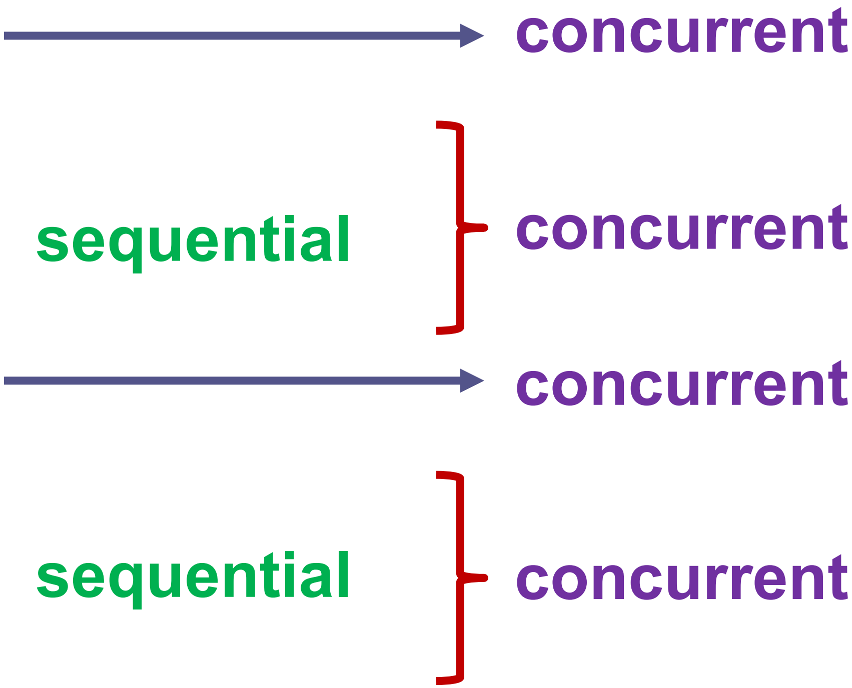


```
--------------------------------------------------
-- architecture
--------------------------------------------------
architecture behavioral of cont_4bits is
 -------------------------------------------------
 -- signal declarations
 -------------------------------------------------
 signal  i_count: unsigned(3 downto 0);

--------------------------------------------------
-- Architecture Body
--------------------------------------------------
begin
  ------------------------------------------------
  -- 4 bits counter
  ------------------------------------------------
  cnt_pr: process(sys_clock, reset)
  begin
    if(reset = '1') then
      i_count <= (others => '0');
    elsif rising_edge(sys_clock) then
      if (low_freq_clock_en = '1')  then
        if(up_down = '1') then
          i_count <= i_count + 1;
        else
          i_count <= i_count - 1;
        end if;
      end if;
    end if;
  end process cnt_pr;
  ------------------------------------------------
  count <= std_logic_vector(i_count);

end behavioral;
```

Understanding Concurrency



```
architecture example of entity_ex is
– architecture declarative part

begin
– architecture descriptive part
        signal assignment concurrent statement;
        signal assignment concurrent statement;
        process concurrent statement;
         begin
                signal assignment sequential statement;
                signal assignment sequential statement;
         end process;
        signal assignment concurrent statement;
        process concurrent statement;
         begin
                signal assignment sequential statement;
                signal assignment sequential statement;
         end process;
end example;
```



concurrent

sequential

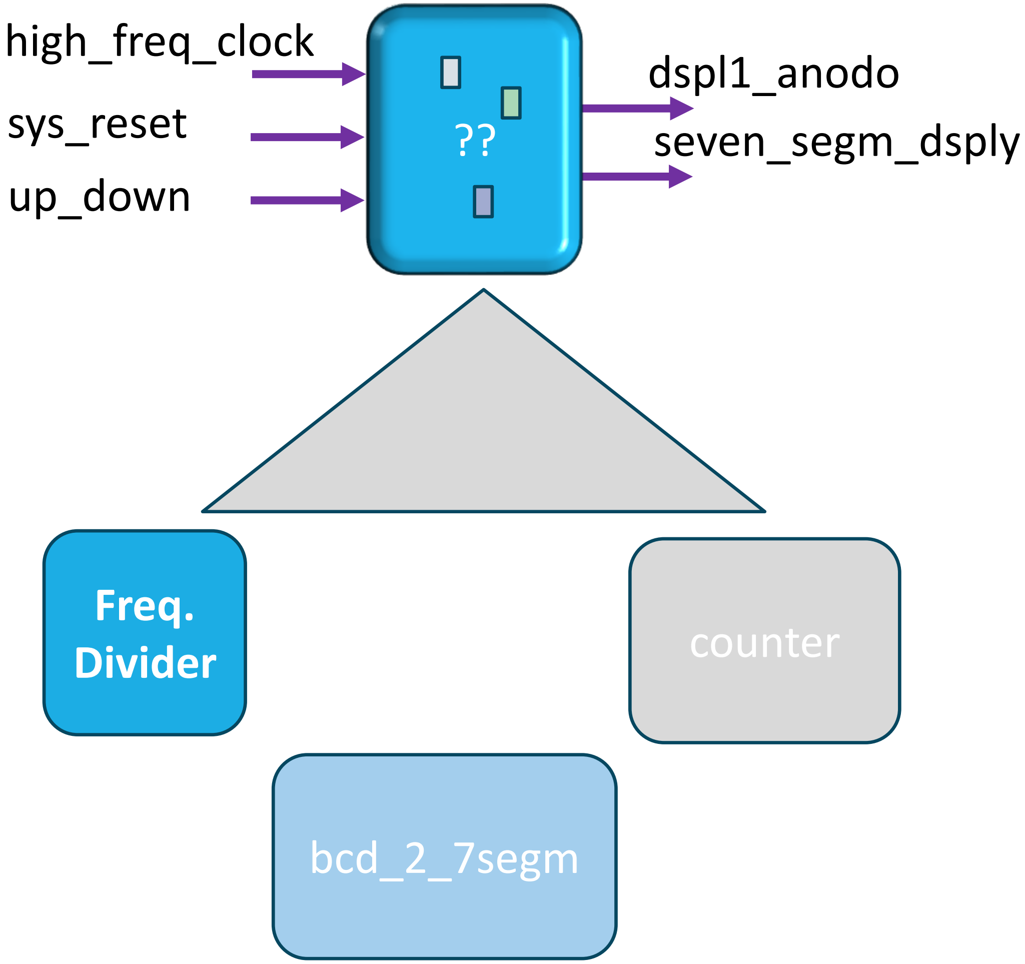
concurrent

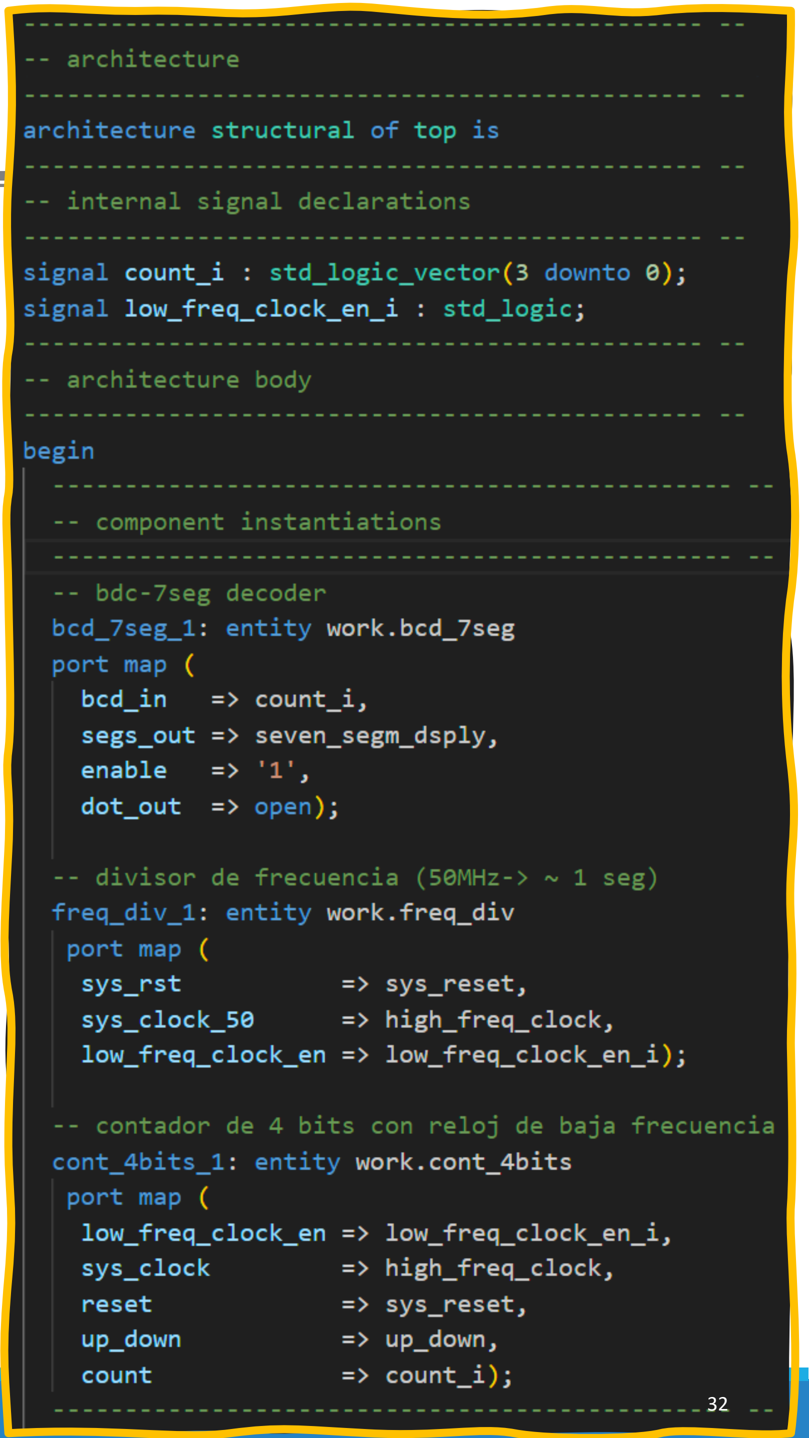
concurrent

sequential

concurrent

Architecture (top)

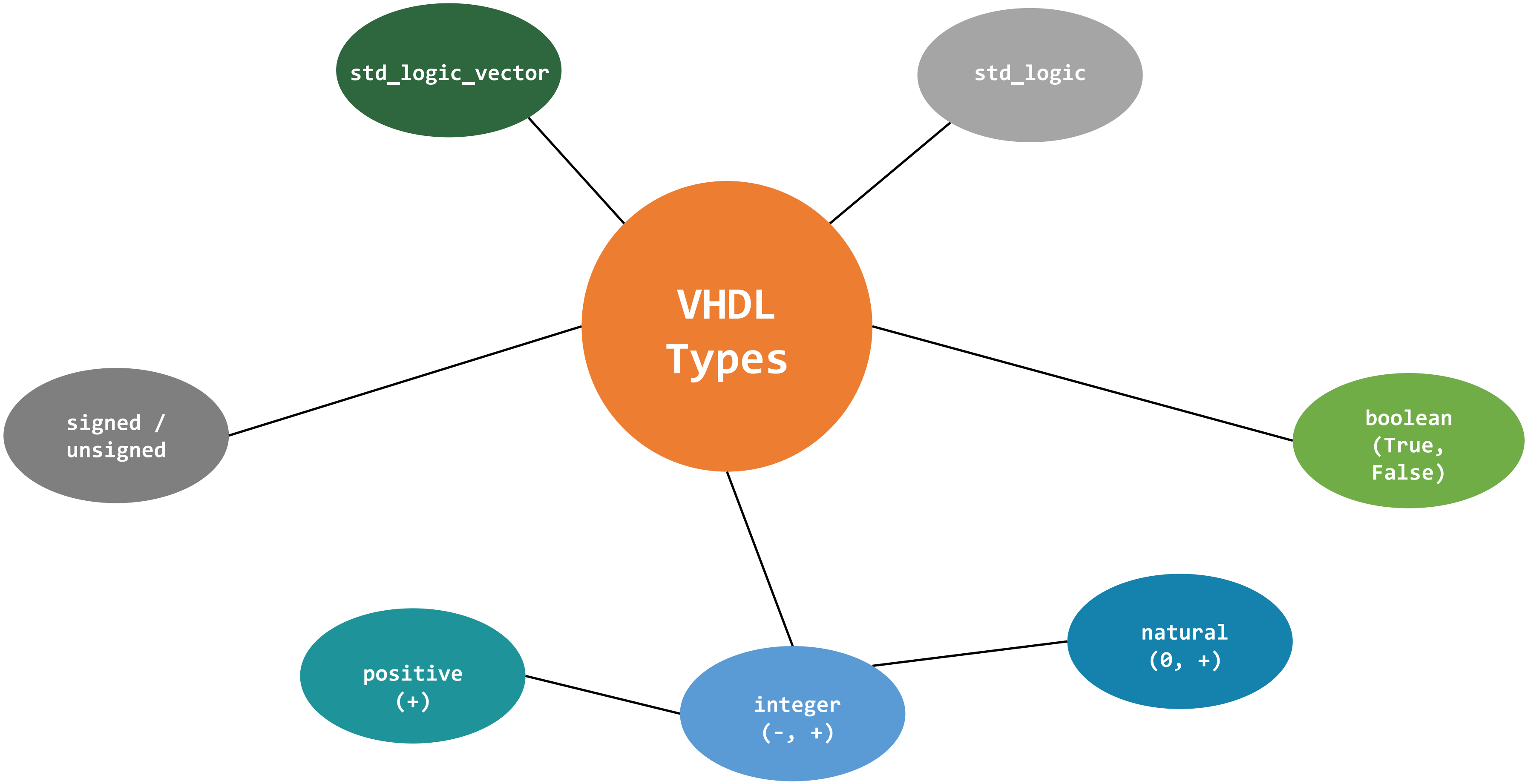




```
---------------------------------------------------------- --
-- architecture
---------------------------------------------------------- --
architecture structural of top is
---------------------------------------------------------- --
-- internal signal declarations
---------------------------------------------------------- --
signal count_i : std_logic_vector(3 downto 0);
signal low_freq_clock_en_i : std_logic;
---------------------------------------------------------- --
-- architecture body
---------------------------------------------------------- --
begin
  -------------------------------------------------------- --
  -- component instantiations
  -------------------------------------------------------- --
  -- bdc-7seg decoder
  bcd_7seg_1: entity work.bcd_7seg
  port map (
    bcd_in    => count_i,
    segs_out  => seven_segm_dsply,
    enable    => '1',
    dot_out   => open);

  -- divisor de frecuencia (50MHz-> ~ 1 seg)
  freq_div_1: entity work.freq_div
   port map (
    sys_rst           => sys_reset,
    sys_clock_50      => high_freq_clock,
    low_freq_clock_en => low_freq_clock_en_i);

  -- contador de 4 bits con reloj de baja frecuencia
  cont_4bits_1: entity work.cont_4bits
   port map (
    low_freq_clock_en => low_freq_clock_en_i,
    sys_clock         => high_freq_clock,
    reset             => sys_reset,
    up_down           => up_down,
    count             => count_i);
  -------------------------------------------------------- --
```

VHDL Types, Objects & Classes

VHDL Data Types



Signal Assignment - strongly typed

```
count       <= count + 1;
carry_out   <= (a and b) or (a and c) or (b and c);
Z           <= y;
```

Left Hand Side (LHS)
Target Signal

Right Hand Side (RHS)
Source Signal(s)

LHS **Signal Data Type** == RHS **Signal Data Type**

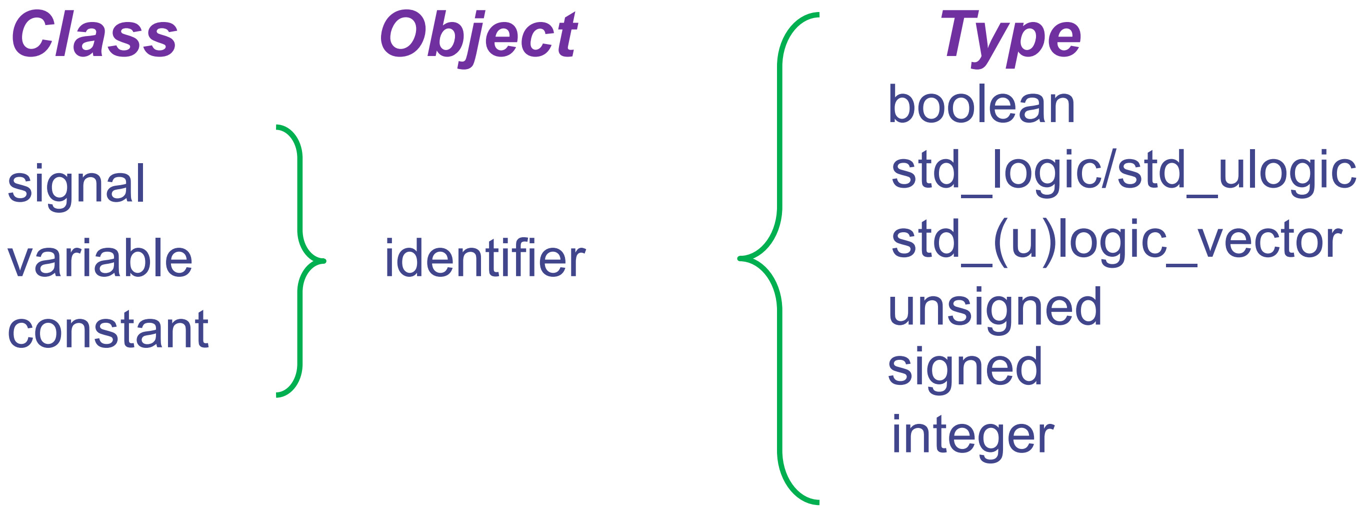
```
signal bandera: integer;
signal flag, enable : std_logic;
. . . .
          bandera <= flag;      -- ?

          enable  <= flag;      -- ?
```

VHDL Object

An **object** holds a value of some specified ***type*** and can be one of the three ***classes***:
signal, variable, constant

Declaration Syntax:

```
object_class <identifier> : type[ := initial_value];
```



Class

signal
variable
constant

Object

identifier

Type

boolean
std_logic/std_ulogic
std_(u)logic_vector
unsigned
signed
integer

std_logic Type

```
PACKAGE std_logic_1164 IS
    ------------------------------------------------
    -- logic state system  (unresolved)
    ------------------------------------------------
    TYPE std_ulogic IS ( 'U',   -- Uninitialized
                         'X',   -- Forcing  Unknown
                         '0',   -- Forcing  0
                         '1',   -- Forcing  1
                         'Z',   -- High Impedance
                         'W',   -- Weak     Unknown
                         'L',   -- Weak     0
                         'H',   -- Weak     1
                         '-'    -- Wild card
                       );
     SUBTYPE std_logic IS resolved std_ulogic;
```

Type Conversion - Casting

VHDL does allow restricted type of CASTING, that is converting values between related types

```
datatype <= type(data_object);
```

```
signal max_rem: unsigned (7 downto 0);
signal more_t: std_logic_vector( 7 downto 0);

max_rem <= more_t;

max_rem <= unsigned(more_t);
```

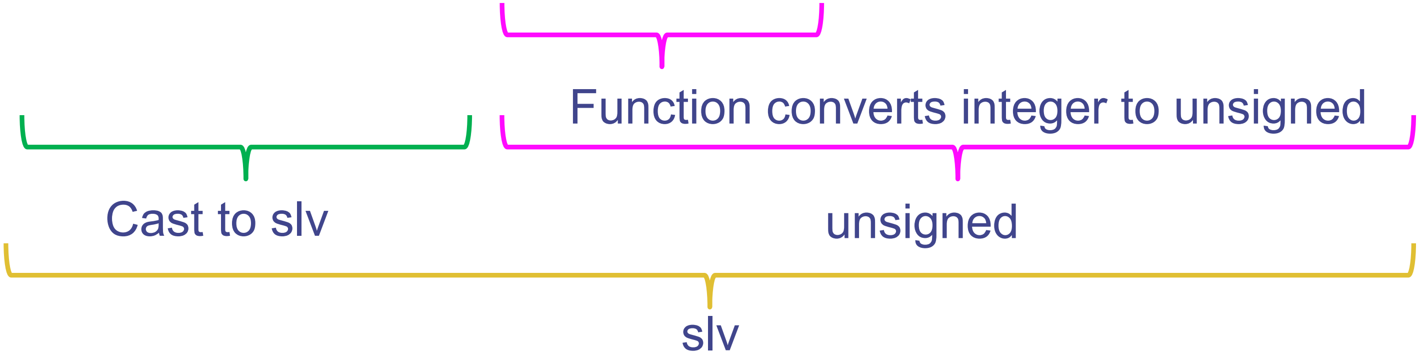
unsigned and ***std_logic_vector*** are both vectors of the same element type, therefore it's possible a direct conversion by ***casting***. When there is not type relationship a conversion ***function*** is used.

