Joint ICTP-IAEA School on Systems-on-Chip based on FPGA for Scientific Instrumentation and Reconfigurable Computing

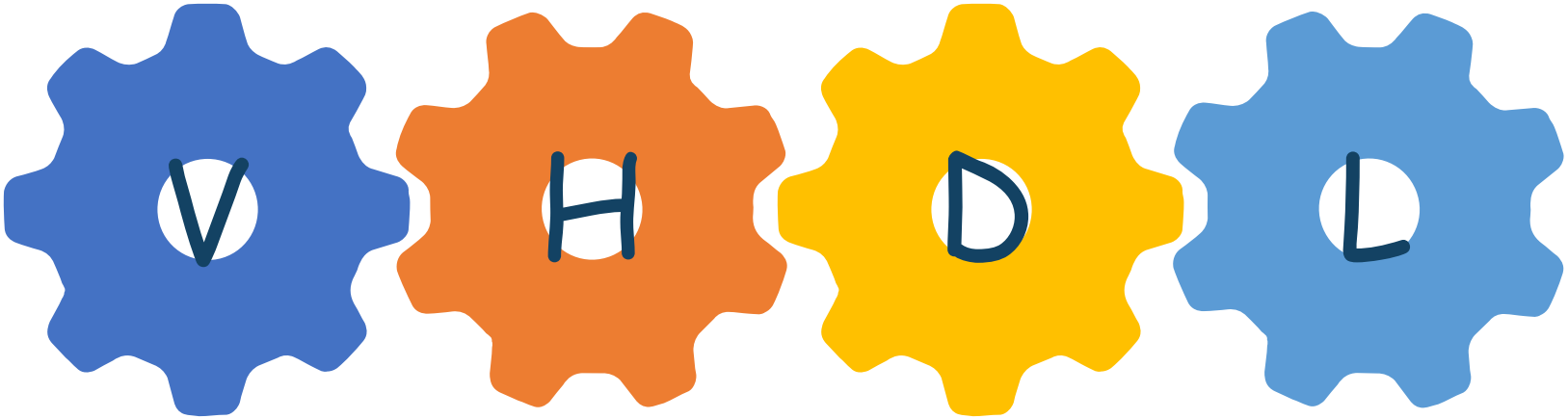
VHDL For Synthesis

MSc. Cristian Sisterna
Universidad Nacional San Juan
Argentina

Intoduction

Hardware

Language



Very High Speed IC

Description

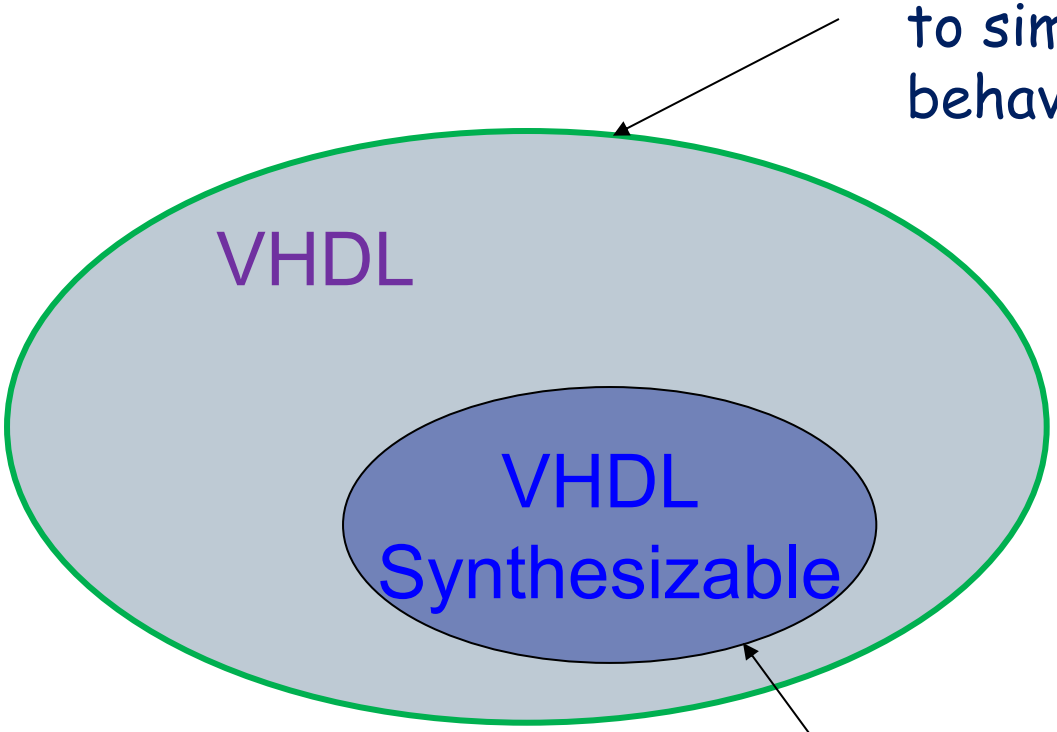
Hardware Description Language

- 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

HDL Synthesis Sub-Set



Used to write code to simulate the behavior of a design

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

HDL Synthesis Sub-Set

- ✓ VHDL is used to **DESCRIBE** the **behavior** and/or structure of a Digital System
- ✓ Be careful ! -> you are ***describing Hardware***

Concurrent Code -> Executed in Paralell

- ✓ With HDL it is possible to describe from a simple combinational circuit to a whole i7 processor

VHDL Describing Digital System

- The operations in real systems are executed *concurrently*.
- The VHDL language describes real systems as a set of components (statements) 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

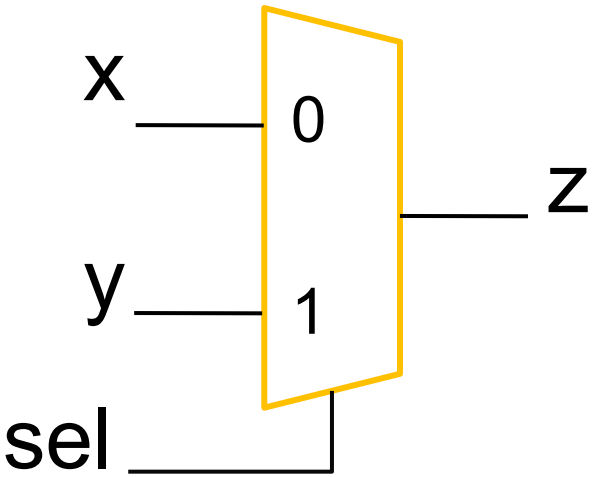
Synthesis versus Simulation

Extremely important to understand that VHLD is both, a *Synthesis* language and a *Simulation* language.

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

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

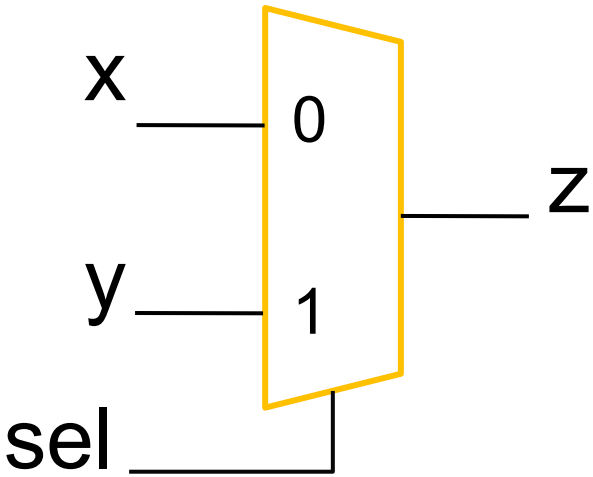
VHDL 'Description' Examples




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

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

VHDL – General Component Structure



mux2x1.vhd

entity

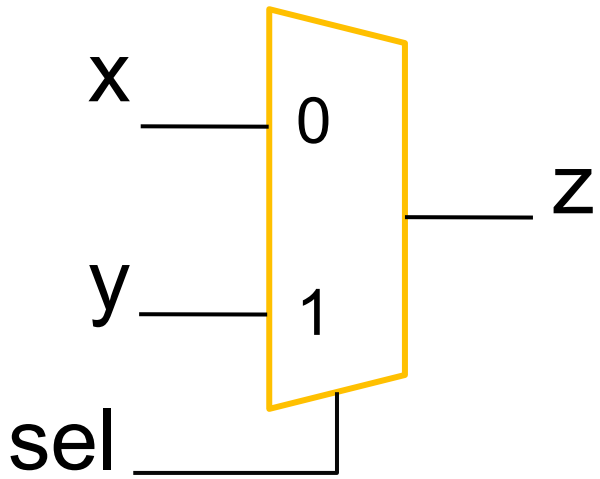
I/O

architecture

Functionality

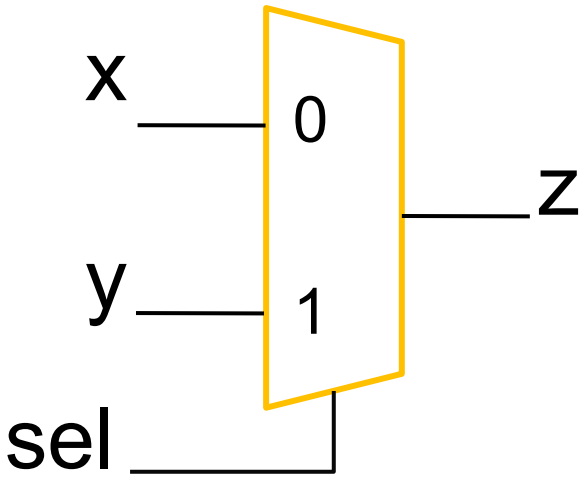
VHDL - General Component Structure

mux2x1.vhd




```
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

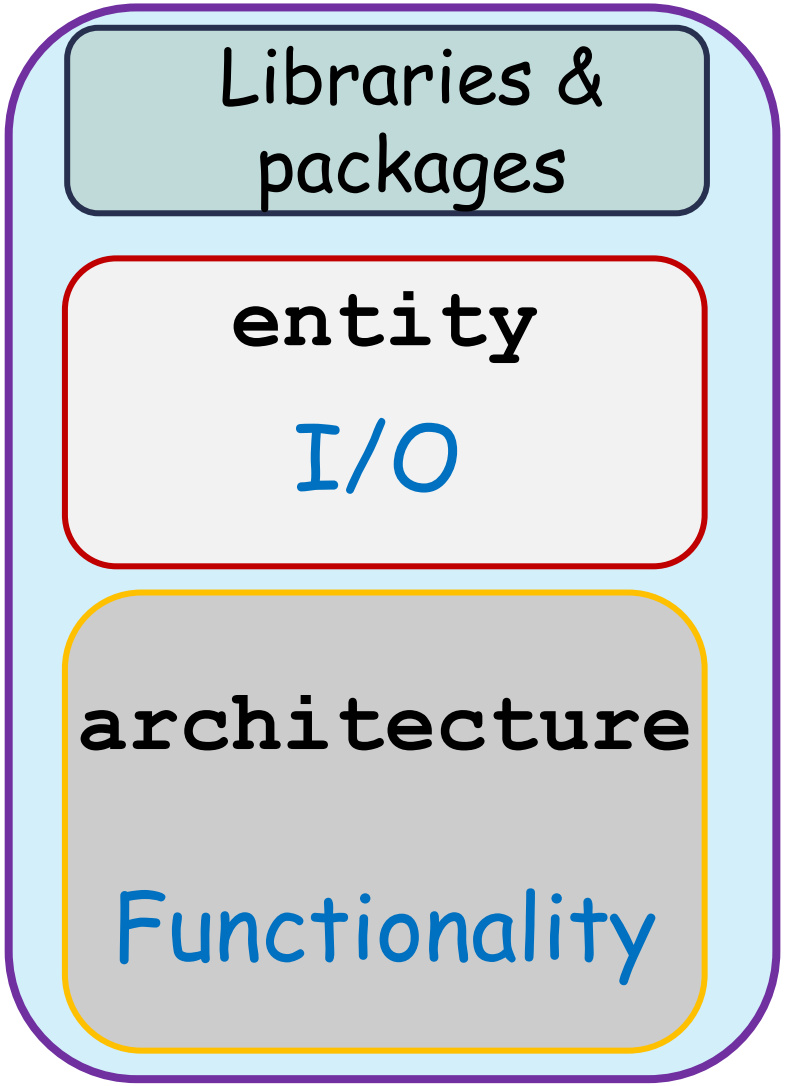


mux2x1.vhd

