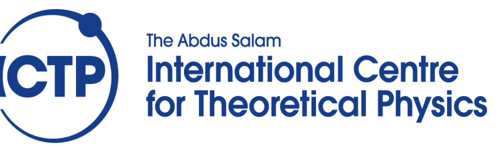


2499-2

International Training Workshop on FPGA Design for Scientific Instrumentation and Computing

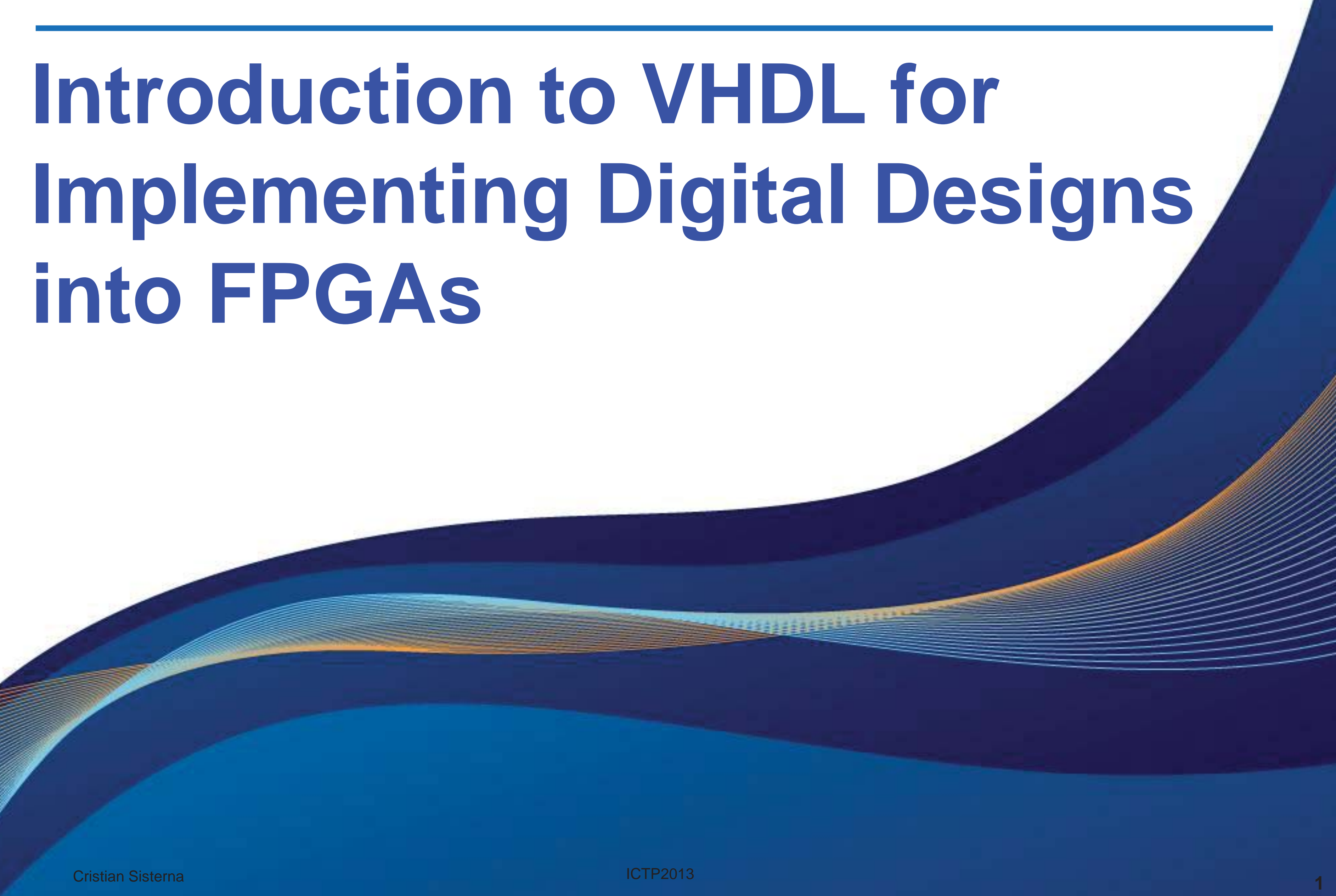
11 - 22 November 2013

Introduction to VHDL for Implementing Digital Designs into FPGAs

Cristian SISTERNA
National University of San Juan
San Juan
Argentina

Strada Costiera, 11 - 34151 - Trieste - Italy • Tel. +39 0402240111 • Fax. +39 040224163 • sci_info@ictp.it • www.ictp.it
ICTP is governed by UNESCO, IAEA, and Italy, and it is a UNESCO Category 1 Institute

Introduction to VHDL for Implementing Digital Designs into FPGAs



Introduction

Very High Speed ICs

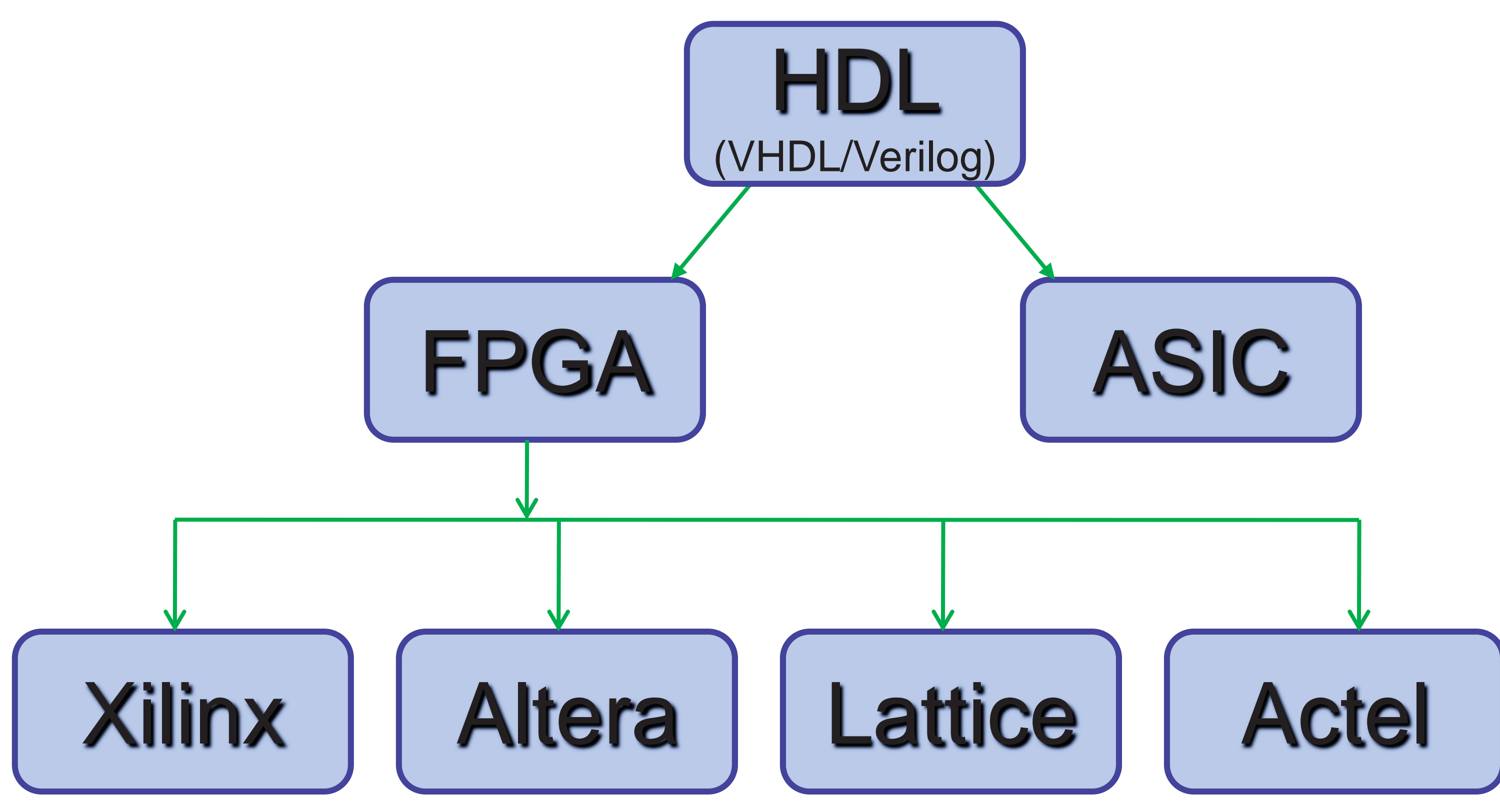
Hardware

Description

Language

V
H
D
L

Hardware Description Language



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

What is not VHDL

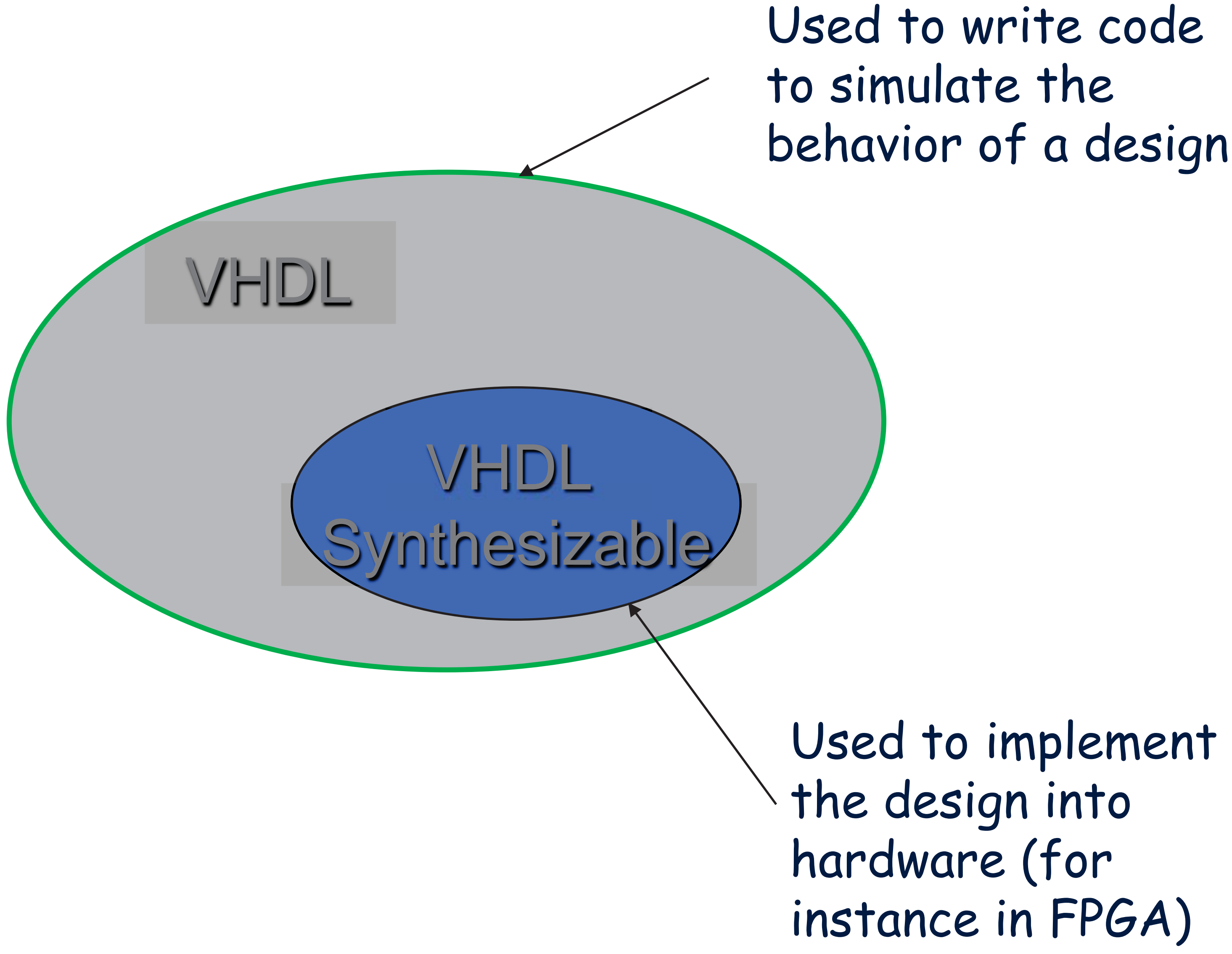
- Verilog o VHDL IS NOT A programming language; IT IS A **HARDWARE** DESCRIPTION LENGUAGE

- Verilog o VHDL is not (yet) a highly abstract language:

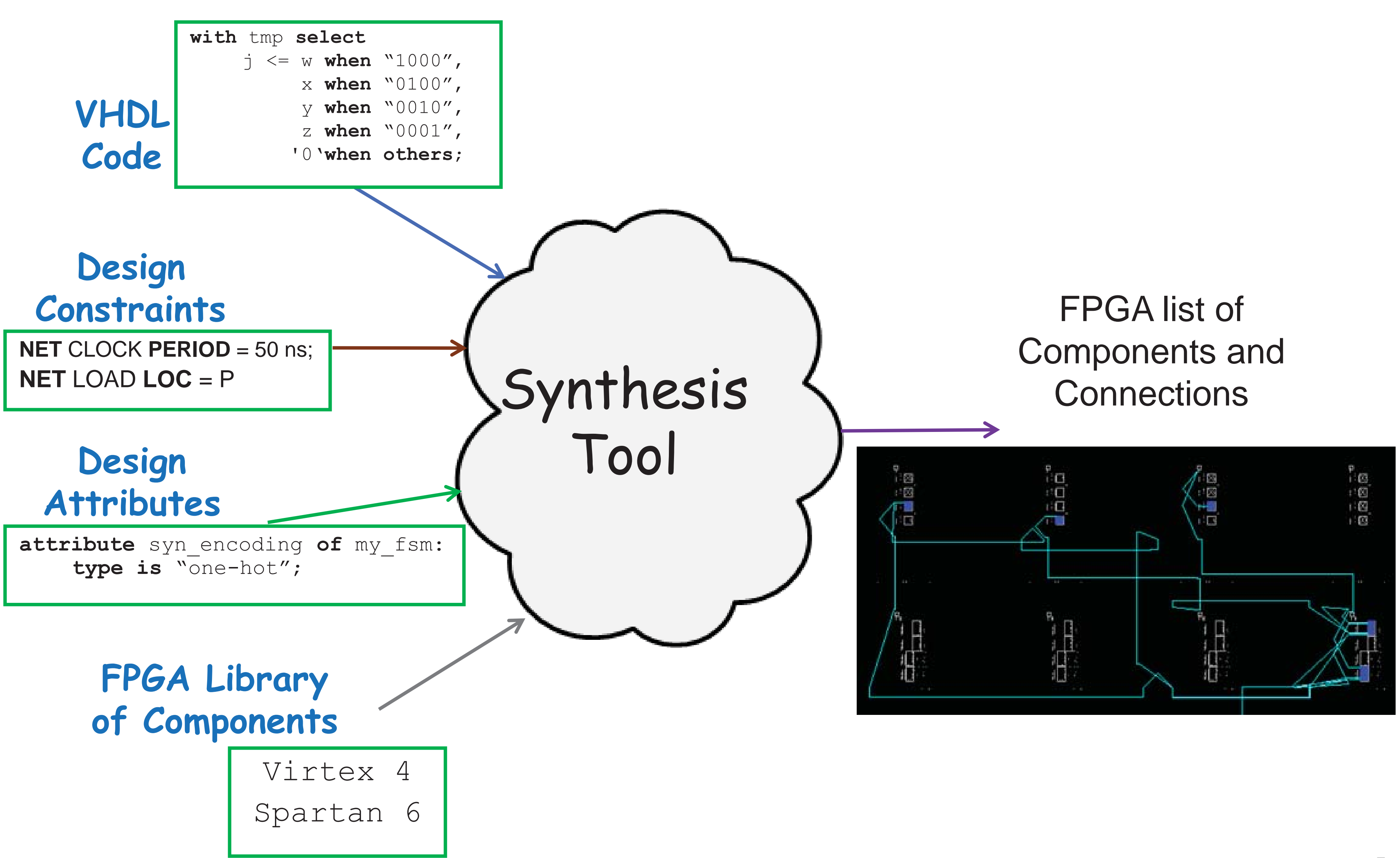
$$y(n) = 0.75y(n-1) + 0.3x(n) ;$$

(Simulink/FPGA design flow)