Type Conversion - Functions

VHDL does have some built-in functions to convert some different data types (not all the types allow conversions)

```
datatype <= to_type(data_object);
```

```
signal internal_counter: integer range 0 to 15;
signal count: std_logic_vector( 3 downto 0);

count <= internal_count;

CoUnT <= std_logic_vector(to_unsigned(internal_count,4));
```



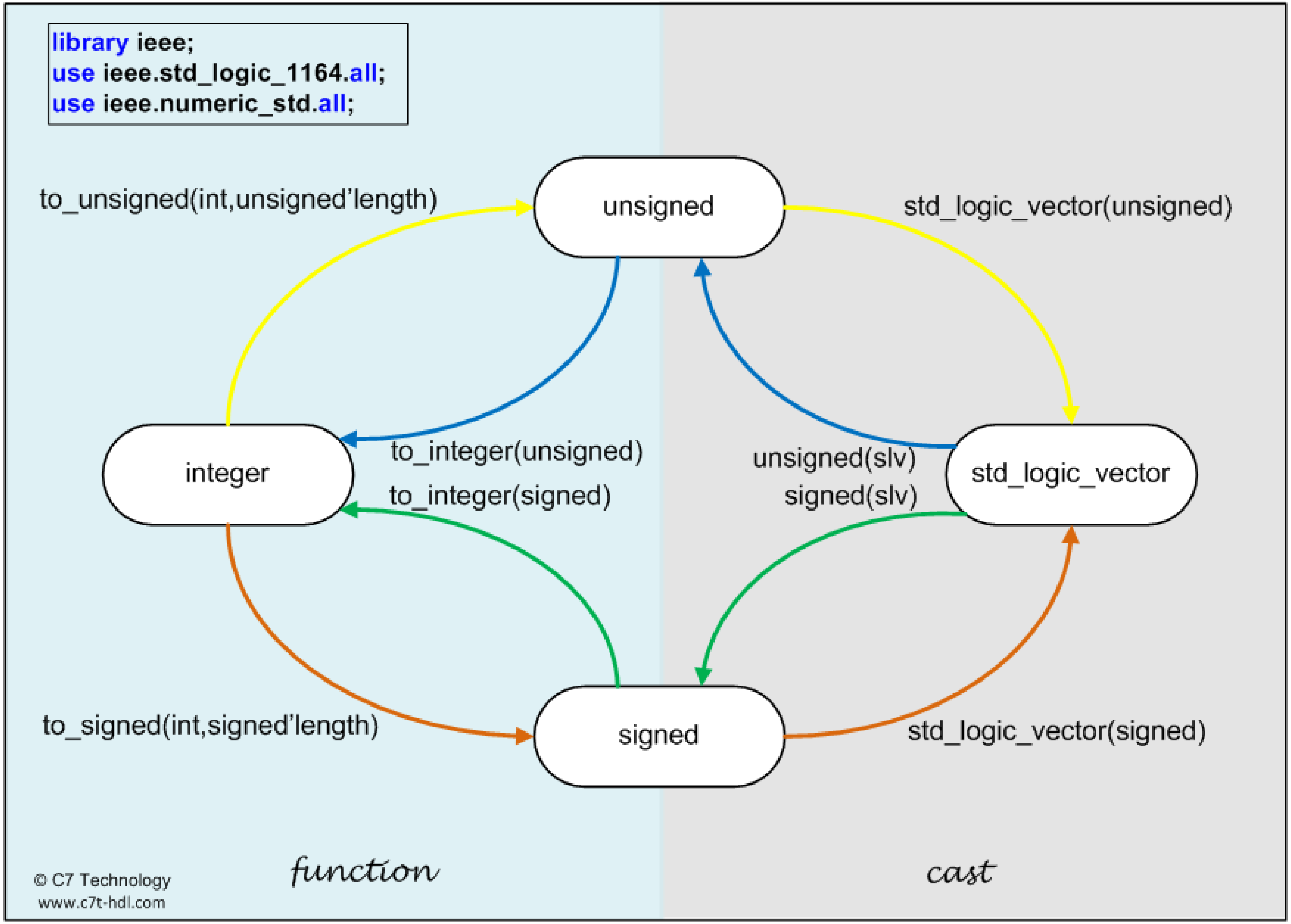
Function converts integer to unsigned

Cast to slv

unsigned

slv

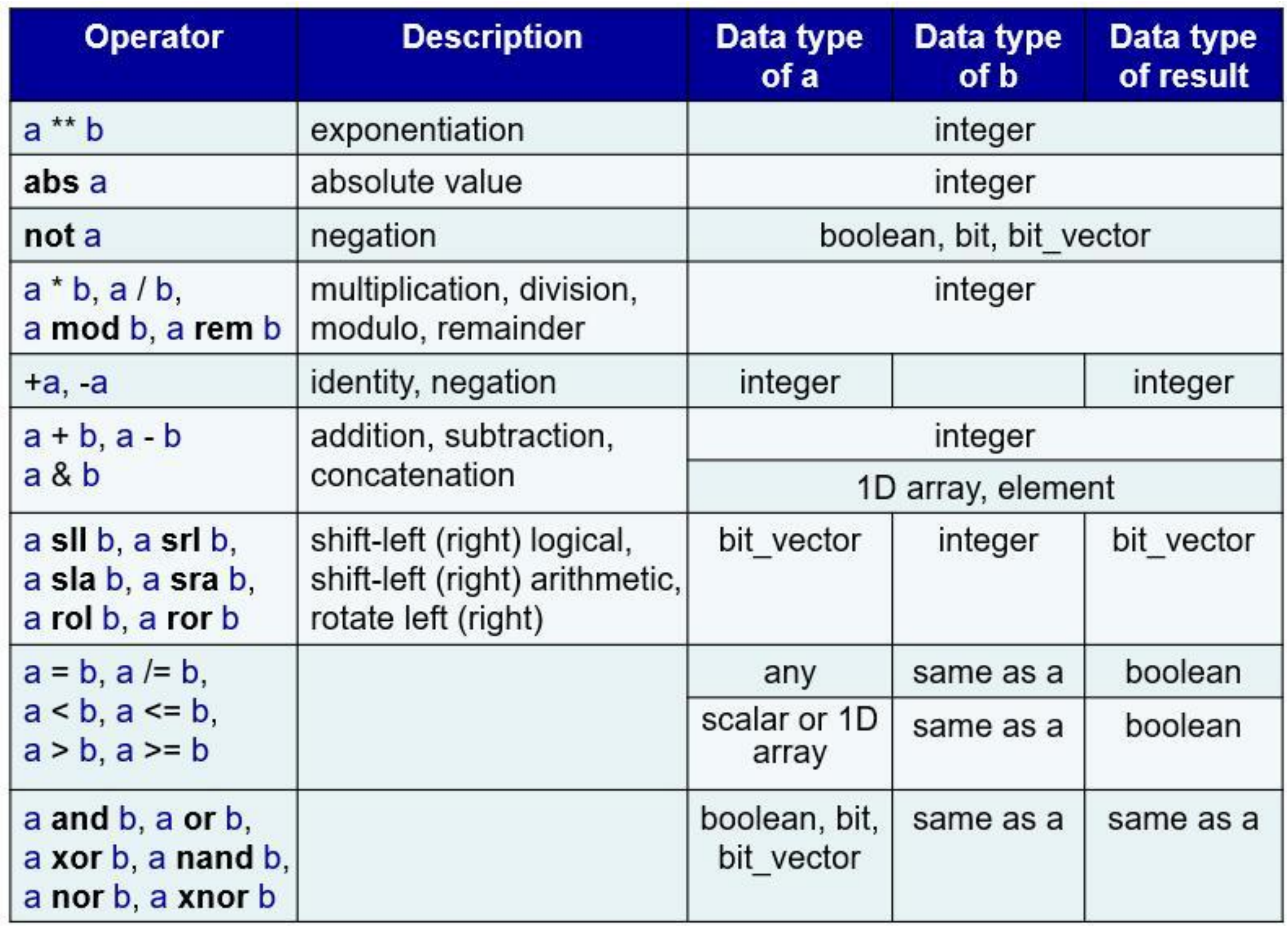
Type Conversion - Cast / Function



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

© C7 Technology
www.c7t-hdl.com

VHDL Operators

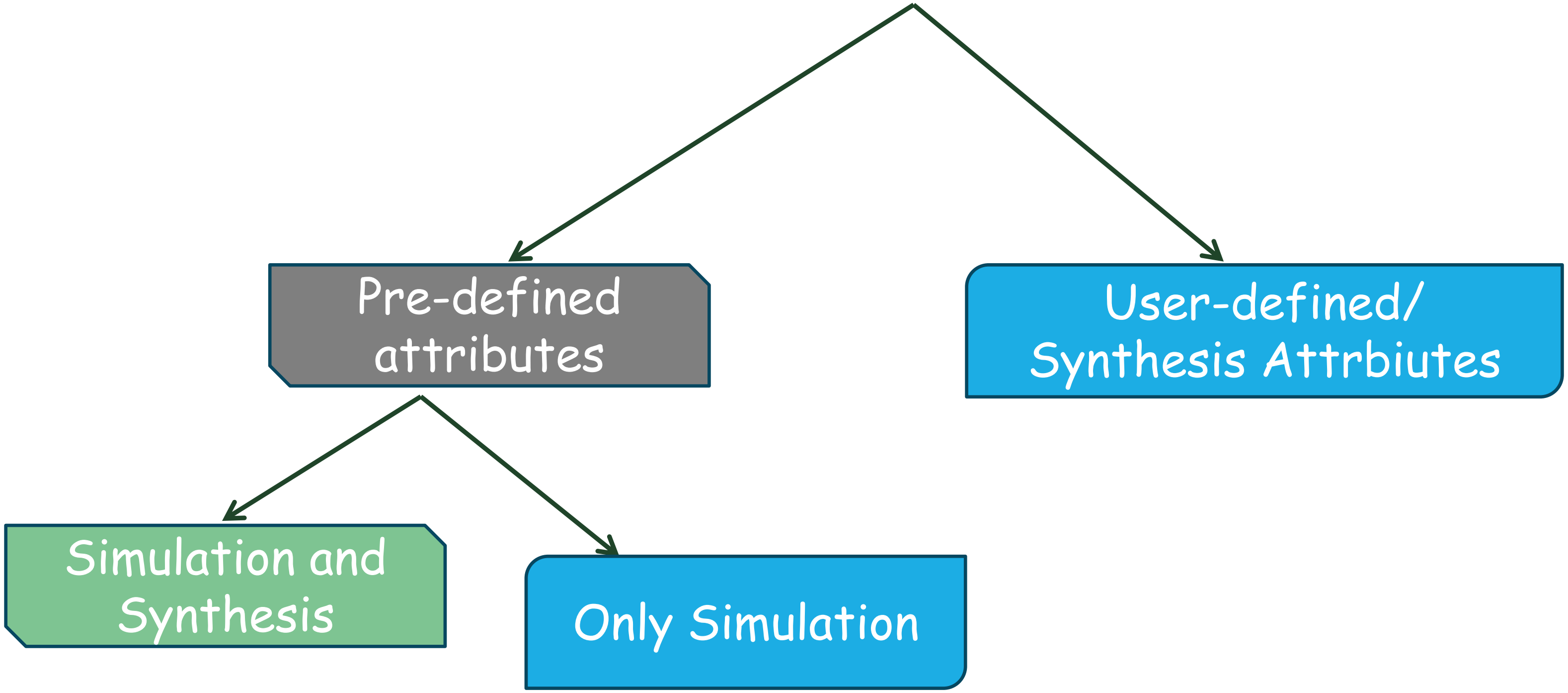


Operator	Description	Data type of a	Data type of b	Data type of result
a ** b	exponentiation	integer		
abs a	absolute value	integer		
not a	negation	boolean, bit, bit_vector		
a * b, a / b, a **mod** b, a **rem** b	multiplication, division, modulo, remainder	integer		
+a, -a	identity, negation	integer		integer
a + b, a - b a & b	addition, subtraction, concatenation	integer		
		1D array, element		
a **sll** b, a **srl** b, a **sla** b, a **sra** b, a **rol** b, a **ror** b	shift-left (right) logical, shift-left (right) arithmetic, rotate left (right)	bit_vector	integer	bit_vector
a = b, a /= b, a < b, a <= b, a > b, a >= b		any	same as a	boolean
		scalar or 1D array	same as a	boolean
a **and** b, a **or** b, a **xor** b, a **nand** b, a **nor** b, a **xnor** b		boolean, bit, bit_vector	same as a	same as a

VHDL Attributes

☑ It’s a way of **extracting** information from a type, from the values of a type

☑ It’s also a way to allow to **assign additional** information to objects in your design description (such as data related to synthesis)



Pre-defined attributes

User-defined/ Synthesis Attrbiutes

Simulation and Synthesis

Only Simulation

Array Attributes

☑ Array attributes are used to obtain information on the size, range and indexing of an array

☑ It’s good practice to use attributes to refer to the size or range of an array. So, if the size of the array is change, the VHDL statement using attributes will automatically adjust to the change

Array Attributes – Range Related	
A’range	Returns the range value of a constrained array
A’reverse_range	Returns the reverse value of a constrained array

Array Attributes

Use of the attributes ***range*** and ***reverse_range***

```
variable w_bus: std_logic_vector(7 downto 0);
```

then:

```
w_bus'range          -- will return:    7 downto 0
```

while:

```
w_bus'reverse_range   -- will return:    0 to 7
```

User-defined/Synthesis Attributes

VHDL provides designers/vendors with a way of adding additional information to the system to be synthesized

- Synthesis tools use this features to add timing, placement, pin assignment, hints for resource locations, type of encoding for state machines and several others physical design information

- The bad side of synthesis attributes is that the VHDL code becomes synthesis tools/FPGA dependant, NO TRANSPORTABLE ….

User-defined/Synthesis Attributes

Syntax

```
attribute attr_name: type;

attribute attr_name of data_object: ObjectType is AttributeValue;
```

Example

```
attribute syn_preserve: boolean;

attribute syn_preserve of ff_data: signal is true;
```

```
type my_fsm_state is (reset, load, count, hold);

attribute syn_encoding: string;

attribute syn_encoding of my_fsm_state: type is “gray”;
```

User-defined/Synthesis Attributes

Example:

```
type ram_type is array (63 downto 0) of
                        std_logic_vector (15 downto 0);
signal ram: ram_type;
attribute syn_ramstyle: string;
attribute syn_ramstyle of ram: signal is “block_ram”;
```

VHDL Statements

Selective Signal Assignment Statement

Syntax

```
with <selection_signal> select
    target_signal <= <expression> when <value1_ss>,
                  <expression> when <value2_ss>,
                   ...
                 <expression> when <last_value_ss>,
                  <expression> when others;
```

A selective signal assignment describes logic based on mutually exclusive combinations of values of the selection signal

Selective Signal Assignment Statement

Example: Truth Table

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	-
1	1	1	-

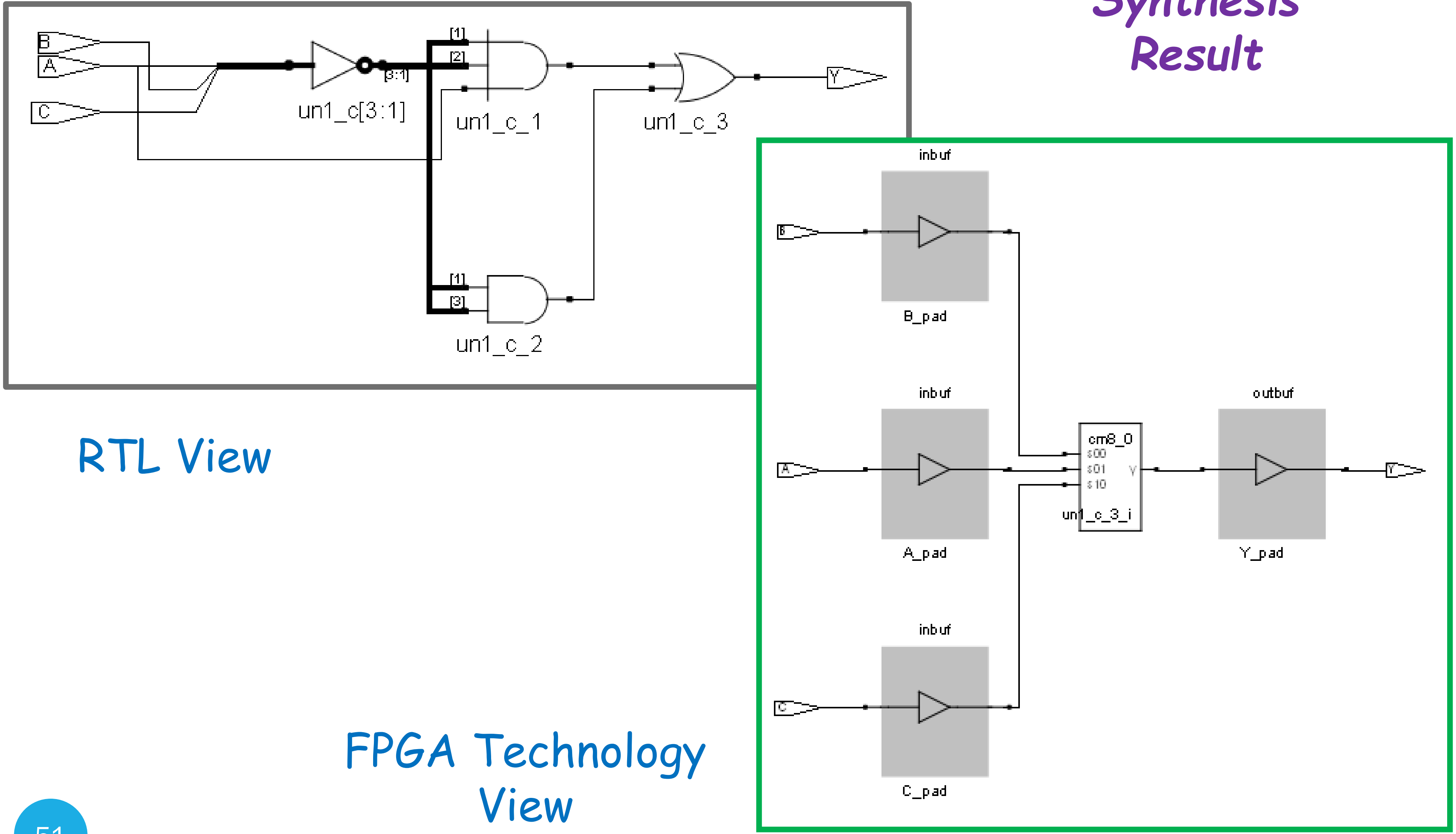
```
library ieee;
use ieee.std_logic_1164.all;
entity TRUTH_TABLE is
  port(A, B, C: in std_logic;
             Y: out std_logic);
end TRUTH_TABLE;
architecture BEHAVE of TRUTH_TABLE is
  signal S1: std_logic_vector(2 downto 0);
begin
  S1 <= A & B & C; -- concatenate A, B, C
  with S1 select




end BEHAVE;
```

"|" means OR only when used in "with" or "case"

'-' means don't care

Selective Signal Assignment Statement

Synthesis Result



RTL View

FPGA Technology View

Conditional Signal Assignment

Syntax

```
target_signal <=
  <expression> when <boolean_condition> else
  <expression> when <boolean_condition> else
  ....
  <expression> when <boolean_condition>[else
      <expression>];
```

A conditional signal assignment describes logic based on unrelated *boolean_conditions*, the *first condition that is true* the value of expression is assigned to the *target_signal*

Conditional Signal Assignment

Main usage

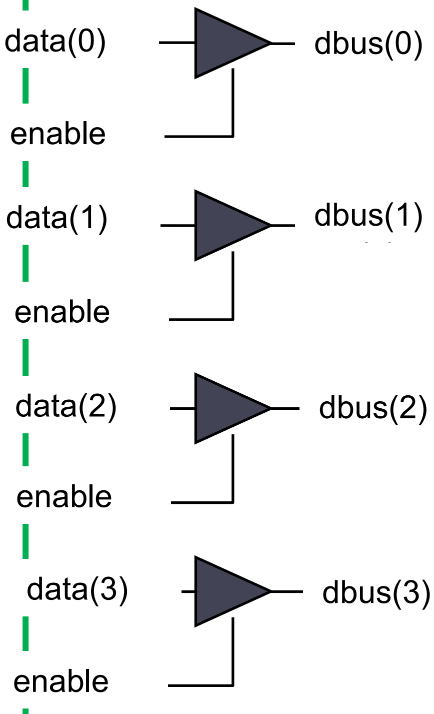
```
dbus <= data   when enable = ‘1’ else ‘Z’;
```

```
dbus <= data when enable = ‘1’ else (others=>‘Z’);
```

Conditional Signal Assignment

Example

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

entity my_tri is
  generic(bus_ancho: integer := 4);
  port(
   data:   in std_logic_vector(bus_ancho-1 downto 0);
   enable: in std_logic;
   dbus :  out std_logic_vector(bus_ancho-1 downto 0)
   );
end my_tri;

architecture behave of my_tri is
begin
  y <= a  when en = ‘1’ else (others => ‘z’) ;
end behave;
```



process Statement

A process is a concurrent statement, but it is the primary mode of introducing **sequential statements**

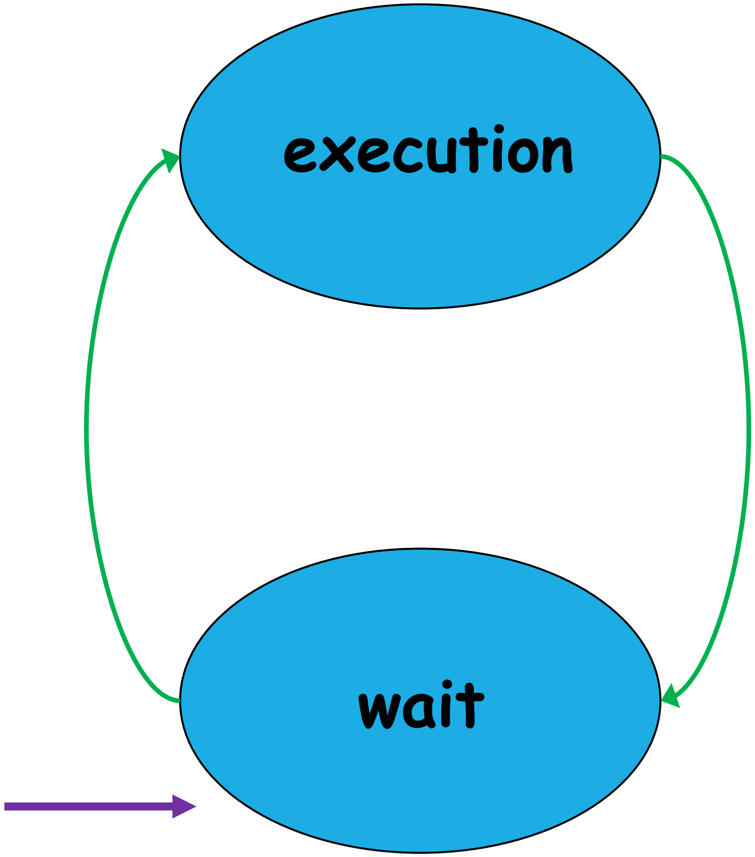
- A process, with all the sequential statements, is a *simple* ***concurrent statement***.

- Multiple ***processes*** can be executed in parallel

- From the traditional programming view,a ***process*** is an *infinite loop*

Process Statement

A process has two states: *execution* and *wait*



Once the process has been executed, it will wait for the next satisfied condition

Until a condition is satisfied

process Statement Syntax