```
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;
```

VHDL - General Component Structure

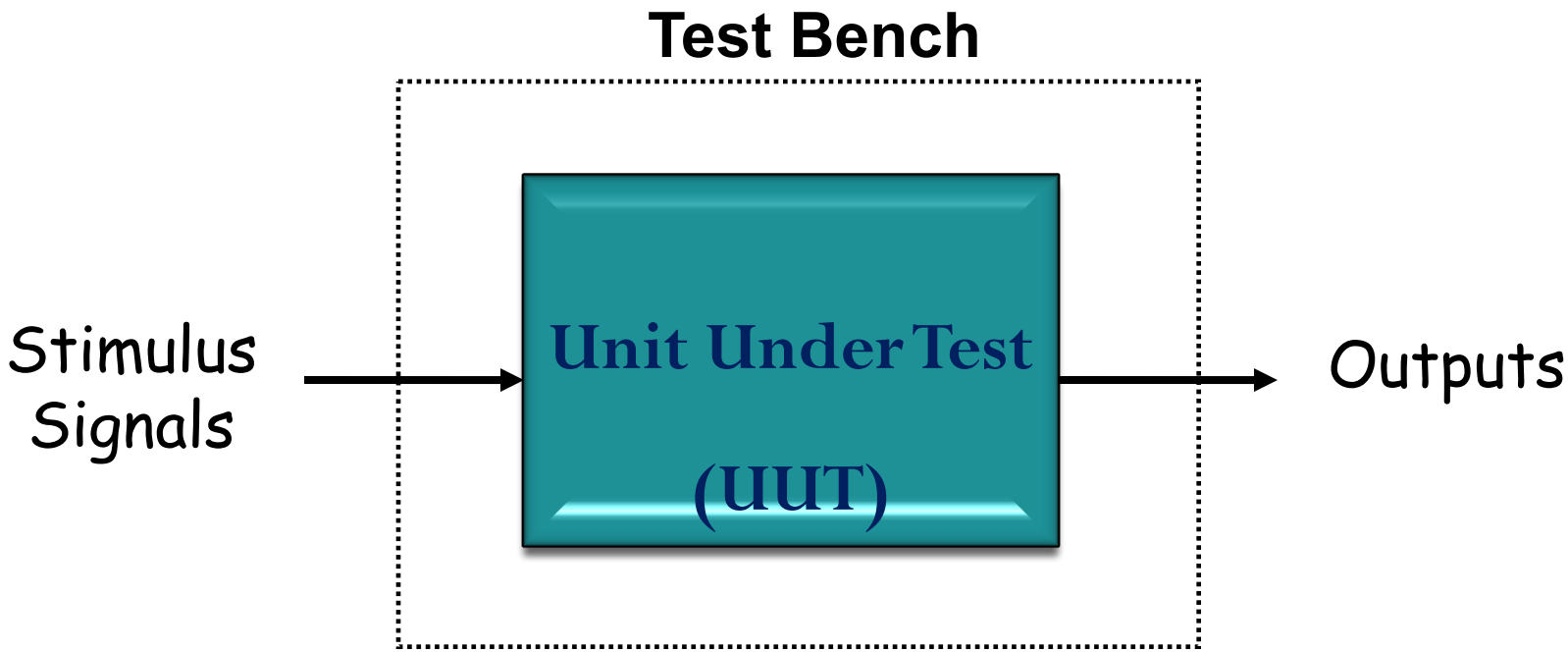


Libraries and packages provides the incorporation of external functions, data types and components to the component to be described

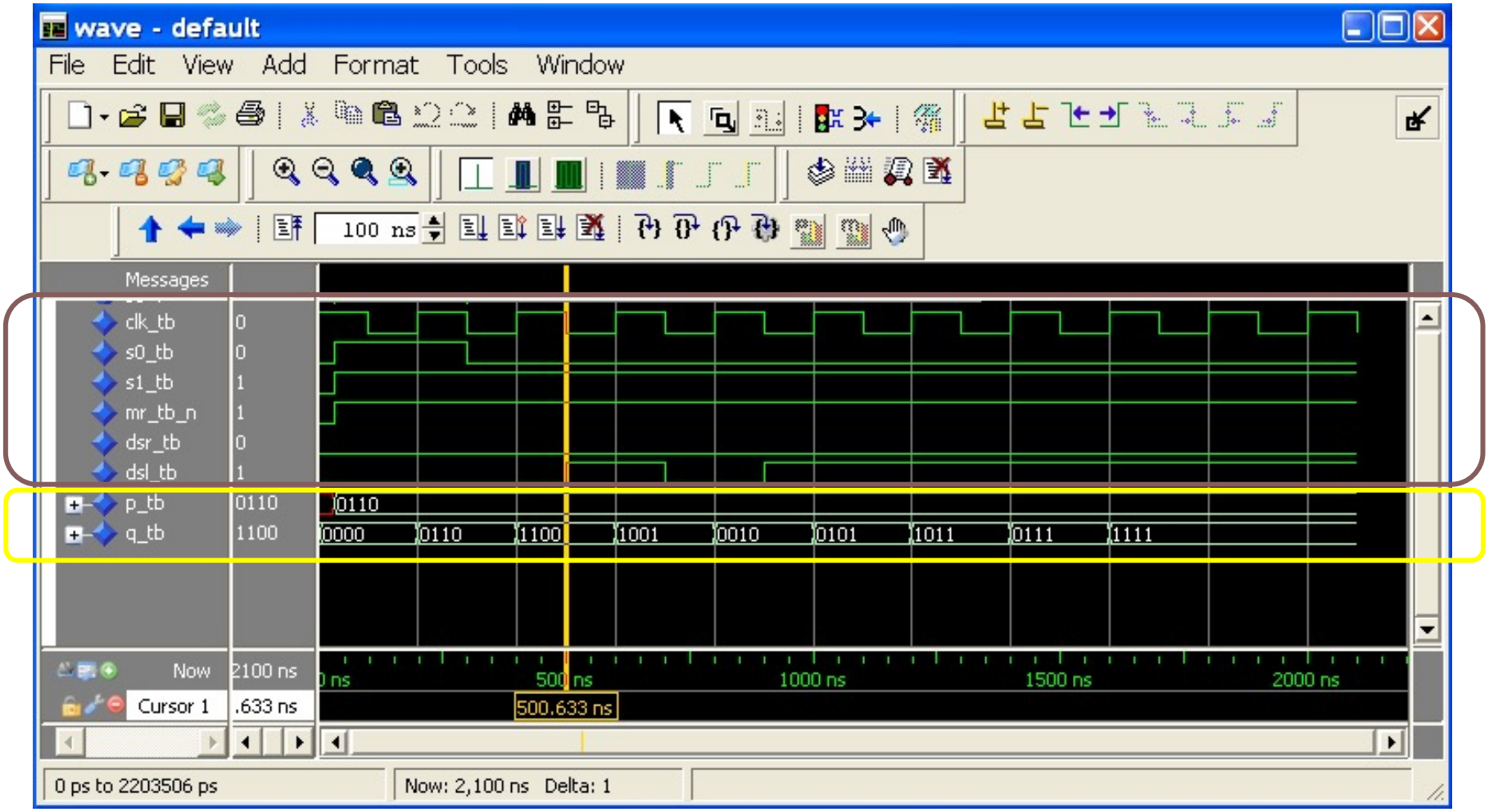
Defines the I/O ports as well as the name of the component. Some times a constant(s) is defined (generic) to write parameterized VHDL code

It's where the hardware **behavior** and/or **structure** is described. It can have from 1 to thousands lines of code… ALL CONCURRENTs !

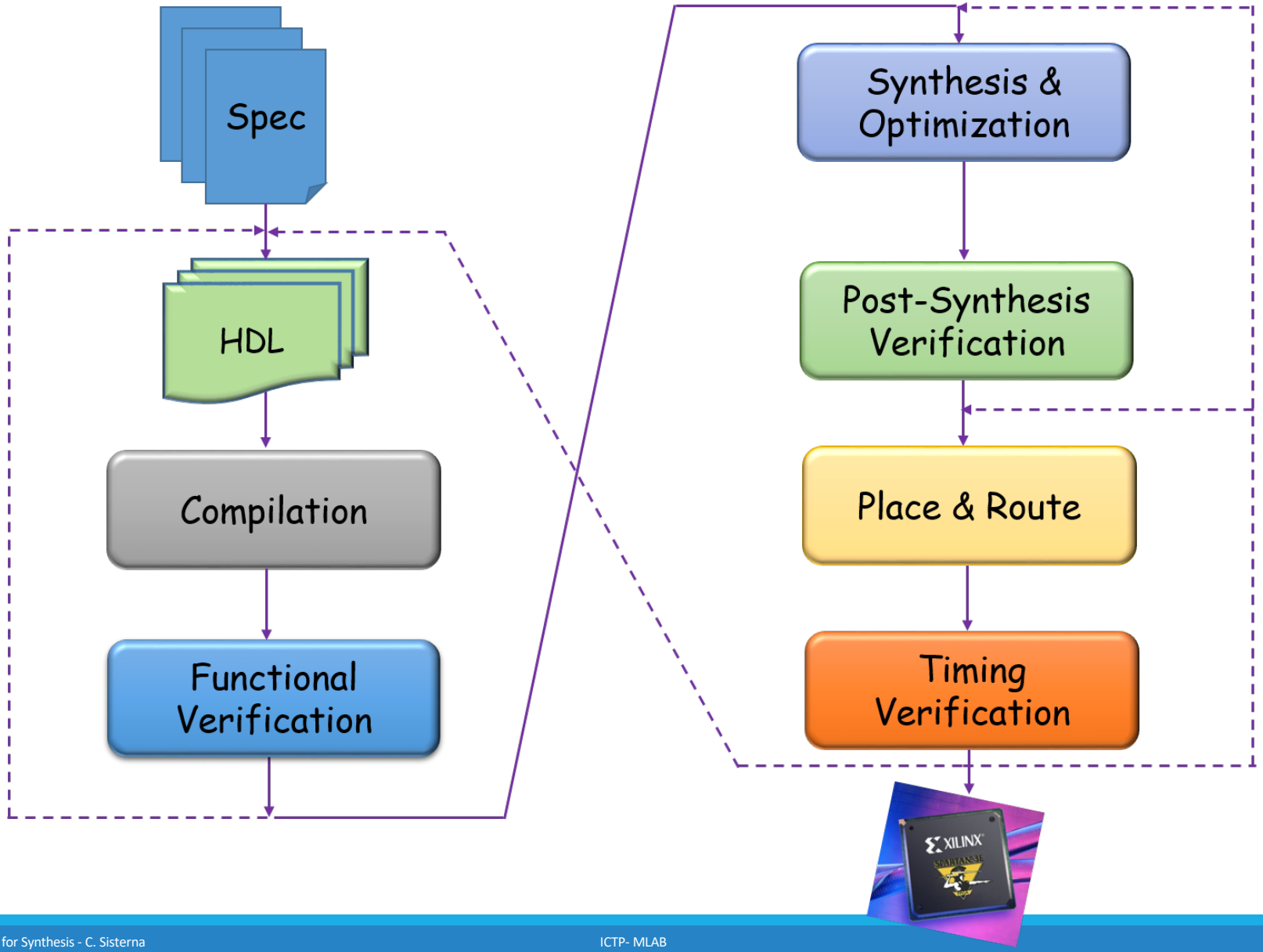
VHDL Code – Is it really Works?



VHDL - Simulation / Verification



VHDL - FPGA Design Flow



VHDL – FPGA: Synthesis + P&R

```
with tmp select
 j <= w when "1000",
      x when "0100",
      y when "0010",
      z when "0001",
     '0'when others;
```

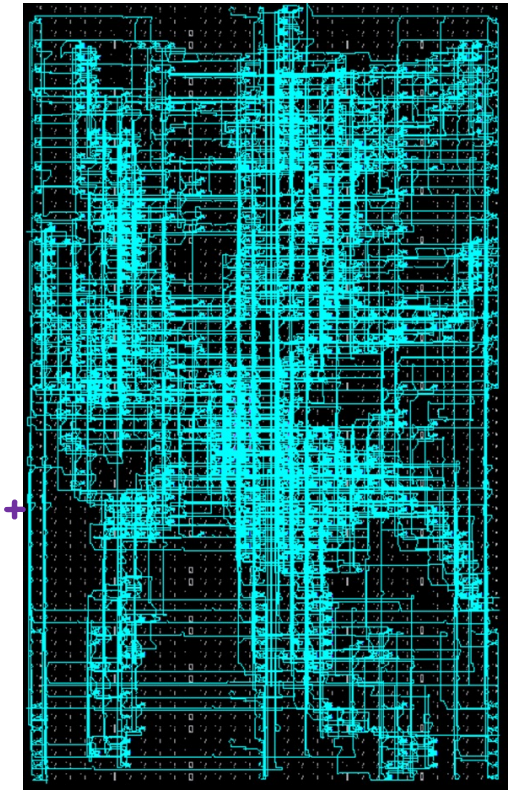
VHDL Code

Design Constraints

NET CLOCK **PERIOD** = 50 ns;
NET LOAD **LOC** = P14

Vivado/Quartus
Libero
EDA Tool

Synthesis + P&R



Digital System implemented in the FPGA

Design Attributes

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

Cyclone
Spartan

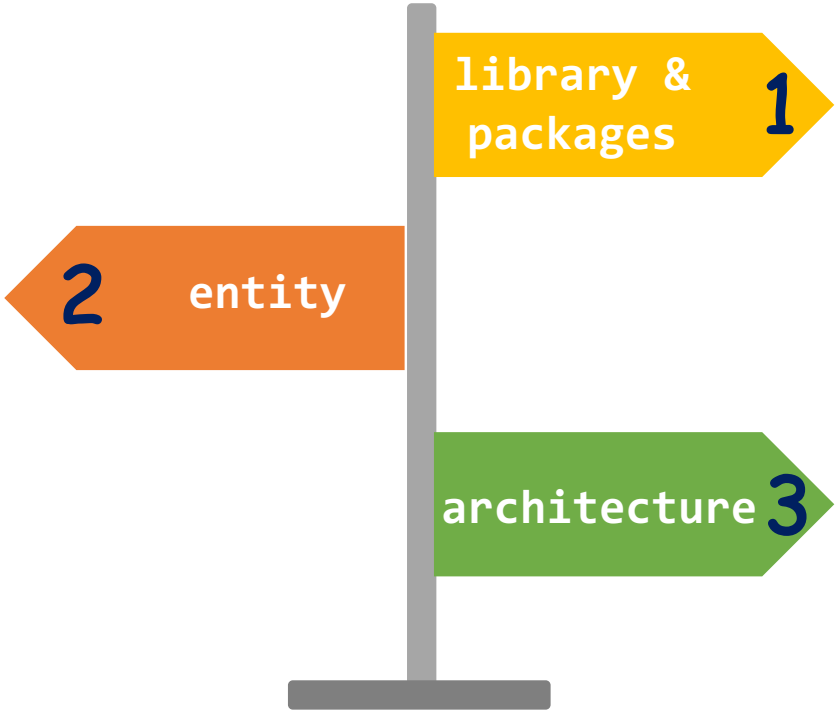
FPGA Library of Components

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;
```

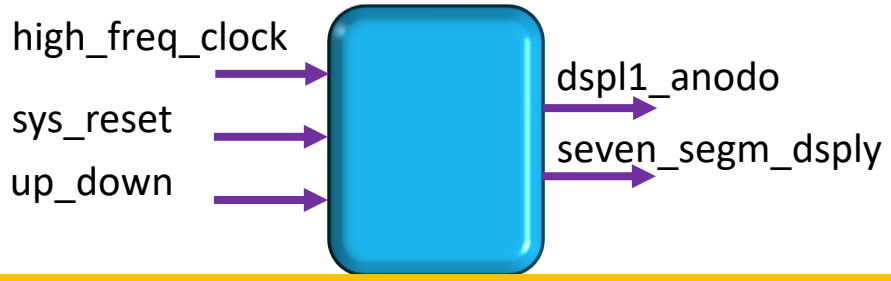
Must be present to add arithmetic functions for ***signed*** and ***unsigned*** types.