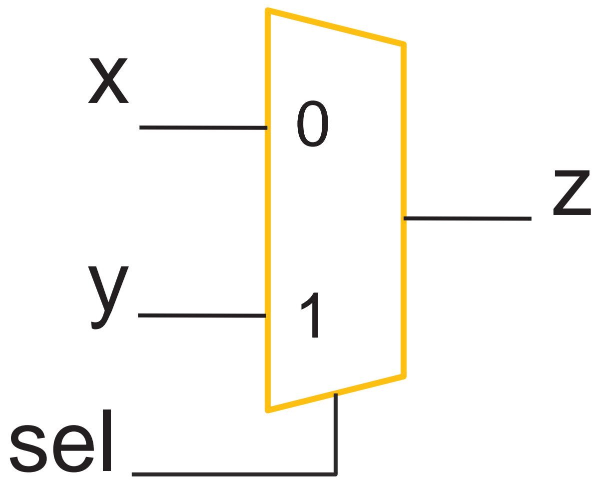
HDL Synthesis Sub-Set



VHDL - Synthesis



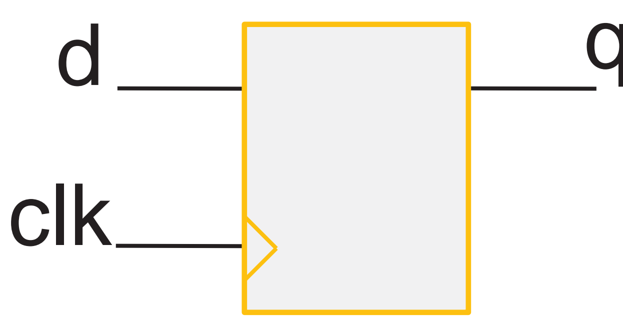
VHDL ‘Description’ Examples




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

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

VHDL 'Description' Ejemplos

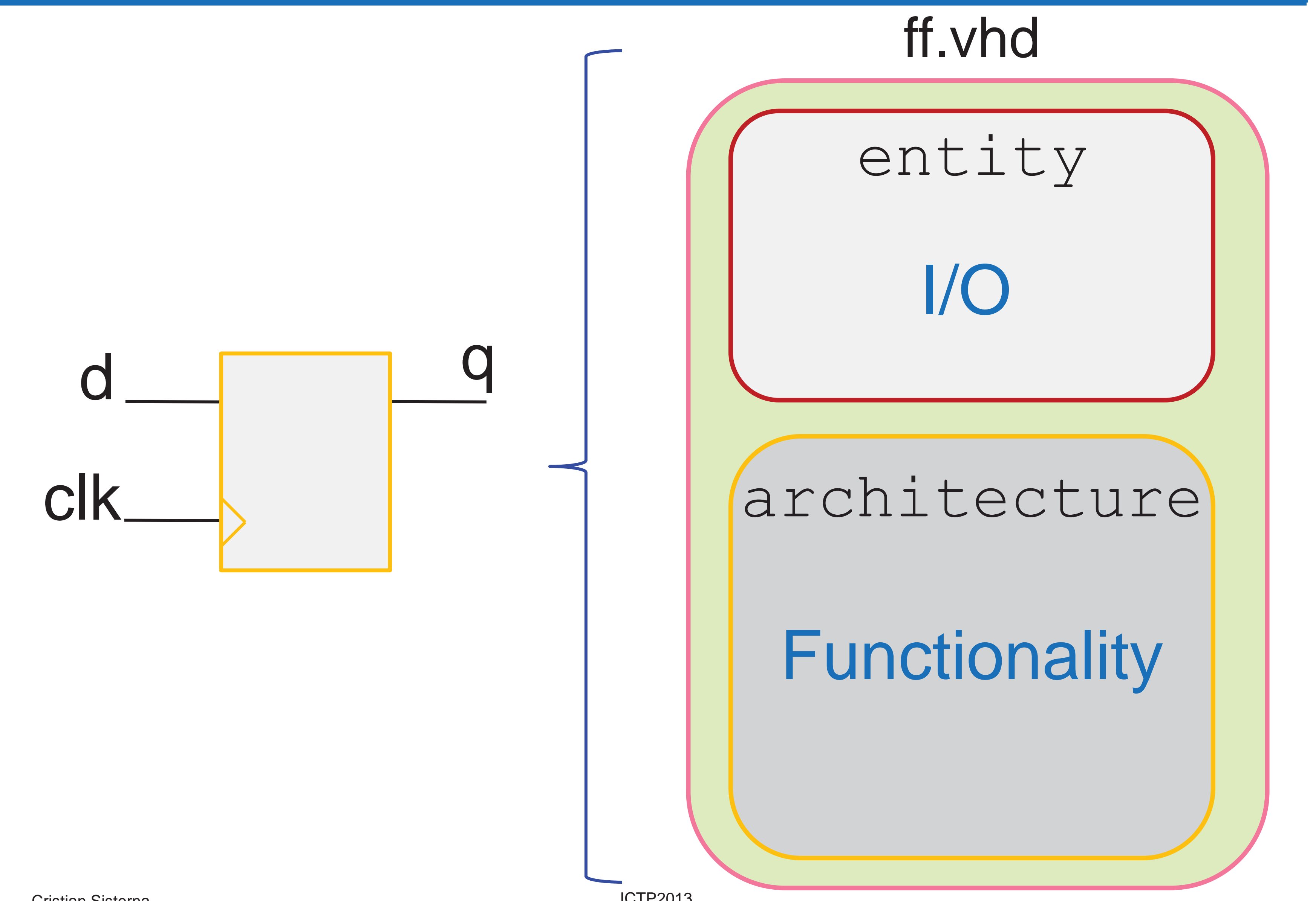



```
if(clk _↑‾) then
      q <= d;
else
      q <= q;
end if;
```

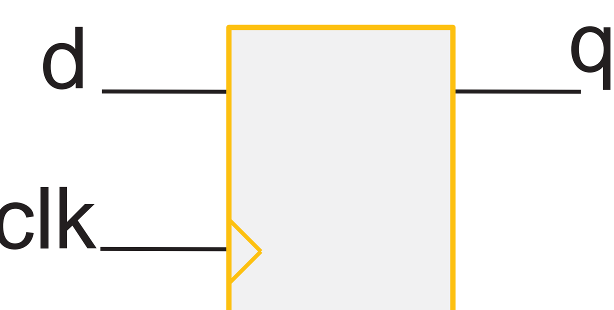
```
if(clk _↑‾) then
      q <= d;
end if;
```

```
if(rising_edge(clk)) then
      q <= d;
end if;
```

VHDL – Module Structure



VHDL–Estructura de un módulo

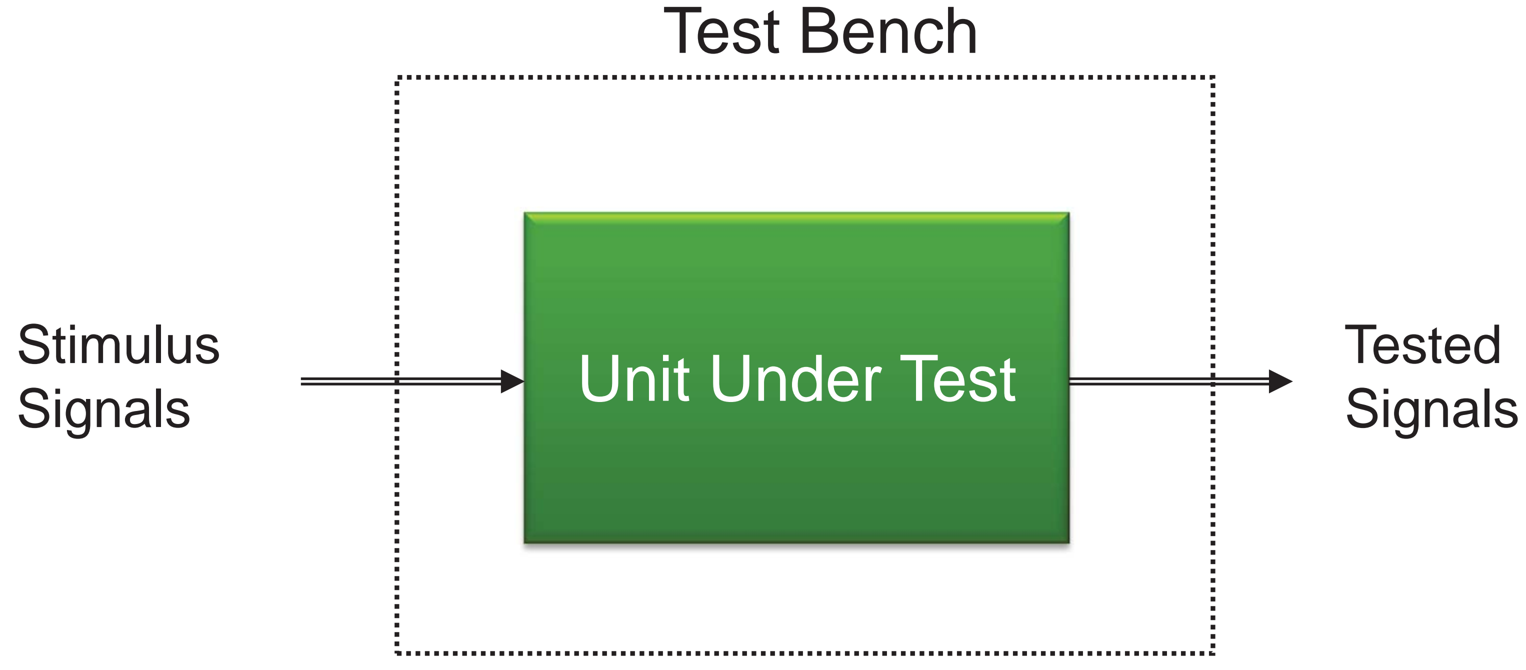


ff.vhd

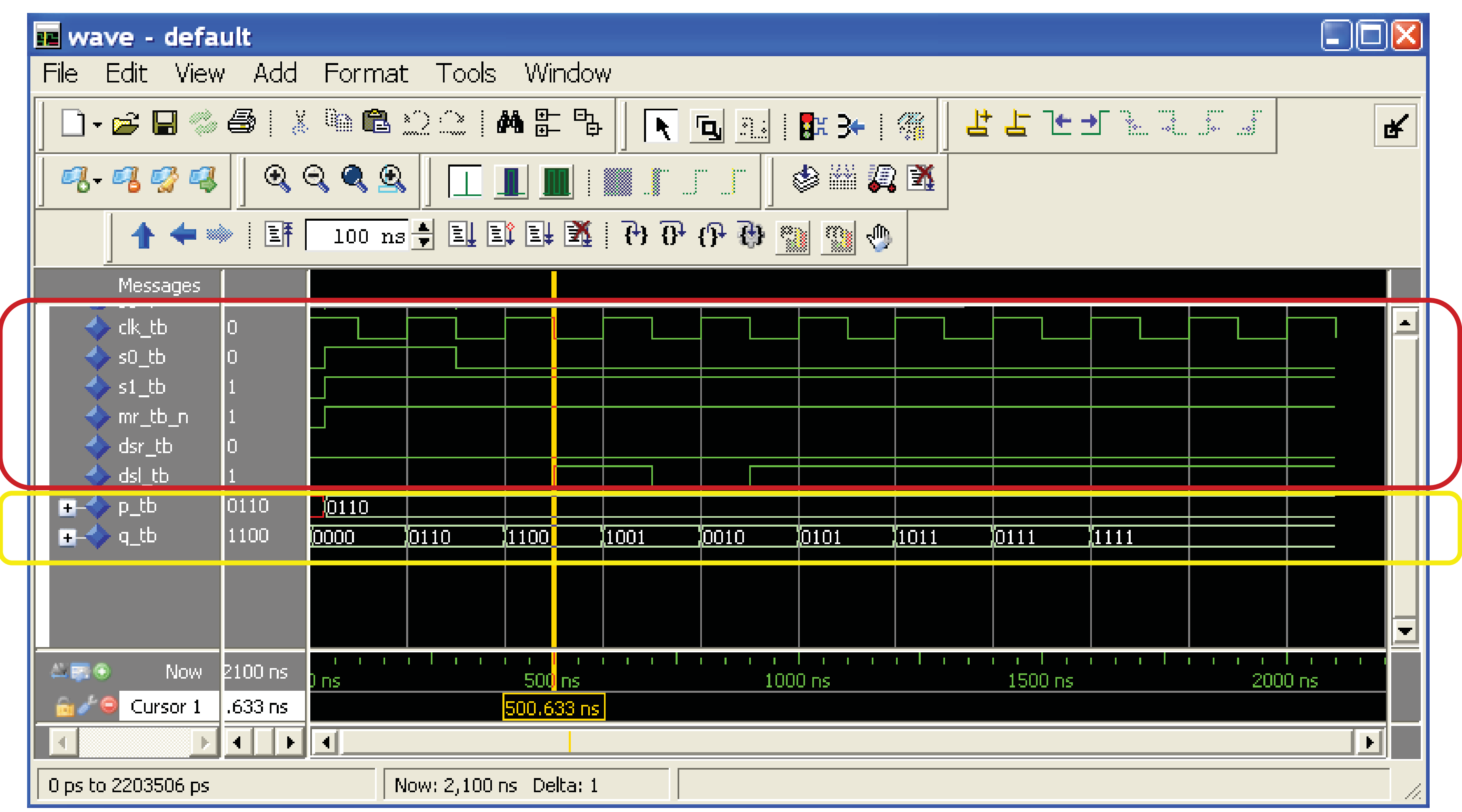
```
entity ff is
port(
    d,clk : in  std_logic;
    q     : out std_logic);
end ff;
```

```
architecture test of ff is
begin
process(clk)
begin
 if(rising_edge(clk)) then
      q <= d;
 end if;
end test;
```

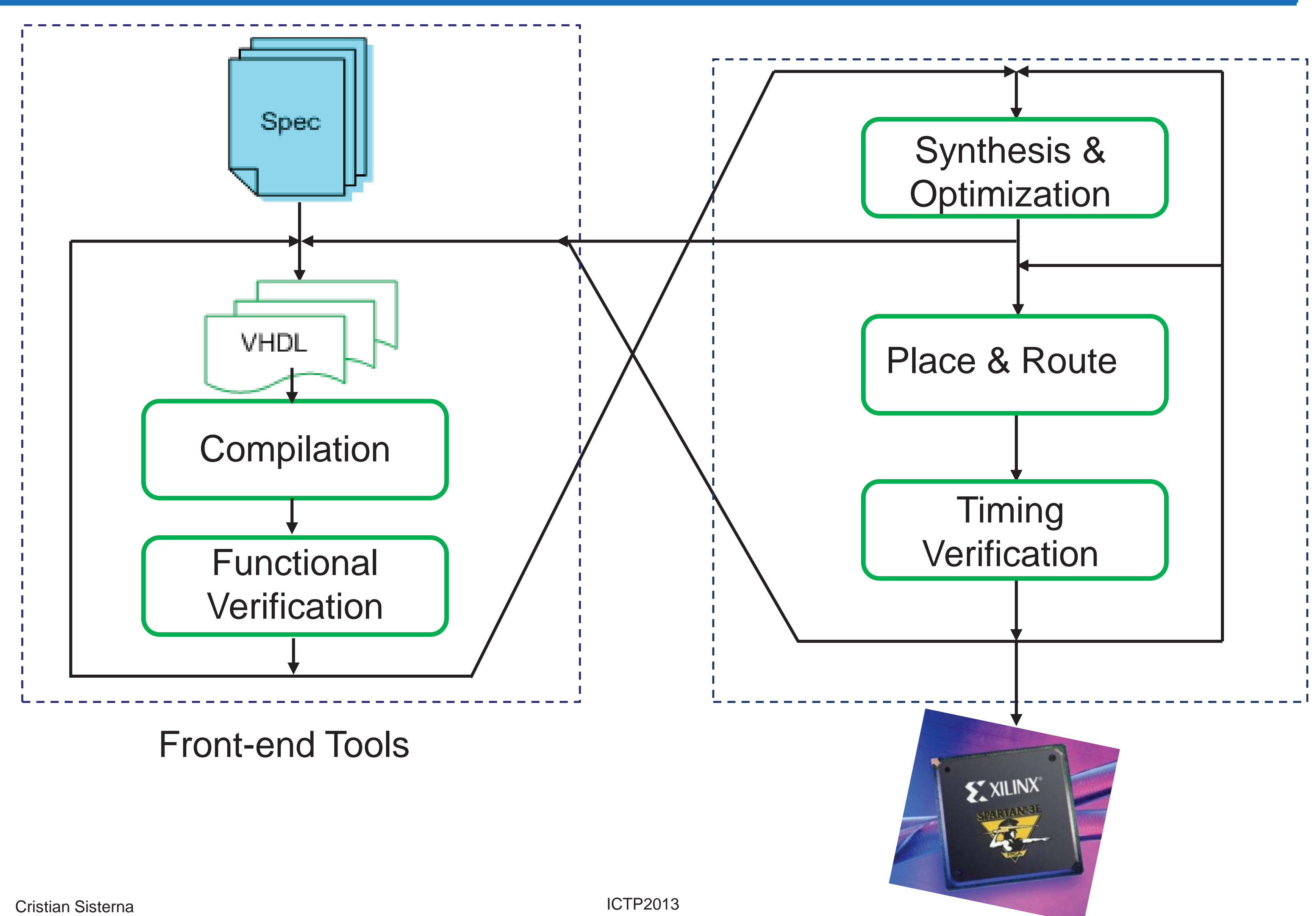
VHDL Code – Is it really Works? ?