```
[process_label:] process [(sensitivity_list)] [is]
[process_data_object_declarations]
begin
   variable_assignment_statement
   signal_assignment_statement
   wait_statement
   if_statement
   case_statement
   loop_statement
   null_statement
   exit_statement
   next_statement
   assertion_statement
   report_statement
   procedure_call_statement
   [wait on sensitivity_list]
end process [process_label];
```

Sequential statements

Parts of the process statement

`sensitivity_list`

- List of all the signals that are able *to trigger the process*
- Simulation tools monitor events on these signals
- Any event on any signal in the sensitivity list will cause to execute the process at least once

- `declarations`
  - Declarative part. Types, functions, procedures and variables can be declared in this part
  - Each declaration is local to the process

- `sequential_statements`
  - All the sequential statements that will be executed each time that the process is activated

Signal Behaviour in a process

While a process is running ALL the SIGNALS in the system **remain unchanged** -> Signals are in effect **CONSTANTS** during process execution, EVEN after a signal assignment, the signal will NOT take a new value

SIGNALS **are updated at the end of a process**

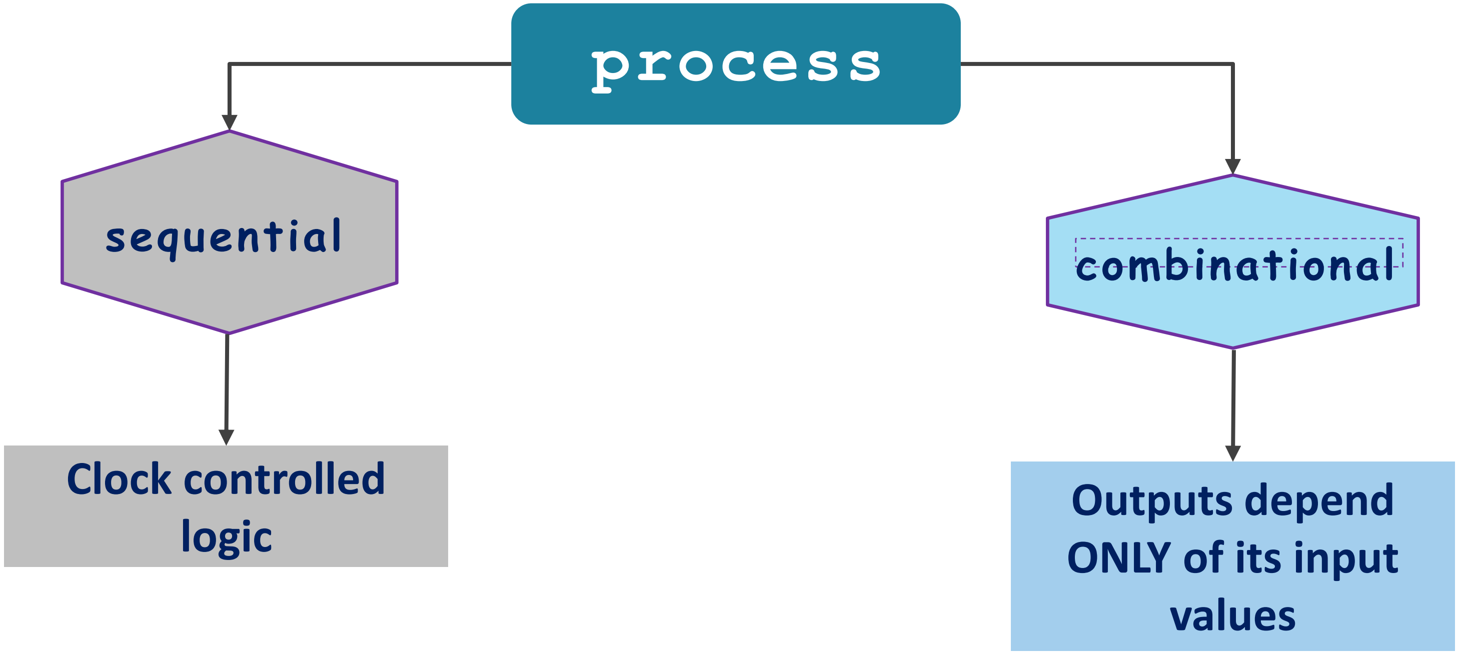
Signals are a mean of communication between processes -> VHDL can be seen as a network of processes intercommunicating via signals

Variable Behavior in a process

While a process is running ALL the Variables in the system are updates **IMMEDIATELY** by a variable assignment statement

process - Combinational/Sequential

When using processes, a key distinction is made between those that model sequential logic (controlled by a clock) and those that model combinational logic.



process

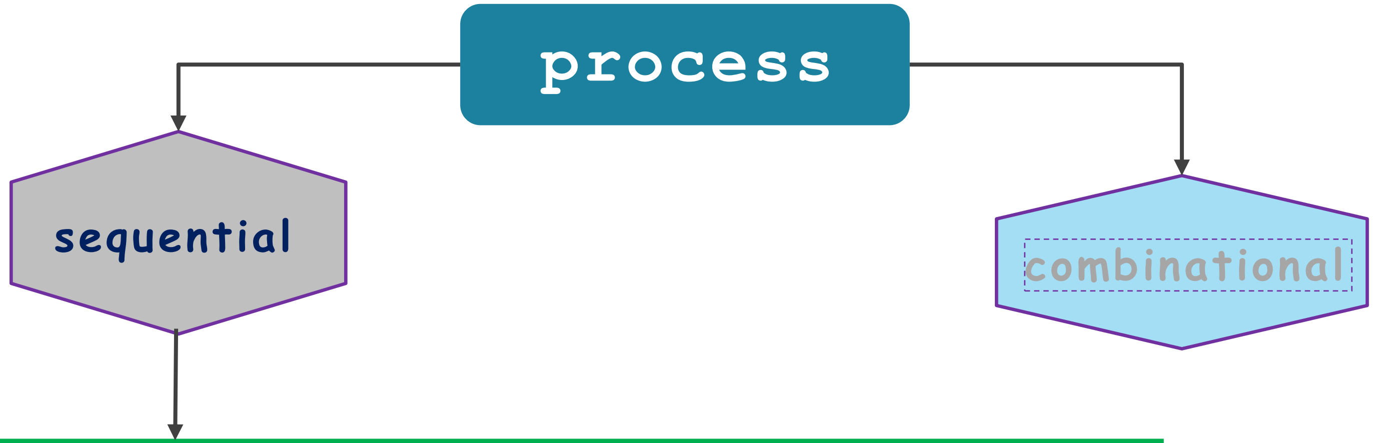
sequential

combinational

Clock controlled logic

Outputs depend ONLY of its input values

process - Combinational/Sequential



process

sequential

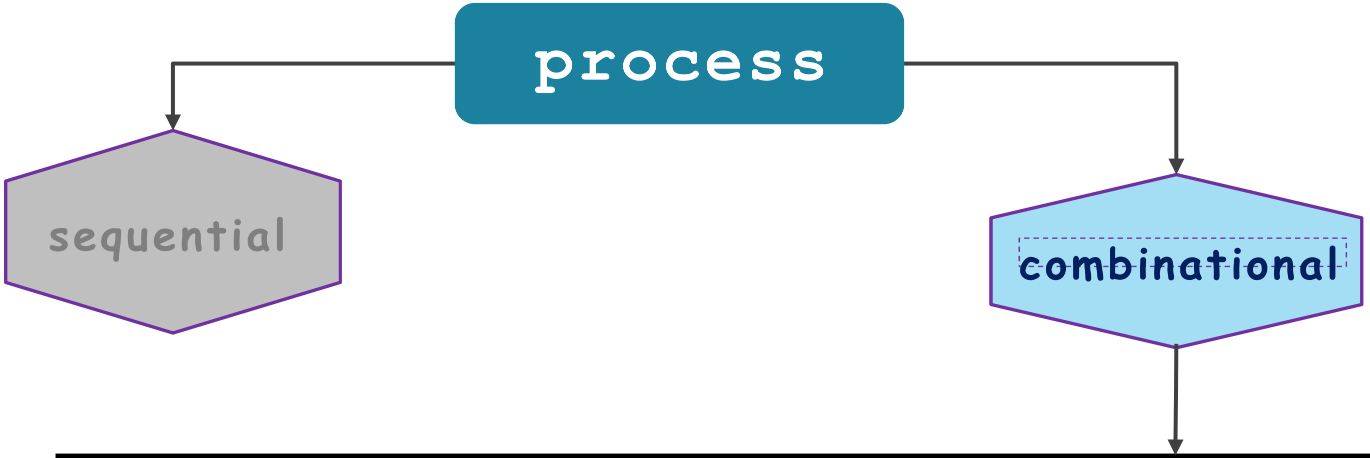
combinational

A clock-controlled process, also known as a **sequential process**, describes logic whose outputs change only at specific edges of a clock signal (e.g., rising edge or falling edge). This type of process is used to model sequential elements like flip-flops, registers, counters, and state machines, which have memory and store state.

Key Characteristics:

- It is **ONLY** sensitive to the **clock signal** and often a **reset signal**.
- It typically contains an IF statement that checks for a clock edge (e.g., rising_edge(clk) or falling_edge(clk)).
- Signal assignments inside the clock edge condition are implemented as some storage element (e.g., flip-flop, memory).

process - Combinational/Sequential

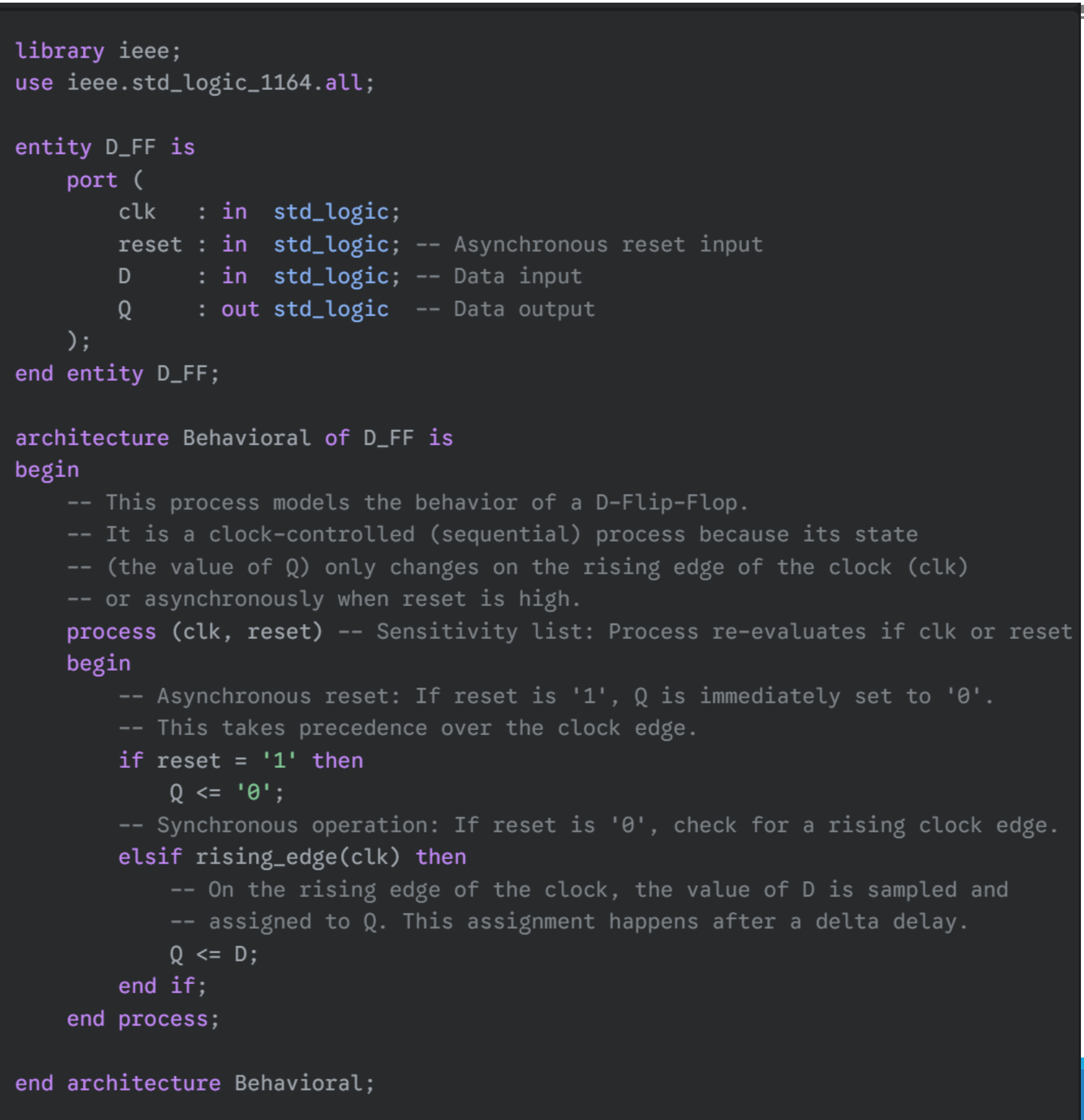


A **combinational process** describes logic whose outputs depend *only* on the current values of its inputs. There is no memory or state involved; if the inputs change, the outputs change (after a propagation delay).

Key Characteristics:

- Its sensitivity list must include **all input signals** that affect the process's outputs. If an input changes, the process must re-evaluate to produce the correct output.
- It **does not** contain clock edge detection (rising_edge or falling_edge).
- Signal assignments are typically **concurrent updates**, as outputs are directly derived from inputs.
- There should be **no unassigned** signals in all possible execution paths within the process; otherwise, the VHDL synthesizer will infer latches, which is generally undesirable for combinational logic. All outputs must be assigned a value for every possible combination of inputs.

Sequential *process* example

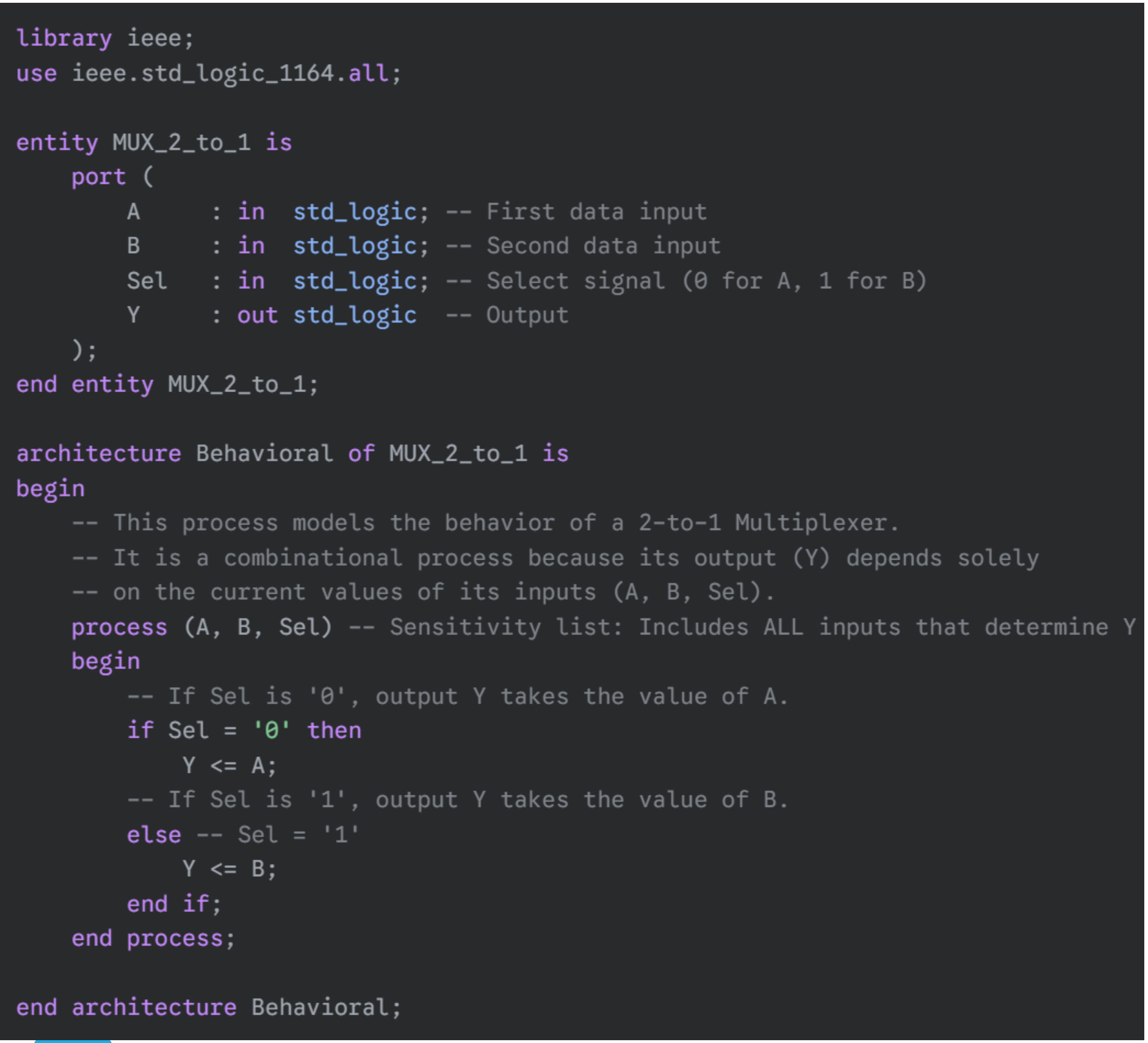


```
library ieee;
use ieee.std_logic_1164.all;

entity D_FF is
    port (
        clk   : in  std_logic;
        reset : in  std_logic; -- Asynchronous reset input
        D     : in  std_logic; -- Data input
        Q     : out std_logic  -- Data output
    );
end entity D_FF;

architecture Behavioral of D_FF is
begin
    -- This process models the behavior of a D-Flip-Flop.
    -- It is a clock-controlled (sequential) process because its state
    -- (the value of Q) only changes on the rising edge of the clock (clk)
    -- or asynchronously when reset is high.
    process (clk, reset) -- Sensitivity list: Process re-evaluates if clk or reset
    begin
        -- Asynchronous reset: If reset is '1', Q is immediately set to '0'.
        -- This takes precedence over the clock edge.
        if reset = '1' then
            Q <= '0';
        -- Synchronous operation: If reset is '0', check for a rising clock edge.
        elsif rising_edge(clk) then
            -- On the rising edge of the clock, the value of D is sampled and
            -- assigned to Q. This assignment happens after a delta delay.
            Q <= D;
        end if;
    end process;

end architecture Behavioral;
```

Description:

- The **`process (clk, reset)`** line defines the sensitivity list. This means the process will execute whenever there's an event (a change in value) on either the clk or reset signal.
- The **if reset = '1'** condition handles the asynchronous reset. If reset is active (high), Q is immediately set to '0'. This happens independently of the clock.
- The **elsif rising_edge(clk)** condition means that if the reset is not active, the statements within this block will only execute when the clk signal transitions from '0' to '1'.
- **Q <= D;** inside the rising_edge block indicates that the output Q will take on the value of the input D at that specific clock edge. This correctly describes the memory element of a D-flip-flop.

Combinational `process` example

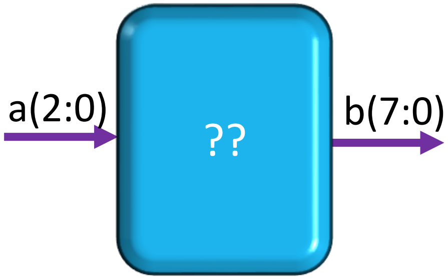


```
library ieee;
use ieee.std_logic_1164.all;

entity MUX_2_to_1 is
    port (
        A      : in  std_logic; -- First data input
        B      : in  std_logic; -- Second data input
        Sel    : in  std_logic; -- Select signal (0 for A, 1 for B)
        Y      : out std_logic  -- Output
    );
end entity MUX_2_to_1;

architecture Behavioral of MUX_2_to_1 is
begin
    -- This process models the behavior of a 2-to-1 Multiplexer.
    -- It is a combinational process because its output (Y) depends solely
    -- on the current values of its inputs (A, B, Sel).
    process (A, B, Sel) -- Sensitivity list: Includes ALL inputs that determine Y
    begin
        -- If Sel is '0', output Y takes the value of A.
        if Sel = '0' then
            Y <= A;
        -- If Sel is '1', output Y takes the value of B.
        else -- Sel = '1'
            Y <= B;
        end if;
    end process;

end architecture Behavioral;
```

Description

- The **`process (A, B, Sel)`** line defines the sensitivity list. The process will execute whenever there's an event on A, B, or Sel. This ensures that Y is always updated whenever any of its inputs change.
- There are no clock edge or reset conditions; the logic simply evaluates based on current inputs.
- The **if Sel = '0'** and **else** branches ensure that the output Y is ***always*** assigned a value, regardless of the **Sel** input. This is critical to avoid inferring a latch.

if Statement – 3 to 8 Decoder



```
entity if_decoder_example is
 port(
  a: in  std_logic_vector(2 downto 0);
  z: out std_logic_vector(7 downto 0);
end entity;

architecture rtl of if_decoder_example is
begin
if_dec_ex: process (a)
 begin
    if    (a = “000”) then
            z <= “00000001”;
    elsif (a = “001”) then
            z <= “00000010”;
            . . .
    else
            z <= (others => ‘0’);
    end if;
 end process if_dec_ex;
end rtl;
```

if Statement

Most common mistakes for describing combinatorial logic

```vhdl
entity example3 is
   port ( a, b, c: in  std_logic;
             z, y: out std_logic);
end example3;

architecture beh of example3 is
begin
 process (a, b)
  begin
   if c='1' then
         z <= a;
   else
         y <= b;
   end if;
 end process;
end beh;
```

case Statement

```
[case label:]case <selector_expression> is
    when <choice_1> =>
        <sequential_statements> -- branch #1
    when <choice_2> =>
        <sequential_statements> -- branch #2
       .  .  .
     [when <choice_n to/downto choice_m > =>
        <sequential_statements>] -- branch #n
       ....
     [when <choice_x | choice_y | .  .  .> =>
        <sequential_statements>] -- branch #...
     [when others =>
        <sequential_statements>]-- last branch
end case [case_label];
```

case Statement

```vhdl
entity mux4 is
  port ( sel            : in std_ulogic_vector(1 downto 0);
         d0, d1, d2, d3 : in std_ulogic;
         z              : out std_ulogic );
end entity mux4;

architecture demo of mux4 is
begin
out_select : process (sel, d0, d1, d2, d3) is
 begin
   case sel is
         when “00” =>
                  z <= d0;
         when “01” =>
                  z <= d1;
         when “10” =>
                  z <= d2;
         when others =>
                  z <= d3;
    end case;
 end process out_select;
end architecture demo;
```

case Statement with if Statement

```
mux_mem_bus :process
  (cont_out,I_P0,I_P1,I_A0,I_A1,Q_P0,Q_P1,Q_A0,Q_A1)
begin
 mux_out <= I_P0;
 case (cont_out) is
  when "00" =>
     if(iq_bus = '0') then
        mux_out <= I_P0;--I_A0;
     else
       mux_out <= Q_P0;--Q_A0;
     end if;
  when "01" =>
     if(iq_bus = '0') then
       mux_out <= I_A0;--I_P0;
     else
       mux_out <= Q_A0;--Q_P0;
          end if;
     . . . . . .
```

for loop-end loop Statement

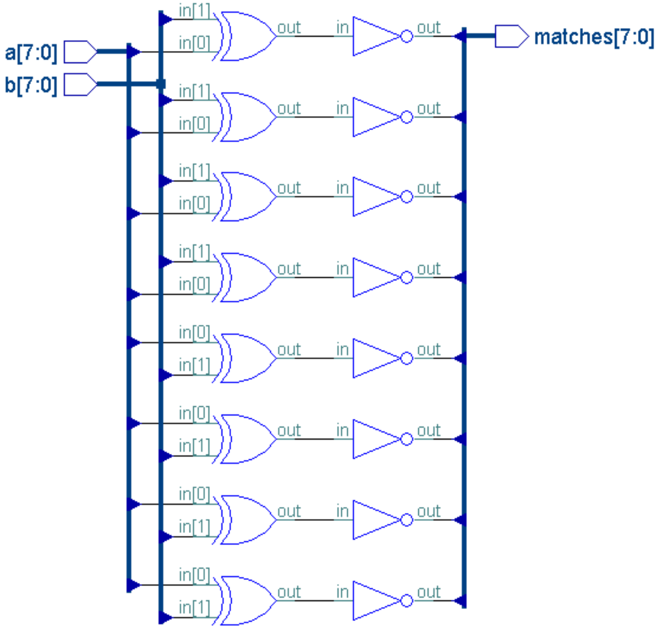
```
[loop_label]: for <identifier> in discrete_range loop
      <sequential_statements>
      end loop [loop_label];
```

```
<identifier>
```

- The identifier is called loop parameter, and for each iteration of the loop, it takes on successive values of the discrete range, starting from the left element
- It is not necessary to declare the identifier
- By default the type is integer
- Only exists when the loop is executing

for-loop Statement

```
entity match_bit is
        port ( a, b    : in  std_logic_vector(7 downto 0);
                matches: out std_logic_vector(7 downto 0));
end entity;
architecture behavioral of match_bit is
begin
process (a, b)
  begin
    for i in a'range loop
        matches(i) <= not (a(i) xor b(i));
    end loop;
  end process;
end behavioral;
```



for-loop Statement

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

entity count_??? is
       port(vec:  in   std_logic_vector(15 downto  0);
           count: out std_logic_vector(3   downto  0))
end count_ones;

architecture behavior of count_???? is
begin
 cnt_ones_proc: process(vec)
   variable result: unsigned(3 downto 0);
 begin
     result:= (others =>'0');
     for i in vec’range loop
       if vec(i)='1' then
          result := result + 1;
       end if;
     end loop;
   count <= std_logic_vector(result);
 end process cnt_ones_proc;
end behavior;
```

The Role of Componentes in VHDL

Hierarchy in VHDL

- Divide & Conquer
- Each subcomponent can be designed and completely tested
- Create library of components (technology independent if possible)
- Third-party available components
- Code for reuse