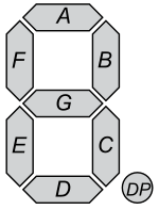
Note: do not do arithmetic operations with std_logic/std_logic_vector

```
-- 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





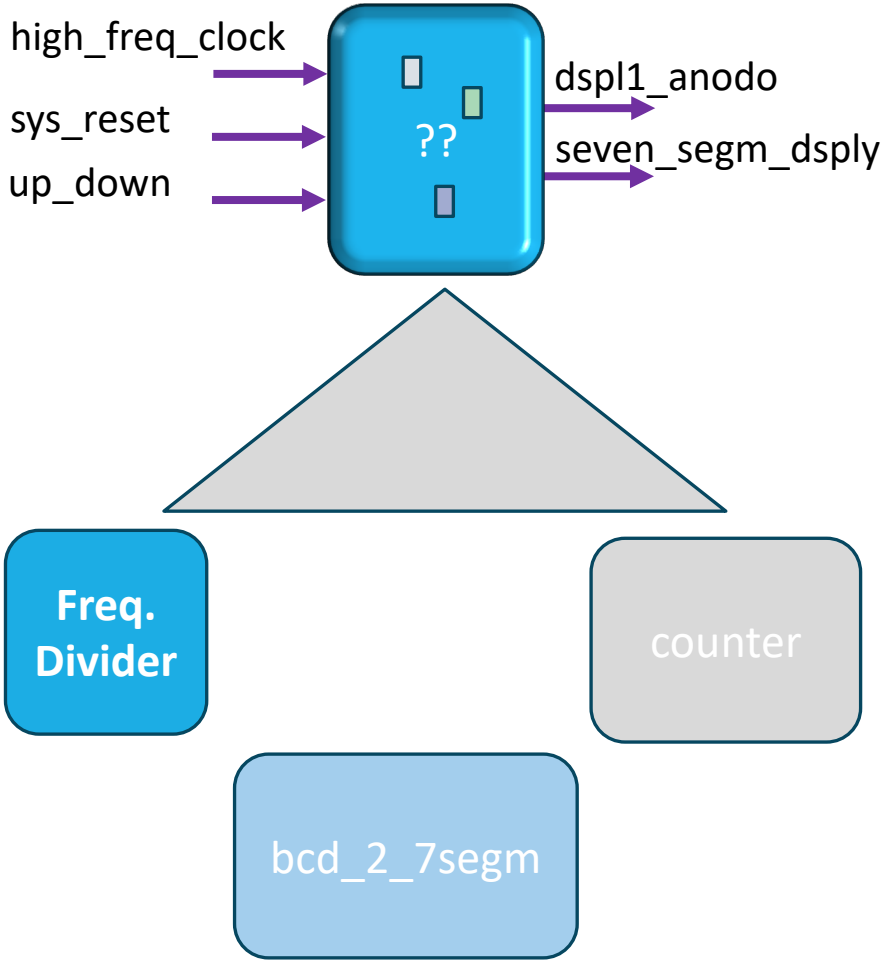

```
-- ------------------------------------------------------ --
-- Entity Declaration
-- ------------------------------------------------------ --
entity top is
  port (
    -- system signals
    high_freq_clock   : in  std_logic;
    sys_reset         : in  std_logic;
    -- control signals
    up_down           : 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

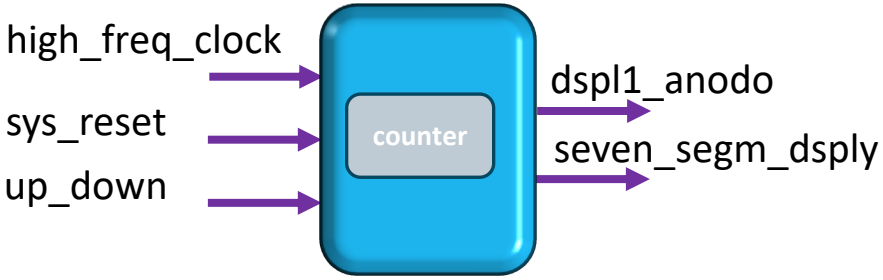
use only std_logic/std_logic_vector types

use only inout mode in the higher lever top-module

Architecture (top)



Counter entity/arch.




```vhdl
-------------------------------------------------------- --
-- 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;
```

```vhdl
-------------------------------------------------------
-- 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;
```

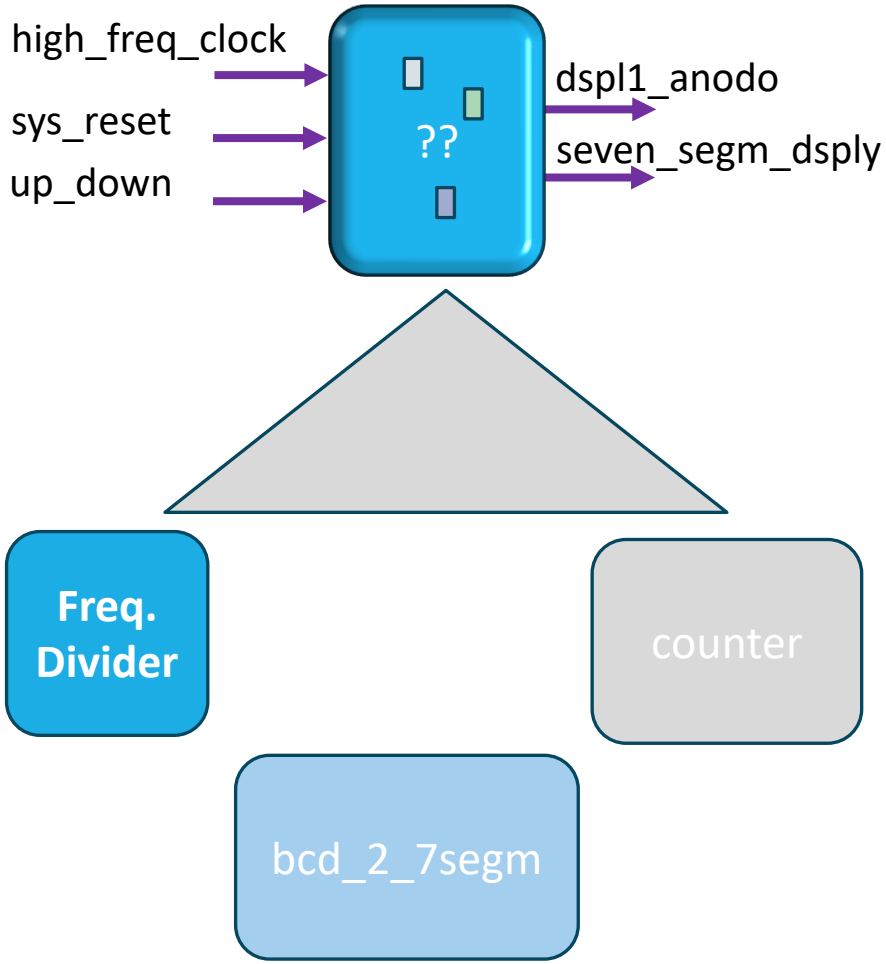
concurrent

sequential — concurrent

concurrent

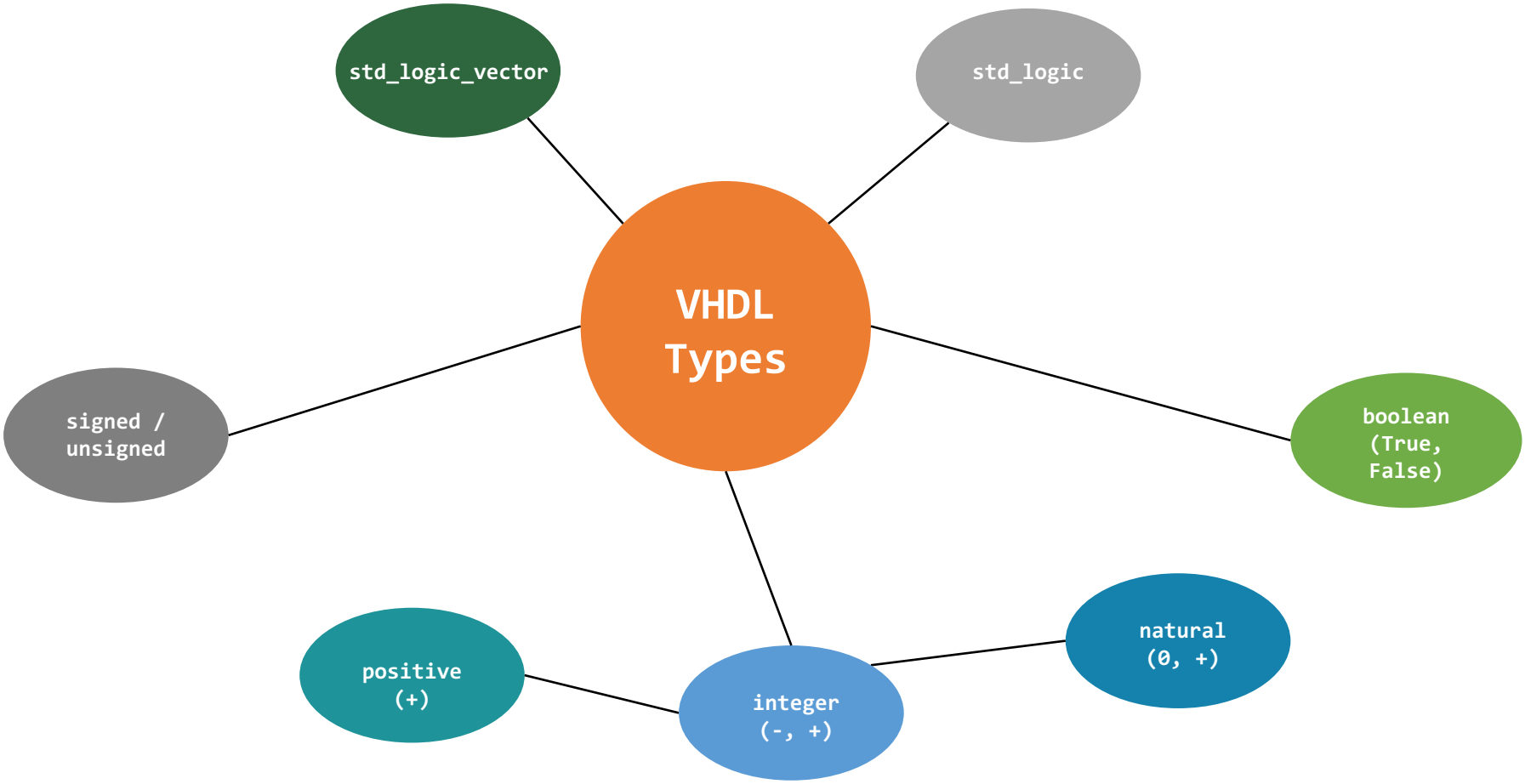
sequential — concurrent

Architecture (top)




```vhdl
-------------------------------------------- --
-- 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 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));
```