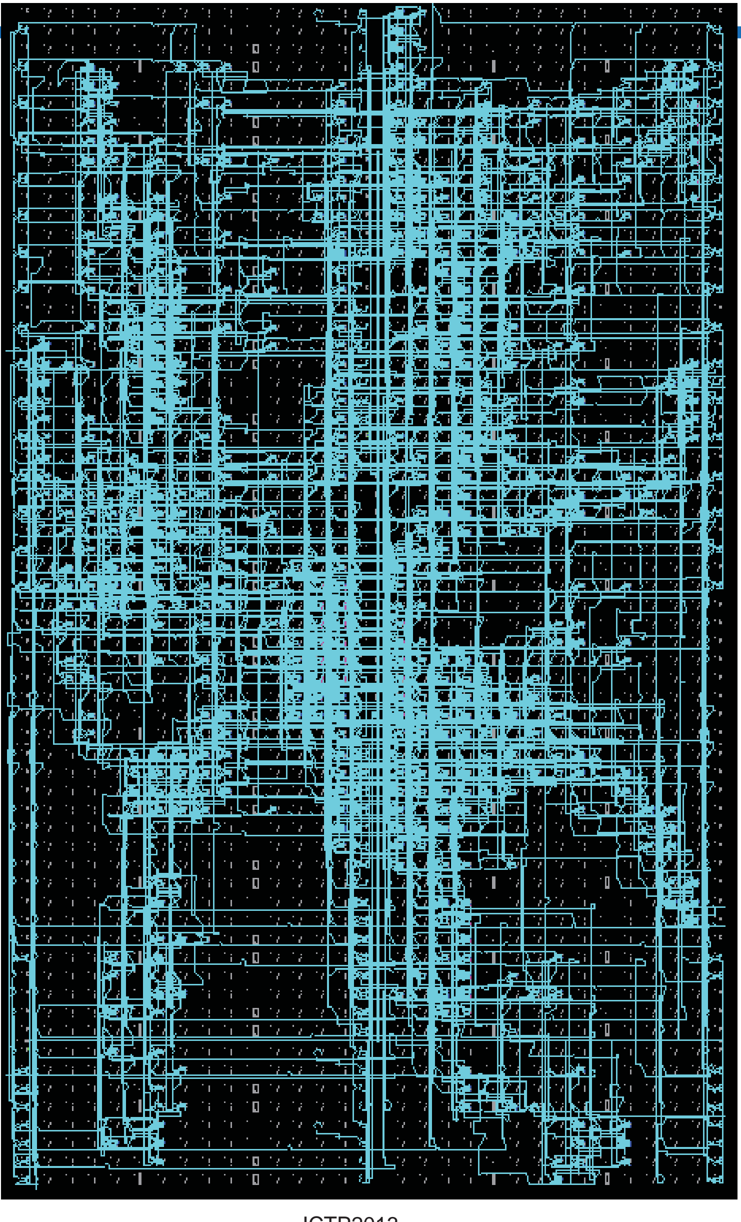
VHDL – Simulation / Verification



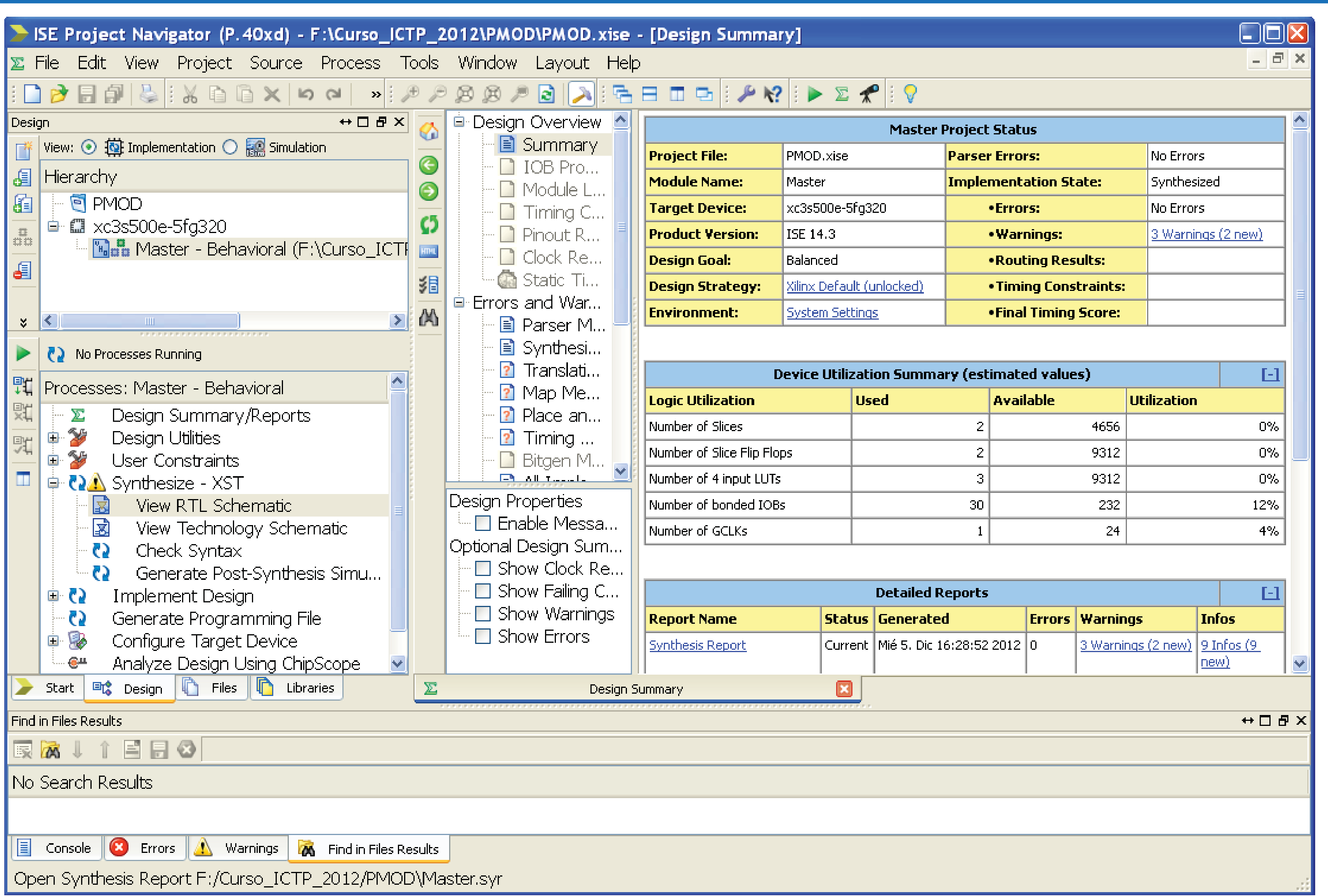
VHDL-FPGA Flujo de Diseño



Design Implemented in the FPGA

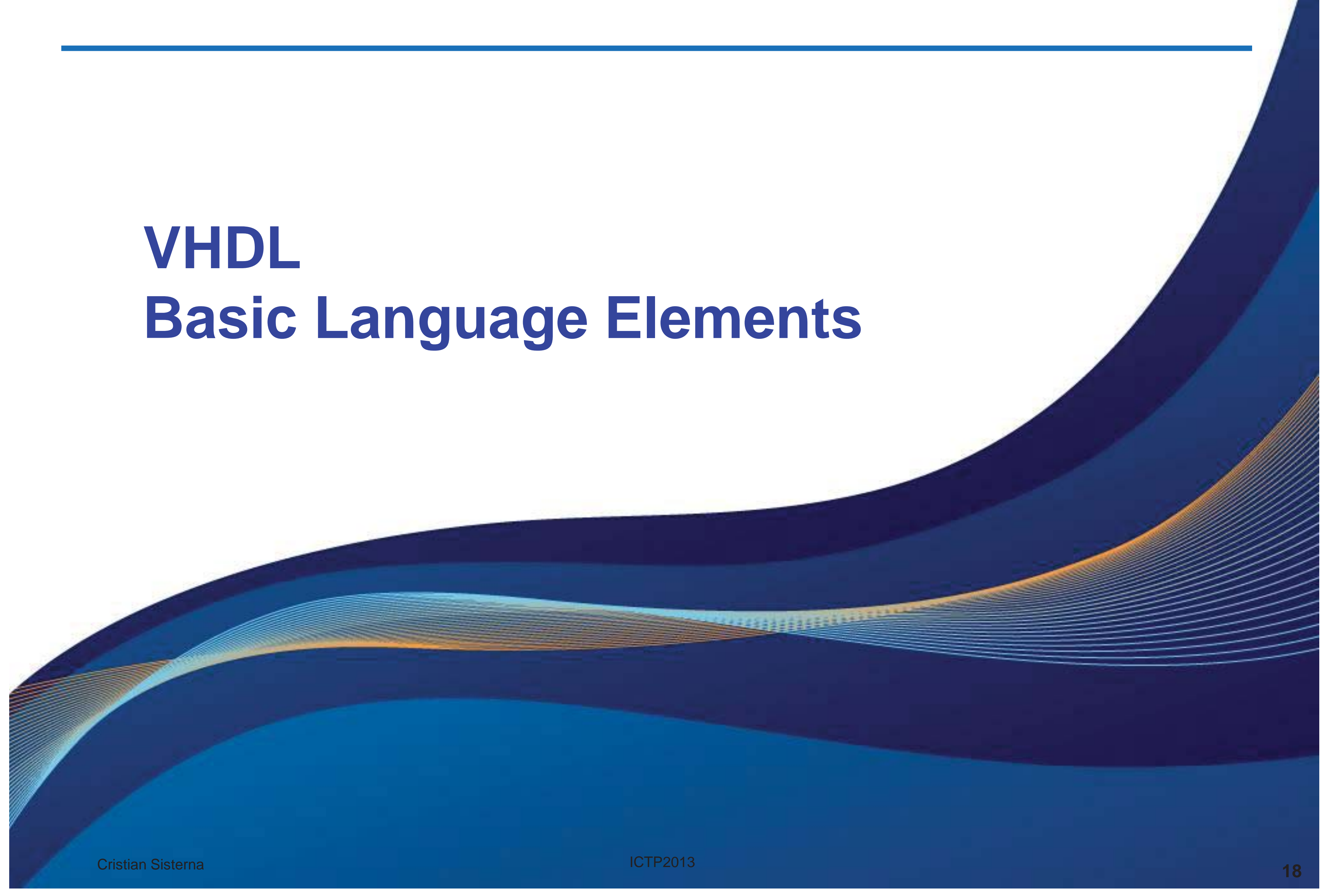


Software to Use: Xilinx ISE 13.3



FPGA Board to Use

VHDL
Basic Language Elements

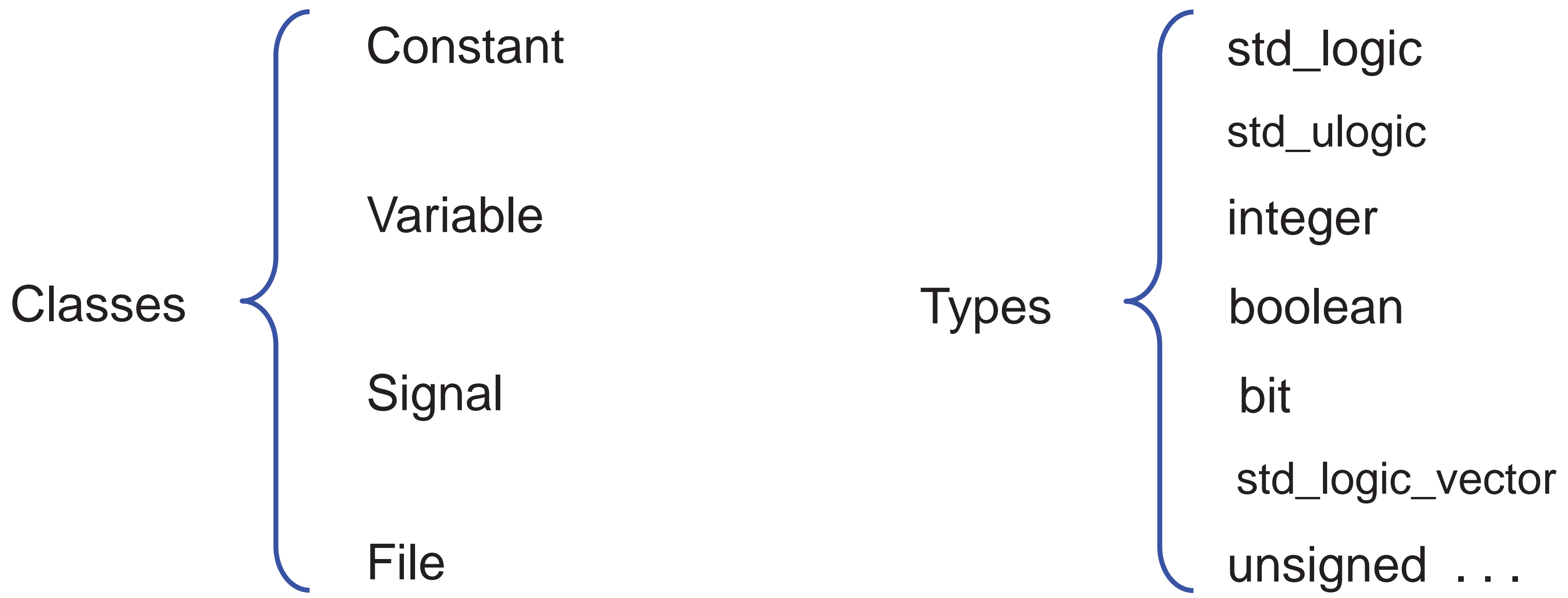


Identifiers

- A *basic identifier:*
  - May only contain alphabetic letters (A to Z and a to z), decimal digits (0 to 9) and the underline character (_)
  - Must start with an alphabetic letter
  - May not end with an underline character
  - May not include two successive underline characters
  - VHDL is not case-sensitive
  - No blank space(s) are allowed
  - Examples:
    - Same identifier
      - `Txclk, TxClk, TXCLK, TxCLK`
    - Legal identifiers
      - `Rst, Three_State_Enable, CS_244, Sel7D`
    - Illegal identifiers
      - `_Set, 80X86, large#bits, m__RAM, add_`

Data Objects

- Each data object has a **type** and a **class**
- The **type** indicates what type of data can be hold by the data object
- The **class** indicates what can be done with the data object



Signal Class

- A Data Object of class ***signal*** can hold information not only about the value of the data, but also the time at which the signal takes the new value

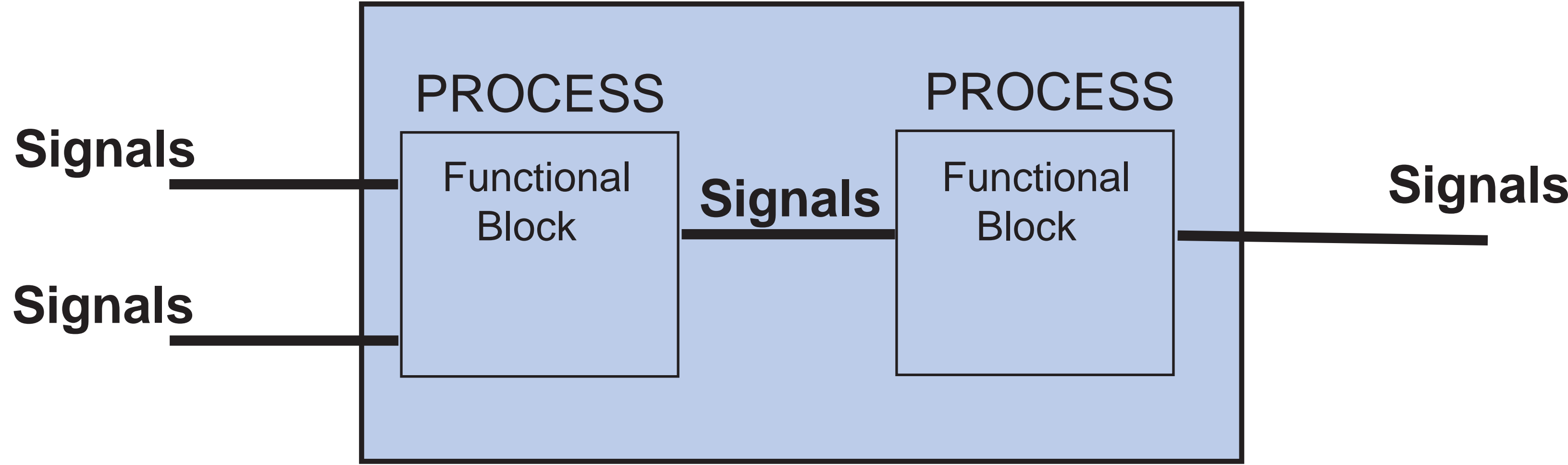
- Different values can be assigned to a signal at different times using *signal assignment statements*

- *A new value is not immediately assigned*

- Signals are used to connect concurrent elements (like wires)