Component Instantiation

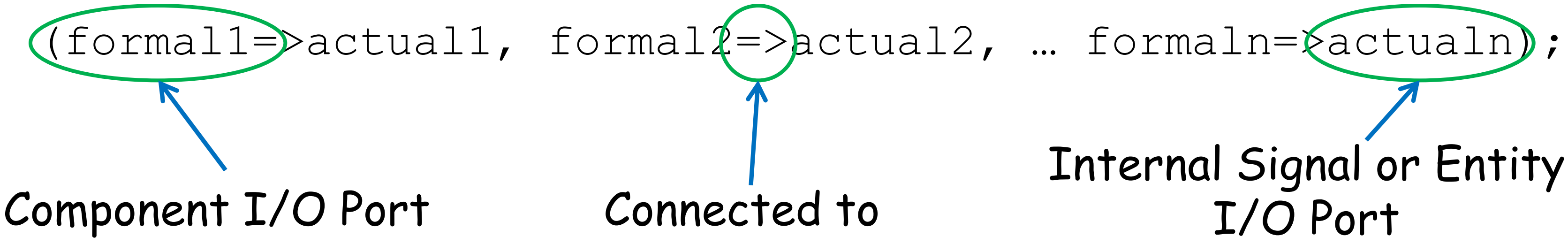
Component instantiation is a concurrent statement that is used to connect a component I/Os to the internal signals or to the I/Os of the higher lever component

```
component_label: entity work.component_name
     [generic map (generic_assocation_list)]
     port map (port_association_list);
```

- `component_label` it labels the instance by giving a name to the instanced
- `generic_assocation_list` assign new values to the default generic values (given in the entity declaration)
- `port_association_list` associate the signals in the top entity/architecture with the ports of the component. There are two ways of specifying the port map:
  ✎ *Positional Association / Name Association*

Association By Name

In named association, an association list is of the form



```
(formal1=>actual1, formal2=>actual2, … formaln=>actualn);
```

Component I/O Port

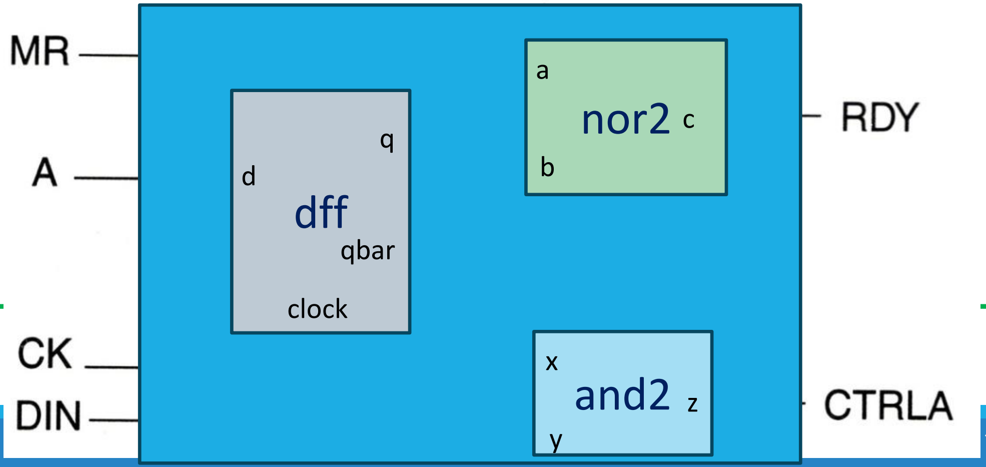
Connected to

Internal Signal or Entity I/O Port

```
-- component declaration
component NAND2
      port (a, b: in  std_logic;
                z: out std_logic);
end component;
-- component instantiation
U1: entity work.NAND2 port map (a=>S1, z=>S3, b=>S2);
-- S1 associated with a, S2 with b and S3 with z
```

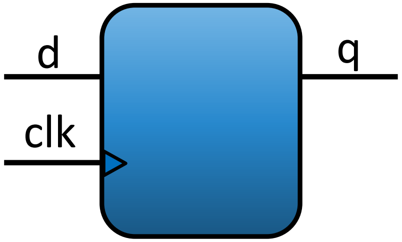
Component Instantiation Example

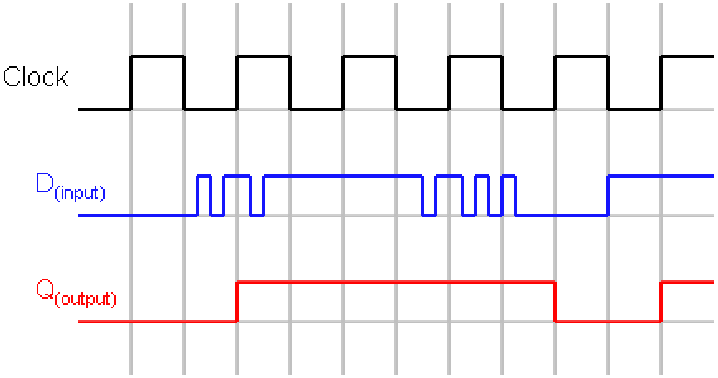
```
library ieee;
use ieee.std_logic_1164.all;
entity glue_logic is
  port (A, CK, MR, DIN: in  std_logic;
        RDY, CTRLA      : out std_logic);
end glue_logic ;
architecture STRUCT of glue_logic is
signal S1, S2: std_logic;
begin
 D1: entity work.DFF  port map (D=>A, CLOCK=>CK, Q=>S1, QBAR=>S2);
 A1: entity work.AND2 port map (X=>S2, Y=>DIN, Z=>CTRLA);
 N1: entity work.NOR2 port map
        (a =>S1,
         b =>MR,
         c =>RD1);
end STRUCT;
```



VHLD for Sequential Logic Design

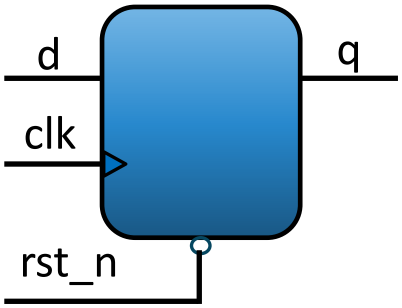
D Flip-Flop - VHDL





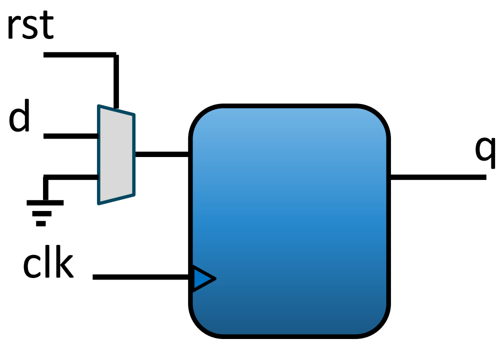
```
entity ff_d_example is
  port(
       d   : in  std_logic;
       clk : in  std_logic;
       q   : out std_logic);
end entity;

architecture rtl of ff_d_example is
begin
 ff_d: process(clk)
 begin
   if (rising_edge(clk)) then
     q <= d;
   end if;
 end process ff_d;
end rtl;
```

D-ff with asynchronous reset



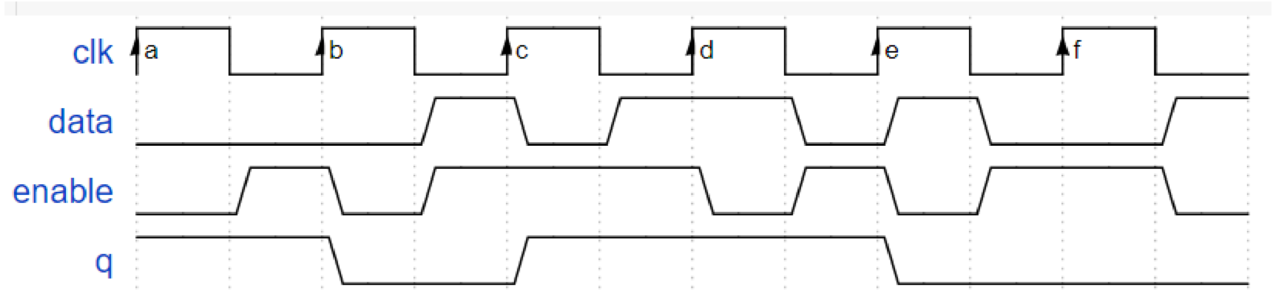
```
entity ff_example is
  port(
     d, clk, rst_n: in  std_logic;
                q: out std_logic);
end entity;
architecture rtl of ff_example is
begin
 ff_d_rst: process (clk, rst_n)



 end process ff_d_rst;
end rtl;
```

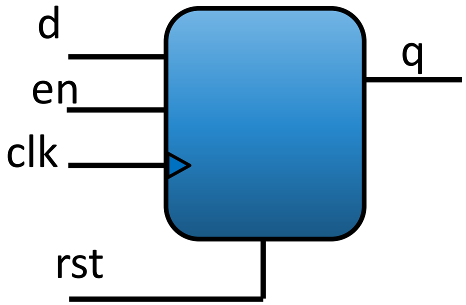
D-ff with synchronous reset



```
entity ff_d_srst is
  port(
  d, clk, rst: in  std_logic;
            q: out std_logic);
end entity;
architecture rtl of ff_d_srst is
begin
 ff_d_srst: process (clk)
begin




end process ff_d_srst;
end rtl;
```

D-ff with async. reset and enable

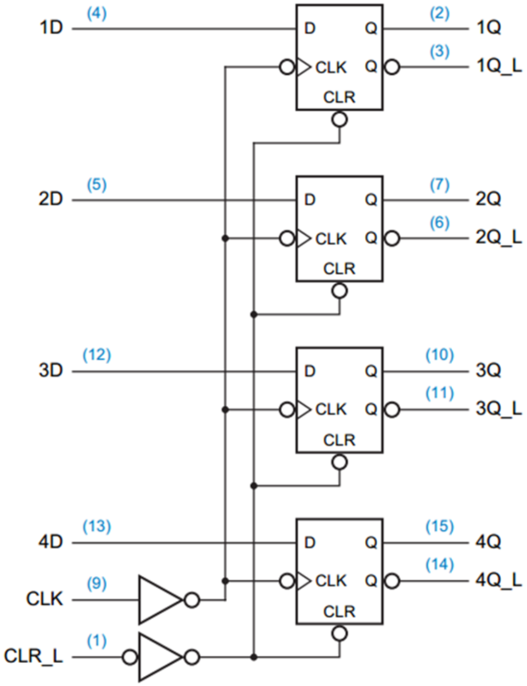




```
entity ff_d_en_rst is
  port(
  d, clk, en, rst: in  std_logic;
  q:                 out std_logic);
end entity;
```

```
end entity;
architecture rtl of ff_d_en_rst is
begin
ff_d_en_rst: process (clk, rst)

 end process ff_d_en_rst;
end rtl;
```

Registers



```
entity reg_d_rst is
 generic(width:= 4);
 port(
   d        : in  std_logic_vector(width-1 downto 0);
   clk, clr_l: in  std_logic;
   q        : out std_logic_vector(width-1 downto 0));
end entity;
architecture rtl of reg_d_rst is
begin
 reg_d_arst: process (clk,clr_l)
 begin
  if (clr_l = '1') then
      q <= (others =>'0');     -- q <= "0000"
  elsif(rising_edge (clk)) then
      q <= d;
  end if;
 end process reg_d_arst;
end rtl;
```

What is the implementation result??

```
library ieee;
use ieee.std_logic_1164.all;
entity shift_pi_po_x8 is
   port(
         clk, clr   : in  std_logic;
        serial_in   : in  std_logic;
        data_out     : out std_logic_vector(7 downto 0);
end shift_pi_po_x8;
architecture behav of shift_si_so_x4 is
  signal data_out_temp: std_logic_vector(3 downto 0);
 begin
 shift_proc: process(clk, clr)
 begin
     if (clr = '0') then
        data_out_temp <= others(=>'0‘);
     elsif (rising_edge(clk)) then
        data_out_temp <= serial_in & data_out_temp(3 downto 1);
     end if;
 end process shift_proc;
 data_out <= data_out_temp;
 end behave;
```

Shift Register : 74x194

```
architecture behav of shift_74x194 is
 signal temp_q: std_logic_vector(3 downto 0);
 signal ctrl  : std_logic_vector(1 downto 0);
begin
ctrl <= s0 & s1;
shift_proc: process(clk, clr_n)
begin
    if (clr_n = '0') then
         temp_q <= (others => '0');
    elsif (rising_edge(clk)) then



end process;
q <= temp_q;
end behav;
```

	Inputs		Next state			
Function	S1	S0	***QA****	***QB****	***QC****	***QD****
Hold	0	0	QA	QB	QC	QD
Shift right	0	1	RIN	QA	QB	QC
Shift left	1	0	QB	QC	QD	LIN
Load	1	1	A	B	C	D

Counter

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

entity counter_nbits is
  generic(cnt_w: natural:= 4)
  port (
         -- clock & reset inputs
         clk      : in std_logic;
         rst      : in std_logic;
         -- ouptuts
         count    : out std_logic_vecto
0));
end counter_nbits;
```

```vhdl
architecture rtl of counter_nbits is
 -- signal declarations
 signal count_i: unsigned(cnt_w-1 downto 0);
begin
count_proc: process(clk, rst)
 begin
  if(rst='0') then
    count_i <= (others => '0');
  elsif(rising_edge(clk)) then
    count_i <= count_i + 1;
  endif;
end process count_proc;
 count <= std_logic_vector(count_i);
end architecture rtl;
```

Up/Down Counter

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

entity counter_ud is
  generic(cnt_w: natural:= 4)
  port (
         -- clock & reset inputs
         clk      : in std_logic;
         rst      : in std_logic;
         -- control input signals
         up_dw    : in std_logic;
         -- ouptuts
         count    : out std_logic_vector(c
0));
end counter_ud;
```

```vhdl
architecture rtl of counter_ud is
-- signal declarations
signal count_i: unsigned(cnt_w-1 downto 0);

begin
 count_proc: process(clk, rst)
 begin
  if(rst='0') then
    count_i <= (others => '0');
  elsif(rising_edge(clk)) then
    if(up_dw = '1') then -- up
       count_i <= count_i + 1;
    else                 -- down
       count_i <= count_i - 1;
    end if;
   end if;
 end process count_proc;

 count <= std_logic_vector(count_i);

end architecture rtl;
```

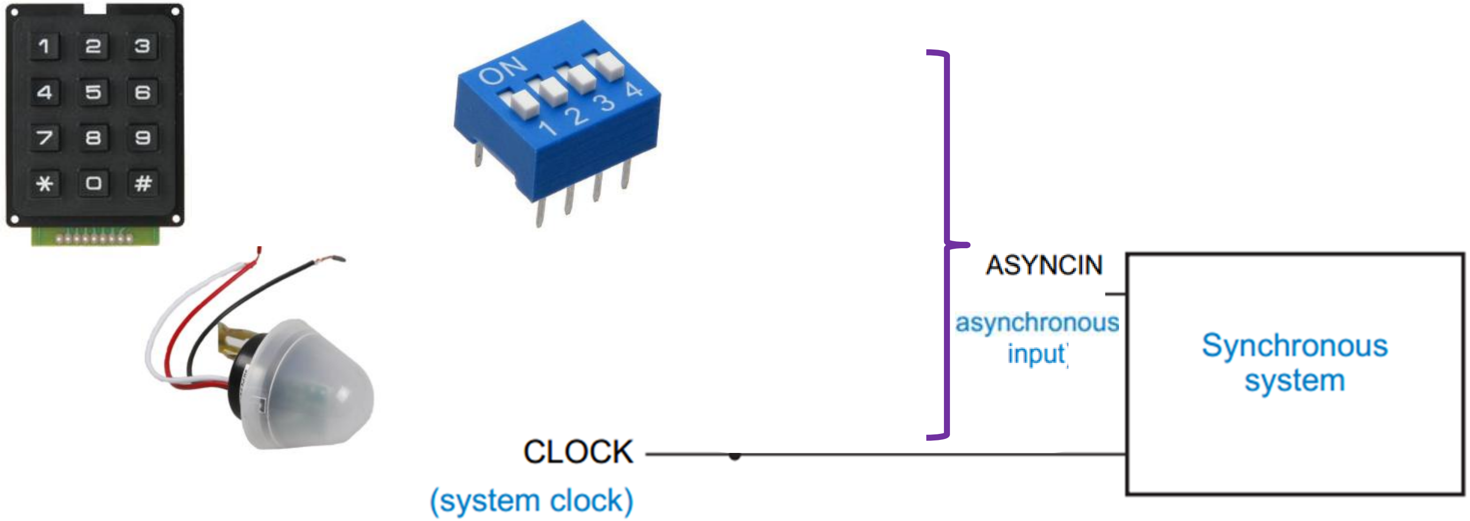
Up/Down Counter - Integers

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

entity counter_ud_i is
  generic(cnt_w: natural:= 4)
  port (
         -- clock & reset inputs
         clk      : in std_logic;
         rst_n    : in std_logic;
         -- ouptuts
         count    : out std_logic_
0));
end counter_ud_i;
```