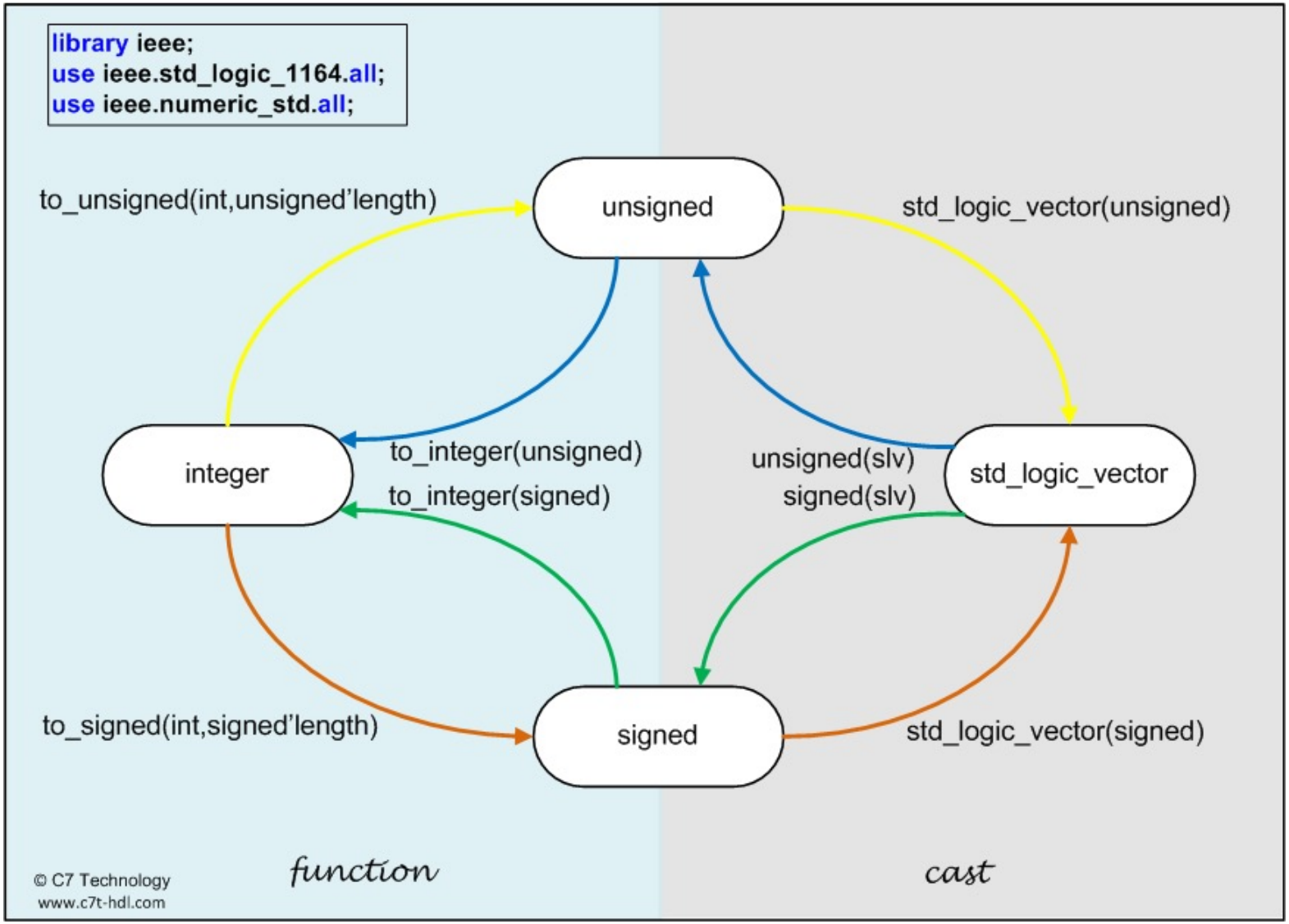
Function converts integer to unsigned

Cast to slv

unsigned

slv

Type Conversion – Cast / Function

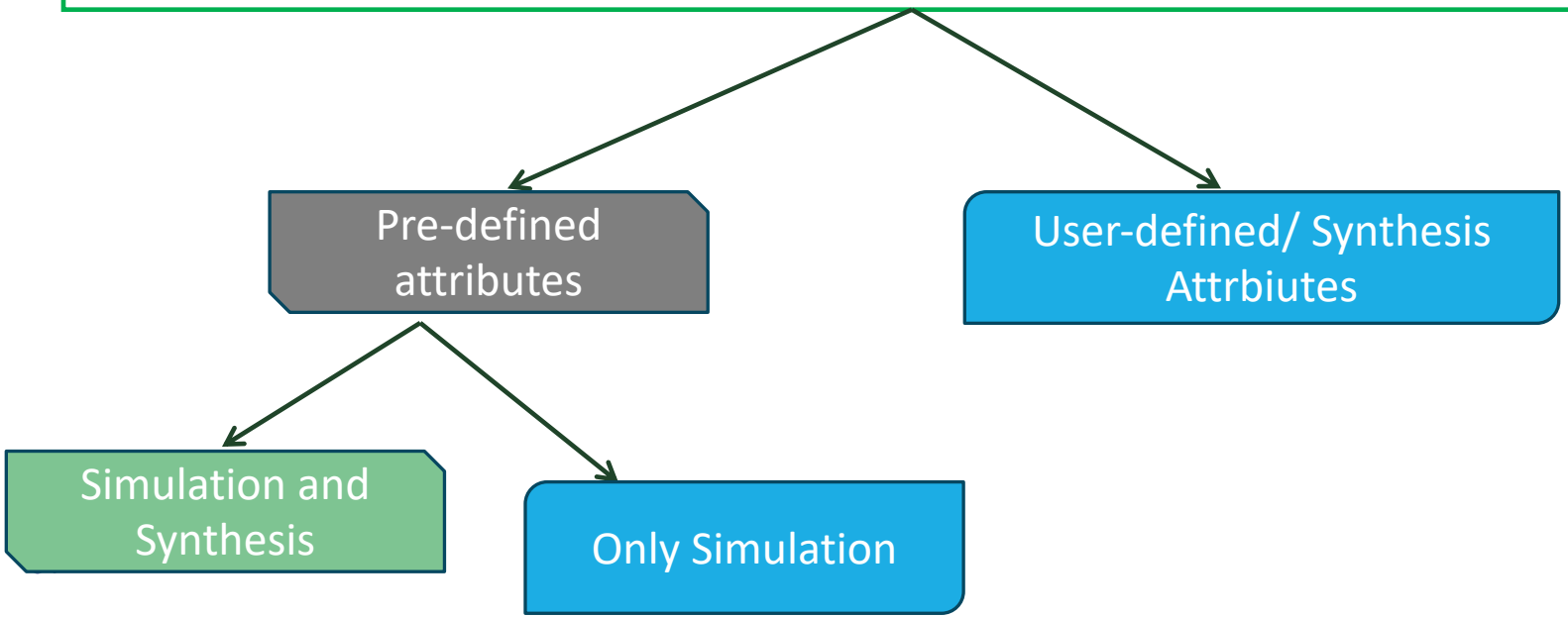


VHDL Operators

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 way of **extracting** information from a type, from the values of a type or it might define new implicit signals from explicitly declared signals
- It's also a way to allow to **assign additional** information to objects in your design description (such as data related to synthesis)



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	-

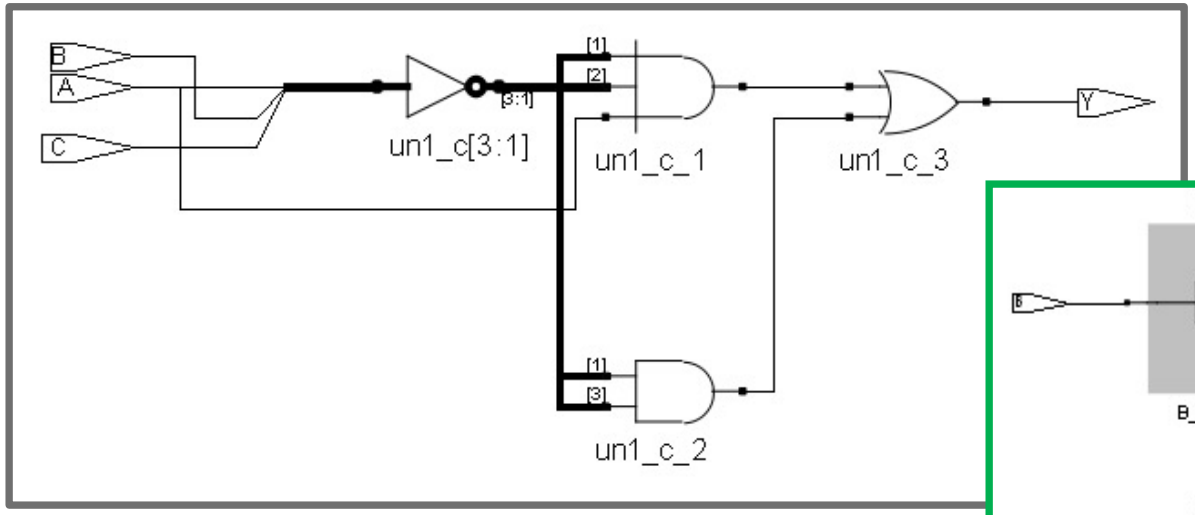
```
library ieee;
use ieee.std_logic_1164.all;

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"

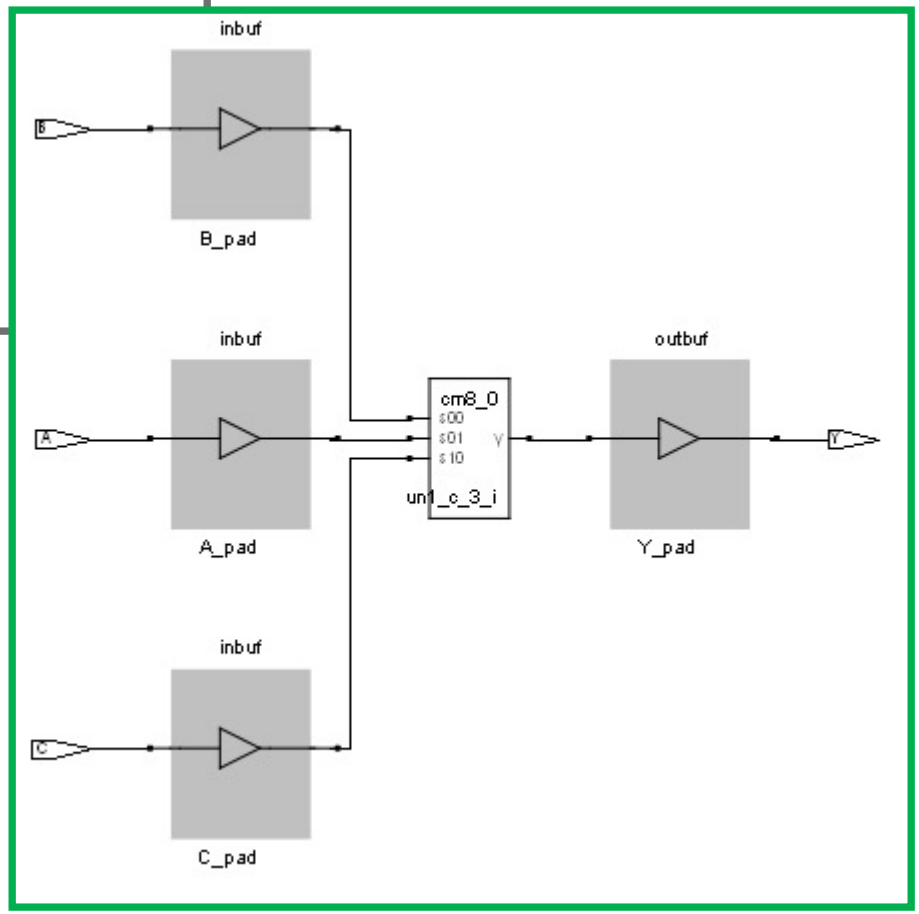
'-' means don't care

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

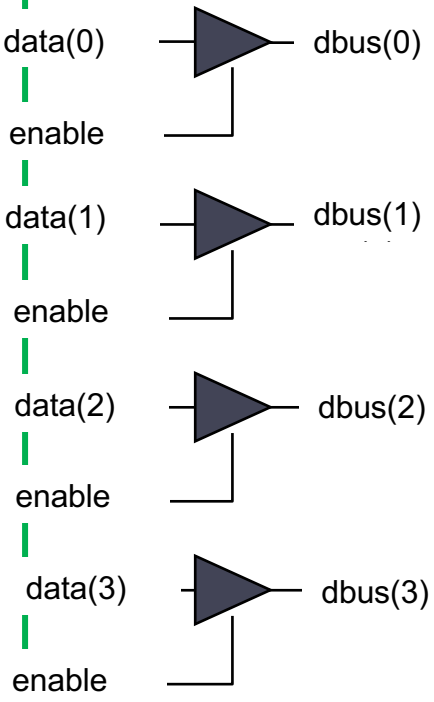
```
dbus <= data    when enable = '1' else 'Z';
```

```
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;
```



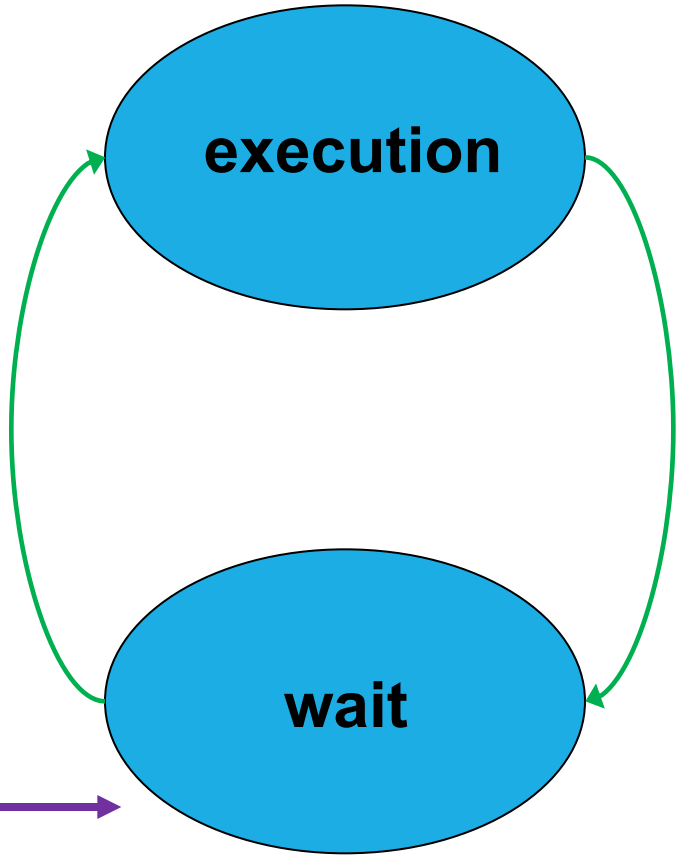
process Statement

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*.
- From the traditional programming view, it is an *infinite loop*
- Multiple processes can be executed in parallel

Process Statement

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



Until a condition is satisfied

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

Process Statement

- Processes are composed of sequential statements, but process declarations are concurrent statements.
- The main features of a process are the following:
  - It is executed in parallel with other processes;
  - It cannot contain concurrent statements;
  - It defines a region of the architecture where statements are executed sequentially
  - It must contain an explicit sensitivity list or a wait statement
  - It allows functional descriptions, similar to the programming languages;

Process Statement

```
[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
  return_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

Combinational Process

- In a combinational process all the input signals must be contained in the sensitivity list
- If a signal is omitted from the sensitivity list, the VHDL simulation and the synthesized hardware will behave differently
- All the output signals from the process must be assigned a value each time the process is executed. If this condition is not satisfied, the signal will retain its value (latch !)

Combinational Process

```
a_process: process (a_in, b_in)
begin
  c_out <= not(a_in and b_in);
  d_out <= not b_in;
end process a_process;
```

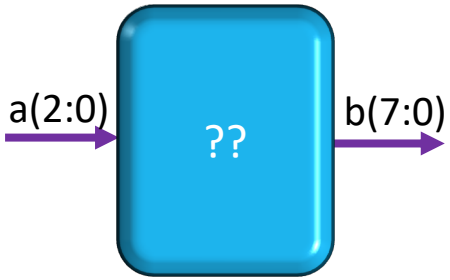
```
. . . .
architecture rtl of com_ex is
begin
 ex_c: process (a,b)
begin
   z <= a and b;
end process ex_c;
end rtl;
```

if Statement

- Syntax

```
if <boolean_expression> then
    <sequential_statement(s)>
[elsif <boolean_expression> then
    <sequential_statement(s)>]
 . . .
[else
    <sequential_statement(s)>]
end if;
```

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

```
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

```
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 Statement

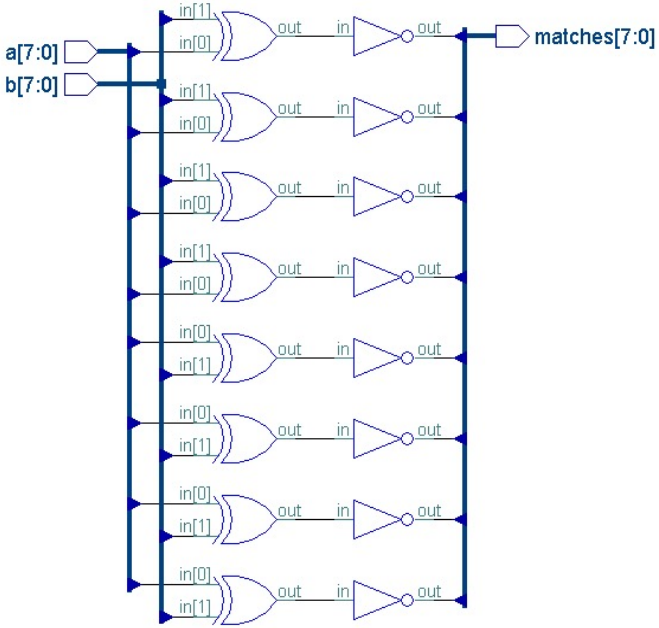
```
[loop_label]: for <identifier> in discrete_range loop
      <sequential_statements>
      end loop [loop_label];
```