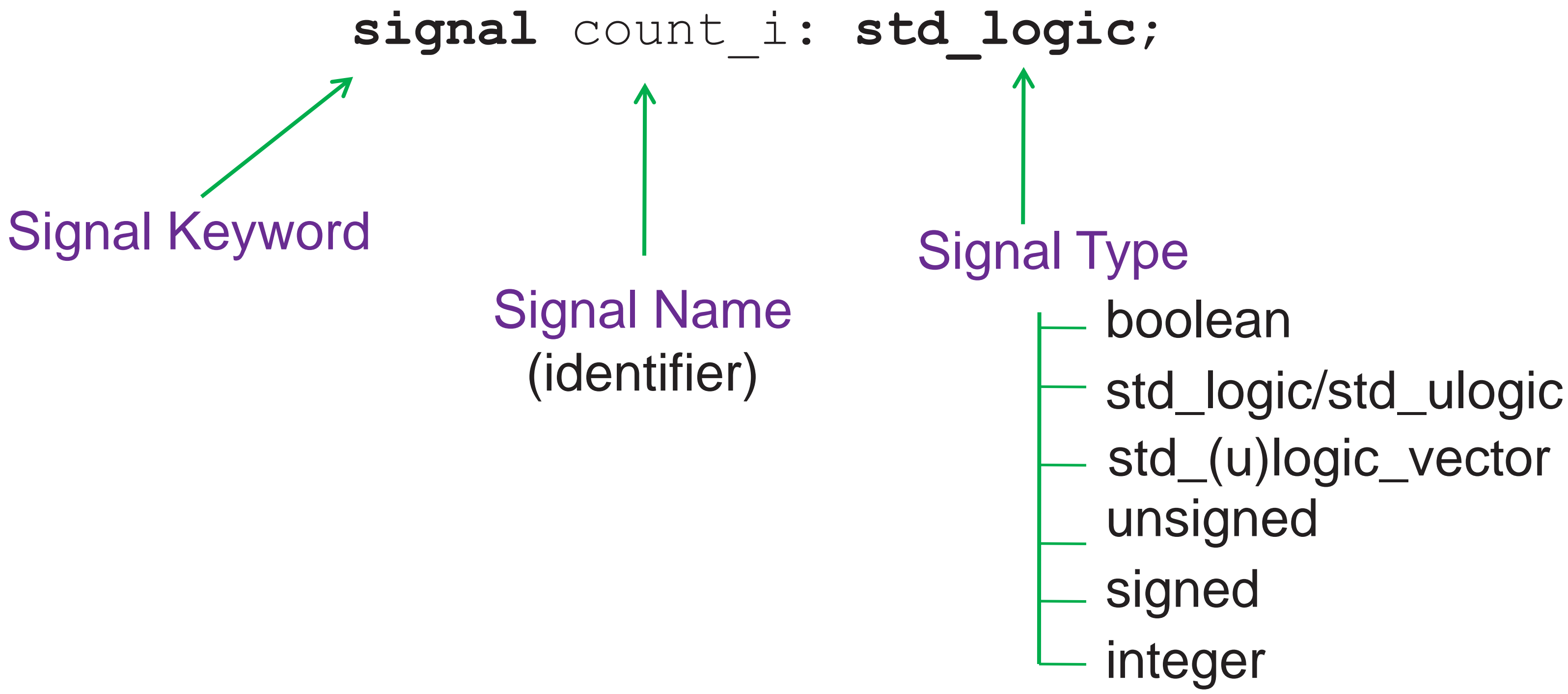
Signal Class

- Signals represent physical interconnect (wire) that communicate between processes as well as the I/O of the system



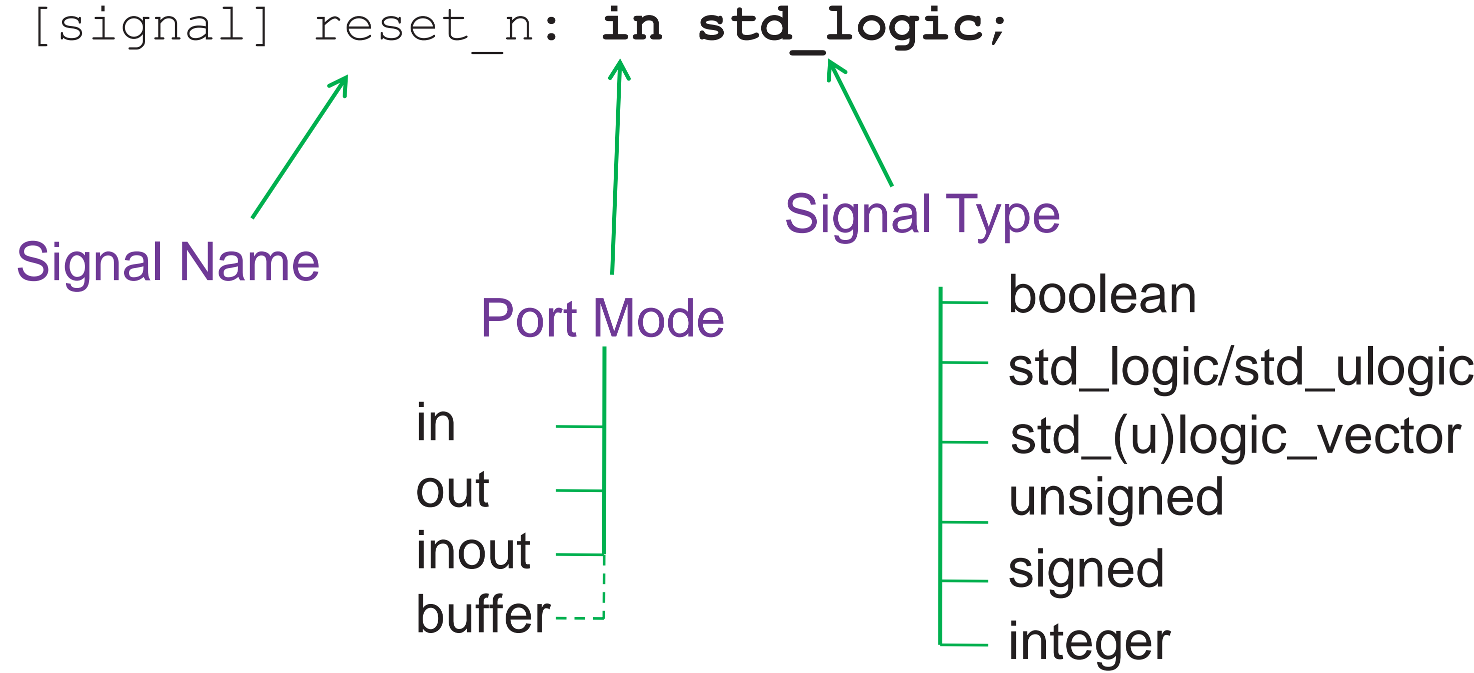
Signal Declaration in the Architecture

- Signal Declarations:
  - A signal is declared in the declarative part of an architecture, and it has two parts:



Signal Declarations–Entity Port Declarations

- Port declarations appear in the port section of an entity declaration
- Each port declaration is separated by semicolon from the others
- A port declaration has three parts:



Port Modes

- Mode ***in***: cannot be assigned a value to, so it can only be read and it appears on the RHS of the signal assignment
- Mode ***out***: can only be assigned to, so it can appear on the LHS of the signal assignment
- Mode ***inout***: can be assigned to and be read, so in can appear on either side of an assignment. It is intended to be used *only* for bidirectional kind of I/O ports
- Mode ***buffer***: can be assigned to and be read, so in can appear on either side of an assignment. However, do not use mode buffer !

Simple Signal Assignment

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

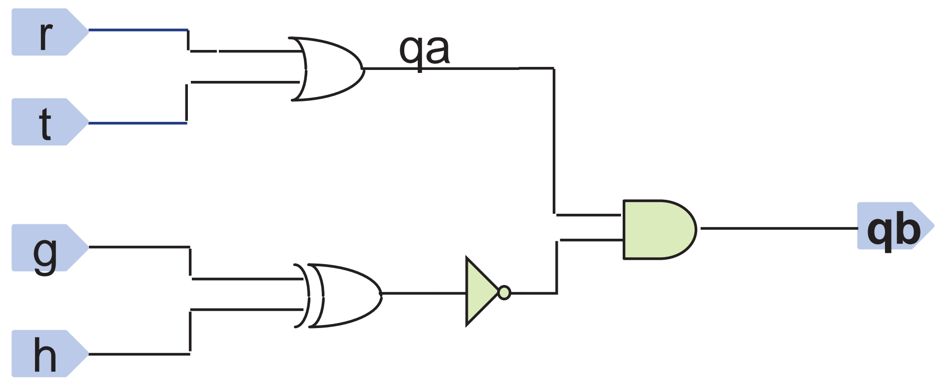
Left Hand Side (LHS)
Target Signal

Right Hand Side (RHS)
Source Signal(s)

LHS **Signal Type** ═══ RHS **Signal Type**

Signal Used as Interconnect

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY simp IS
   PORT (
      r, t, g, h : IN STD_LOGIC;
      qb             : OUT   STD_LOGIC
                 );
END ENTITY simp;

ARCHITECTURE logic OF simp IS
   SIGNAL qa : STD_LOGIC;
BEGIN

    qa <= r OR t;
    qb <= (qa AND NOT (g XOR h));

END ARCHITECTURE logic;
```



Signal declaration inside architecture

- **r, t, g, h**, and **qb** are signals (by default)
- **qa** is a buried signal and needs to be declared

Signals – Initial Values

- All signal have an initial value by the simulator when simulation begins (elaboration phase of the simulation):
  - It can be assigned in the signal declaration

    ```
    signal ini_cnt: std_logic_vector(3 downto 0):= "1111";
    ```

  - It will be given by default depending on the type of the signal:
    - It will be the first value of the type: for type bit is '0'; for type std_logic is 'U'

```
signal counter: std_logic_vector(7 downto 0);
signal flag: boolean;
constant bus_width: integer := 16;
signal full_fifo: bit;
signal addr: std_logic_vector(bus_width-1 downto 0);
signal data: std_logic_vector(bus_width*2-1 downto 0);
```

Signals – Initial Values

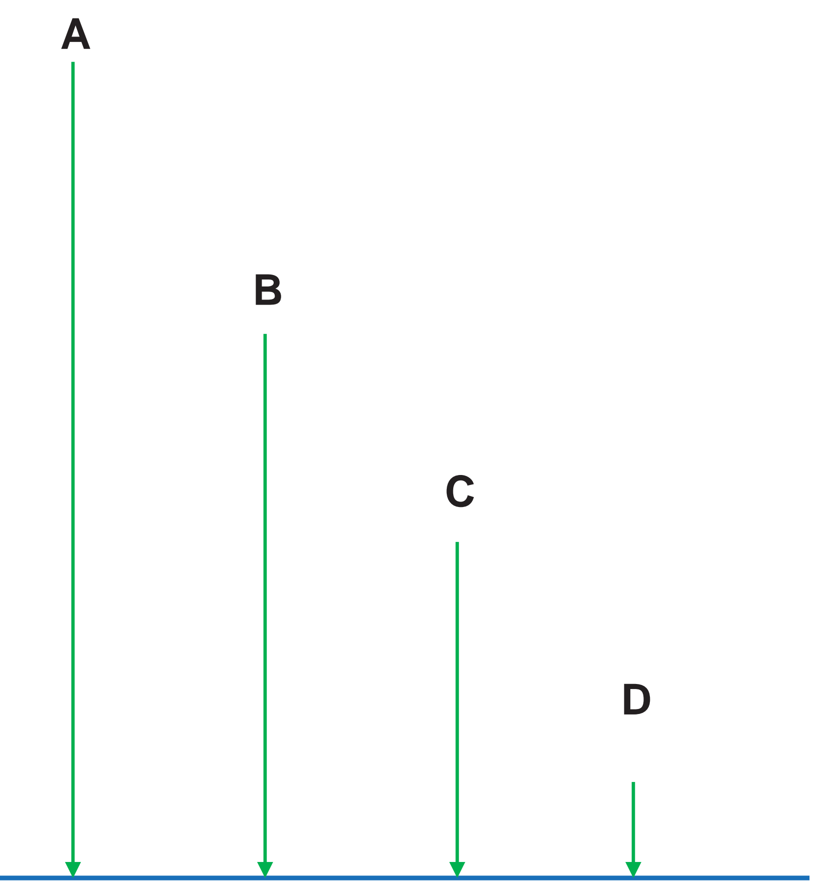
- For synthesis*, there is no hardware interpretation* of an intial value -> SYNHESIS IGNORES INTIAL VALUES

Missmatch between SIMULATION & HARDWARE !

Designer is responsable for setting the Hardware in a known intial state

Signal Declarations and Visibility

```
package my_packg
        signal A;
------------------------------
use my_packg.all;
entity test
        port (B ...
------------------------------
architecture example of test is
        signal C: ...
begin
...
my_process: process
        signal D: …
```



Assigning Values to Signals

```
signal temp : std_logic_vector (7 downto 0);
```

- Examples
  - All bits

    ```
    temp  <=  "10101010";
    temp  <= x"aa" ; -- Hexadecimal
    ```

    - VHDL also supports 'o' for octal and 'b' for binary
  - Bit-slicing

    ```
    temp (7 DOWNTO 4)  <=  "1010";
    ```
  - Single bit

    ```
    temp(7)  <=  '1';
    ```
  - Different values at different times

    ```
    rst <= x"00", x"AA" after 50 ns, x"FF" after 100 ns; -- just for simulation
    ```

- Use double-quotes (" ") to assign multi-bit values and single-quotes (' ') to assign single-bit values

Signals Resume

- Not immediately updated
- Declared within:
  - Architecture
  - Blocks
  - Packages
  - Implicit declaration in the entity declaration
- Visible for all the processes in the architecture
- In a combinatorial process *synthesize as combinatorial logic*
- In a sequential process *synthesize as registers*
- Used to communicate between processes and as I/O

Variable Class

- A Data Object of class variable can hold a simple value of a specific type

- Different values can be assigned to a variable at different times using variable assignment statement

- Variable takes a new value immediately

- Variables are used to hold intermediate values between sequential instructions (like variable in the conventional software languages)

- Variables are allowed in process, procedures and functions

Variable Class

- Variable declaration syntax:

```
variable <do_identifier> : <type> [:=initial value];

variable counter: integer range 0 to 10;
variable ASK: boolean;
```

- Variable assignment statement

```
<variable> := <expression>;

counter := 5;
ASK     := TRUE;
```

Variable Class

- Can be declared in an architecture only inside a process or in a subprogram
- A process declared variable is never reinitialized
- A sub-program declared variable is initialized for each call to the subprogram
- Value assignment has immediate effect

Variable Class

- The variable assignments have no time dimension associated with them. That is, the assignments take their value immediately

- *Variables has no direct analogue in hardware*

- Variable can only be used within sequential areas, within a process, in subprograms (functions and procedures) but not in concurrent statements in the architecture body

Example: Variable vs Signal

```
--====================--
-- example of
-- bad use of signal
-- ====================--

entity cmb_var is
   port (i0: in std_logic;
         i1: in std_logic;
          a: in std_logic;
         q: out std_logic);
end cmb_var;
```

```
architecture bad of cmb_var is
signal val:integer range 0 to 1;
begin
test: process(i0, i1, a)
begin
       if (a='0') then
          val <= 1;
       else
          val <= 0;
       end if;
       case val is
          when 0 =>
             q <= i0;
          when 1 =>
             q <= i1;
       end case;
end process test;
end bad;
```

--Example de MAL USO de SIGNALS
-- Referencia: ALTERA pg. 70

Example: Variable vs Signal

```vhdl
-- ---------------------- --
-- example of good use
-- of variables
-- ---------------------- --
entity cmb_var is
   port (i0: in std_logic;
         i1: in std_logic;
          a: in std_logic;
         q: out std_logic);
end cmb_var;
```

```vhdl
architecture good of cmb_var is
begin
test: process(i0, i1, a)
variable val:integer range 0 to 1;
begin
   if (a='0') then
        val := 1;
   else
        val := 0;
   end if;
   case val is
        when 0 => q <= i0;
        when 1 => q <= i1;
   end case;
end process;
end good;
```

Signals vs Variables

	SIGNALS	VARIABLES
Assign utility	**<=** Represent circuit interconnection	**:=** Represent local storage
Scope	Global scope, comunication between process	Local scope
Behavior	Update at the end of the process statement	Update immediately

Constant Class

- A Data Objects of class constant holds a single value of a specific type
- The value is assigned to the constant during its declaration and the value can not be changed
- Constant declaration:

```
constant <identifier>: type := value;
```

Constants - Examples

```
constant Bus_Width: integer := 16;
constant GNDLogic:  bit := '0';
constant AnchoBus:  integer := 16;
constant Message:   string := "End of simulation";
constant error_flag:boolean := true;
constant ClkPeriod: time:= 50 ns;
constant RiseTime:  time:= 5 ns;
```

Constant Examples

```
-- case 1
architecture ej_constant of ej1 is
        constant tpo_delay: time := 10 ns;
begin
        ....
end architecture ej_constant;
```

```
-- case 2
proc_ejemplo: process(A, B)
        constant t_hold: time:= 5 ns;
          begin
                ....
          end process;
```

Constant Examples