```
architecture rtl of counter_ud_i is
begin
 count_proc: process(clk, rst)
    variable count_i: integer range 0 to 255;
begin
  if(rst_n = '0') then
     count_i := 0;
  elsif(rising_edge(clk)) then
    if(count_i = 255) then
       count_i := 0;
    else
       count_i := count_i + 1;
    end if;
   end if;
 end process count_proc;
 count <= std_logic_vector(to_unsigned(count_i,8));
end architecture rtl;
```

?

Asynchronous Inputs



ASYNCIN

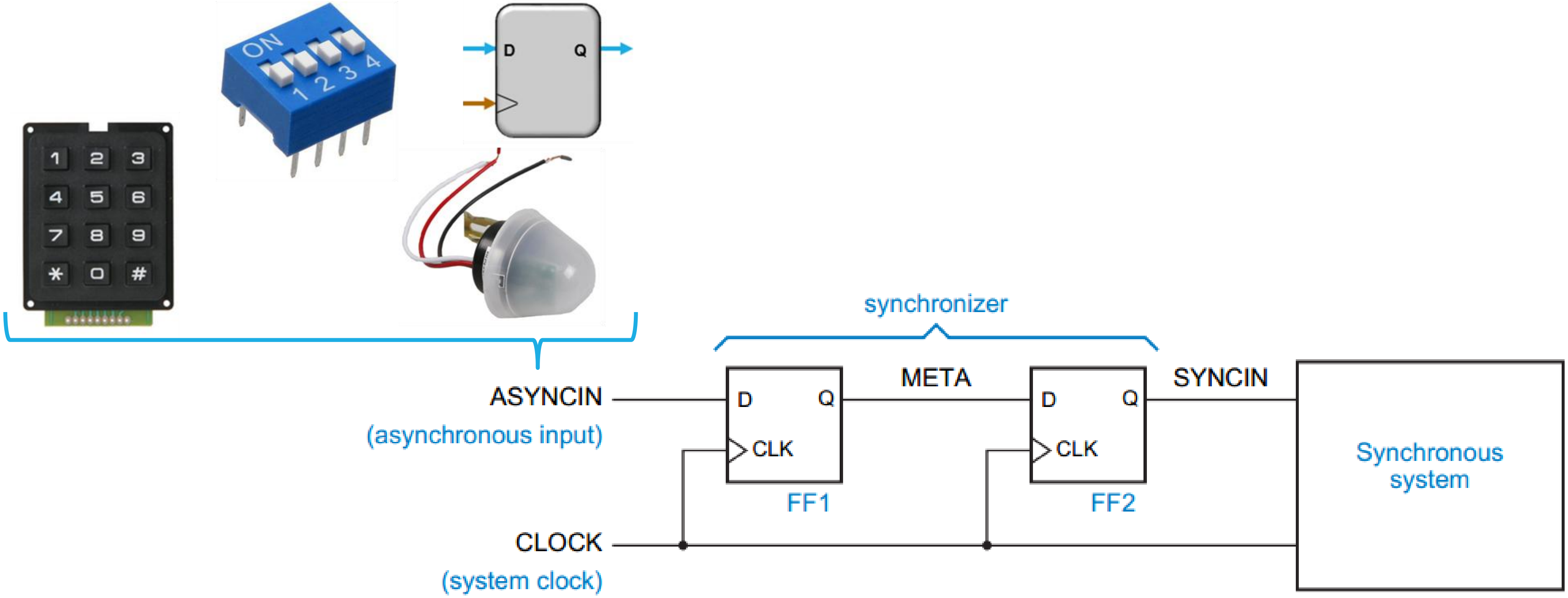
asynchronous input

Synchronous system

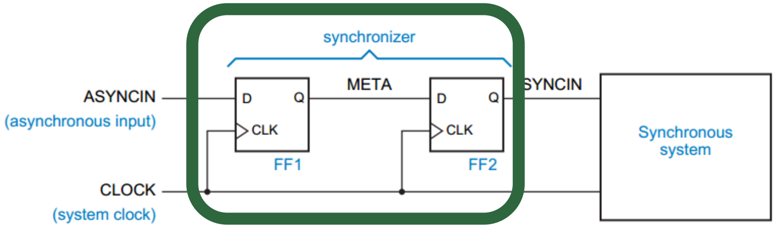
CLOCK

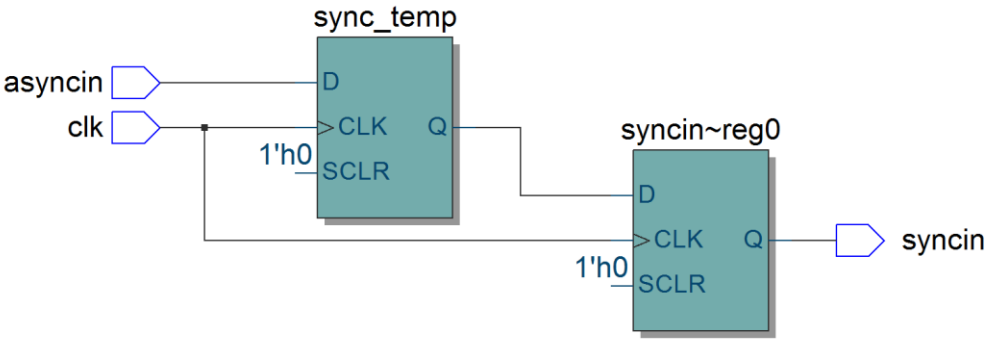
(system clock)

Synchronizer



Synchronizer

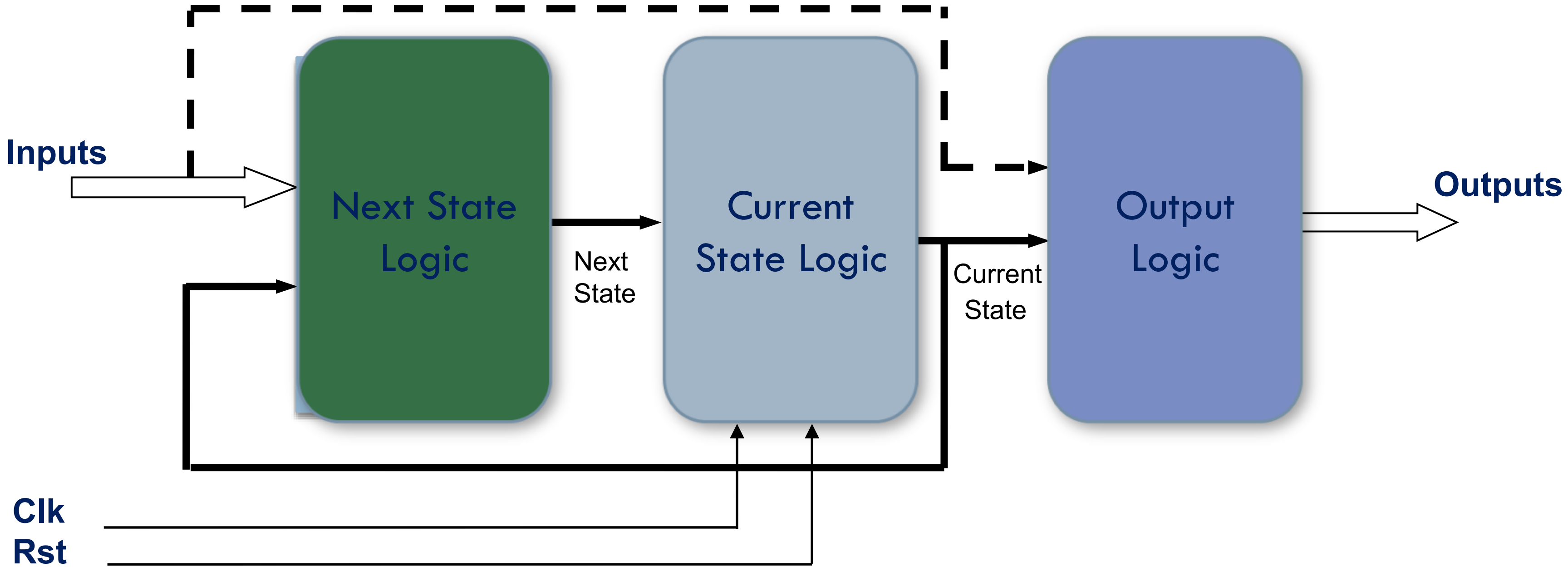




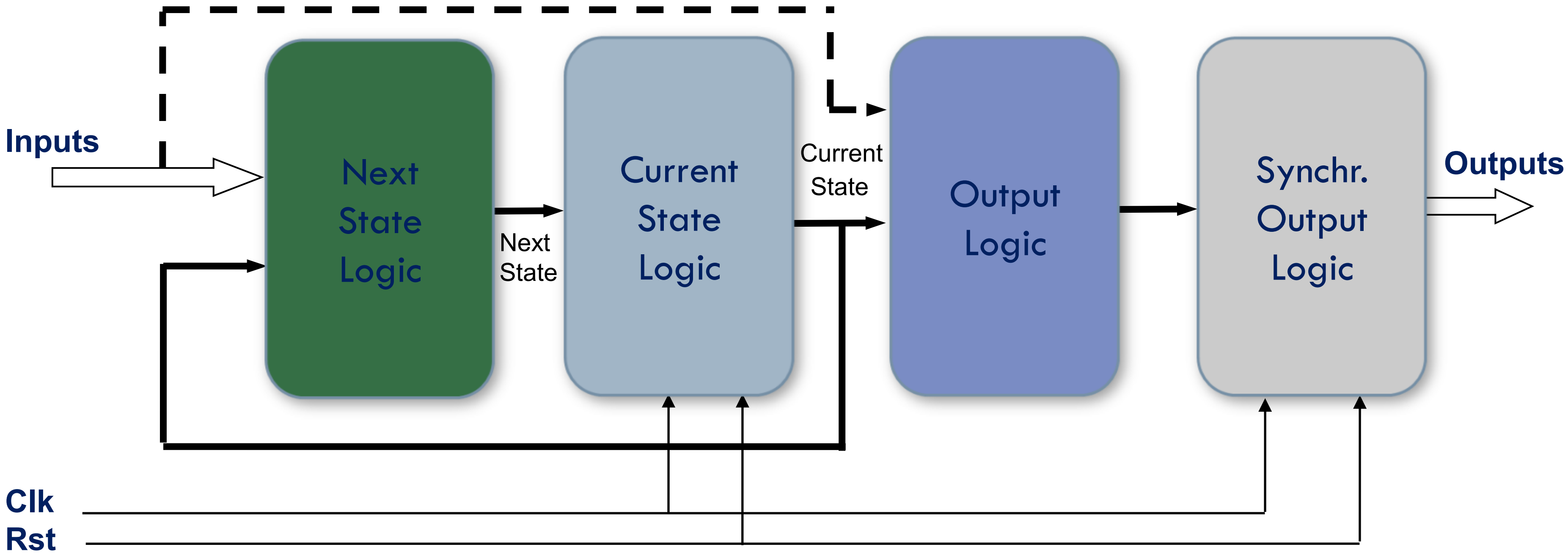
```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity synchronizer is
   port(
         clk        : in  std_logic;
         asyncin    : in  std_logic;
          syncin    : out std_logic);
end synchronizer;
architecture behave of synchronizer is
  signal sync_temp: std_logic;
 begin
 sync_proc: process(clk)
 begin
      if (rising_edge(clk)) then
          sync_temp <= asyncin;
          syncin    <= sync_temp;
      end if;
  end process;
 end behave;
```

FINITE STATE MACHINES (FSM) DESCRIPTION IN VHDL

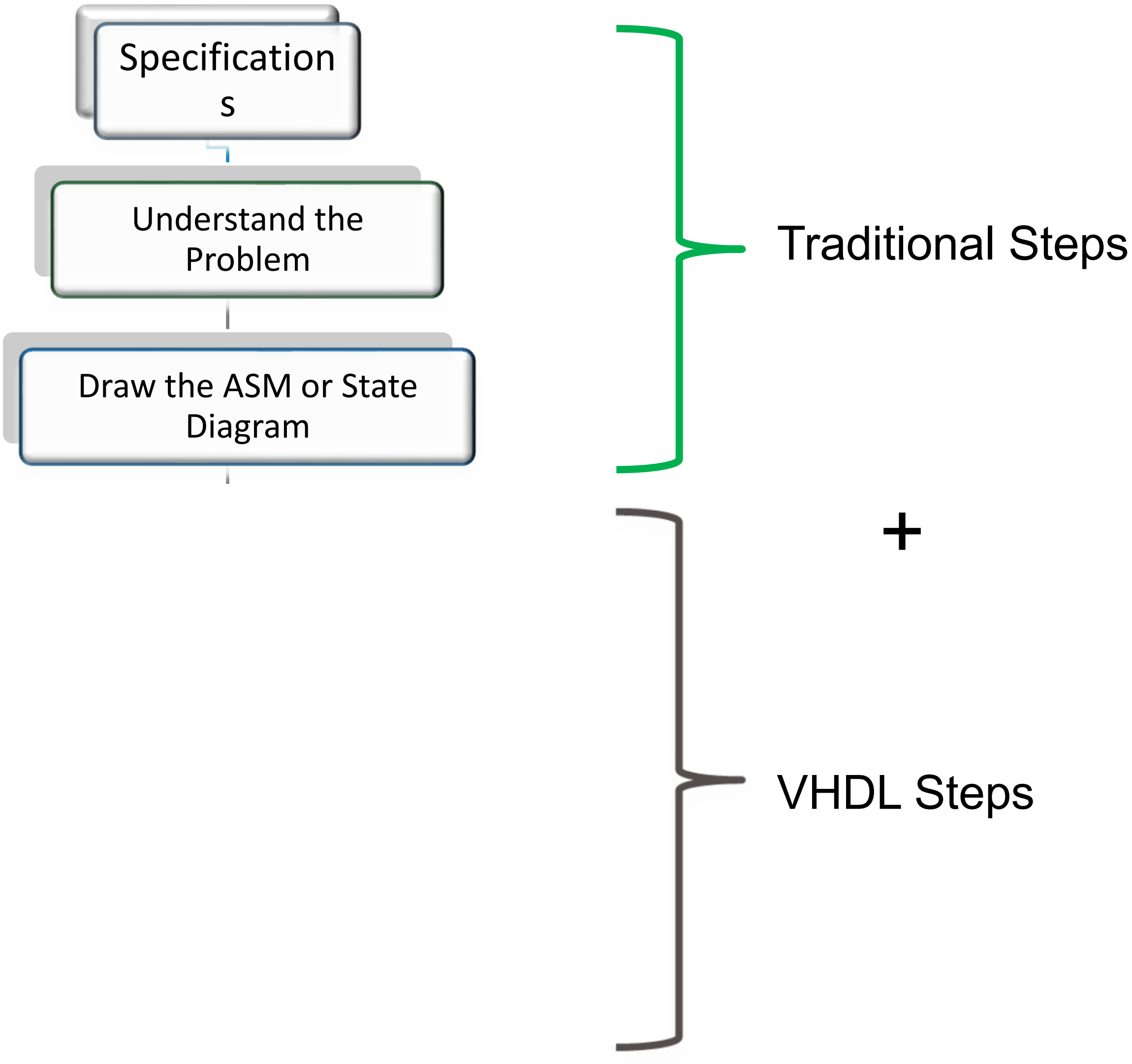
State Machine General Scheme 1



State Machine General Scheme 2



FSM VHDL General Design Flow



Specifications

Understand the Problem

Draw the ASM or State Diagram

Traditional Steps

+

VHDL Steps

FSM Enumerated Type Declaration

Declare an enumerated data type with *values (names) that symbolize the states of the state machine*

Symbolic State Names

```
-- declare the states of the state-machine
-- as enumerated type
type FSM_States is(IDLE,START,STOP_1BIT,PARITY,SHIFT);
```

Declare the signals for the next state and current state of the state machine as signal of the enumerated data type already defined for the state machine

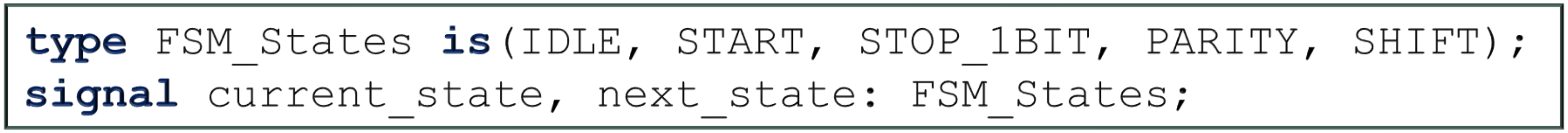
```
-- declare signals of FSM_States type
signal current_state, next_state: FSM_States;
```

The only values that `current_state` and `next_state` can hold are:
`IDLE,START,STOP_1BIT,PARITY,SHIFT`

FSM Encoding Techniques

State Assignment

- During synthesis each ***symbolic state name*** has to be mapped to a ***unique binary representation***



```
type FSM_States is(IDLE, START, STOP_1BIT, PARITY, SHIFT);
signal current_state, next_state: FSM_States;
```

- A good state assignment can ***reduce*** the circuit ***size*** and ***increase*** the ***clock rate*** (by reducing propagation delays)

- The hardware needed for the implementation of the next state logic and the output logic is ***directly related*** to the state assignment selected

FSM Encoding Schemes

An FSM with *n* symbolic states requires at least [$\log_2 n$] bits to encode all the possible symbolic values

Commonly used state assignment schemes:

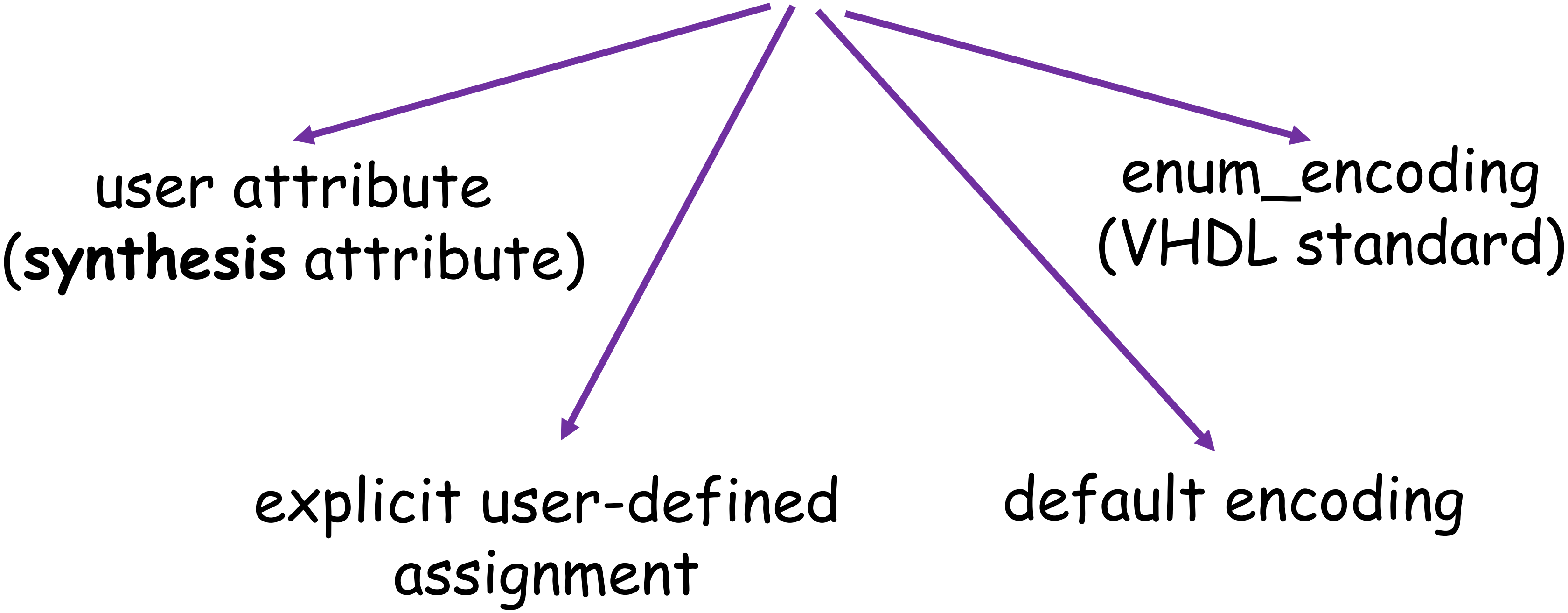
- □ **Binary**: assign states according to a binary sequence
- □ **Gray**: use the Gray code sequence for assigning states
- □ **One-hot**: assigns one ‘hot’ bit for each state
- □ **Almost one-hot**: similar to one-hot but add the all zeros code (initial state)

FSM Encoding Schemes

	Binary	Gray	One-Hot	Almost One-hot
idle	000	000	00001	0000
start	001	001	00010	0001
stop_1bit	010	011	00100	0010
parity	011	010	01000	0100
shift	100	110	10000	1000

Encoding Schemes in VHDL

During ***synthesis*** each ***symbolic state name*** has to be mapped to a ***unique binary representation***



user attribute
(**synthesis** attribute)

enum_encoding
(VHDL standard)

explicit user-defined
assignment

default encoding

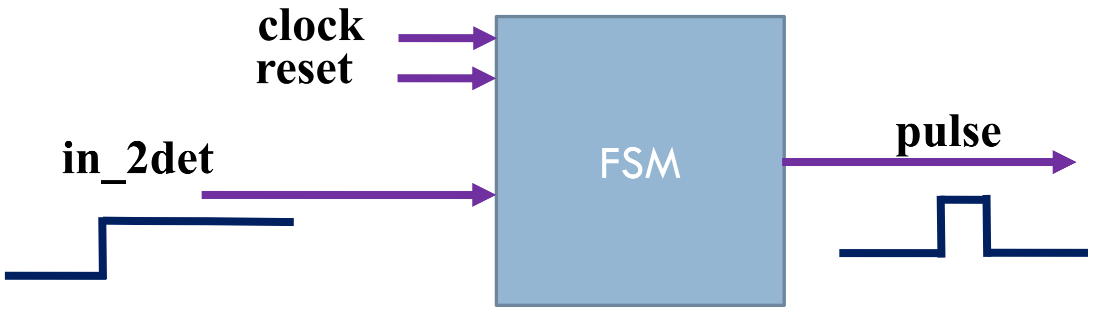
Results for Different Encoding Schemes

19 states, state machine

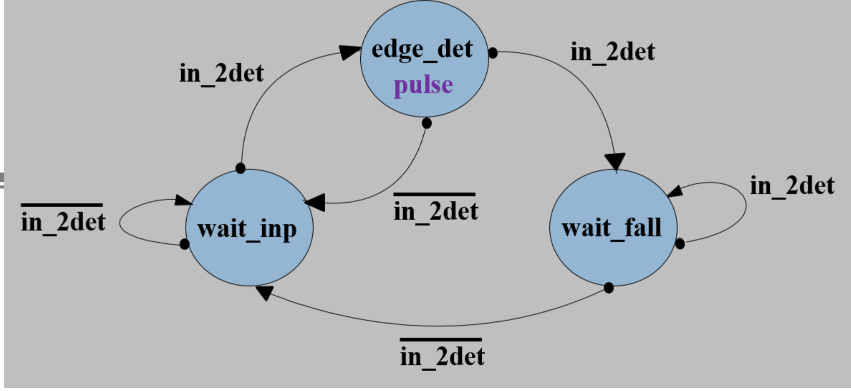
	One-hot safe	One-hot	Gray	Gray-Safe	Binary	Johnson
Total combinational functions	556	523	569	566	561	573
Dedicated logic registers	215	215	201	201	201	206
Max. frq.	187.3	175.22	186.39	180.6	197.63	186.22

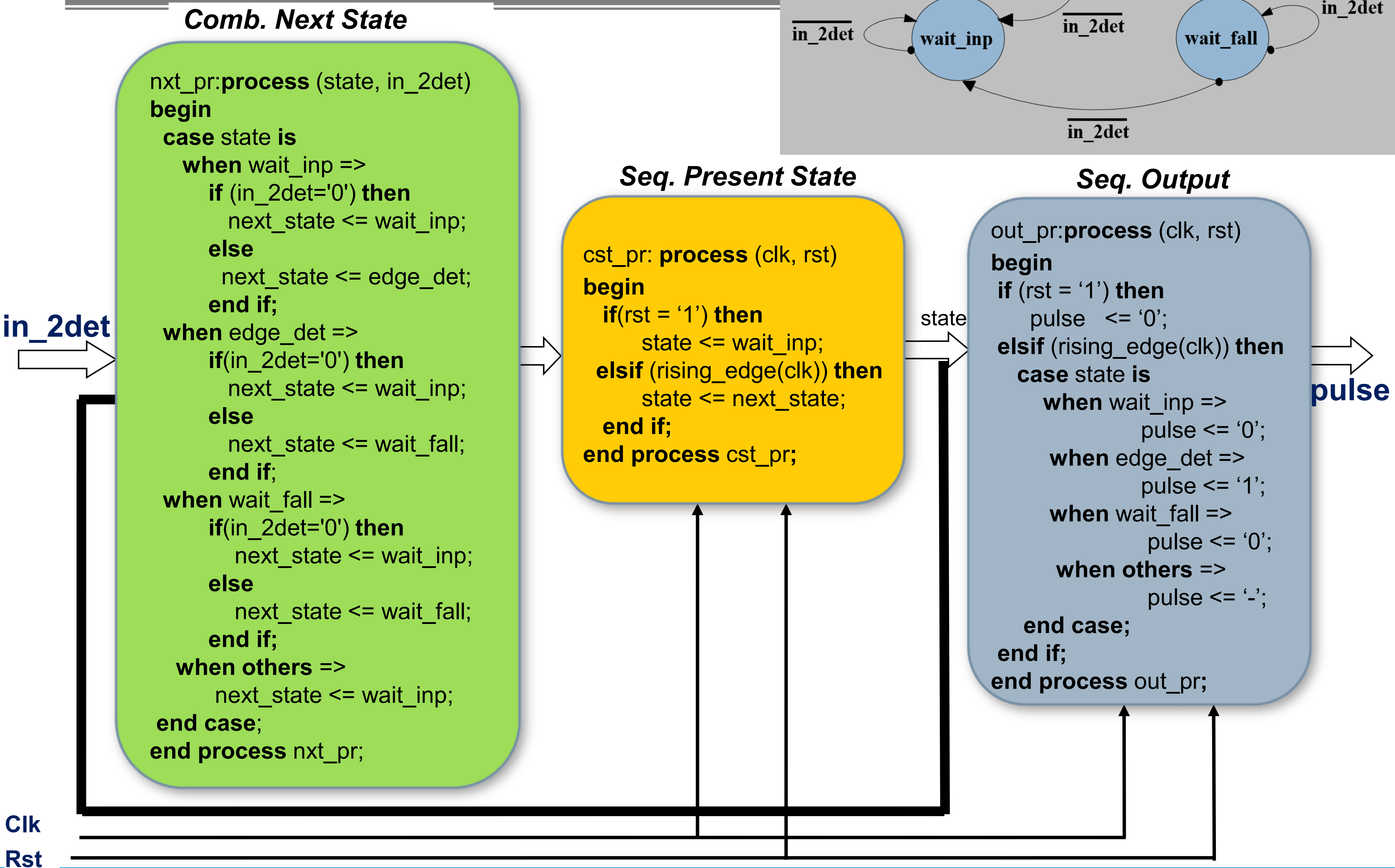
State Machine VHDL Coding - Example

Describe in VHDL an FSM that generate a pulse per each rising edge of the input.



FSM VHDL Coding





Comb. Next State

```
nxt_pr:process (state, in_2det)
begin
  case state is
     when wait_inp =>
         if (in_2det='0') then
            next_state <= wait_inp;
         else
           next_state <= edge_det;
         end if;
  when edge_det =>
         if(in_2det='0') then
            next_state <= wait_inp;
         else
            next_state <= wait_fall;
         end if;
  when wait_fall =>
         if(in_2det='0') then
             next_state <= wait_inp;
         else
             next_state <= wait_fall;
         end if;
    when others =>
           next_state <= wait_inp;
 end case;
end process nxt_pr;
```

Seq. Present State