`<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

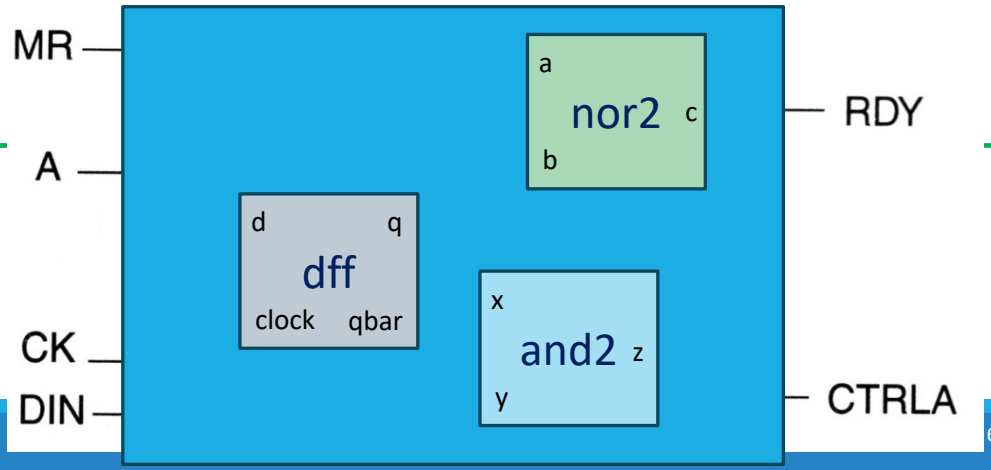
Connected to

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
```

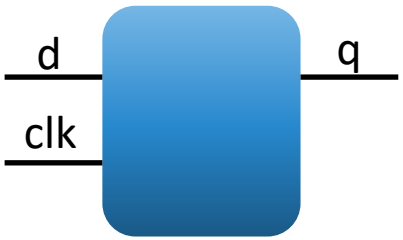
Component Instantiation Example

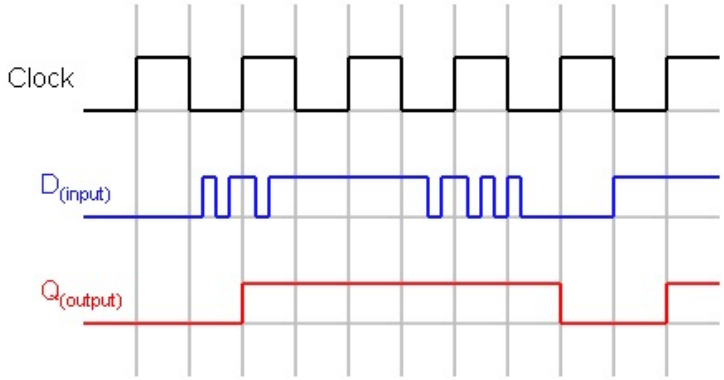
```
library ieee;
use ieee.std_logic_1164.all;
entity glue_logic is
  port (A, CK, MR, DIN: in std_logic;
        RDY, CTRLA: out std_logic);
end glue_logic ;
architecture STRUCT of glue_logic is
signal S1, S2: BIT;
begin
 D1: entity work.DFF  port map (D=>A, CLOCK=>CK, Q=>S1, QBAR=>S2);
 A1: entity 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 Flop-Flop with . . .

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

 end process ff_d_rst;
end rtl;
```

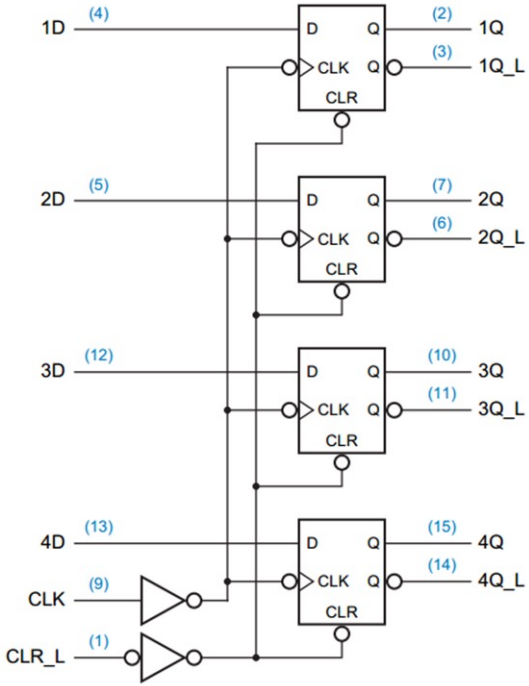
D Flip-Flop with . . . .

```
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 Flip-Flop with . . . .

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

 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, rst: 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)
 begin
  if (rst='1') then
    q <= (others =>'0');     -- q <= '00000000'
  elsif(rising_edge (clk)) then
      q <= d;
  end if;
 end process reg_d_arst;
end rtl;
```

??

```
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

Function	Inputs S1	Inputs S0	Next state QA*	Next state QB*	Next state QC*	Next state 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

```
194 is
r(3 downto 0);
r(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_e
      case ctrl is
     when "11" =>
        when "10" =
        when "01" =
     when others =>
      end case;
    end if;
end process;
q <= temp_q;
end behav;
```

Counter

```
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;
```

```
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;

end process count_proc;
 count <= std_logic_vector(count_i);
end architecture rtl;
```

?

Up/Down Counter

```
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;
```

```
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;
```

Contador Up/Down en VHDL Enteros

```
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_v
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;
```

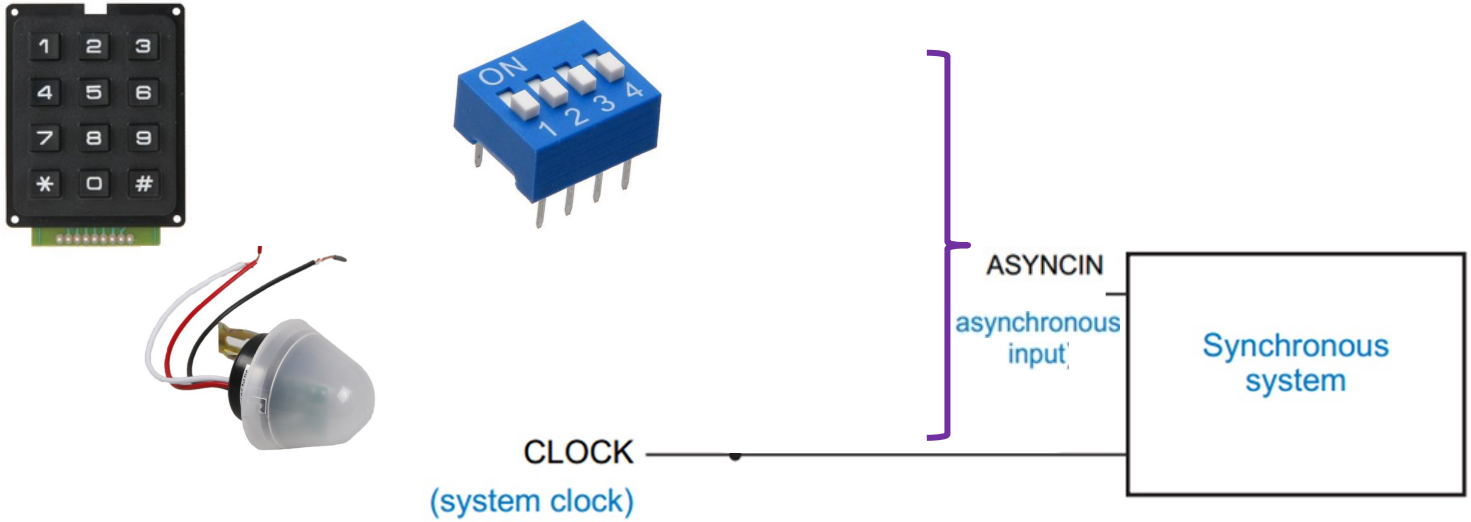
Up/Down Counter - Integers

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

entity counter_ud_i is
  generic(cnt_w: natural:= 8)
  port (
        -- clock & reset inputs
        clk      : in std_logic;
        rst_n    : in std_logic;
        -- control input signals
        up_dw    : in std_logic;
        -- ouptuts
        count    : out std_logic_v
0));
end counter_ud_i;
```

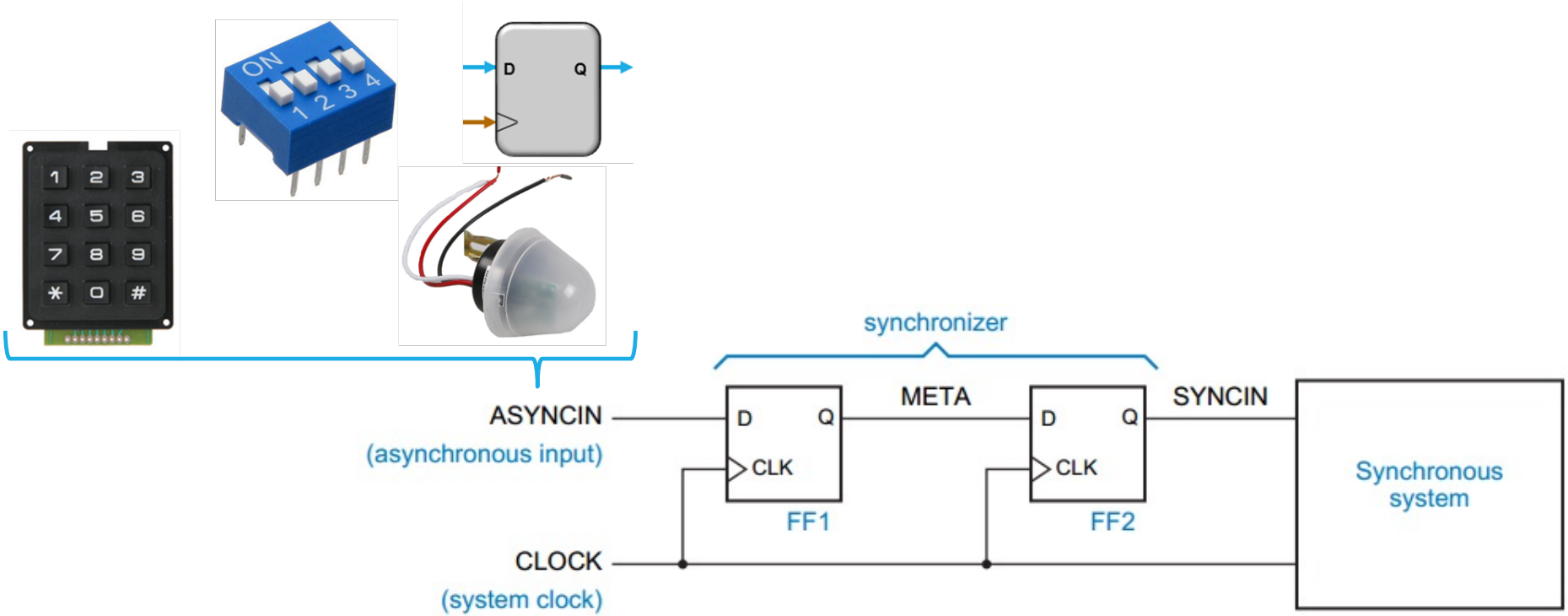
?