```
-- case 3
package my_package is
  constant mi_contador: integer:= 5;
  constant max_cuenta: std_logic_vector(3 downto 0):= "1111";
end package;
```

```
-- case 4
entity CPU is
   generic (Add_width:integer:= 15);-- implicit constant
   port(Input_CPU: in bit_vector(Add_width-1 downto 0);
        Output_CPU: out bit_vector(31 downto 0));
end CPU;
```

Literals

Values assigned to objects or used within expressions.
The value is not always explicit

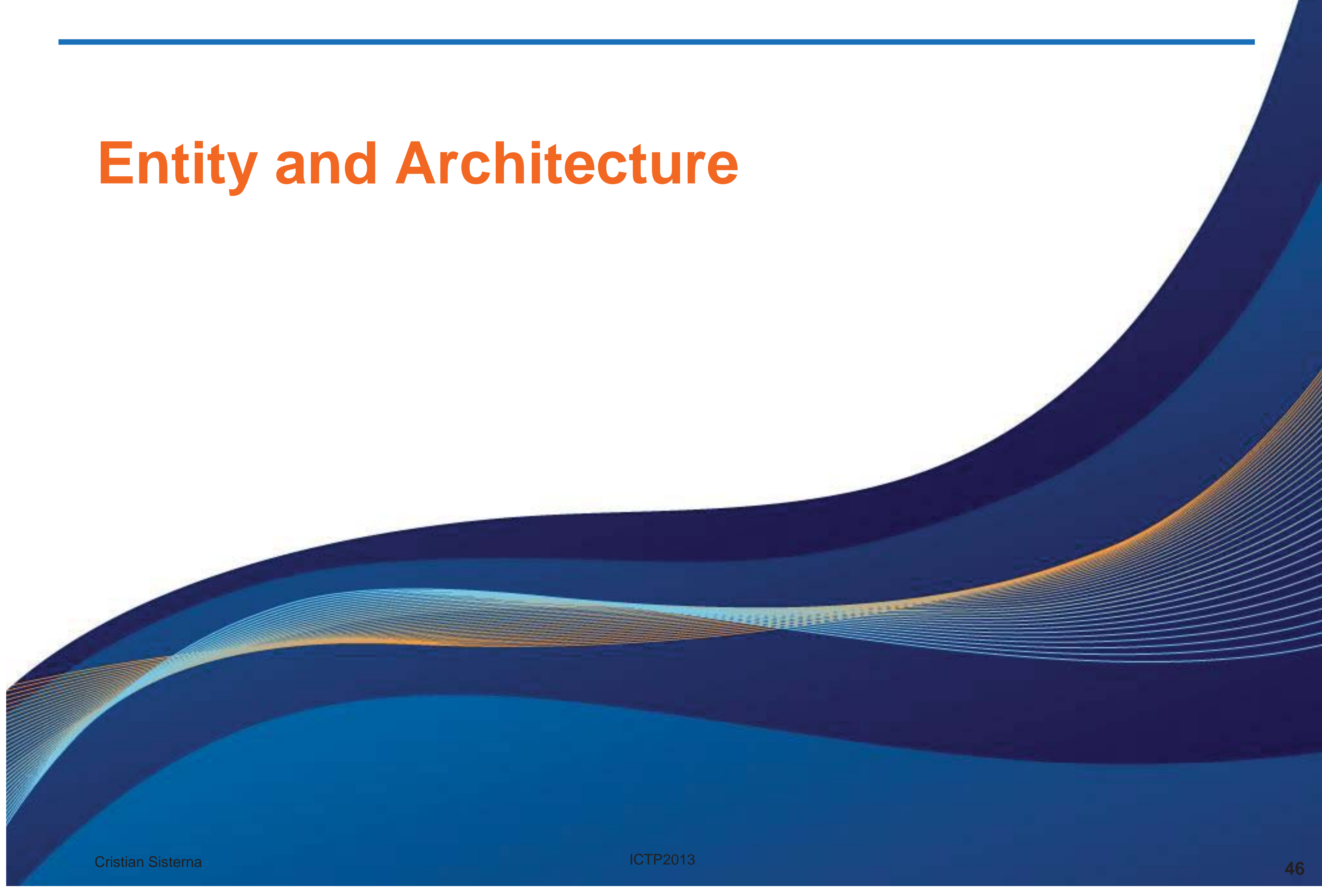
- Characters: An ASCII character enclosed in single quotation marks
  - '1', 'Z', 'A'
  - The character is valid depending on the type of the data object

- String: A sequence of characters enclosed in double quotation marks. A string may be assigned to an array of data objects (bit_vector, std_logic_vector, string)
  - "00100", "ZZZZZZZZ", "a string"

Literals

- Bit string: represent a sequence of ‘1’ and ‘0’ enclosed by double quotation marks. A base specifier can be used
  - B”10010110”   B”1001_0110”
  - X”FA”
- Numeric: Integer and real
  - 0 /= 0.0
- Based:
  - 2#100100001# = 145   16#FFCC# = 65484

Entity and Architecture



ENTITY Declaration

```
ENTITY <entity_name>  IS
     Generic declarations
     Port Declarations
END ENTITY <entity_name> ; (1076-1993 version)
```

ENTITY : Port Declarations

```
ENTITY  <entity_name>  IS
    GENERIC declarations
    PORT (
        SIGNAL clk        : IN    bit;
        --Note: SIGNAL is assumed and is not required
        q                 : OUT bit
    );
END ENTITY <entity_name> ;
```

- Structure : <class> object_name : <mode> <type> ;
  - <class> : what can be done to an object
  - object_name : identifier (name) used to refer to object
  - <mode> : directional
    - **IN** (input) **OUT** (output)
    - **INOUT** (bidirectional) **BUFFER** (output w/ internal feedback)
  - <Type> : what can be contained in the object (discussed later)

ENTITY : Generic Declaration

```
ENTITY  <entity_name> IS
    GENERIC (
        CONSTANT tmax_cnt    : integer = 324;
        -- Note CONSTANT is assumed and is not required
        default_value        : INTEGER := 1;
        cnt_dir              : STRING := "up"
            );
    PORT declarations
END ENTITY <entity_name> ; --( 1076-1993 version)
```

- Generic values can be overwritten during compilation
  - i.e. Passing in parameter information
- Generic must resolve to a constant during compilation

ARCHITECTURE

- Analogy- schematic
  - Describes the functionality and timing of a model
- Must be associated with an **ENTITY**
- **ENTITY** can have multiple architectures
- **ARCHITECTURE** statements execute concurrently (processes)
- **ARCHITECTURE** styles
  - Behavioral : how designs operate
    - RTL : designs are described in terms of registers
    - Functional : no timing
  - Structural : netlist
    - Gate/component level
  - Hybrid : mixture of the two styles
- End architecture with
  - **END ARCHITECTURE <architecture_name>;** -- VHDL '93 & later
  - **END ARCHITECTURE;** -- VHDL '93 & later
  - **END;** -- All VHDL versions

ARCHITECTURE

```
ARCHITECTURE <identifier> OF <entity_identifier> IS
--ARCHITECTURE declaration section (list does not include all)
    SIGNAL temp : INTEGER := 1;  -- signal declarations with optional default values
    CONSTANT load : boolean := true;  --constant declarations
    --Type declarations (discussed later)
    --Component declarations (discussed later)
    --Subprogram declarations (discussed later)
    --Subprogram body (discussed later)
    --Subtype declarations
    --Attribute declarations
    --Attribute specifications
BEGIN
   PROCESS statements
   Concurrent procedural calls
   Concurrent signal assignment
   Component instantiation statements
   Generate statements
END ARCHITECTURE <architecture_identifier>;
```

VHDL - Basic Modeling Structure

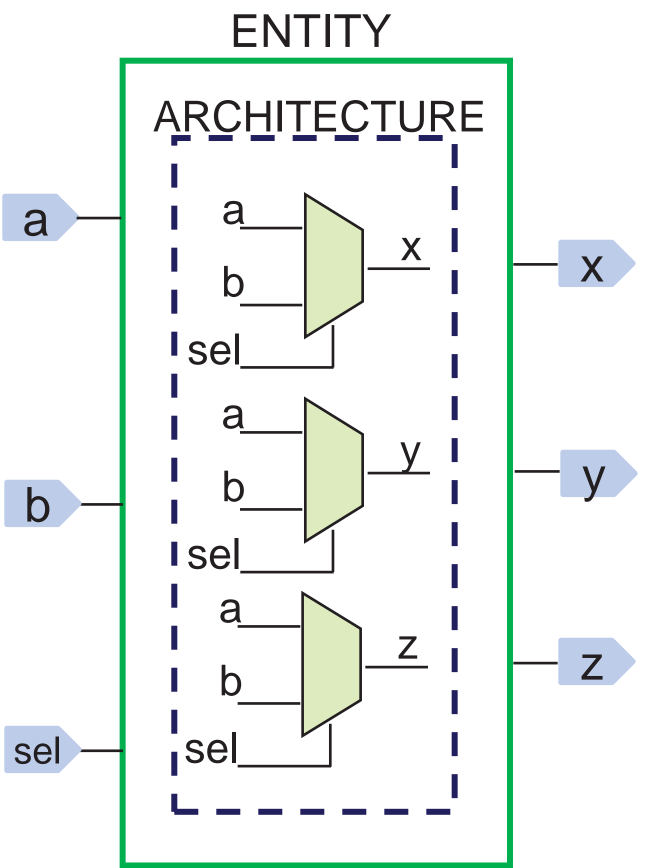
```
ENTITY entity_name IS
        generics
        port declarations
END ENTITY entity_name;
```

```
ARCHITECTURE arch_name OF entity_name IS
        internal signal declarations
        enumerated data type declarations
        component declarations
BEGIN
        signal assignment statements
        PROCESS statements
        component instantiations
END ARCHITECTURE arch_name;
```

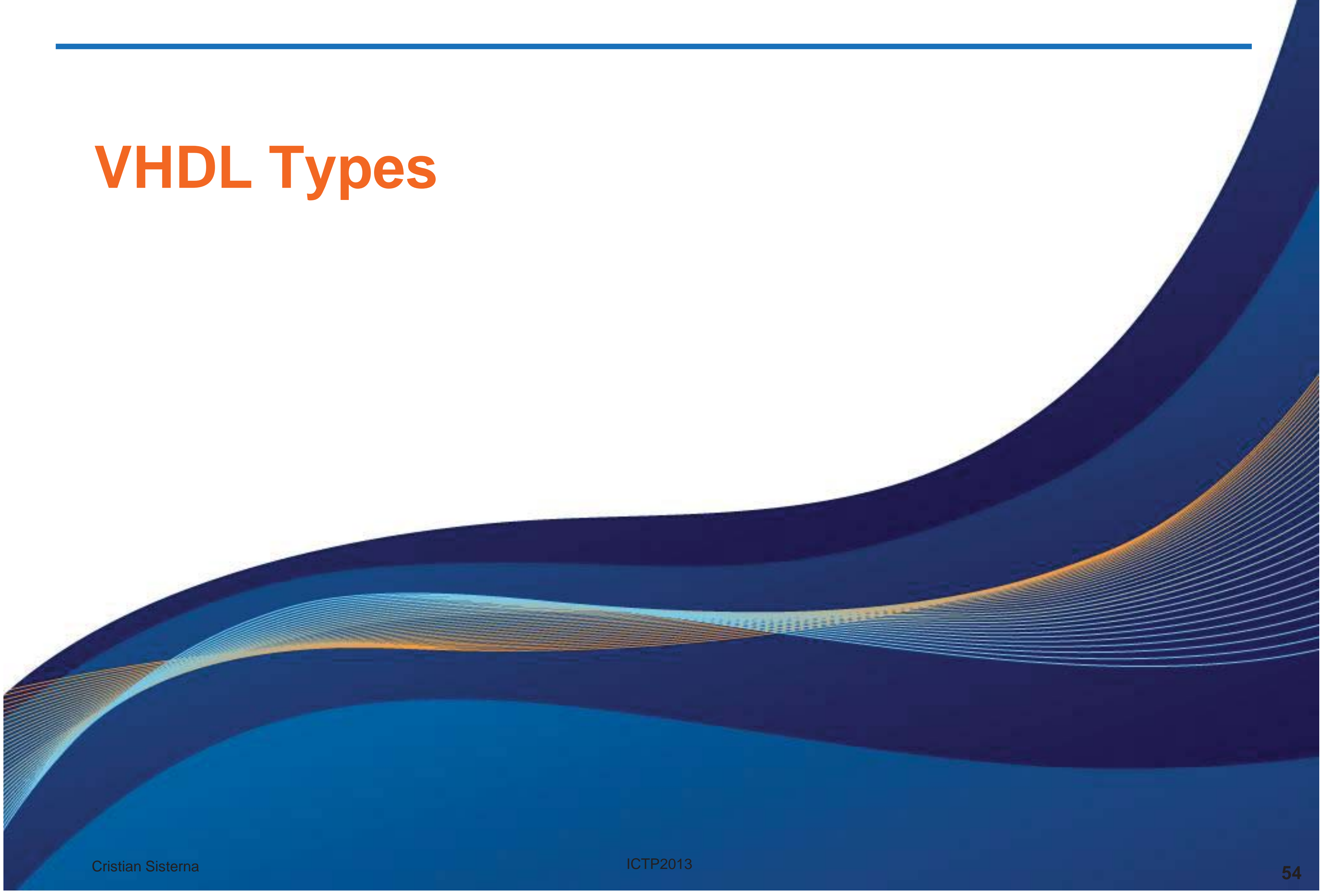
Putting It All Together

```
ENTITY cmpl_sig IS
    PORT (
        a, b, sel       : IN    BIT;
        x, y, z         : OUT BIT
    );
END ENTITY cmpl_sig;
```

```
ARCHITECTURE logic OF cmpl_sig IS
BEGIN
     -- simple signal assignment
    x <= (a AND NOT sel) OR (b AND sel);
    -- conditional signal assignment
    y <= a WHEN sel='0' ELSE
            b;
     -- selected signal assignment
    WITH sel SELECT
        z <= a WHEN '0',
                b WHEN '1',
               '0' WHEN OTHERS;
END ARCHITECTURE logic;
```



VHDL Types



VHDL Types Defined in STANDARD Package

- Type **BIT**
  - 2 logic value system ('0', '1')

    ```
    SIGNAL a_temp : BIT;
    ```
  - Bit_vector array of bits

    ```
    SIGNAL temp : BIT_VECTOR (3 DOWNTO 0);
    SIGNAL temp : BIT_VECTOR (0 TO 3);
    ```
- Type **BOOLEAN**
  - (False, true)
- Type **INTEGER**
  - Positive and negative values in decimal

    ```
    SIGNAL int_tmp   : INTEGER; -- 32-bit number
    SIGNAL int_tmp1 : INTEGER RANGE 0 TO 255; --8 bit number
    ```

Other Types Defined in Standard Package

- Type **NATURAL**
  - Integer with range 0 to 2^{32}
- Type **POSITIVE**
  - Integer with range 1 to 2^{32}
- Type **CHARACTER**
  - ASCII characters
- Type **STRING**
  - Array of characters
- Type **TIME**
  - Value includes units of time (e.g. ps, us, ns, ms, sec, min, hr)
- Type **REAL**
  - Double-precision floating point numbers

Types Defined in STD_LOGIC_1164 Package

- Type **STD_LOGIC**
  - 9 logic value system ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-')
    - '1': Logic high
    - '0': Logic low
    - 'X: Unknown
    - 'Z': (not 'z') Tri-state
    - '-': Don't Care
    - 'U': Undefined
    - 'H': Weak logic high
    - 'L': Weak logic low
    - 'W': Weak unknown
  - Resolved type: supports signals with multiple drivers
    - Driving multiple values onto same signal results in known value
- Type **STD_ULOGIC**
  - Same 9 value system as STD_LOGIC
  - Unresolved type: Does not support multiple signal drivers
    - Driving multiple values onto same signal results in error

Types

- VHDL has built-in data types to model hardware (e.g. BIT, BOOLEAN, STD_LOGIC)
- VHDL also allows creation of brand new types for declaring objects (i.e. constants, signals, variables)

- Subtype
- Enumerated Data Type
- Array

Subtype

- A constrained type
- Synthesizable if base type is synthesizable
- Use to make code more readable and flexible
  - Place in package to use throughout design

```
ARCHITECTURE logic OF subtype_test IS

    SUBTYPE word IS std_logic_vector (31 DOWNTO 0);
    SIGNAL mem_read, mem_write : word;

    SUBTYPE dec_count IS INTEGER RANGE 0 TO 9;
    SIGNAL ones, tens : dec_count;

BEGIN
```

Enumerated Data Type

- Allows user to create data type ***name*** and ***values***
  - Must create constant, signal or variable of that type to use
- Used in
  - Making code more readable
  - Finite state machines
- Enumerated type declaration

```
TYPE <your_data_type> IS
      (data type items or values separated by commas);
```