```
cst_pr: process (clk, rst)
begin
   if(rst = '1') then
         state <= wait_inp;
  elsif (rising_edge(clk)) then
         state <= next_state;
   end if;
end process cst_pr;
```

Seq. Output

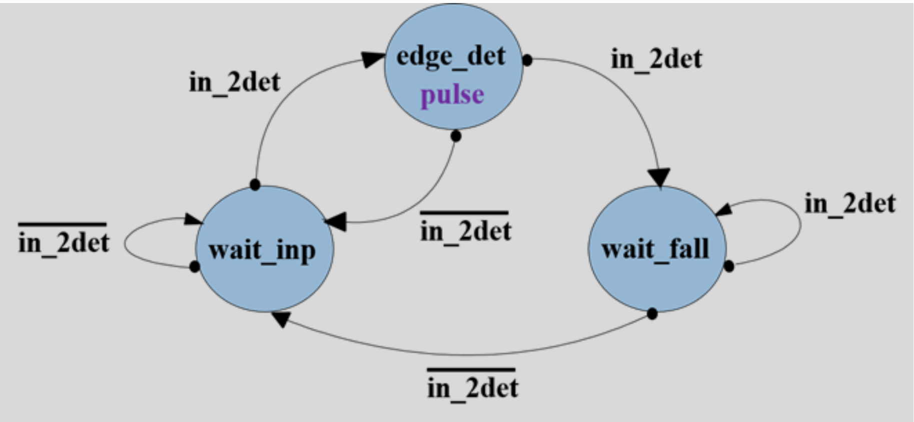
```
out_pr:process (clk, rst)
begin
 if (rst = '1') then
      pulse    <= '0';
 elsif (rising_edge(clk)) then
    case state is
        when wait_inp =>
                       pulse <= '0';
         when edge_det =>
                       pulse <= '1';
         when wait_fall =>
                        pulse <= '0';
          when others =>
                        pulse <= '-';
     end case;
 end if;
end process out_pr;
```

State Machine VHDL Coding (complete)

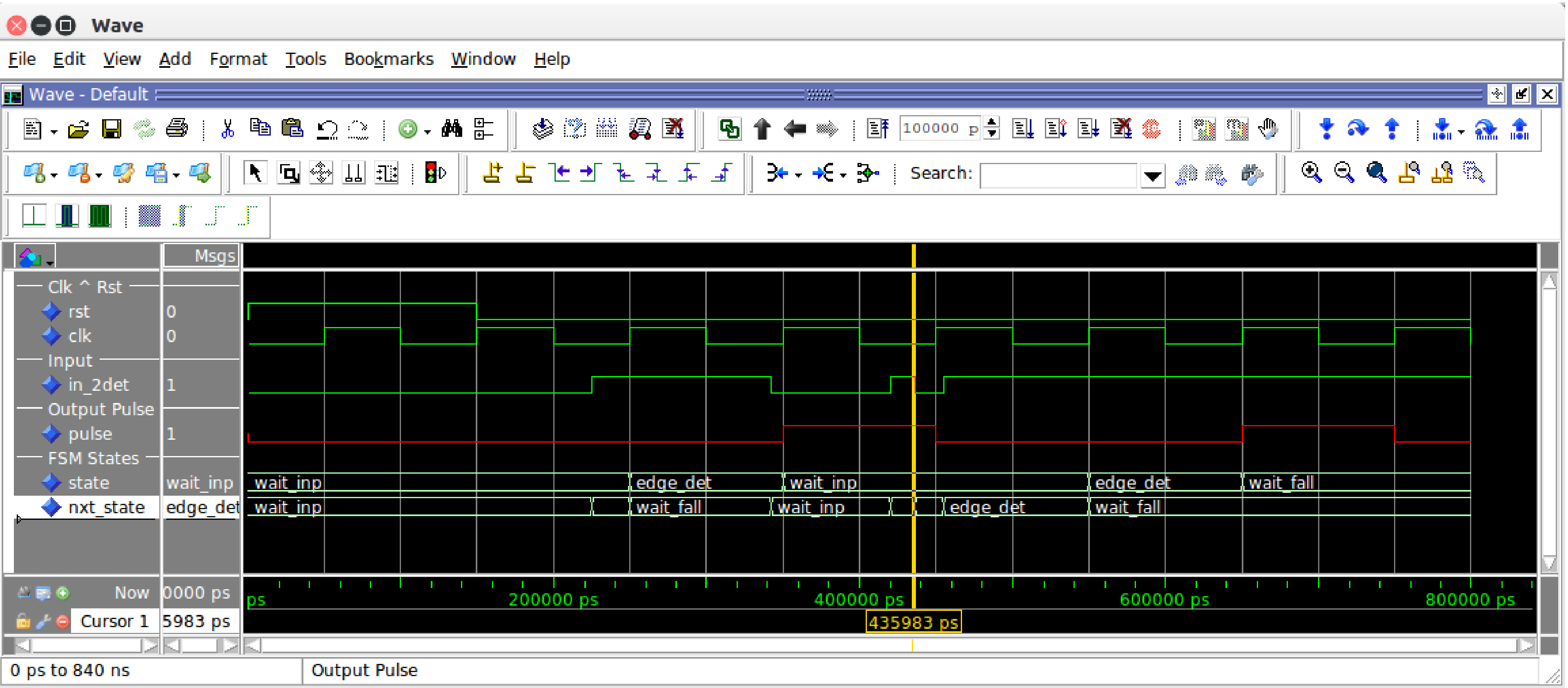
```vhdl
-- VHDL code example for an FSM
library ieee;
use ieee.std_logic_1164.all;

entity fsm _edge_detect is
  port(
        in_2det : in   std_logic;
        clk          : in  std_logic;
        rst          : in  std_logic;
        pulse       : out std_logic );
end entity fsm_edge_detect;

architecture beh of my_fsm  is

 -- fsm enumerated type declaration
 type fsm_states is (wait_inp, edge_det, wait_fall);

 -- fsm signal declarations
 signal next_state, state: fsm_states;

begin

-- current state logic
cst_pr: process (clk, rst)
begin
   if(rst = '1') then
         state <= wait_inp;
   elsif (rising_edge(clk)) then
         state <= next_state;
   end if;
end process cst_pr;
```

```vhdl
-- next state logic
nxt_pr:process (state, in_2det)
begin
  case state is
     when wait_inp =>
        if(in_2det='0') then
              next_state <= wait_inp;
        else
              next_state <= edge_det;
       end if;
     when edge_det =>
        if ….
              next_state <= .. ;
          ….
      when others =>
          ….
  end case;
end process nxt_pr;
- - output logic
out_pr:process (clk, rst)
begin
   if(rst = '1') then
          pulse   <= '0';
   elsif (rising_edge(clk)) then
     case state is
          when wait_inp  =>   pulse <= '0';
           . . .
          when others     =>   pulse <= '-';
     end case;
 end process out_pr;
end architecture beh;
```

FSM Simulation



edge_det
pulse
in_2det
in_2det
$\overline{\text{in\_2det}}$
wait_inp
$\overline{\text{in\_2det}}$
in_2det
wait_fall
$\overline{\text{in\_2det}}$



	Msgs
Clk ^ Rst	
rst	0
clk	0
Input	
in_2det	1
Output Pulse	
pulse	1
FSM States	
state	wait_inp
nxt_state	edge_det
Now	0000 ps
Cursor 1	5983 ps

state: wait_inp, edge_det, wait_inp, edge_det, wait_fall

nxt_state: wait_inp, wait_fall, wait_inp, edge_det, wait_fall

0 ps to 840 ns | Output Pulse

Another Ex.: Memory Controller FSM

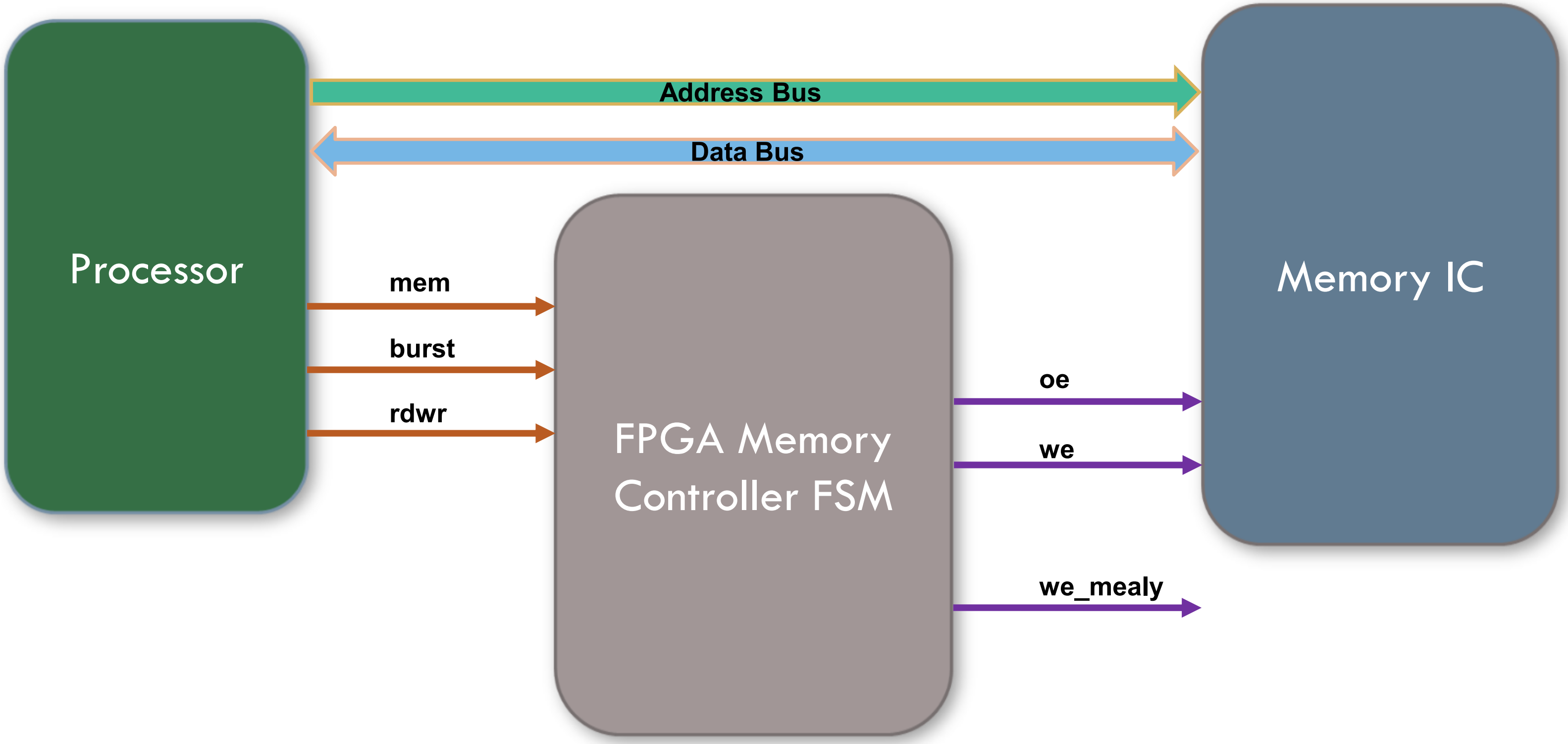
Let's try to obtain an state diagram of a hypothetical memory controller FSM that has the following specifications:

The controller is between a processor and a memory chip, interpreting commands from the processor and then generating a control sequence accordingly. The commands, ***mem***, ***rw*** and ***burst***, from the processor constitute *the input signals* of the FSM. The ***mem*** signal is asserted to high when a memory access is required. The ***rdwr*** signal indicates the type of memory access, and its value can be either '1' or '0', for memory read and memory write respectively. The ***burst*** signal is for a special mode of a memory read operation. If it is asserted, four consecutive read operations will be performed. The memory chip has two control signals, **oe** (for output enable) and ***we*** (for write enable), which need to be asserted during the memory read and memory write respectively. The two output signals of the FSM, **oe** and ***we***, are connected to the memory chip's control signals. For comparison purpose, let also add an artificial Mealy output signal, ***we_mealy*** , to the state diagram. Initially, the FSM is in the ***idle*** state, waiting for the mem command from the processor. Once ***mem*** is asserted, the FSM examines the value of ***rdwr*** and moves to either the ***read1*** or the **write** state. The input conditions can be formalized to logic expressions, as shown below:

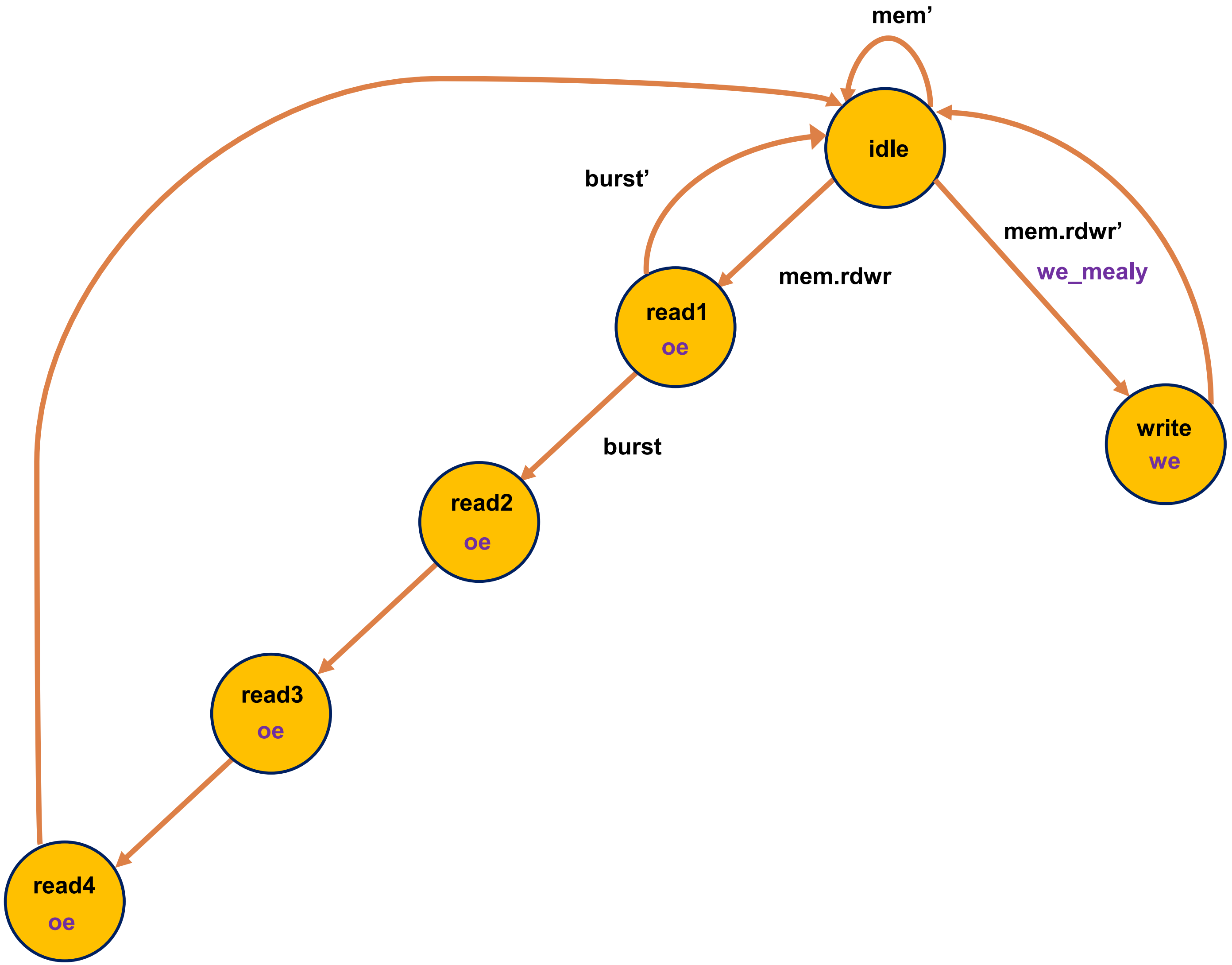
- ***mem'*** : represents that *no* memory operation is required (mem='0')
- ***mem.rdwr***: represents that a memory *read* operation is required (mem=rdwr='1').
- ***mem.rdwr'***: represents that a memory *write* operation is required (mem='1'; rdwr='0')

Based on an example from the "RTL Hardware Design Using VHDL" book, By Pong Chu

Memory Controller FSM



Memory Controller FSM



Memory Controller FSM - VHDL Code

```
library ieee ;
use ieee.std_logic_1164.all;
entity mem_ctrl is
port (
       clk, reset      : in  std_logic;
       mem, rdwr, burst: in  std_logic;
       oe, we, we_mealy: out std_logic
      );
end mem_ctrl ;
architecture mult_seg_arch of mem_ctrl is
 type fsm_states_type is
       (idle, read1, read2, read3, read4, write);
 signal crrnt_state, next_state: fsm_states_type;
begin
```

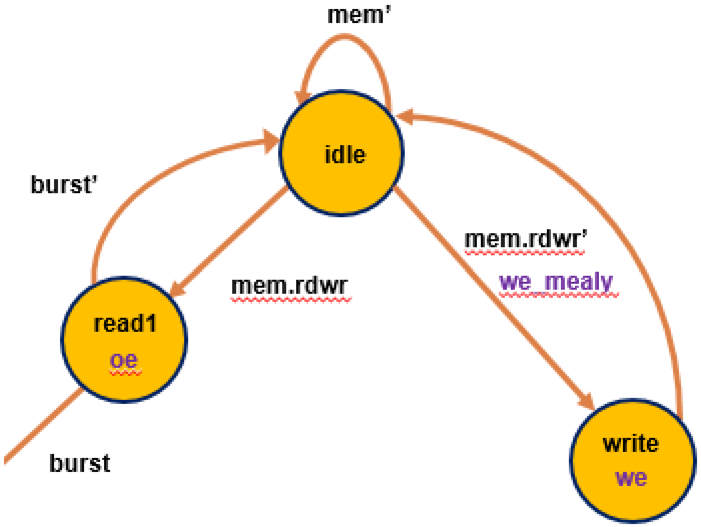
Memory Controller FSM - VHDL Code

```
-- current state process
cs_pr: process (clk, reset)
begin
 if(reset = ’1’) then
     crrnt_state <= idle ;
 elsif(rising_edge(clk))then
     crrnt_state <= next_state;
 end if;
end process cs_pr;
```

Memory Controller FSM – VHDL Code

- Next state process (1)

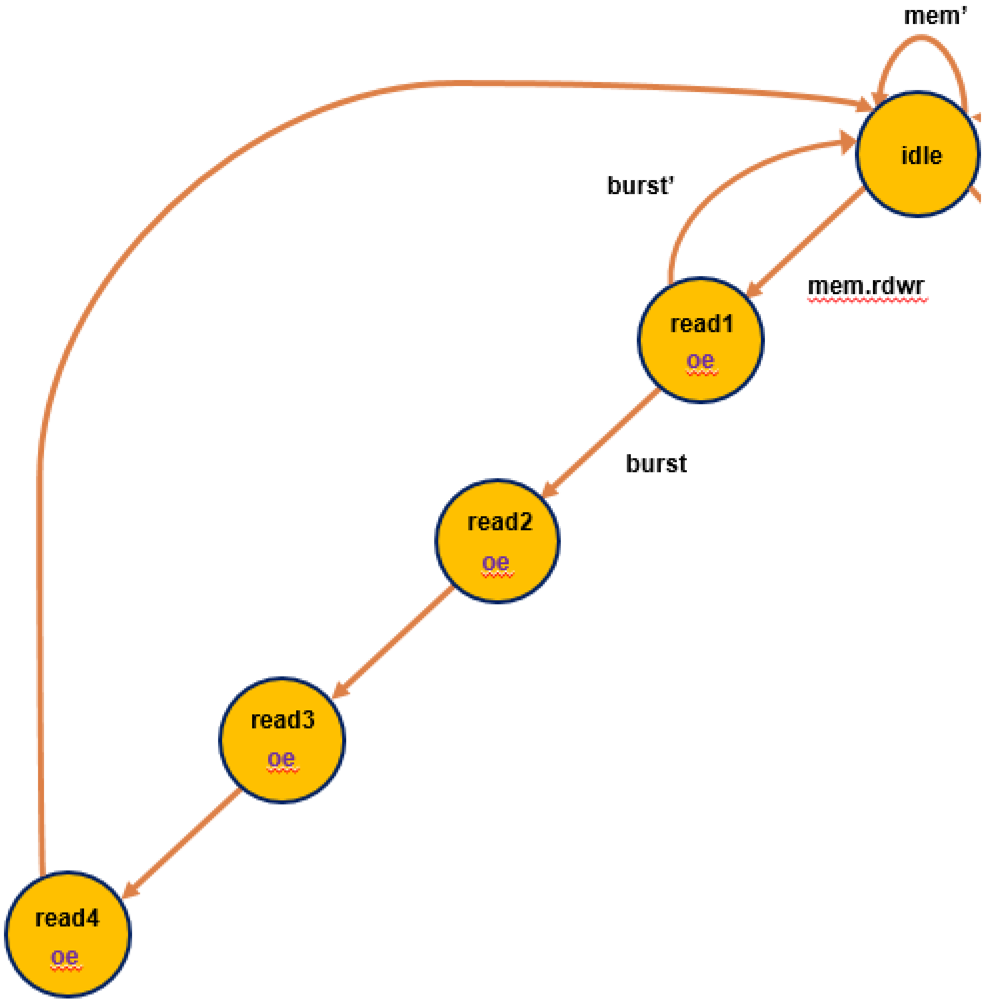
```
-- next-state logic
nxp:process(crrnt_state,mem,rdwr,burst)
begin
 case crrnt_state is
   when idle =>
     if mem = ’1 ’ then
       if rdwr = ’1’ then
         next_state <= read1;
       else
         next_state <= write;
       end if;
     else
        next_state <= idle;
     end if;
   when write =>
     next_state <= idle;
```



Memory Controller FSM – VHDL Code

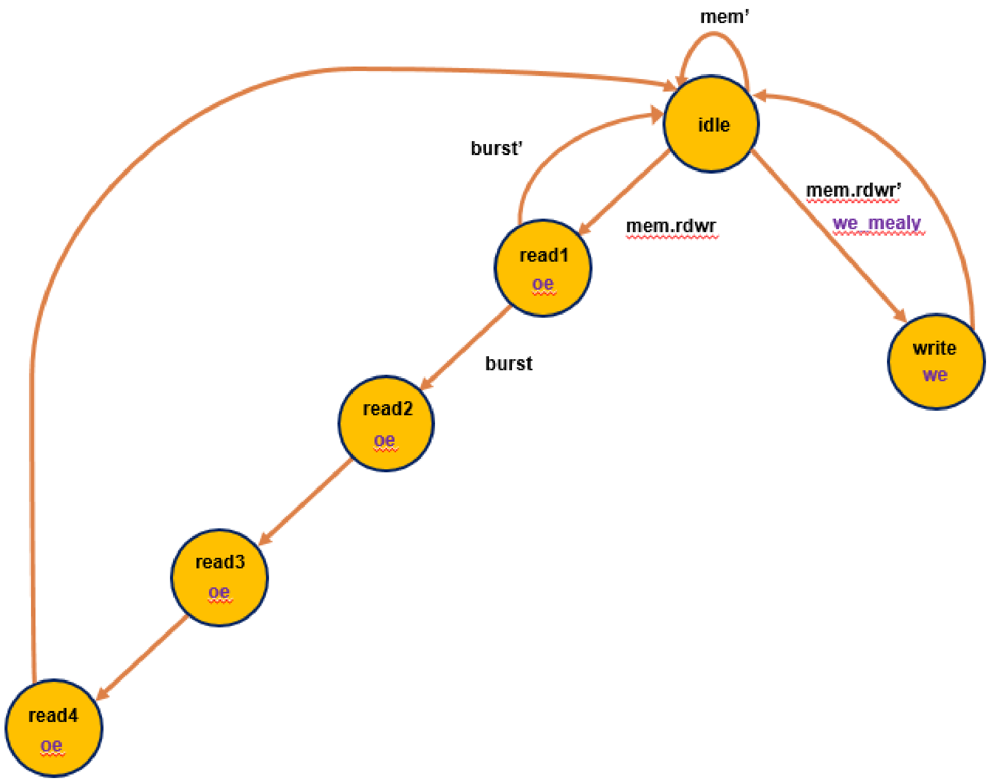
- Next state process (2)