```
architecture rtl of counter_ud_i is
begin
 count_proc: process(clk, rst)
    variable count_i: integer range 0 to 199;
 begin
 if(rst_n = '0') then
   count_i := 0;
 elsif(rising_edge(clk)) then
   if(count_i = 200) 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;
```

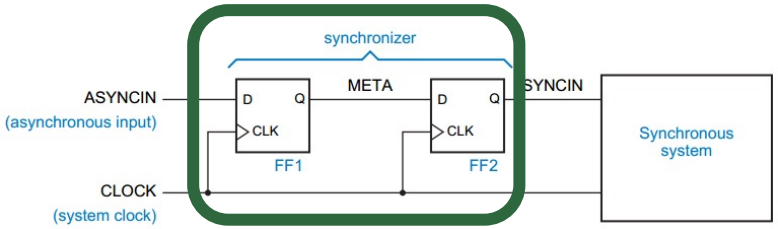
Asynchronous Inputs

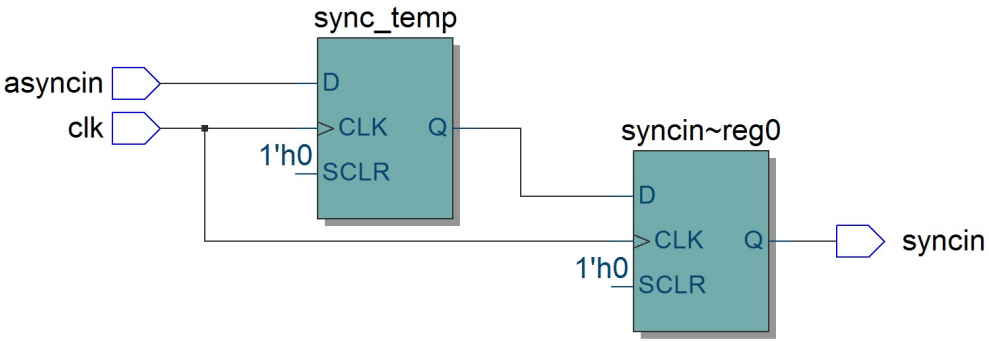


Synchronizer



Synchronizer

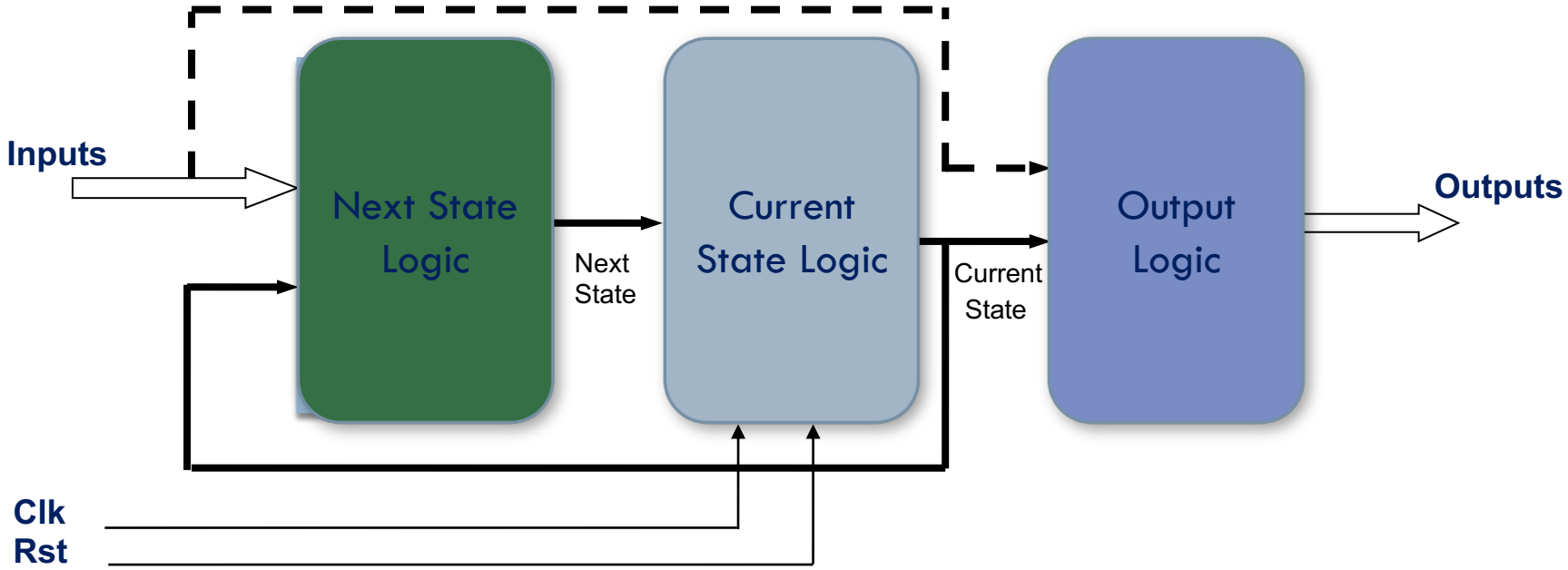




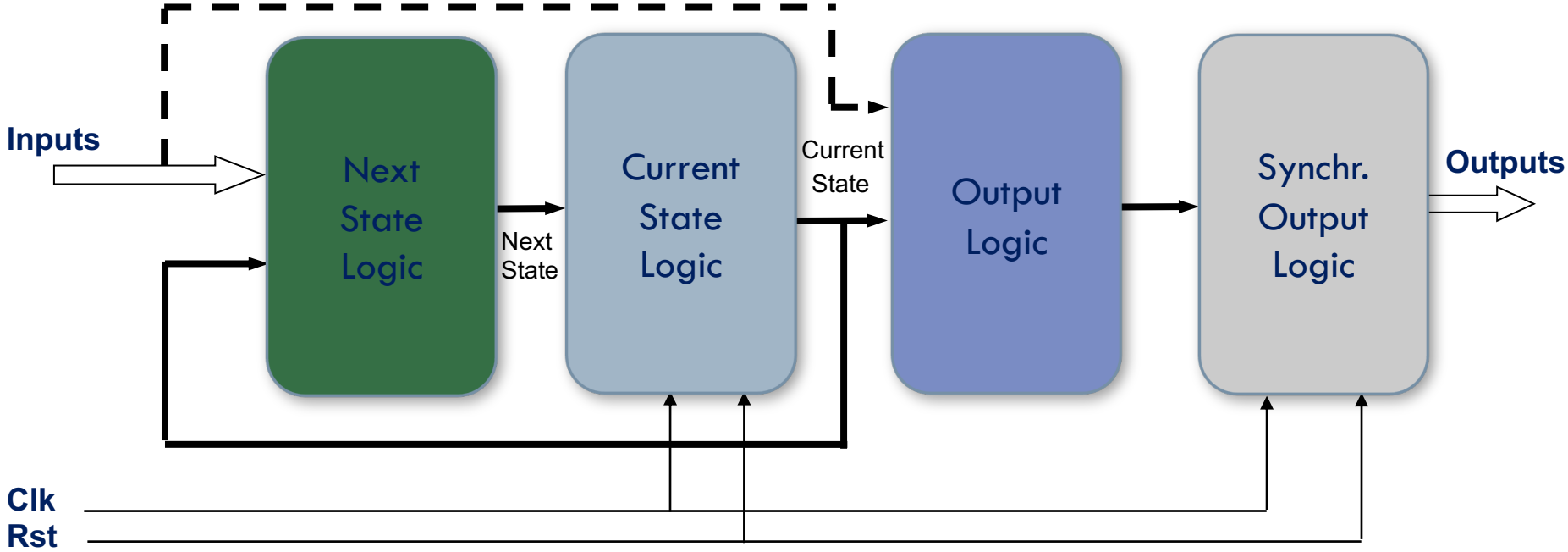

```
library ieee;
use ieee.std_logic_1164.all;
entity synchronizer is
   port(
       clk       : in  std_logic;
       asyncin   : in  std_logic;
       syncin    : out std_logic);
end synchronizer;

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

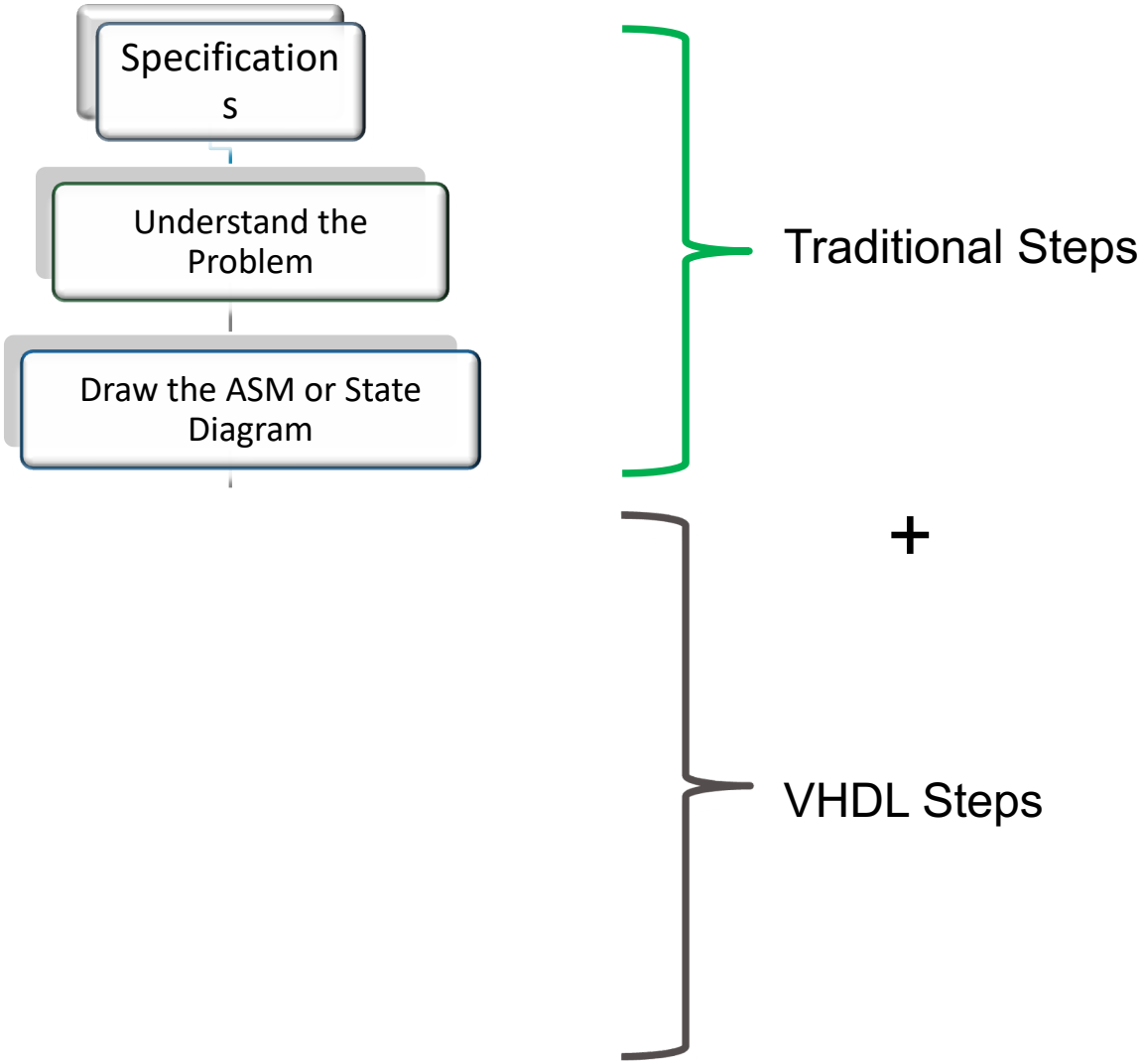
State Machine General Scheme 1



State Machine General Scheme 2



FSM VHDL General Design Flow



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 $\lceil \log_2 n \rceil$ 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)

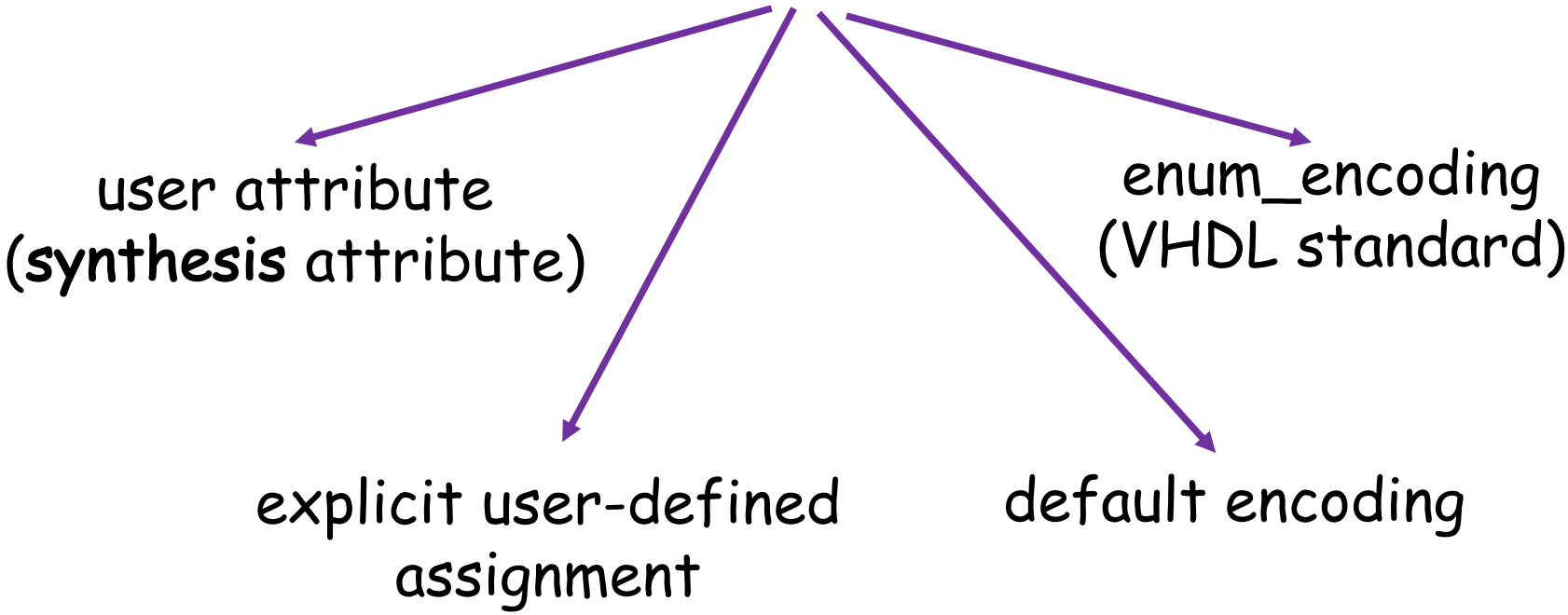
FSM Encoding Schemes

	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

How is the map process done ?

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



syn_encoding – Quartus & Synplify

- *syn_encoding* is the synthesis *user-attribute* of Quartus (Synplify) that specifies encoding for the states modeled by an enumeration type
- To use the *syn_encoding* attribute, it must first be declared as *string* type. Then, assign a value to it, referencing the current state signal.

```
-- declare the (state-machine) enumerated type
type my_fms_states is (IDLE,START,STOP_1BIT,PARITY,SHIFT);
-- declare signals as my_fsm_states type
signal nxt_state, current_state: my_fsm_states;

-- set the style encoding
attribute syn_encoding: string;
attribute syn_encoding of my_fms_states : type is "one-hot";
```

Results for Different Encoding Schemes

Simple, 5 states, state machine

	One-hot safe	One-hot	Gray	Gray-Safe	Binary	Johnson
Total combinational functions	76	66	66	68	66	68
Dedicated logic registers	45	45	43	43	43	43
Max. frq.	352.24	340.95	331.02	335.01	338.34	311.72