```
TYPE enum IS (idle, fill, heat_w, wash, drain);
SIGNAL fsm_st : enum;
      …
drain_led <= '1' WHEN fsm_st = drain ELSE '0';
```

Array

- Creates multi-dimensional data type for storing values
  - Must create constant, signal or variable of that type
- Used to create memories and store simulation vectors
- VHDL 2008 allows unconstrained elements
- Array type Declaration

array depth

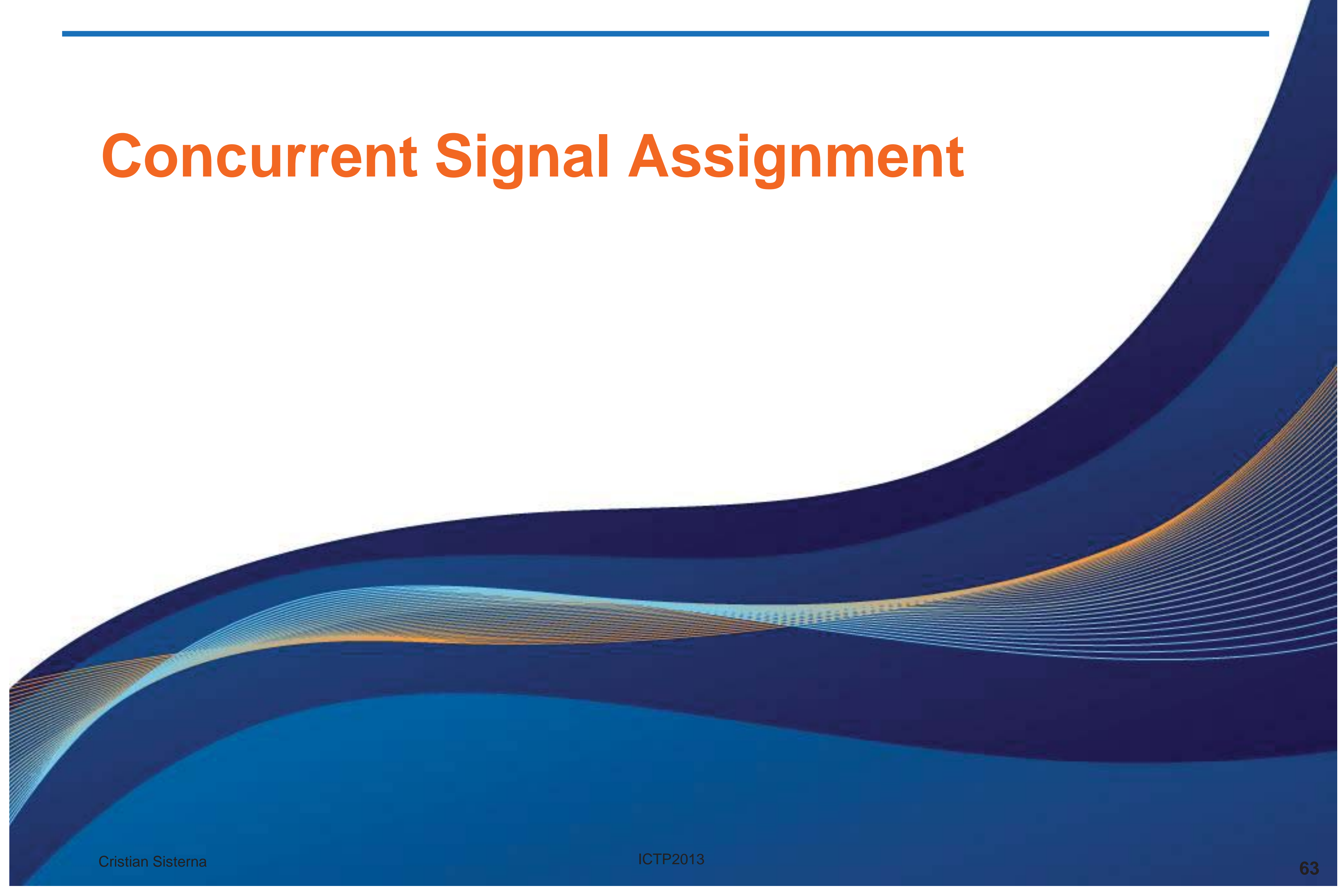
```
TYPE <array_type_name> IS ARRAY (<integer_range>) OF
     <data_type>;
```

what can be stored in each array address

Array Example

```
ARCHITECTURE logic OF my_memory IS
    -- Creates new array data type named mem which has 64
    --   address locations each 8 bits wide
    TYPE mem IS ARRAY (0 to 63) OF std_logic_vector (7 DOWNTO 0);

    -- Creates 2 - 64x8-bit array to use in design
    SIGNAL mem_64x8_a, mem_64x8_b : mem;

BEGIN
    ...
    mem_64x8_a(12) <= x“AF”;
    mem_64x8_b(50) <= “11110000”;
    …
END ARCHITECTURE logic;
```

Concurrent Signal Assignment



Concurrent Signal Assignments

- Used to assign values to signals using expressions
- Represent ***implied*** processes that execute in parallel
  - Process sensitive to anything on read (right) side of assignment
- Three types
  - Simple signal assignment
  - Conditional signal assignment
  - Selected signal assignment

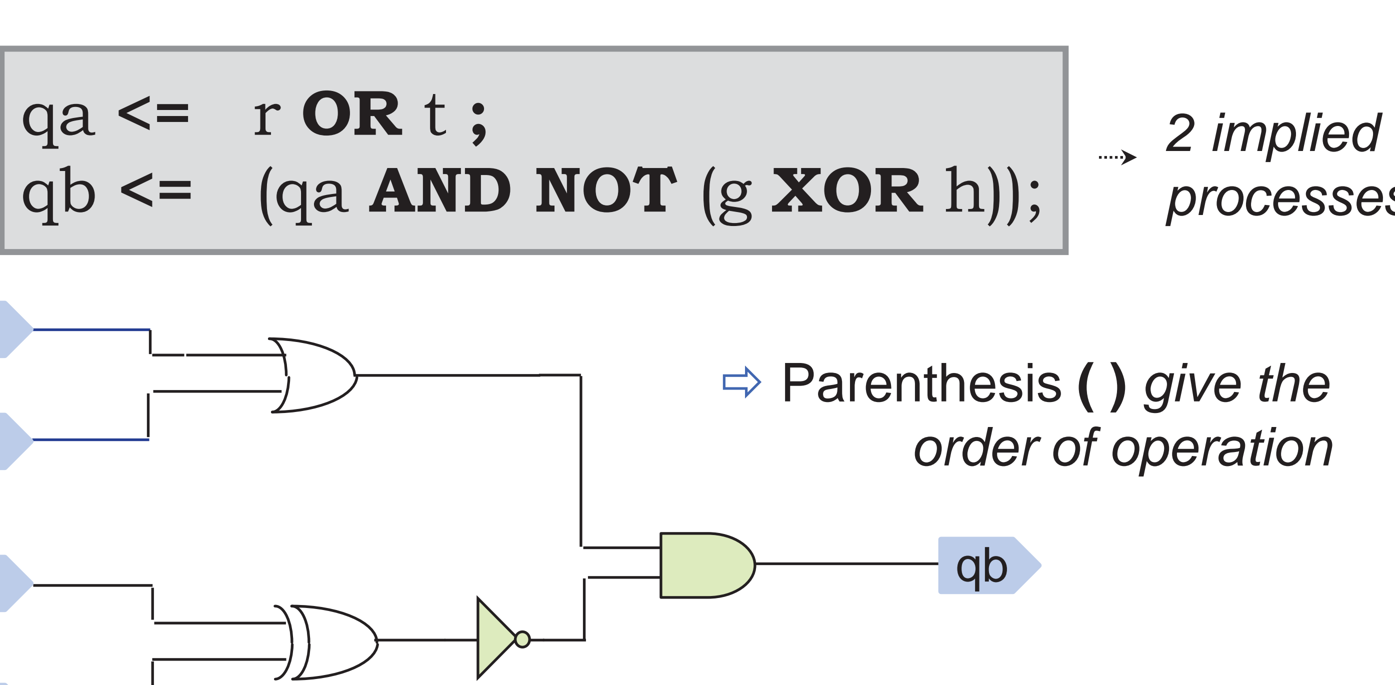
Simple Signal Assignment Statement

- Format: *<signal_name>* **<=** *<expression>***;**

- Example:

```
qa <=  r OR t ;
qb <=  (qa AND NOT (g XOR h));
```

→ *2 implied processes*



- Expressions use VHDL operators to describe behavior

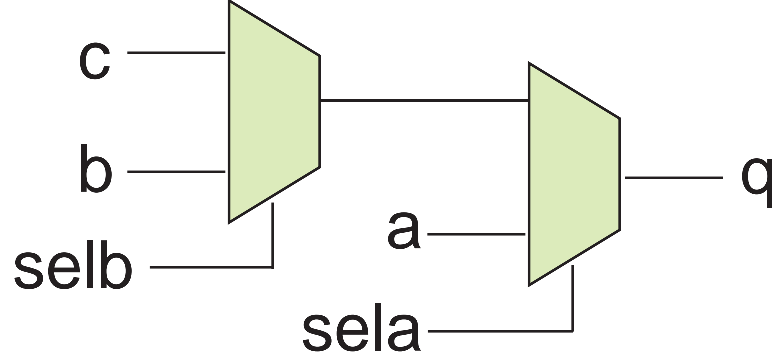
Conditional Signal Assignment Statement

- Format:

```
<signal_name>  <=  <signal/value> WHEN <condition_1> ELSE
                   <signal/value> WHEN <condition_2> ELSE
                                  ...
                   <signal/value> WHEN <condition_n> ELSE
                   <signal/value>;
```

- Example:

```
q <=  a WHEN sela = '1' ELSE
      b WHEN selb = '1' ELSE
      c;
```



Selected Signal Assignments

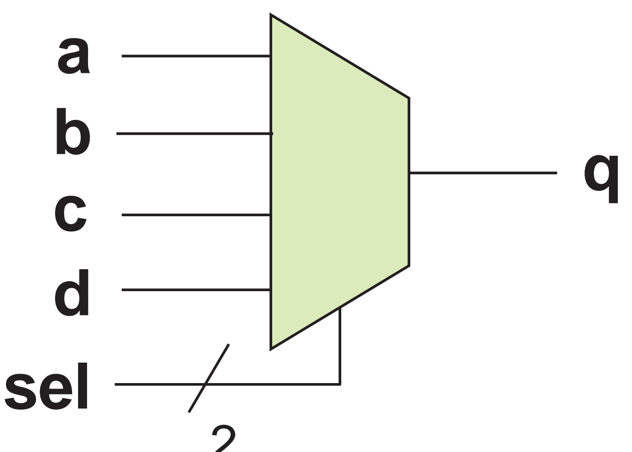
- Format:

```
WITH <expression> SELECT
    <signal_name> <=<signal/value> WHEN <condition_1>,
                    <signal/value> WHEN <condition_2>,
                    ...
                    <signal/value> WHEN OTHERS;
```

- Example:

```
WITH sel SELECT
    q <=   a WHEN "00",
           b WHEN "01",
           c WHEN "10",
           d WHEN OTHERS;
```

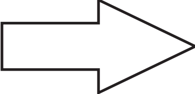
Implied process



Selected Signal Assignments

- **All** possible conditions must be considered
- **WHEN OTHERS** clause evaluates all other possible conditions that are not specifically stated

See next slide



Selected Signal Assignment

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY cmpl_sig IS
    PORT (
        a, b, sel : IN STD_LOGIC;
        z         : OUT STD_LOGIC
    );
END ENTITY cmpl_sig;

ARCHITECTURE logic OF cmpl_sig IS
BEGIN
      -- Selected signal assignment
    WITH sel SELECT
        z <=  a WHEN '0',
              b WHEN '1',
              '0' WHEN OTHERS;
END ARCHITECTURE logic;
```

sel is of **STD_LOGIC** data type

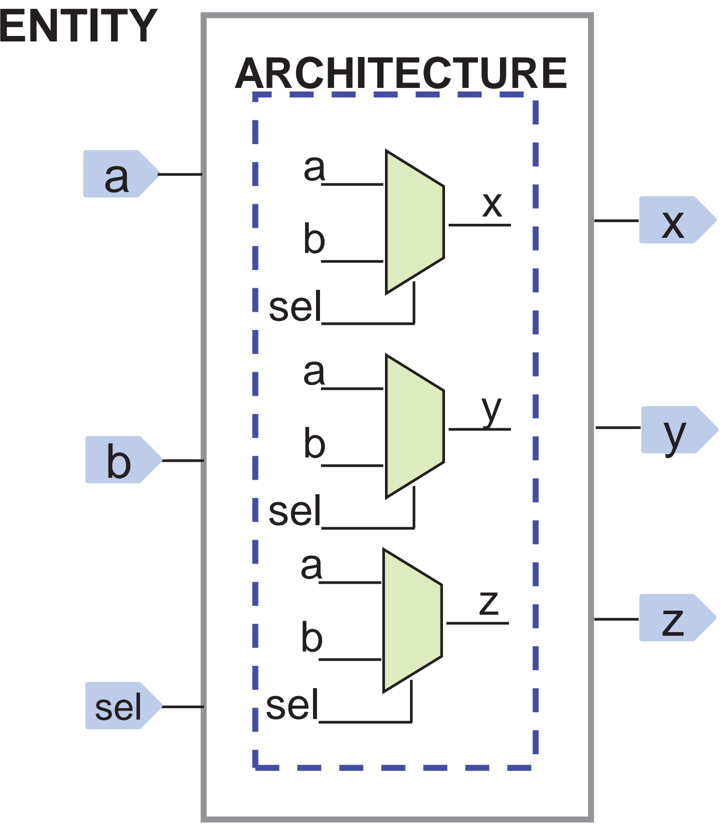
- What are the values for a **STD_LOGIC** data type
- Answer: **{'0','1','X','Z' … }**

- Therefore, is the **WHEN OTHERS** clause necessary?
- Answer: **YES**

VHDL Model - Concurrent Signal Assignments

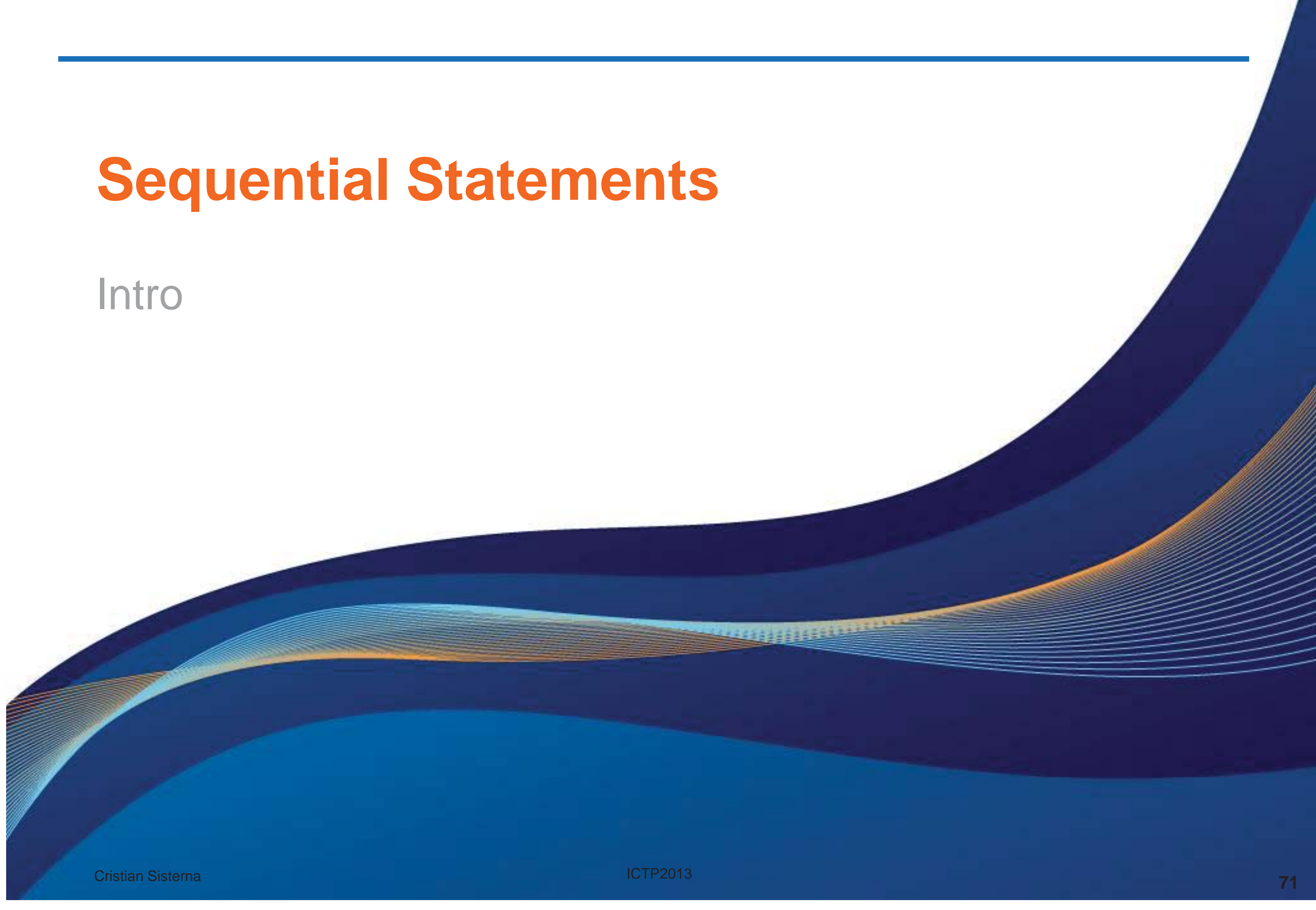
```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY cmpl_sig is
     PORT (
          a, b, sel : IN STD_LOGIC;
          x, y, z : OUT STD_LOGIC
     );
END ENTITY cmpl_sig;

ARCHITECTURE logic OF cmpl_sig IS
BEGIN
     -- Conditional signal assignment
     y <= a WHEN sel='0' ELSE
            b;
     -- Selected signal assignment
     WITH sel SELECT
          z <=  a WHEN '0',
                b WHEN '1',
                'X' WHEN OTHERS;
   -- Simple signal assignment
     x <= (a AND NOT sel) OR (b AND sel);

END ARCHITECTURE logic;
```