```
 when read1 =>
   if (burst = '1') then
     next_state <= read2;
   else
     next_state <= idle;
   end if;
 when read2 =>
   next_state <= read3;
 when read3 =>
   next_state <= read4;
 when read4 =>
   next_state <= idle;
 when others =>
   next_state <= idle;
end case;
end process nxp;
```



Memory Controller FSM – VHDL Code

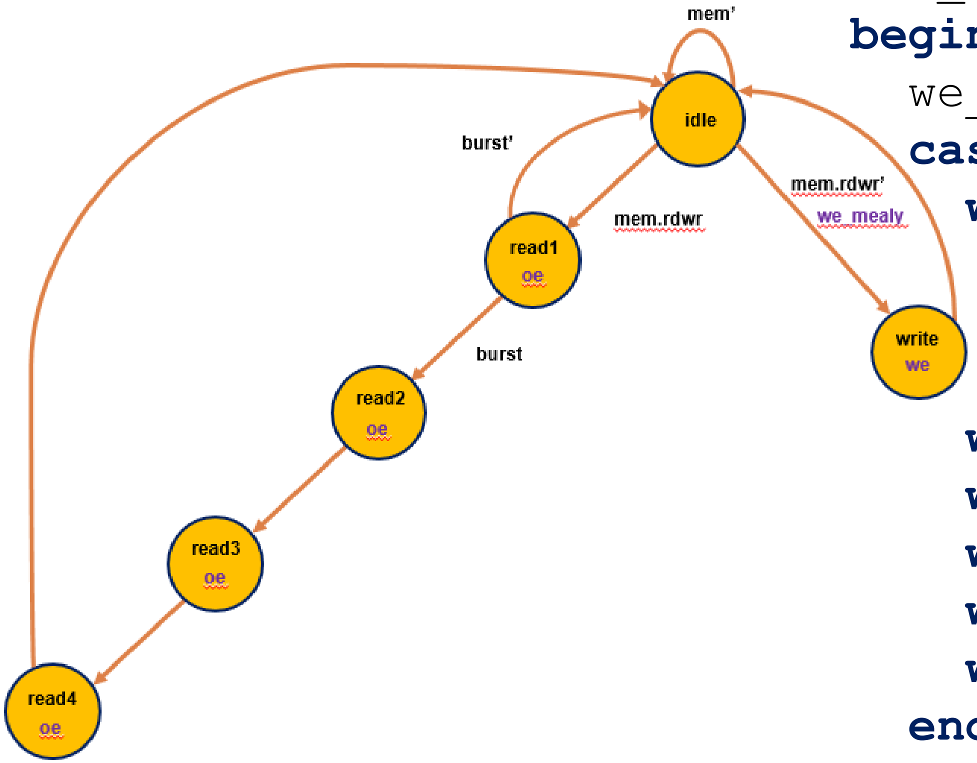
- Moore outputs process



```
-- Moore output logic
moore_pr: process (crrnt_state)
begin
  we <= '0'; -- default value
  oe <= '0'; -- default value
  case crrnt_state is
     when idle => null;
     when write =>
            we <= '1';
     when read1 =>
            oe <= '1';
     when read2 =>
            oe <= '1';
     when read3 =>
            oe <= '1';
     when read4 =>
            oe <= '1';
     when others => null;
  end case ;
end process moore_pr;
```

Memory Controller FSM – VHDL Code

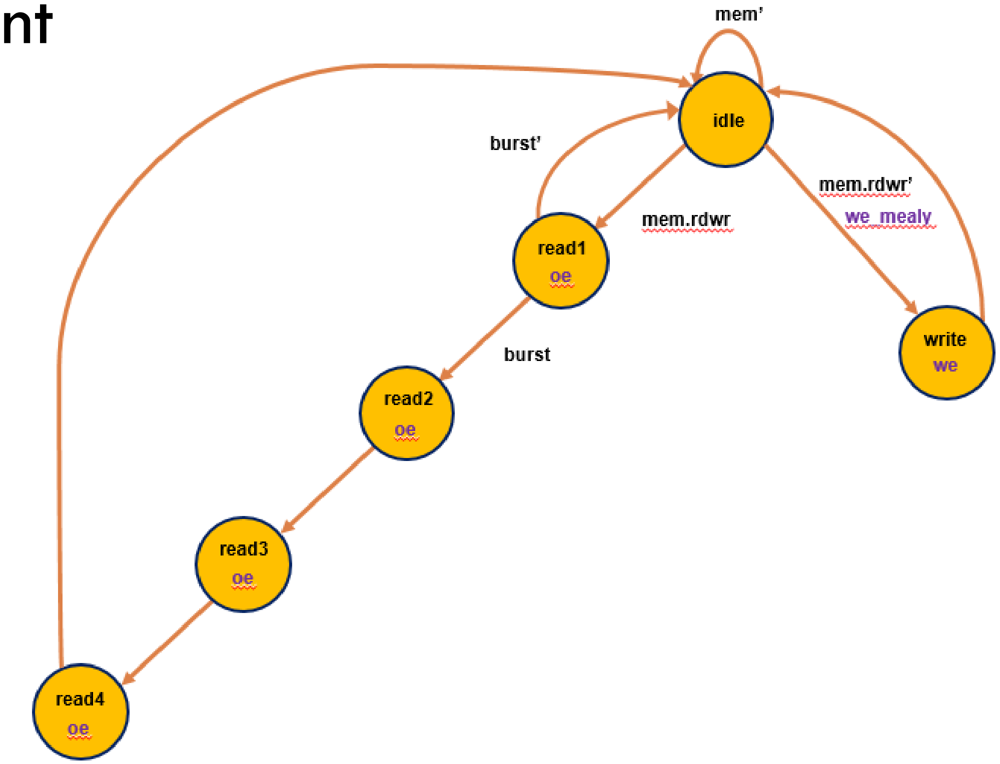
- Mealy output process



```
-- Mealy output logic
mly_pr: process(crrt_state,mem,rdwr)
begin
  we_me <= ’0’; -- default value
  case state_reg is
    when idle =>
      if (mem=’1’)and(rdwr =’0’)then
         we_me <= ’1’;
      end if;
    when write => null;
    when read1 => null;
    when read2 => null;
    when read3 => null;
    when read4 => null;
  end case;
end process mly_pr;
```

Memory Controller FSM – VHDL Code

- Mealy output statement

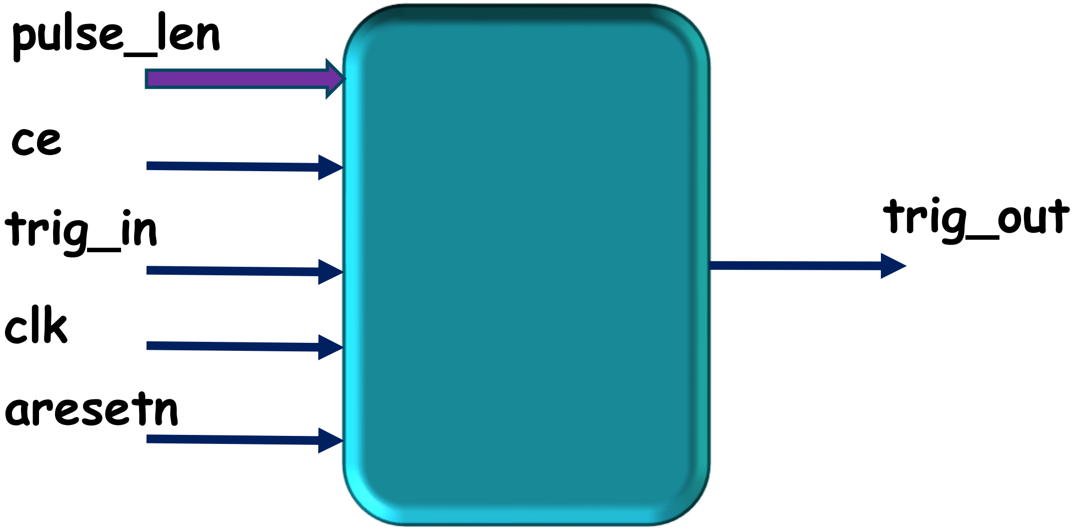


```
-- Mealy output logic
we_me <= '1' when ((crrnt_state=idle) and (mem='1') and(rdwr='0'))
             else
          '0';
```

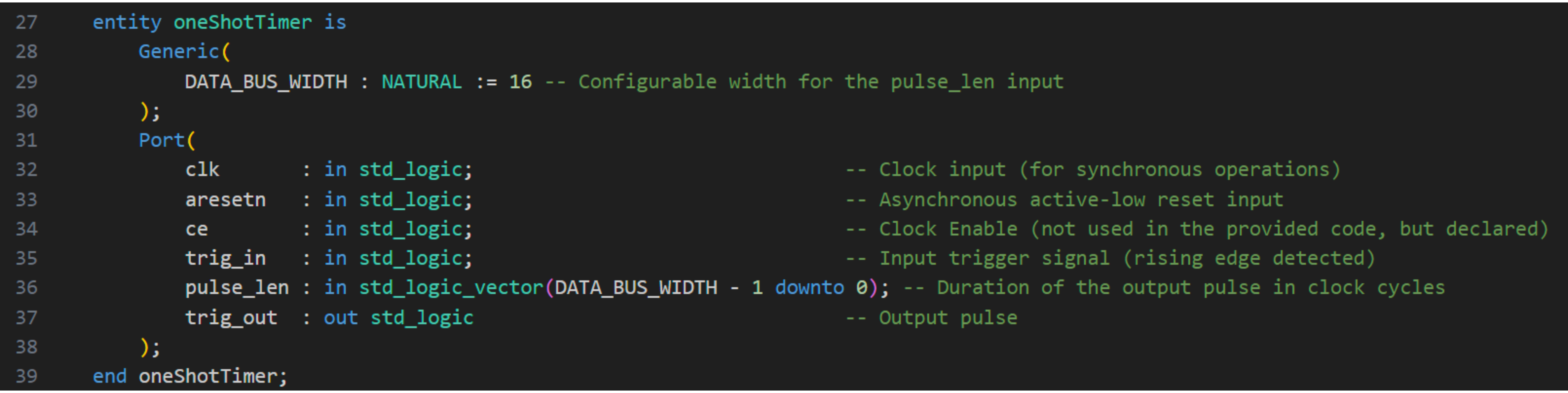
VHDL Code to be used in the Labs

Synchronous one-shot-timer

The purpose of the module **oneShotTimer** is to generate a single output pulse of a predefined duration (**pulse_len**) every time it receives a rising edge on its **trig_in** input.



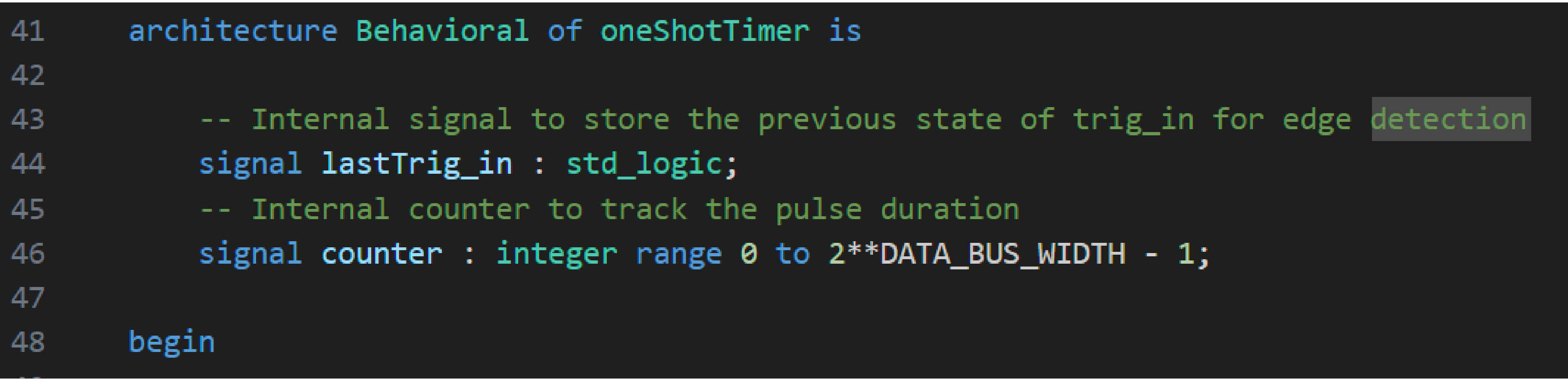
Synchronous one-shot-timer



```vhdl
entity oneShotTimer is
    Generic(
        DATA_BUS_WIDTH : NATURAL := 16 -- Configurable width for the pulse_len input
    );
    Port(
        clk       : in std_logic;                                   -- Clock input (for synchronous operations)
        aresetn   : in std_logic;                                   -- Asynchronous active-low reset input
        ce        : in std_logic;                                   -- Clock Enable (not used in the provided code, but declared)
        trig_in   : in std_logic;                                   -- Input trigger signal (rising edge detected)
        pulse_len : in std_logic_vector(DATA_BUS_WIDTH - 1 downto 0); -- Duration of the output pulse in clock cycles
        trig_out  : out std_logic                                   -- Output pulse
    );
end oneShotTimer;
```

- **generic(DATA_BUS_WIDTH : NATURAL := 16):**
  - **generic** is a way to pass constant values into an entity from its instantiation.
  - **DATA_BUS_WIDTH** defines the bit width of the **pulse_len** input. It defaults to 16 bits, meaning **pulse_len** can specify a pulse length from 0 to 65535 clock cycles.
- **Port (...):** Defines the input and output signals of the block:
  - **clk**: Standard clock input. All internal state changes occur on its rising edge.
  - **aresetn**: Asynchronous active-low reset. When aresetn is '0', the timer is reset.
  - **ce**: Clock Enable. *Note: This port is declared but not used yet.*
  - **trig_in**: The input signal that, upon a rising edge, initiates the one-shot pulse.
  - **pulse_len**: An input vector that defines the desired duration of the output pulse in terms of clock cycles.
  - **trig_out**: The output signal that generates a pulse (goes high) for the specified **pulse_len** duration.

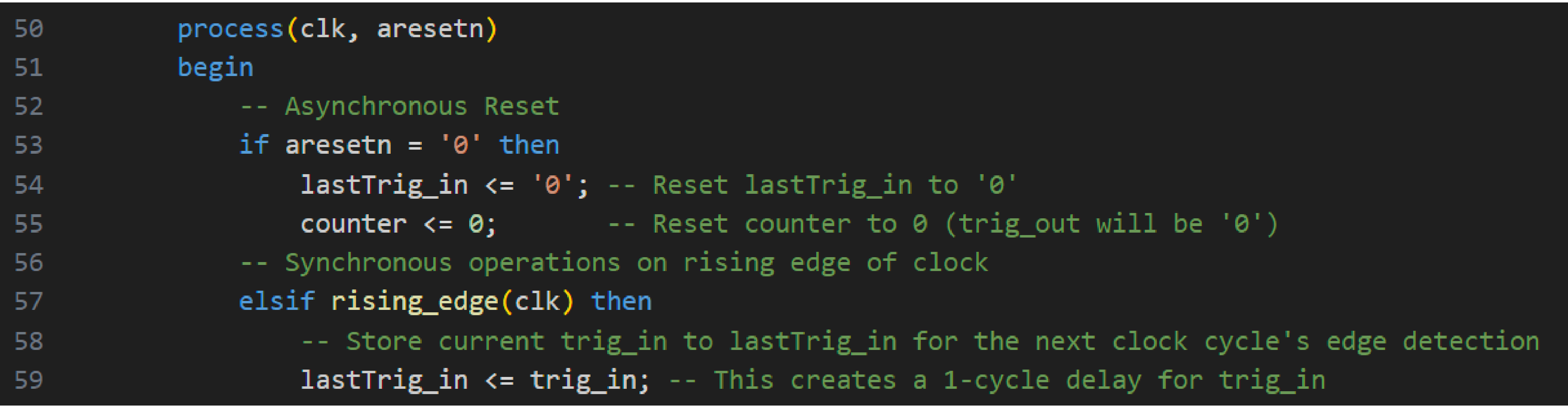
Synchronous one-shot-timer



```
architecture Behavioral of oneShotTimer is

    -- Internal signal to store the previous state of trig_in for edge detection
    signal lastTrig_in : std_logic;
    -- Internal counter to track the pulse duration
    signal counter : integer range 0 to 2**DATA_BUS_WIDTH - 1;

begin
```

- **architecture Behavioral of oneShotTimer** is: Defines the behavior or implementation of the **oneShotTimer** entity.
- **signal lastTrig_in : std_logic;**
    - Declares an internal signal named **lastTrig_in**.
    - This signal is crucial for **rising edge detection** of **trig_in**. It will store the value of **trig_in** from the *previous* clock cycle.
- **signal counter : integer range 0 to 2**DATA_BUS_WIDTH - 1;**
    - Declares an internal signal named **counter**.
    - It's declared as an INTEGER with a specified range, from 0 up to the maximum value that can be represented by DATA_BUS_WIDTH bits (e.g., if DATA_BUS_WIDTH is 16, the range is 0 to 65535). This counter tracks the duration of the output pulse.

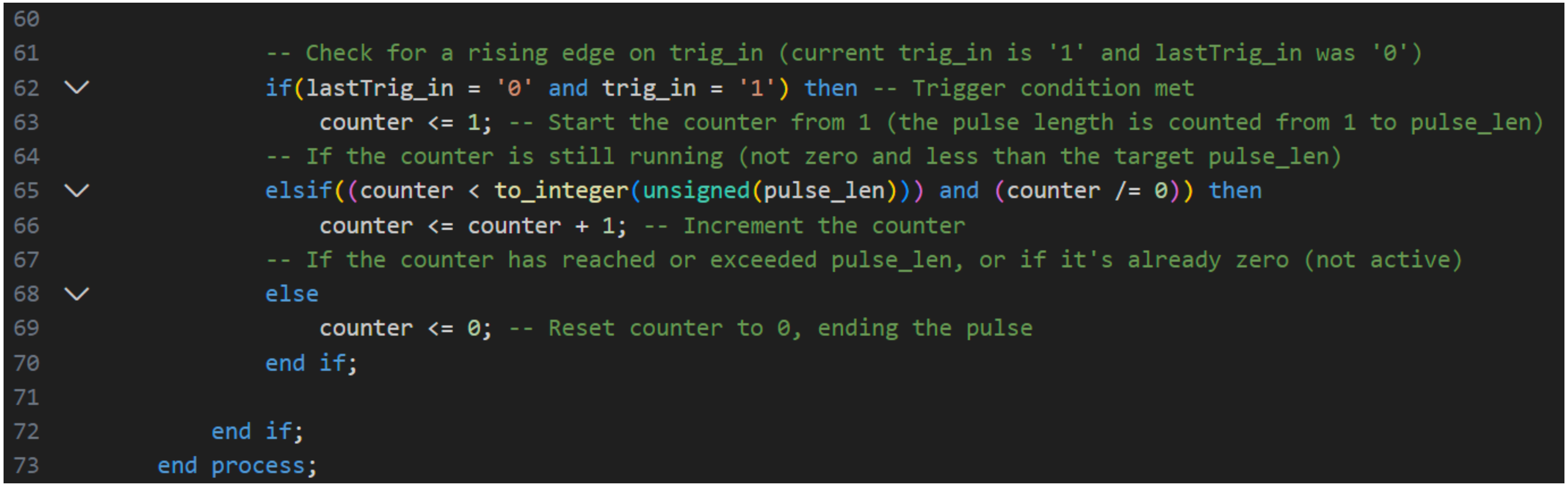
Synchronous one-shot-timer



```
        process(clk, aresetn)
        begin
            -- Asynchronous Reset
            if aresetn = '0' then
                lastTrig_in <= '0'; -- Reset lastTrig_in to '0'
                counter <= 0;       -- Reset counter to 0 (trig_out will be '0')
            -- Synchronous operations on rising edge of clock
            elsif rising_edge(clk) then
                -- Store current trig_in to lastTrig_in for the next clock cycle's edge detection
                lastTrig_in <= trig_in; -- This creates a 1-cycle delay for trig_in
```

- **process(clk, aresetn)**: This is a **sequential (clock-controlled) process**.
    - It's sensitive to **clk** and **aresetn**. This means the code inside the process will execute whenever **clk** or **aresetn** changes value.
- **if aresetn = '0' then** ...: This handles the **asynchronous reset**. If **aresetn** goes low, both **lastTrig_in** and **counter** are immediately forced to '0'. This initializes the timer to an inactive state.
- **elsif rising_edge(clk) then** ...: This block executes only on the **rising edge of the clock**, *after* checking the reset condition. All the logic inside this **elsif** is **synchronous**.
- **lastTrig_in <= trig_in;** On each rising clock edge, the current value of **trig_in** is stored into **lastTrig_in**. This is the standard way to create a one-clock-cycle delayed version of a signal for edge detection.

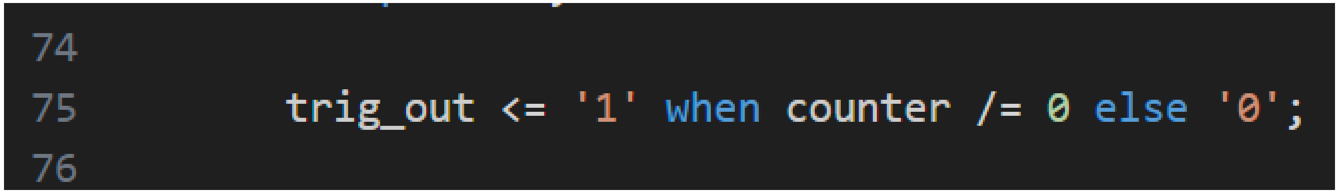
Synchronous one-shot-timer



```
                -- Check for a rising edge on trig_in (current trig_in is '1' and lastTrig_in was '0')
                if(lastTrig_in = '0' and trig_in = '1') then -- Trigger condition met
                    counter <= 1; -- Start the counter from 1 (the pulse length is counted from 1 to pulse_len)
                -- If the counter is still running (not zero and less than the target pulse_len)
                elsif((counter < to_integer(unsigned(pulse_len))) and (counter /= 0)) then
                    counter <= counter + 1; -- Increment the counter
                -- If the counter has reached or exceeded pulse_len, or if it's already zero (not active)
                else
                    counter <= 0; -- Reset counter to 0, ending the pulse
                end if;

            end if;
        end process;
```

- **if(lastTrig_in = '0' and trig_in = '1')** then -- Trigger condition:
    - This is the **rising edge detection** logic. It checks if **trig_in** was '0' in the *previous* clock cycle (**lastTrig_in**) and is '1' in the *current* clock cycle (**trig_in**).
    - If a rising edge is detected**, counter <= 1;** initializes the counter. This makes the **trig_out** go high starting from the next clock cycle (because trig_out is 1 when counter /= 0).
- **elsif((counter < to_integer(unsigned(pulse_len))) and (counter /= 0)) then**: The **counting** phase.
    - **to_integer(unsigned(pulse_len)):** Converts the STD_LOGIC_VECTOR **pulse_len** into an INTEGER so it can be compared with **counter**. unsigned() treats the vector as an unsigned number.
    - **counter < ...:** Checks if the counter has not yet reached the desired pulse length.
    - **counter /= 0**: Ensures that the counter is actually active (has been triggered).
- If both conditions are true**, counter <= counter + 1;** increments the counter, continuing the pulse.
- else counter <= 0;: This is the default case.

Synchronous one-shot-timer