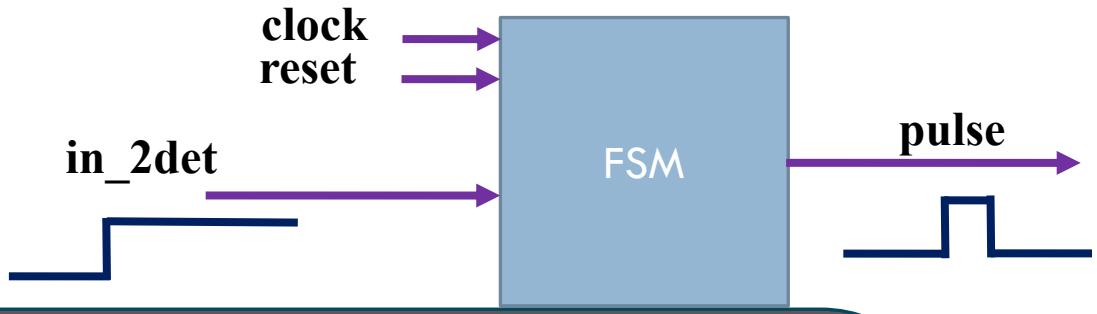
Results for Different Encoding Schemes

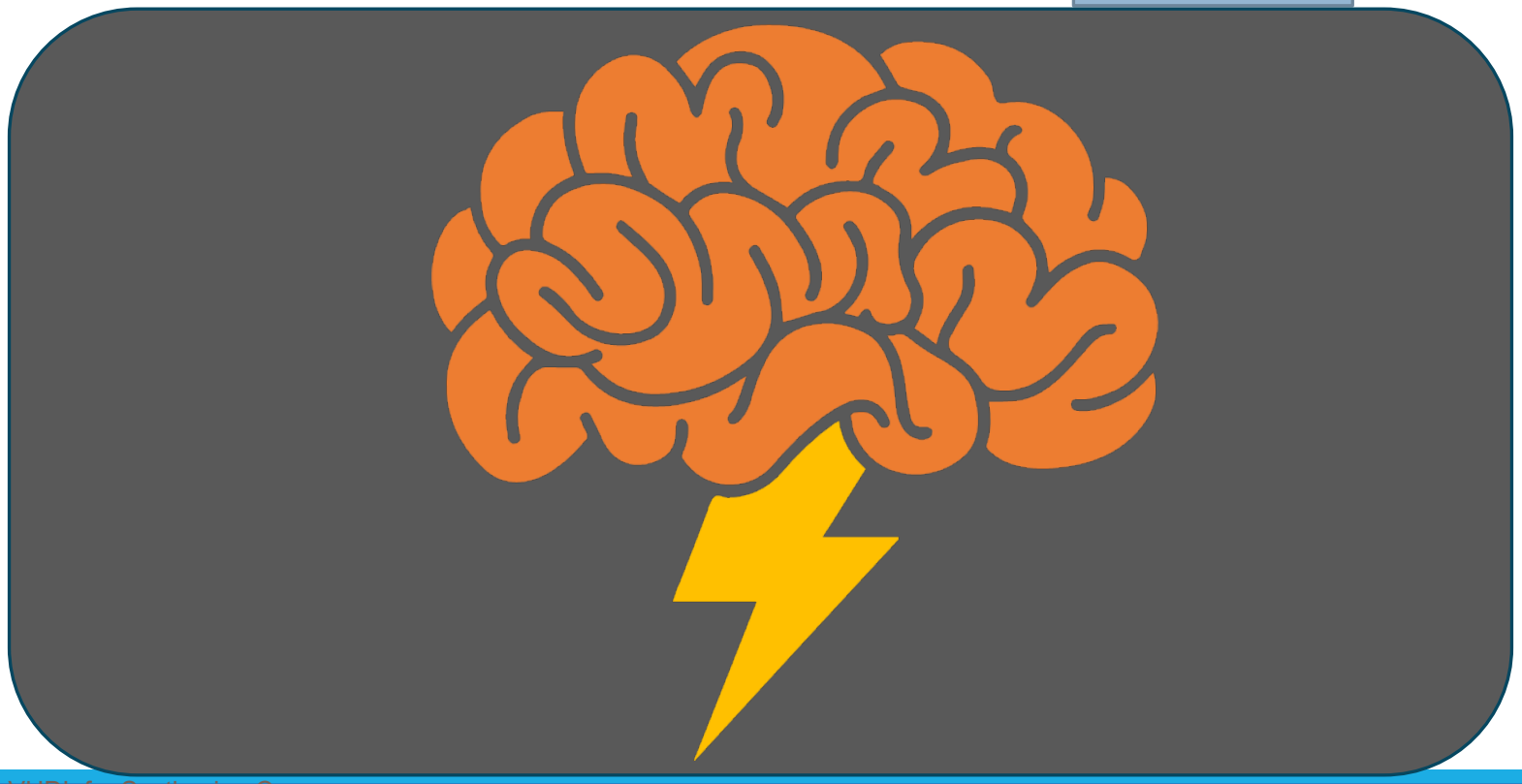
19 states, state machine

	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

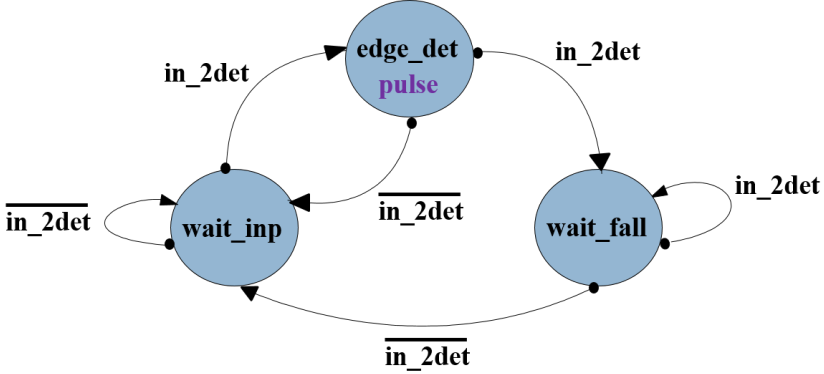
State Machine VHDL Coding - Example

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;
```

in_2det

state

pulse

Clk

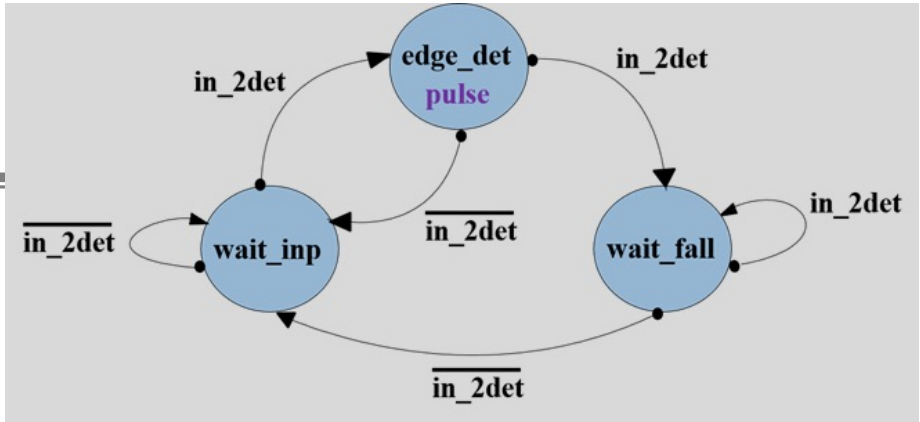
Rst

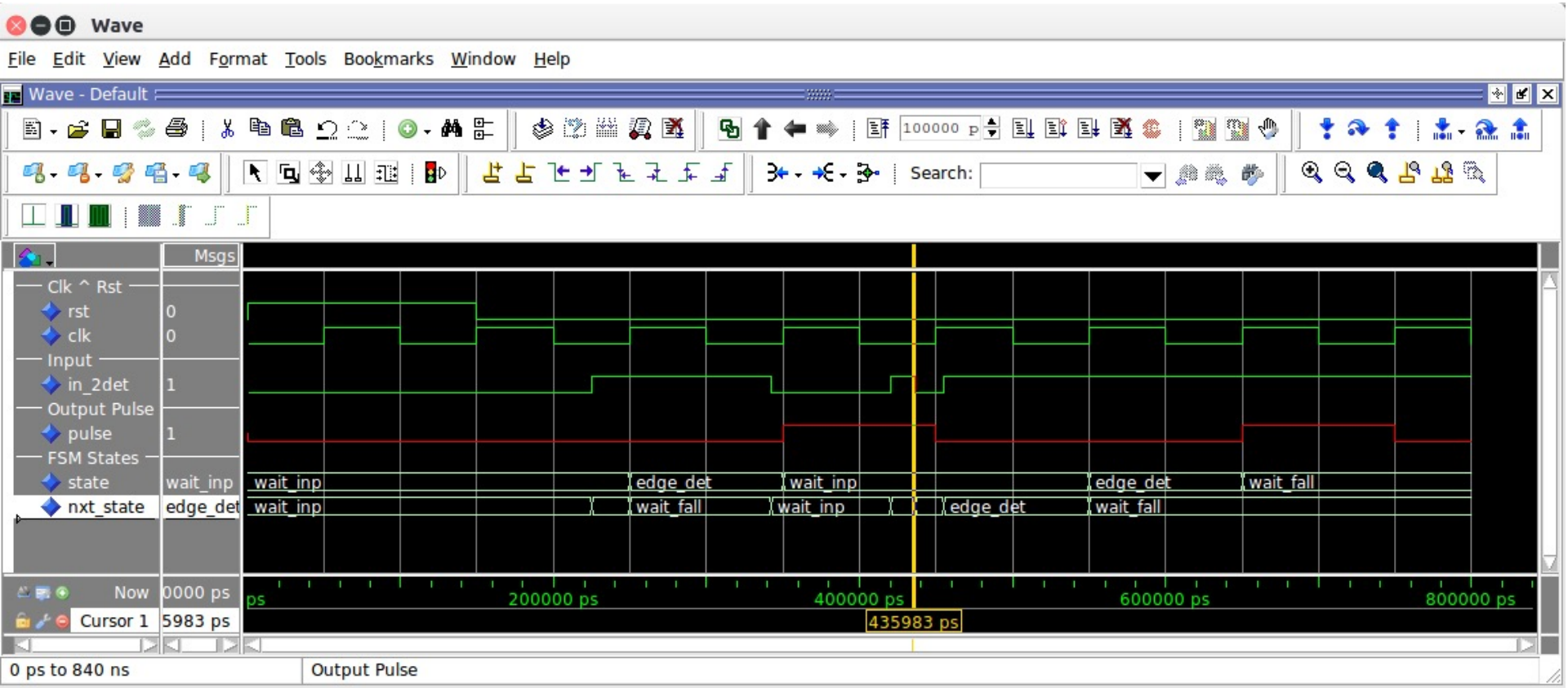
State Machine VHDL Coding (complete)

```
-- 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;
```

```
-- 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





Another Ex.: Memory Controller FSM

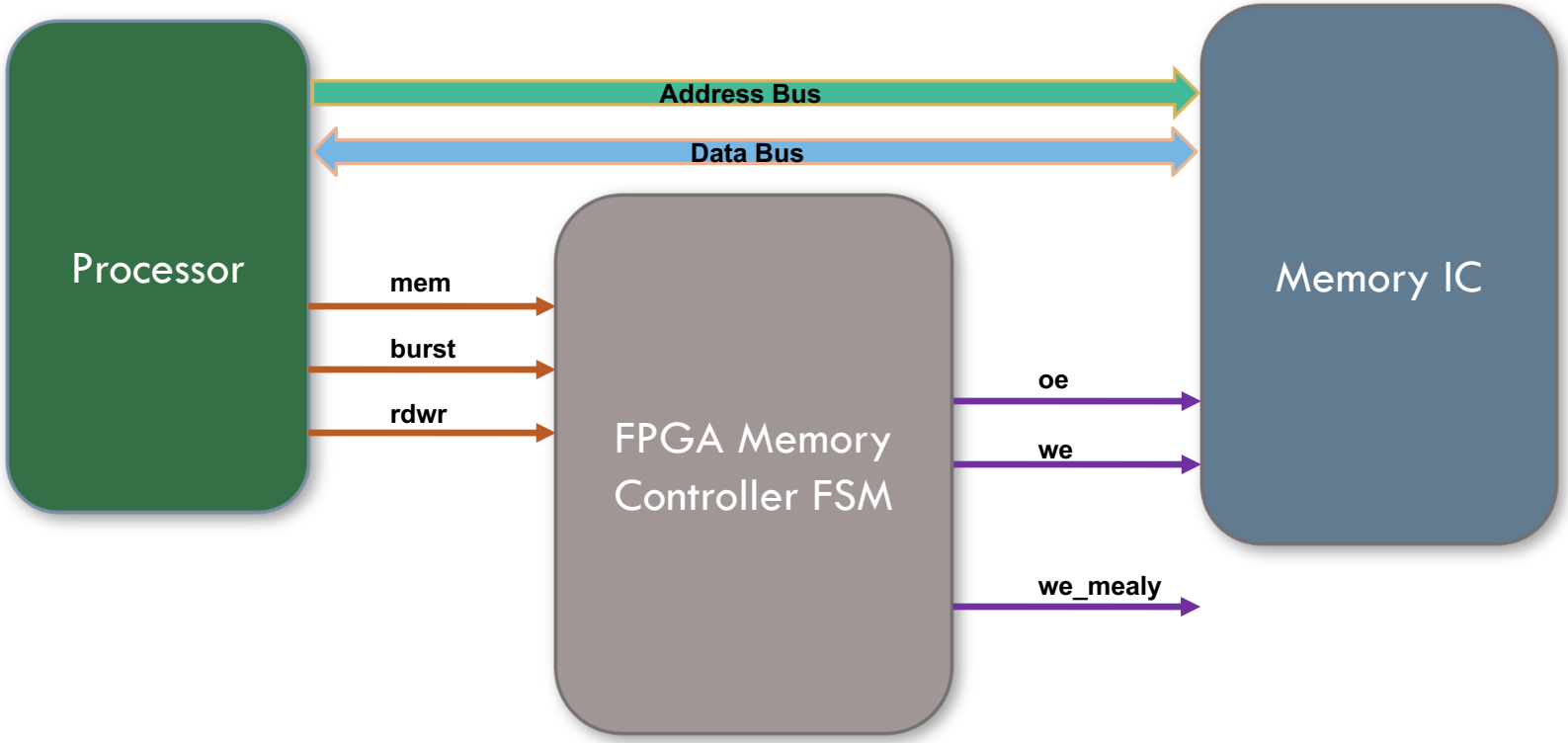
Let's try to obtain an state diagram of a hypothetical memory controller FSM that has the following specifications:

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:

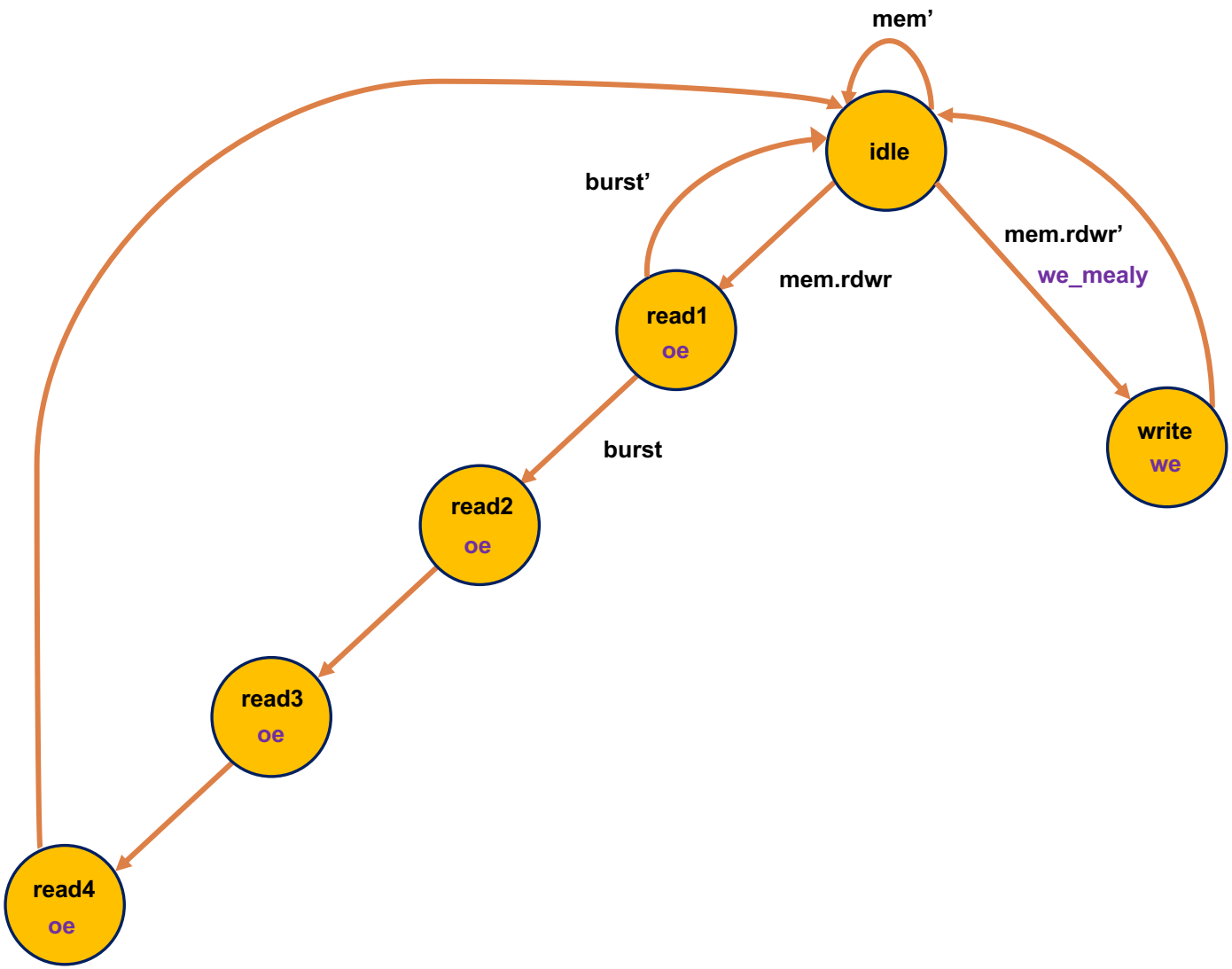
- ***mem'*** : represents that *no* memory operation is required (mem='0')
- ***mem.rdwr***: represents that a memory *read* operation is required (mem=rdwr='1').
- ***mem.rdwr'***: represents that a memory *write* operation is required (mem='1'; rdwr='0')

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
```

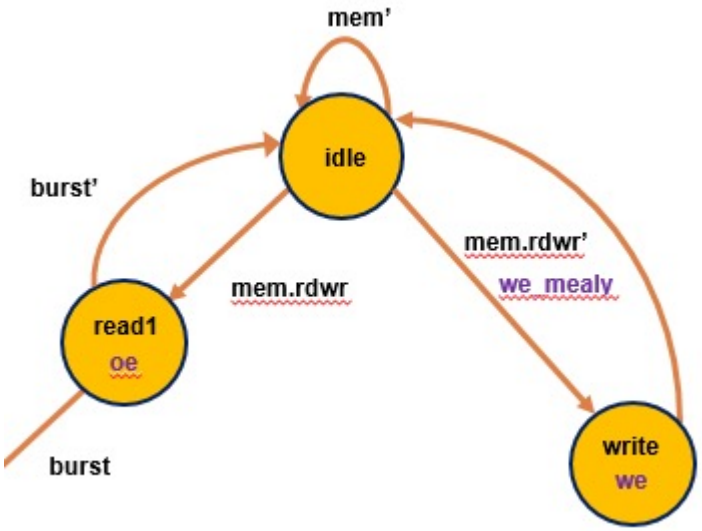
Memory Controller FSM - VHDL Code

```
-- 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)

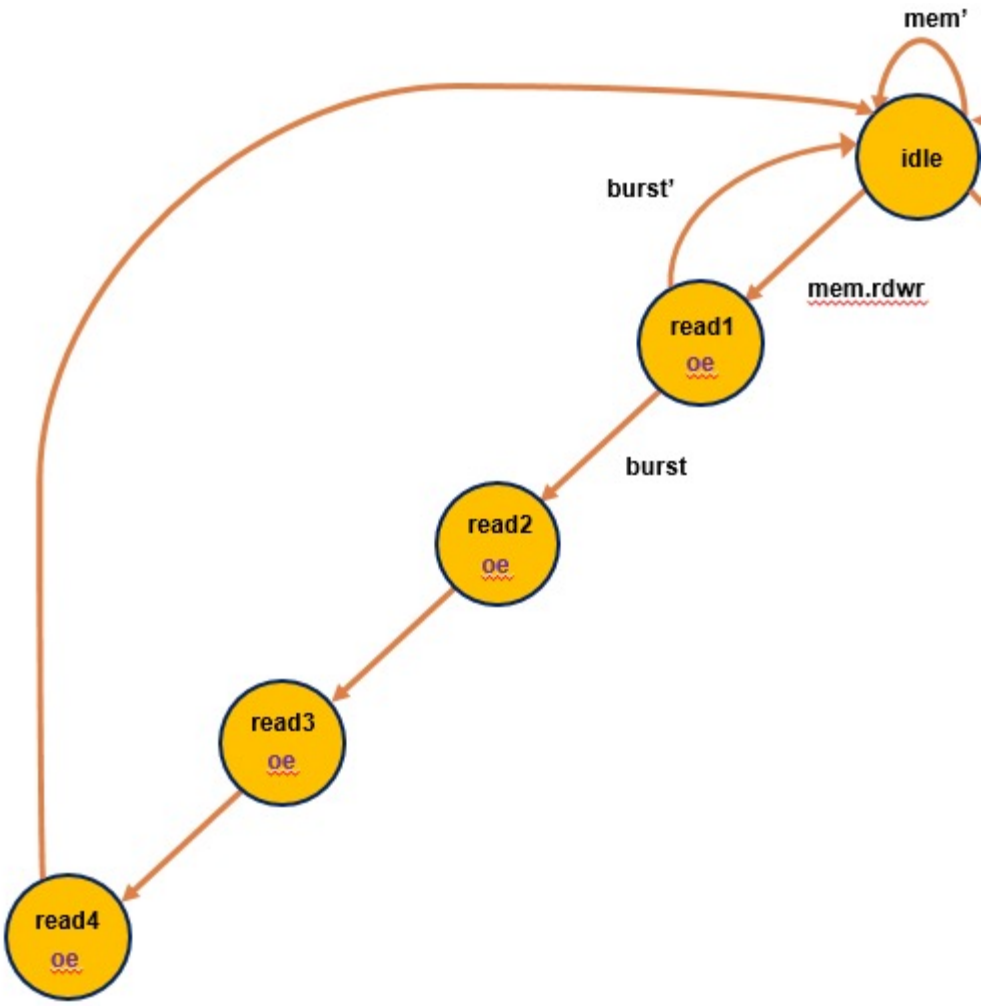
```
-- next-state logic
nxp:process(crrnt_state,mem,rdwr,burst)
begin
 case crrnt_state is
   when idle =>
     if mem = '1 ' then
       if rdwr = '1' then
         next_state <= read1;
       else
         next_state <= write;
       end if;
     else
        next_state <= idle;
     end if;
   when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



Memory Controller FSM – VHDL Code

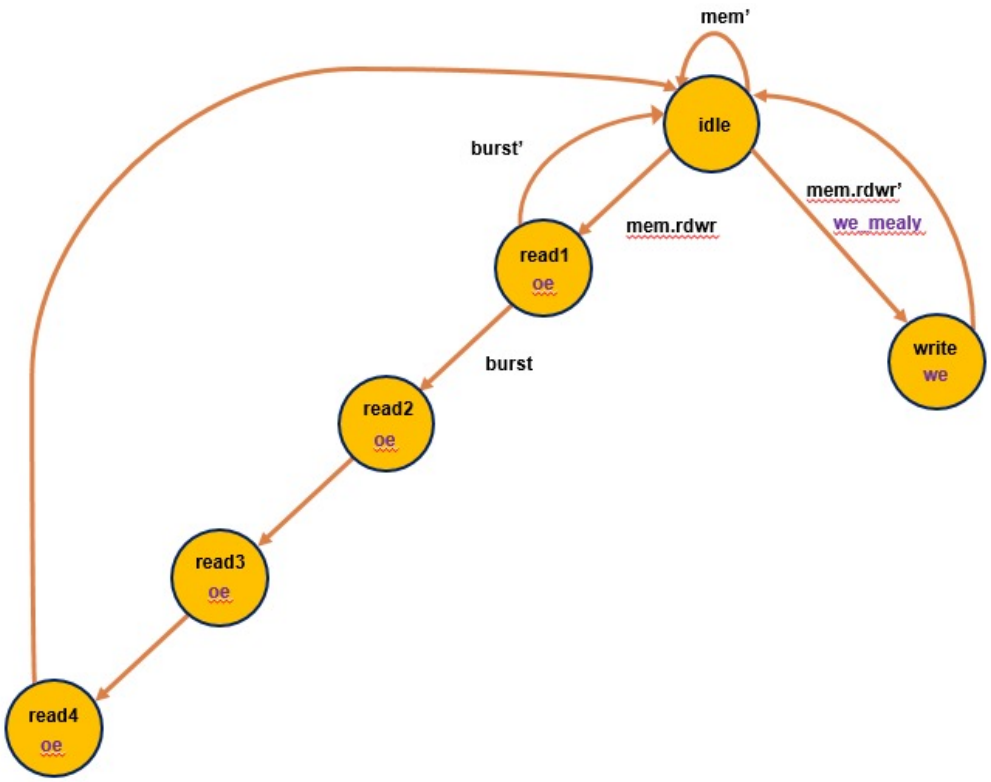
- Next state process (2)

```
  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

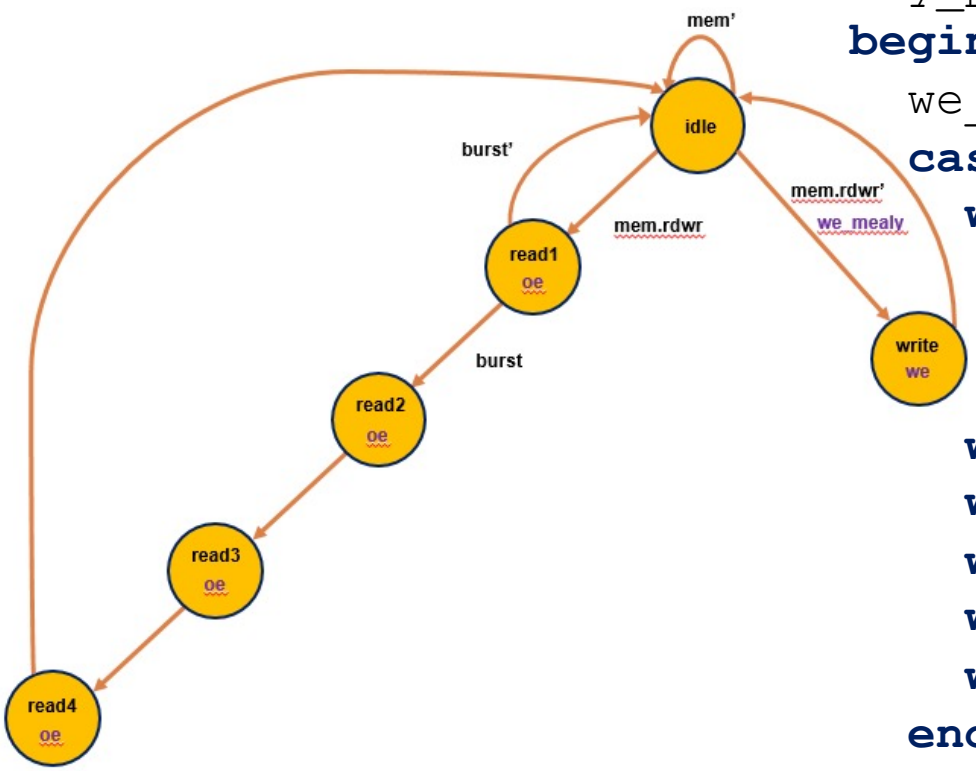
- Moore outputs process




```
-- 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

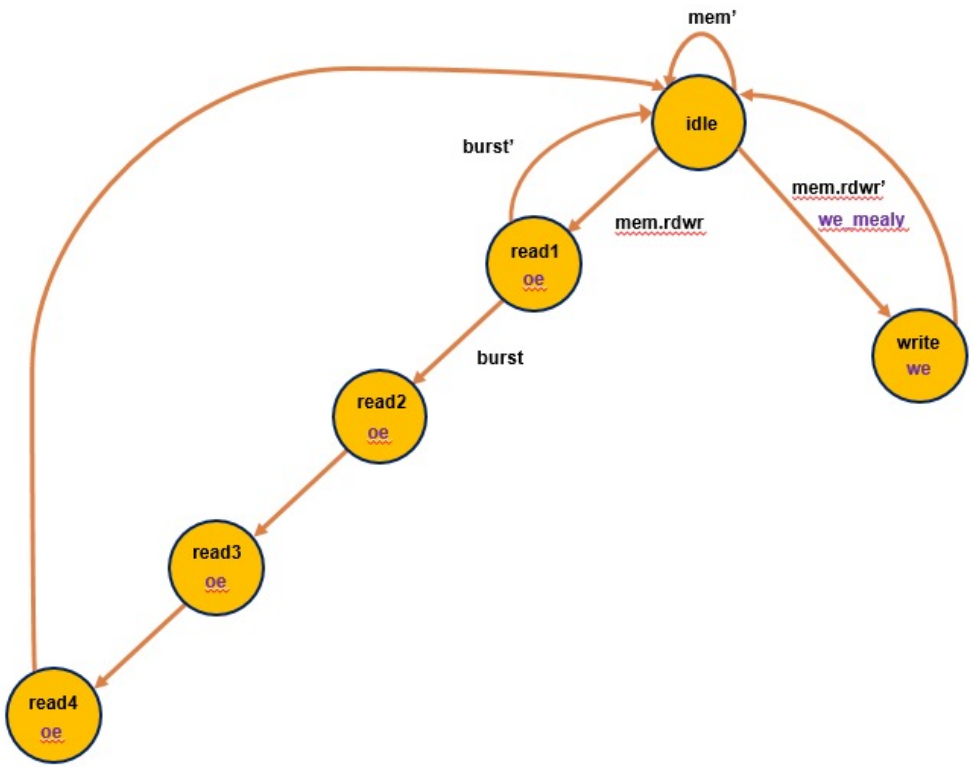
- Mealy output process




```
-- 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 statemer




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