- The signal assignments execute in parallel, and therefore the order we list the statements should not affect the outcome



Sequential Statements

Intro



Sequential Statements

- Allow to describe the behavior of a circuit as a sequence of related events

- Can be used to model, simulate and synthesize:
  - Combinational logic circuits
  - Sequential logic circuits

- They are used inside of:
  - Processes
  - Functions
  - Procedures

The order or sequence of sequential statements is very important

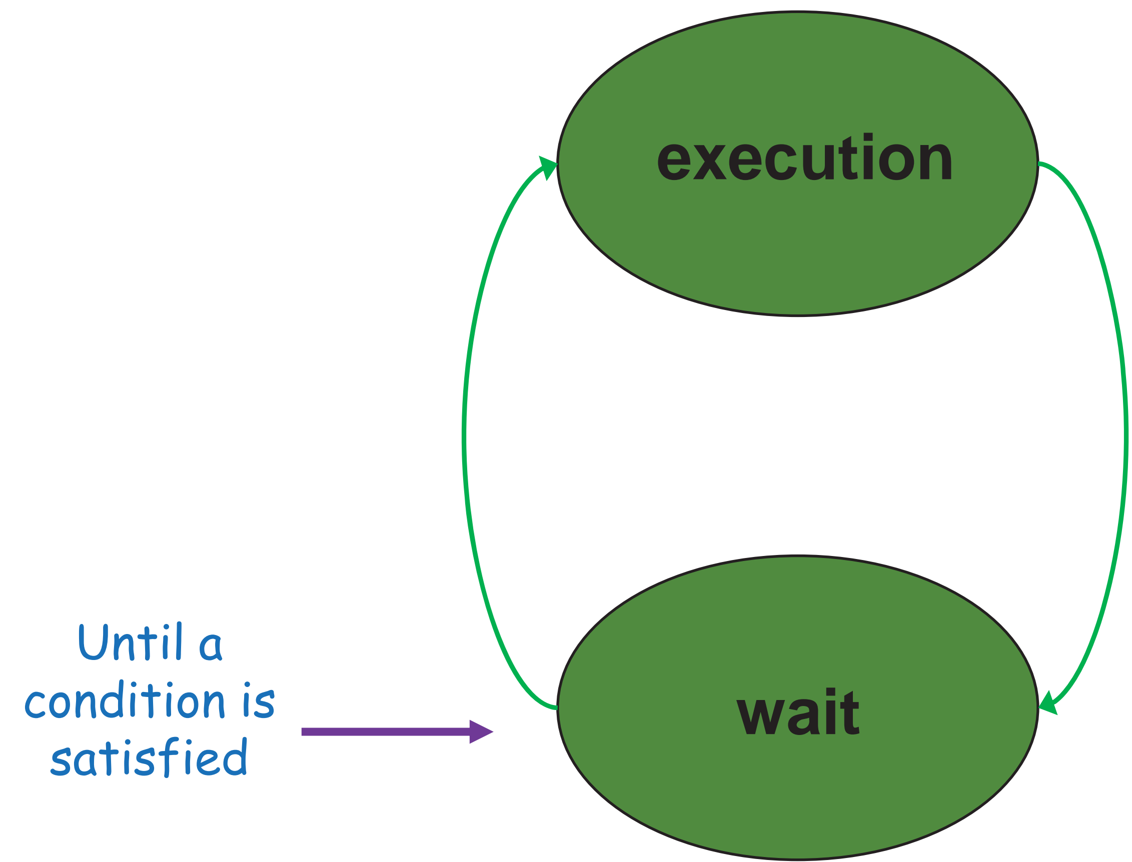
Process Statement

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*. That is, a process is executed in just *one delta time*
- From the traditional programming view, it is an infinite loop
- Multiple processes can be executed in parallel

Anatomy of a Process

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



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

process - Syntax

```
process sensitivity_list
            [declarations;]
begin
     sequential_statements;
end process;
```

2

```
process
        [declarations;]
begin
   sequential_statements;
   wait on sensitivity_list;
end process;
```

process 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
        return_statement
         [wait on sensitivity_list]
end process [process_label];
```

Sequential statements

Parts of a Process

- `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 Behavior 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* or in a *wait* statement**

Signal - *process*

Signals are the interface between VHDL's concurrent domain and the sequential domain within 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

Clocked `process`

- Clocked processes lead to all the *signals* assigned inside the process resulting in a flip-flop

- Variables can also give a flip-flop in a clocked process. *If a variable is read before it is assigned a value*, it will result in a flip-flop for the variable (bad practice!)

- If a signal is not assigned a value in a clocked process, the signal will retain the old value

- A clocked process can result in combinational logic besides the flip-flop(s). All logic caused by a signal assignment in a clocked process will end up on the "left" of the flip-flop (before the flip-flop's input)

Clocked *process*

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

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

Clocked *process*

```vhdl
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_srst: process (clk)
  begin
        if (rising_edge(clk)) then
            if (rst='1') then
               q <= '0';
            else
               q <= d;
            end if;
        end if;
 end process ff_d_srst;
end rtl;
```

Clocked *process*

```
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_arst: process (clk, rst)
  begin
        if (rst='1') then
            q <= '0';
        else if (rising_edge (clk)) then
            q <= d;
        end if;
  end process ff_d_arst;
end rtl;
```

Clocked *process*

```
architecture rtl of example is
signal q: std_logic;
begin
FD1_B1: process (clk, rst)
   begin
        if (rst='1') then
                q <= '0';
        elsif (clk'event and clk='1') then
                if (en='1') then
                        q <= …;-- some boolean expression(B1)
                end if;
        end if;
   end process;
   q1 <= q and .....          -- some boolean expression(B2)
end rtl;
```

Clocked *process* – 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(cnt_w-1 downto 0));
end counter_ud;
```

Clocked *process* – Counter

```
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;
```

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*

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

Combinational vs Clocked *process*

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

```
. . . .
architecture rtl of reg_ex is
begin
 ex_r: process (clk)
   begin
        if (rising_edge(clk)) then
            z <= a and b;
        end if;
end process ex_r;
end rtl;
```

Sequential Statement

if-then-elsif-end if

if-then-elsif-else-end if

Syntax

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

if-then-elsif-else-end if

- An ***if*** statement can have one or more branches, controlled by one or more conditions.
- There can be any number of elsif branches to an ***if*** statement.
- There may be only one ***else*** branch to the ***if*** statement, and if it's present it must be the last branch. It can also be omitted.
- Each branch of an ***if*** statement can contain any number of statements, it is not limited to a single statement.
- An ***if*** statement must always be terminated with an ***end if***

if-then-elsif-else-end if

Combinational process example:

```
entity if_example_1 is
  port(
         a,b: in std_logic_vector(7 downto 0);
         z  : out std_logic);
end entity;
architecture rtl of if_example_1 is
begin
 if_ex: process (a,b)
   begin
         if (a = b) then
                z <= '1';
         else
                z <= '0';
         end if;
 end process if_ex;
end rtl;
```

if-then-elsif-else-end if

Assigning values by default :

```
entity if_example_2 is
  port(
        a,b: in std_logic_vector(7 downto 0);
        z  : out std_logic);
end entity;
architecture rtl of if_example_2 is
begin
 if_ex2: process (a,b)
   begin
        z <= '0';
        if (a = b) then
            z <= '1';
        end if;
 end process if_ex2;
end rtl;
```

if-then-elsif-else-end if

```
entity if_example_3 is
  port(
        a,b,c,sel1,sel2: in std_logic;
        z              : out std_logic);
end entity;
architecture rtl of if_example_3 is
begin
 if_ex3: process (a,b,c,sel1,sel2)
   begin
        if (sel1 = '1') then
            z <= a;
        elsif (sel2 = '1') then
            z <= b;
        else
            z <= c;
        end if;
 end process if_ex3;
end rtl;
```

if-then-elsif-else-end if

```
library ieee;
use ieee.std_logic_1163.all;

entity if_example_9 is
  port(
         sel1,a1,b1: in std_logic;
          sel2,a2,b2: in std_logic;
          sel3,a3,b3: in std_logic;
         y1,y2,y3   : out std_logic);
end entity;

architecture rtl of if_example_9 is
begin
   y1 <= a1 when sel1='1' else b1;
```

if-then-elsif-else-end if

```
 if_ex9_1: process (sel2,a2,b2)
  begin
       y2 <= b2;
       if (sel2 = '1') then
            y2 <= a2;
       end if;
 end process if_ex9_1;
if_ex9_2: process (sel3,a3,b3)
       if (sel3 = '1') then
            y3 <= a3;
       else
            y3 <= b3;
       end if;
 end process if_ex9_2;
end rtl;
```

if-then-elsif-else-end if

```
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";
        elsif (a = "010") then z <= "00000100";
        . . .
        elsif (a = "110") then z <= "010000000";
        else                   z <= "10000000";
        end if;
 end process if_dec_ex;
end rtl;
```

if-then-elsif-else-end if

- The conditions in the successive branches of an ***if*** statement are evaluated independently

- There can be any number of conditions, each of which will be independent of the others

- Due to the structure of the if statement, the earlier conditions are tested first => **There is a priority**

if-then-elsif-else-end if

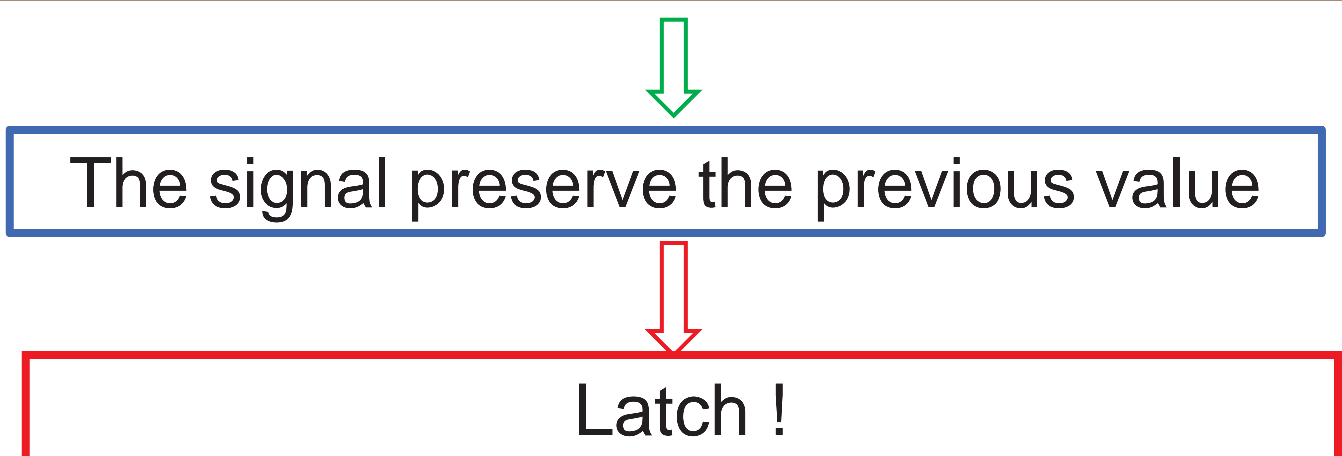
- In a multi-branch ***if*** statements, each condition can be, and will normally be, dependent on different signals and variables.

- A ***case*** statement should be used when every branch is dependent on the same signal or variable.

- It's important to remember the prioritization of the conditions to avoid redundant tests

if-then-elsif-else-end if

- If the target signal is part of a **combinatorial circuit**, it has to gets a value under all possible conditions of the if statement



if-then-elsif-else-end if

Two ways of avoiding generating unwanted latches

- Make sure that every signal that gets a value in an ***if*** statement also get assigned on every branch of the ***if*** and in the ***else*** part

- Initialize the signals in the if statement in an unconditional assignment before the ***if***

if-then-elsif-else-end if

Clocked process example:

```
entity if_example_4 is
  port(
        d, clk: in  std_logic;
        q     : out std_logic);
end entity;
architecture rtl of if_example_4 is
begin
 if_ex4: process (clk)
   begin
        if (clk'event and clk = '1') then
             q <= d;
        end if;
 end process if_ex4;
end rtl;
```

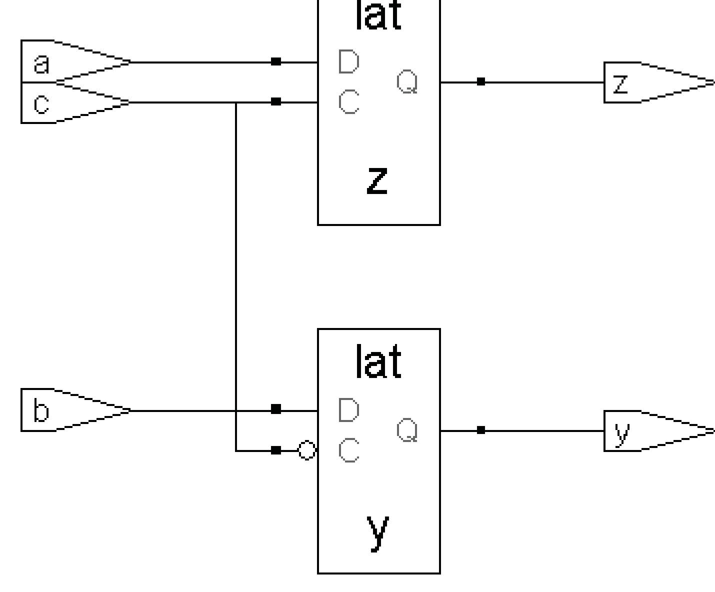
if-then-elsif-else-end if

Clocked process example:

```
entity if_example_5 is
  port(
       d, clk: in std_logic;
       q     : out std_logic);
end entity;
architecture rtl of if_example_5 is
begin
 if_ex5: process (clk)
  begin
       if (clk'event and clk = '1') then
            q <= d;
       else
            q <= d;
       end if;
 end process if_ex5;
end rtl;
```