```
        trig_out <= '1' when counter /= 0 else '0';
```

- This is a **concurrent signal assignment statement**. It is outside the process block, meaning it's continuously evaluated.
- **trig_out <= '1' when counter /= 0 else '0';**
  - The **trig_out** signal will be '1' whenever the counter is *not equal to* 0.
  - Otherwise (when counter is 0), **trig_out** will be '0'.
  - This effectively means that **trig_out** is high for the duration that counter is counting (from 1 up to **pulse_len**), and low when the timer is idle or reset.

Synchronous one-shot-timer

How it Works Together:

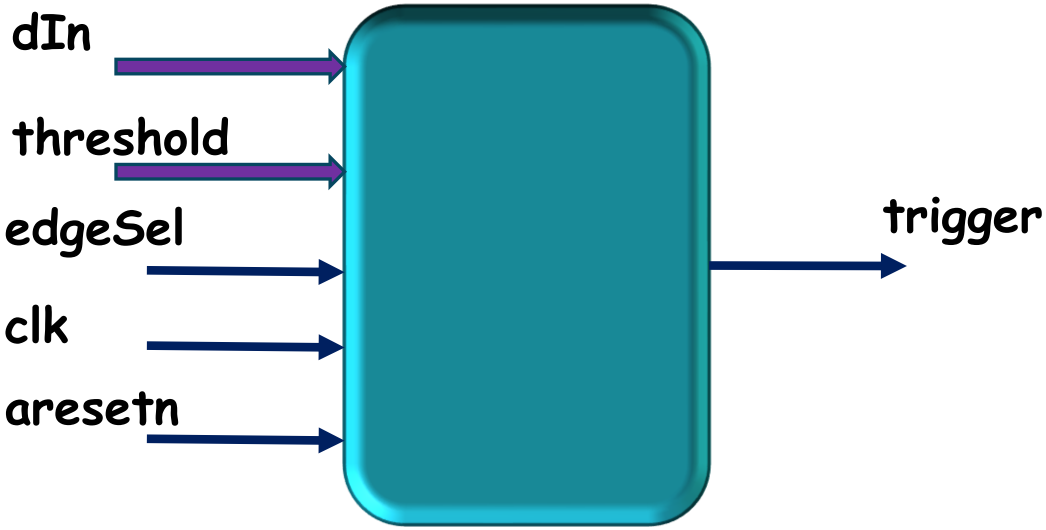
1. **Idle State:** counter is 0, trig_out is '0'.
2. **Trigger:** A rising edge on **trig_in** is detected.
3. **Start Pulse:** On the next rising **clk** edge after the **trigger**, **counter** is set to 1. **trig_out** immediately becomes '1'.
4. **Pulse Duration:** For subsequent clock cycles, if counter is less than **pulse_len** and not 0, counter increments. **trig_out** remains '1'.
5. **Pulse End:** When counter becomes equal to **pulse_len** (meaning pulse_len clock cycles have passed since the trigger), in the *next* clock cycle, the **elsif** condition (**counter < to_integer(unsigned(pulse_len)))** becomes false. The **else** branch executes, resetting **counter** to 0.
6. **Output Low:** As counter becomes 0, **trig_out** immediately reverts to '0', ending the pulse.
7. **Reset:** If **aresetn** goes low at any point, **counter** and **lastTrig_in** are reset to 0, forcing **trig_out** to '0' and stopping any ongoing pulse.

This design creates a reliable, retriggerable one-shot timer with a configurable pulse length, synchronized to the system clock.

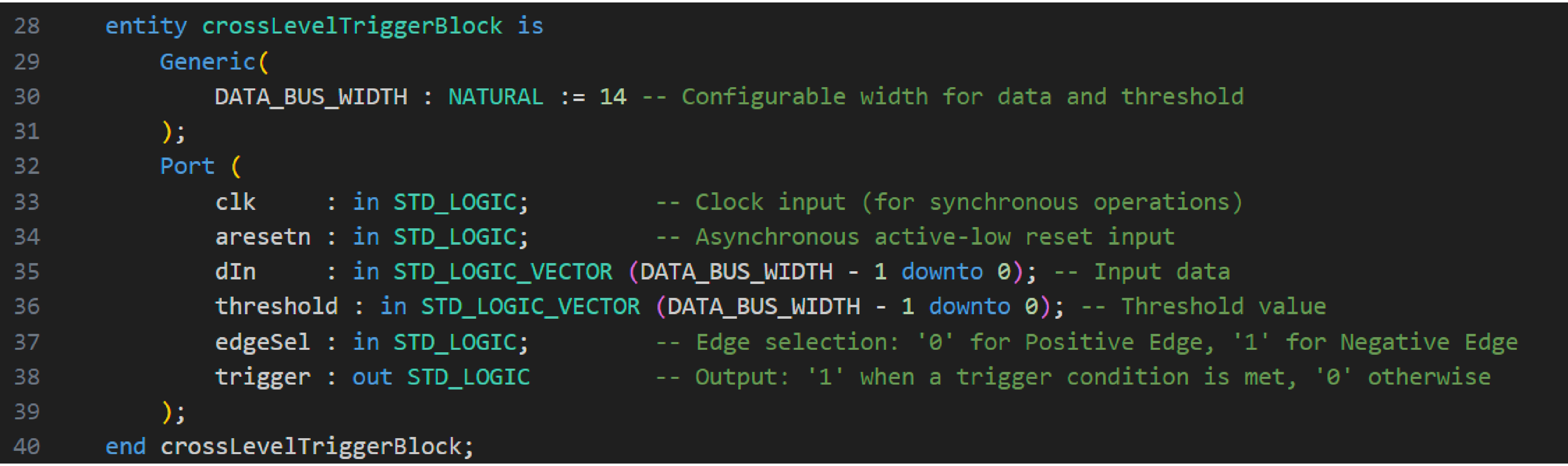
Synchronous level-crossing trigger

Its purpose is to detect when an input data value, **dIn**, crosses a specified **threshold** value, specifically looking for either a positive-going (rising) edge crossing or a negative-going (falling) edge crossing, depending on the **edgeSel** input.

The output, **trigger**, will be set to '1' during one clock cycle whenever a trigger event is detected



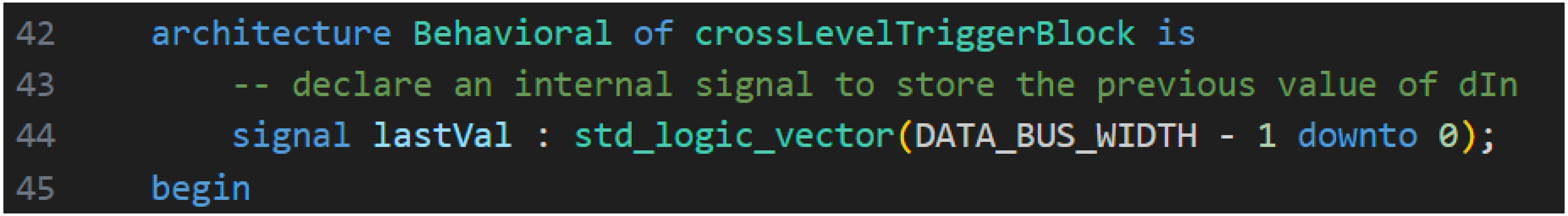
Synchronous level-crossing trigger



```vhdl
    entity crossLevelTriggerBlock is
        Generic(
            DATA_BUS_WIDTH : NATURAL := 14 -- Configurable width for data and threshold
        );
        Port (
            clk     : in STD_LOGIC;         -- Clock input (for synchronous operations)
            aresetn : in STD_LOGIC;         -- Asynchronous active-low reset input
            dIn     : in STD_LOGIC_VECTOR (DATA_BUS_WIDTH - 1 downto 0); -- Input data
            threshold : in STD_LOGIC_VECTOR (DATA_BUS_WIDTH - 1 downto 0); -- Threshold value
            edgeSel : in STD_LOGIC;         -- Edge selection: '0' for Positive Edge, '1' for Negative Edge
            trigger : out STD_LOGIC         -- Output: '1' when a trigger condition is met, '0' otherwise
        );
    end crossLevelTriggerBlock;
```

- **generic**(DATA_BUS_WIDTH : NATURAL := 14):
    - Generic is a way to pass constant values into an entity from its instantiation.
    - DATA_BUS_WIDTH is a generic parameter defining the width of the **dIn** and threshold buses. It's set to a default value of 14 bits. This makes the design *reusable for different data widths without modifying the core code*.
- Port (...): Defines the input and output signals of the block:
    - **clk**: Standard clock input. All internal state changes will occur on its edge.
    - **aresetn**: Asynchronous active-low reset. When aresetn is '0', the circuit is reset. *The n suffix typically indicates active-low.*
    - **dIn**: Input data bus, DATA_BUS_WIDTH bits wide.
    - **threshold**: Reference value for comparison, DATA_BUS_WIDTH bits wide.
    - **edgeSel**: A single bit to select the type of trigger: '0' for positive-going edge (crossing threshold from below), '1' for negative-going edge (crossing threshold from above).
    - **trigger**: A single-bit output that goes '1' when a trigger condition is detected.

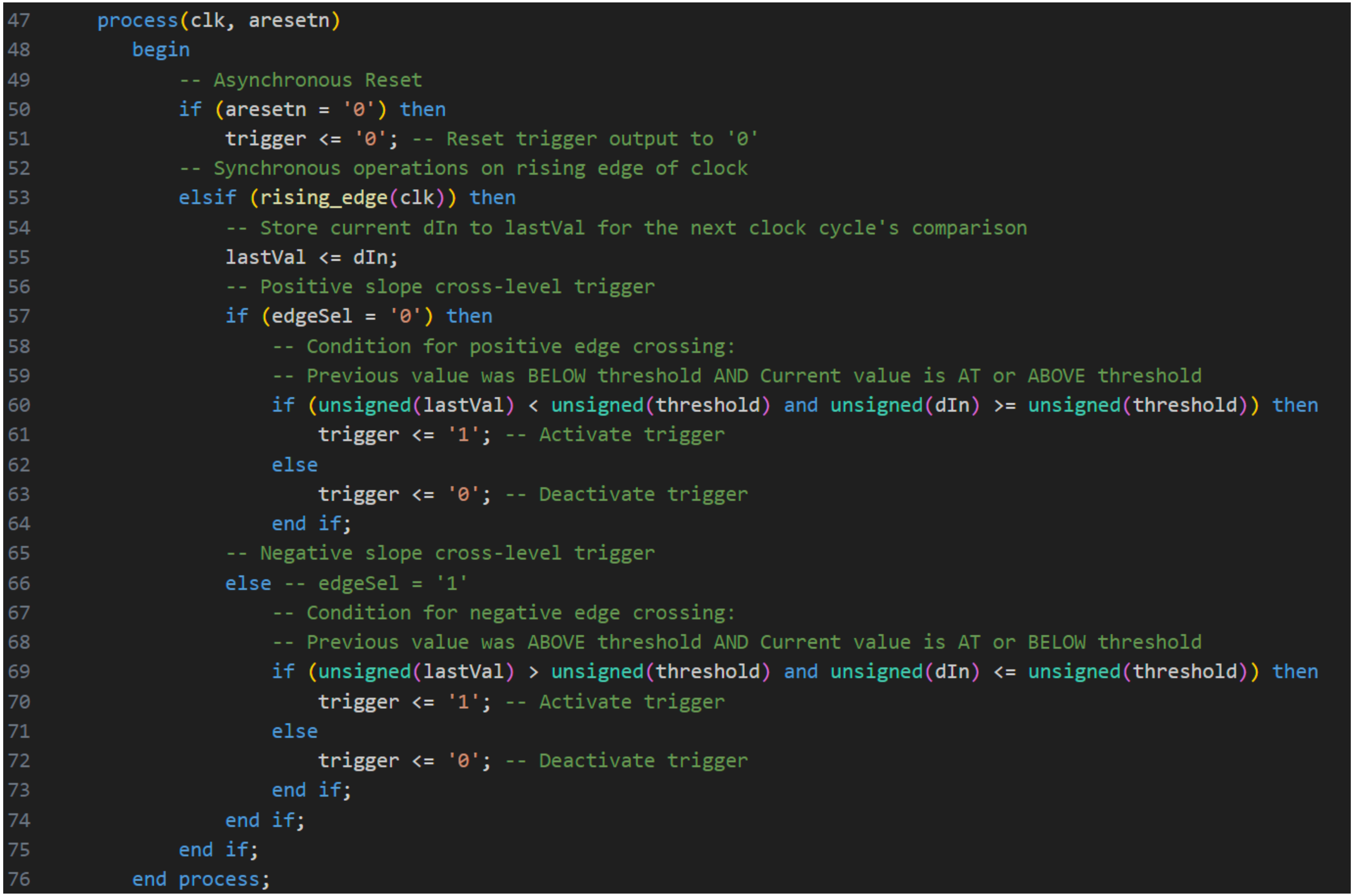
Synchronous level-crossing trigger



```
architecture Behavioral of crossLevelTriggerBlock is
    -- declare an internal signal to store the previous value of dIn
    signal lastVal : std_logic_vector(DATA_BUS_WIDTH - 1 downto 0);
begin
```

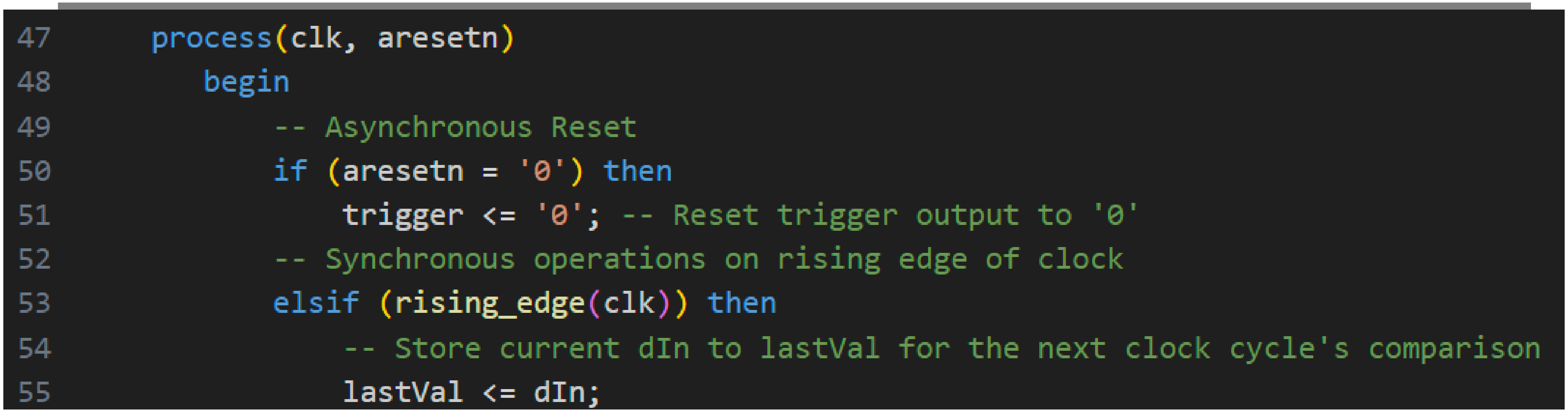
- **`architecture Behavioral of crossLevelTriggerBlock`** is: Defines the behavior or implementation of the **`crossLevelTriggerBlock`** entity.
- signal **`lastVal`** `: std_logic_vector(DATA_BUS_WIDTH - 1 downto 0);`
  - Declares an internal signal named last **`lastVal`**.
  - **`lastVal`** is a STD_LOGIC_VECTOR of the same width as **dIn** and **threshold**.
  - This signal is crucial for detecting *edge* crossings: it stores the value of **dIn** from the *previous* clock cycle.

Synchronous level-crossing trigger



```vhdl
  process(clk, aresetn)
    begin
      -- Asynchronous Reset
      if (aresetn = '0') then
        trigger <= '0'; -- Reset trigger output to '0'
      -- Synchronous operations on rising edge of clock
      elsif (rising_edge(clk)) then
        -- Store current dIn to lastVal for the next clock cycle's comparison
        lastVal <= dIn;
        -- Positive slope cross-level trigger
        if (edgeSel = '0') then
          -- Condition for positive edge crossing:
          -- Previous value was BELOW threshold AND Current value is AT or ABOVE threshold
          if (unsigned(lastVal) < unsigned(threshold) and unsigned(dIn) >= unsigned(threshold)) then
            trigger <= '1'; -- Activate trigger
          else
            trigger <= '0'; -- Deactivate trigger
          end if;
        -- Negative slope cross-level trigger
        else -- edgeSel = '1'
          -- Condition for negative edge crossing:
          -- Previous value was ABOVE threshold AND Current value is AT or BELOW threshold
          if (unsigned(lastVal) > unsigned(threshold) and unsigned(dIn) <= unsigned(threshold)) then
            trigger <= '1'; -- Activate trigger
          else
            trigger <= '0'; -- Deactivate trigger
          end if;
        end if;
      end if;
    end process;
```

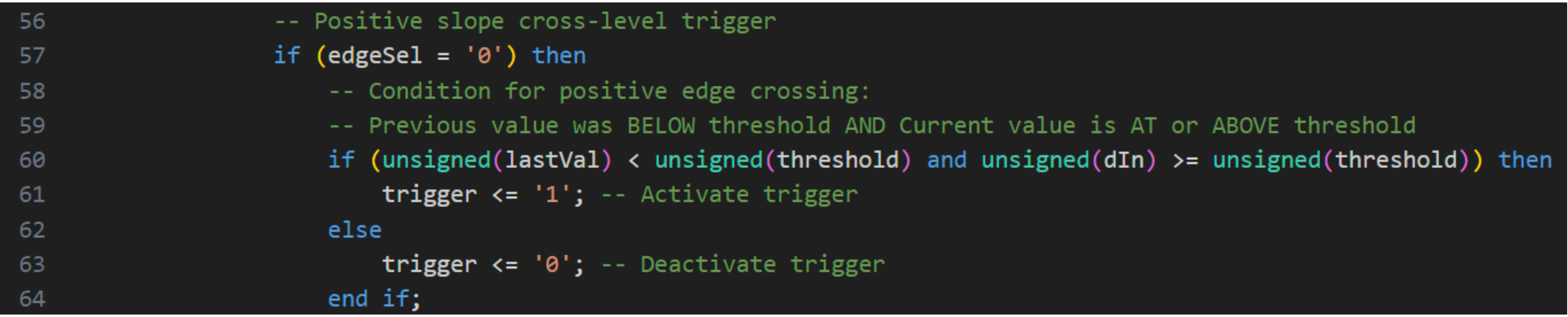
Synchronous level-crossing trigger



```
    process(clk, aresetn)
       begin
           -- Asynchronous Reset
           if (aresetn = '0') then
               trigger <= '0'; -- Reset trigger output to '0'
           -- Synchronous operations on rising edge of clock
           elsif (rising_edge(clk)) then
               -- Store current dIn to lastVal for the next clock cycle's comparison
               lastVal <= dIn;
```

- **process(clk, aresetn):** This is a **sequential (clock-controlled) process**.
    - It's sensitive to **clk** and **aresetn**. This means the code inside the process will execute whenever **clk** or **aresetn** changes value (aka, an event on one of those signals).
- **if (aresetn = '0')** then trigger <= '0’; This handles the **asynchronous reset**. If **aresetn** goes low,the output **trigger** is immediately forced to '0', *regardless of the clock*. This is common for initial circuit states.
- **elsif (rising_edge(clk)) then**: This block executes only on the **rising edge of the clock**, *after* checking the reset condition. All the logic inside this **elsif** is **synchronous**.
- **lastVal <= dIn;**
    - This is a synchronous assignment. On each rising clock edge, the current value of **dIn** is stored into **lastVal**.
    - This means **lastVal** will always hold the value of **dIn** from the *previous* clock cycle, allowing for edge detection.

Synchronous level-crossing trigger

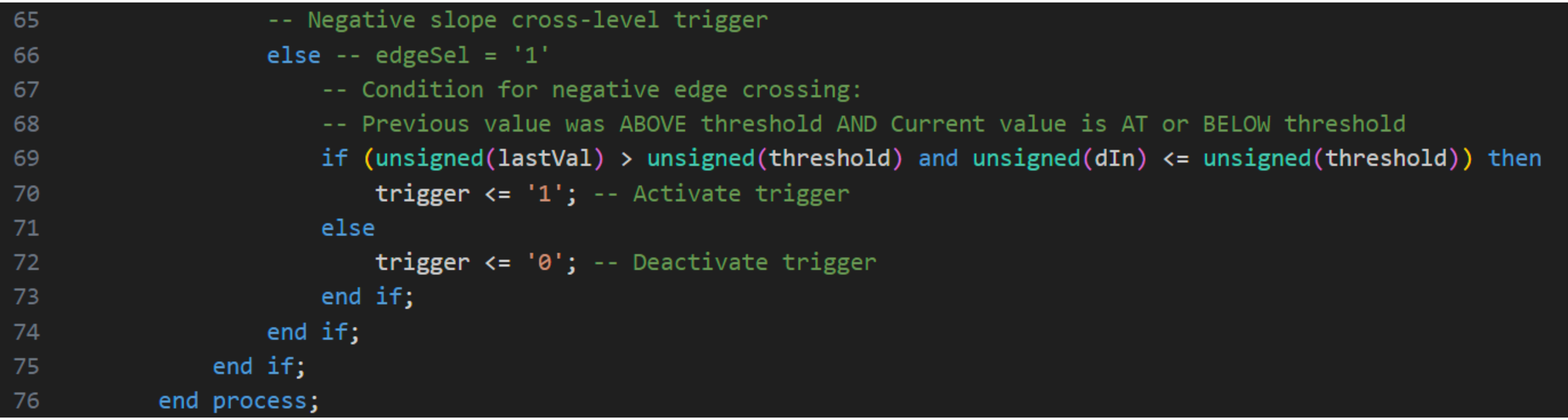


```vhdl
-- Positive slope cross-level trigger
if (edgeSel = '0') then
    -- Condition for positive edge crossing:
    -- Previous value was BELOW threshold AND Current value is AT or ABOVE threshold
    if (unsigned(lastVal) < unsigned(threshold) and unsigned(dIn) >= unsigned(threshold)) then
        trigger <= '1'; -- Activate trigger
    else
        trigger <= '0'; -- Deactivate trigger
    end if;
```

if (edgeSel = '0') then -- Positive slope cross-level trigger:

- If **edgeSel** is '0', the block is configured to detect a positive-going (rising) edge crossing.
- **if (unsigned(lastVal) < unsigned(threshold) and unsigned(dIn) >= unsigned(threshold)) then**: This is the core detection logic for a positive edge.
    - **unsigned(): cast** that converts STD_LOGIC_VECTOR signals to UNSIGNED type for ***numerical comparison.***
    - The condition checks if the *previous* data value (**lastVal**) was strictly **less than** the **threshold**, **AND** the ***current* data value (dIn)** is **greater than or equal to** the **threshold**. This accurately defines a ***positive cross-level event***.
    - If this condition is met, **trigger** is set to '**1**'.
    - **else trigger <= '0';** Otherwise, **trigger** is '0'. *This ensures **trigger** is only '1' for one clock cycle when the condition is met.*

Synchronous level-crossing trigger



```vhdl
                -- Negative slope cross-level trigger
                else -- edgeSel = '1'
                    -- Condition for negative edge crossing:
                    -- Previous value was ABOVE threshold AND Current value is AT or BELOW threshold
                    if (unsigned(lastVal) > unsigned(threshold) and unsigned(dIn) <= unsigned(threshold)) then
                        trigger <= '1'; -- Activate trigger
                    else
                        trigger <= '0'; -- Deactivate trigger
                    end if;
                end if;
            end if;
        end process;
```

else -- Negative slope cross-level trigger (**edgeSel = '1'**):

- If **edgeSel** is '1', the block is configured to detect a negative-going (falling) edge crossing.
- **if (unsigned(lastVal) > unsigned(threshold) and unsigned(dIn) <= unsigned(threshold))** then: This is the core detection logic for a **negative edge**.
    - The condition checks if the *previous* data value (**lastVal**) was strictly **greater than** the **threshold**, AND the ***current*** data value (**dIn**) is **less than or equal to** the **threshold**. This defines a negative cross-level event.
    - If this condition is met, **trigger** is set to '**1**'.
    - else trigger <= '0';: Otherwise, **trigger** is '**0**'.

Synchronous level-crossing trigger

In Summary:

This **crossLevelTriggerBlock** VHDL design acts as a programmable level-crossing detector:

1. It stores the previous sample of the input data (**dIn**) in **lastVal** synchronously on each clock edge.
2. On *each clock edge*, it compares the current **dIn** and the previous **lastVal** against a **threshold**.
3. Based on the **edgeSel** input, it determines if a positive-going or negative-going level crossing has occurred.
4. If a crossing is detected, the **trigger** output goes *high for one clock cycle*.
5. It also includes an asynchronous active-low reset to initialize the trigger output to '0'.

This type of module is common in digital signal processing or control systems where events need to be detected based on signal levels crossing certain thresholds