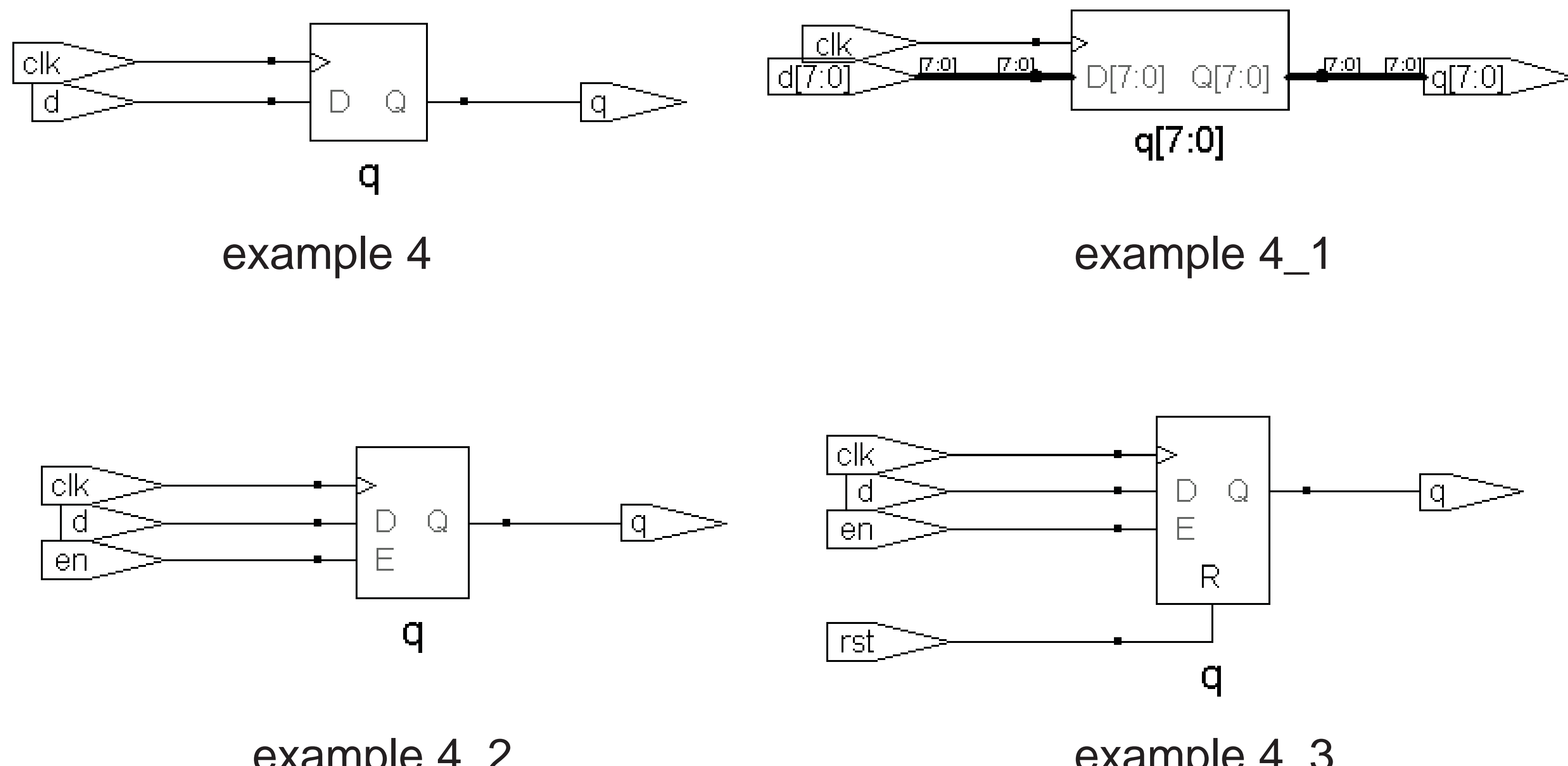
if-then-elsif-else-end if

```
architecture rtl of if_expl_6 is
begin
 if_ex6: process (a,b,c)
  begin
      if (c = '1') then
          z <= a;
      else
          y <= b;
      end if;
 end process if_ex6;
end rtl;
```



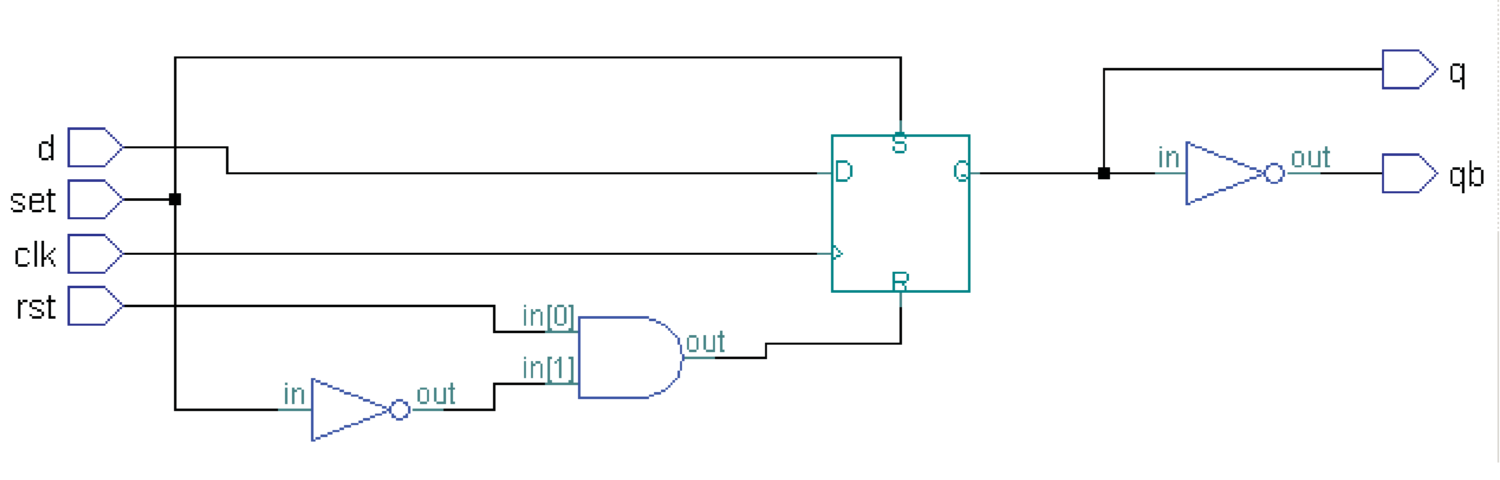
if-then-elsif-else-end if

Describe in VHDL the following circuits:



if-then-elsif-else-end if

Describe in VHDL the following circuit:
Case A – set and rst asynchronous
Case B - set and rst synchronous

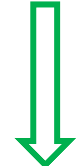


example 4_4

if-then-elsif-else-end if

Use of variables within *if* statements

- The same rules we have seen for signals, apply for variables. However, unlike signals, the reading and writing of variables in the same process will result in feedback only if the read occurs earlier in the process then the write



Variables get registered when there is a feedback of a previous variable value

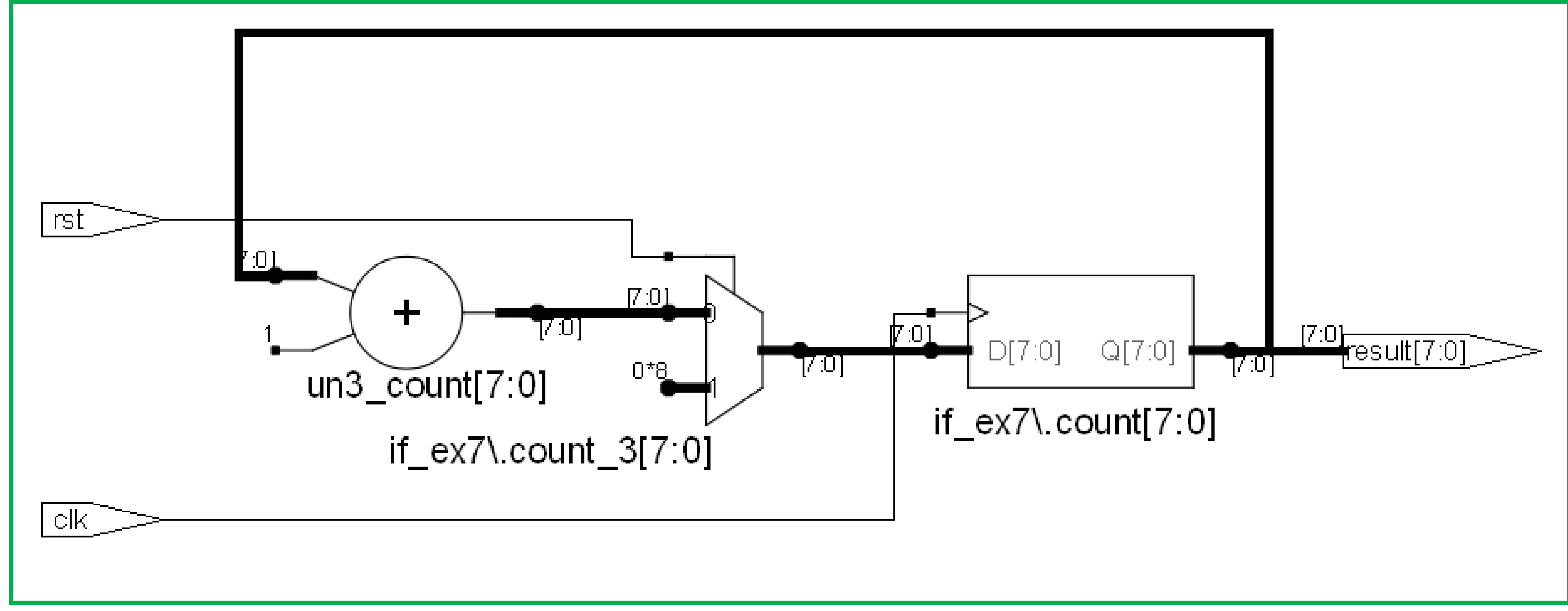
if-then-elsif-else-end if

Use of variables within *if* statements, example:

```
architecture rtl of if_expl_7 is
begin
 if_ex7: process (clk)
 variable count: unsigned(7 downto 0);
  begin
      if (rising_edge (clk)) then
         if(rst = '1') then
           count := (others => '0');
         else
           count := count + 1;
         end if;
      end if;
      result <= count;
 end process if_ex7;
end rtl;
```

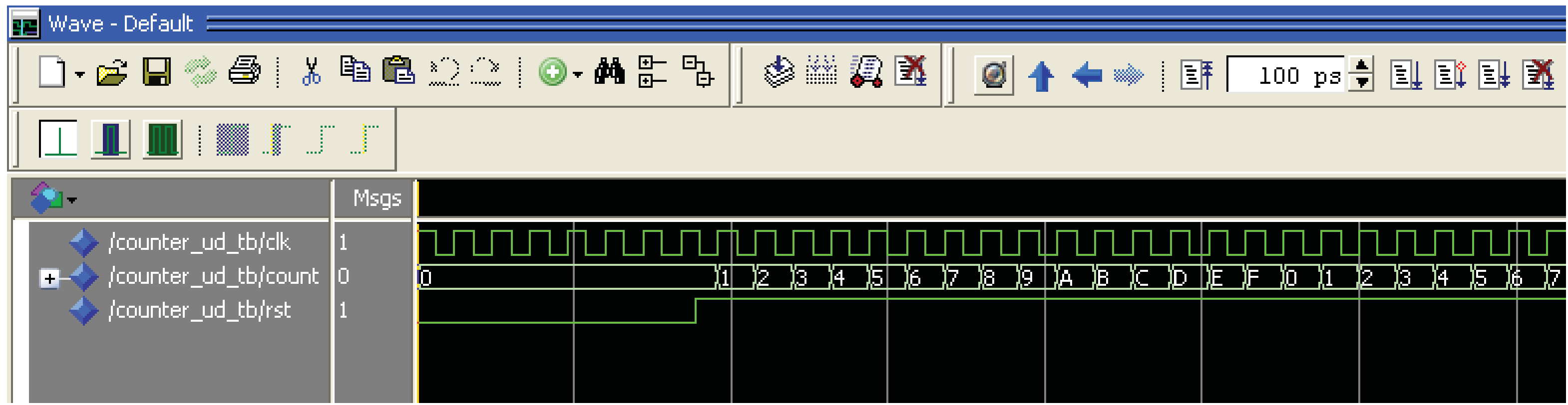
if-then-elsif-else-end if

Synthesis result



Example Counter – Using Integer

```
--------------------------------------------------
-- Architecture Body
--------------------------------------------------
begin

  count_proc: process(clk, rst)
  variable count_i: integer range 0 to 16;
  begin
    if(rst='0') then
      count_i := 0;
    elsif(rising_edge(clk)) then
        count_i := count_i + 1;
        if (count_i = 16) then
           count_i := 0;
        end if;
    end if;

  count <= std_logic_vector(to_unsigned(count_i,4));
  end process count_proc;
```



Example Counter – Using Usigned

F:/Actelprj/counter_ud/src/counter_ud_unsigned.vhd

```
Ln#
89   ----------------------------------------------
90    signal count_i: unsigned(3 downto 0);
91
92   ----------------------------------------------
93   -- Architecture Body
94   ----------------------------------------------
95   begin
96
97     count_proc: process(clk, rst)
98     begin
99       if(rst='0') then
100        count_i <= (others => '0');
101      elsif(rising_edge(clk)) then
102        if(up_dw = '1') then -- up
103          count_i <= count_i + 1;
104        else
105          count_i <= count_i - 1;
106        end if;
107      end if;
108    end process count_proc;
109
110    count <= std_logic_vector(count_i);
111
112  end architecture counter_ud_beh;
```

Wave

Messages	
/counter_ud_tb/clk	U
/counter_ud_tb/count	X
/counter_ud_tb/rst	U
/counter_ud_tb/up_dw	U

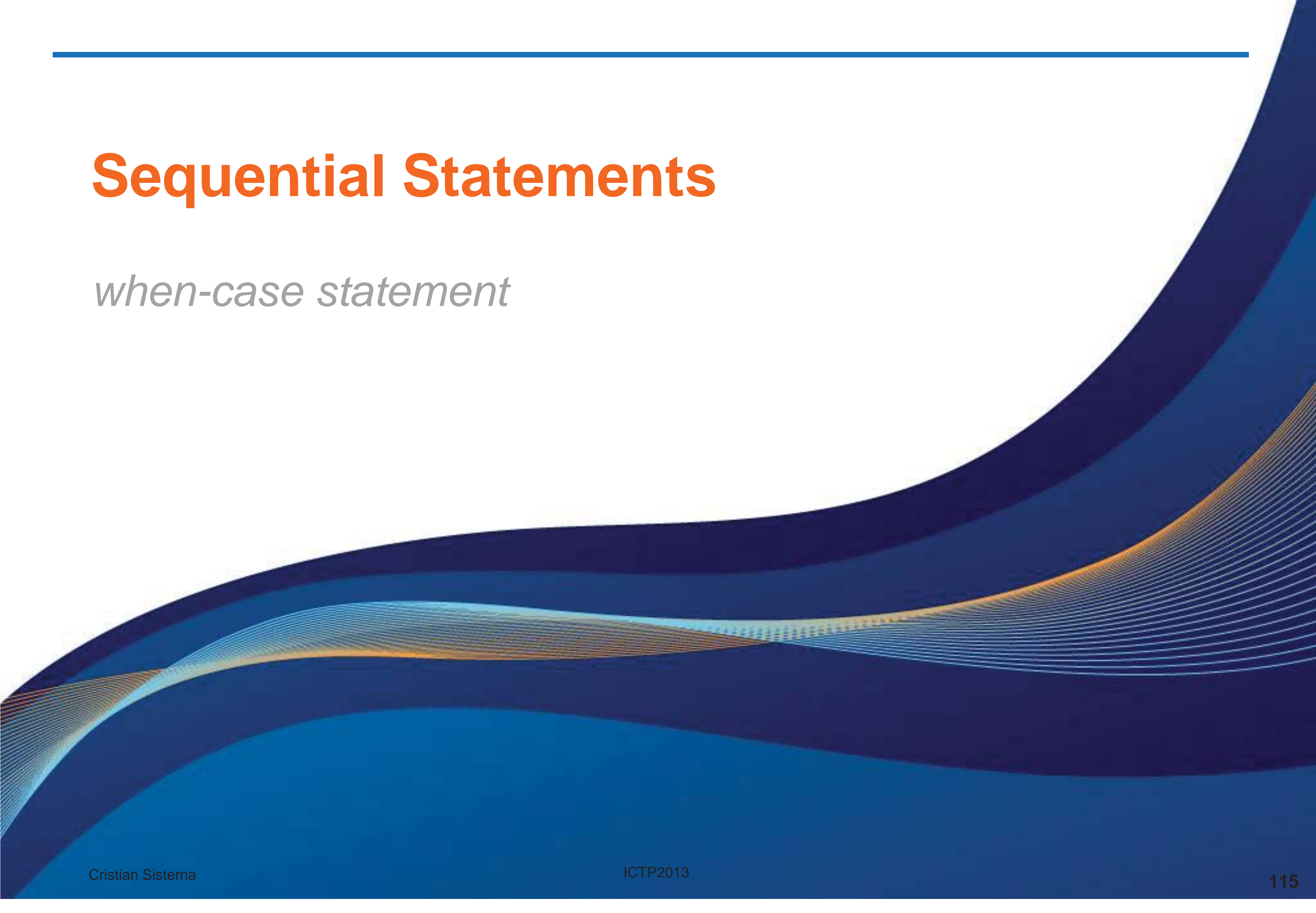
count: 0 1 2 3 4 5 6 7 8 9 A B C D E F 0 1 2 3 4 5 6 7 8 9 A B C D E F 0 1 2

if-then-elsif-else-end if

- Conclusions
  - The inadequate use of the if statement can generate unwanted logic
  - Using nested if-then it is possible:
    - Introduce priority
    - Use more logic
    - Generate latches

Sequential Statements

when-case statement



Statement: *case* – Syntax

```
[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];
```

Statement: *case*

- The value of the `<selector_expression>` is used to select which statements to execute
  - The `<selector_expression>` can be a signal, variable, any discrete type: integer or enumerated, or an array

- The <choices> are values that are compared with the value of the <selector_expression>

- There must exactly one choice for each possible value of the <selector_expression> (mutually exclusive)

Statement: *case*

- The ***case*** statement finds the branch with exactly the same choice value as the <selector_expression> and executes the statements in that branch

- The type of each choice must be of the same type as the type resulting from the <selector_expression>

- More than one choice can be included in each branch by writing the choices separated by the "|" symbol (it can be read as 'or')

Statement: *case*

- The special choice ***others*** is used to handle all possible values of <selector_expression> not mentioned on previous alternatives.

- If it is included, there must only be one branch that uses the ***others*** choice, and it must be the last alternative in the case statement.

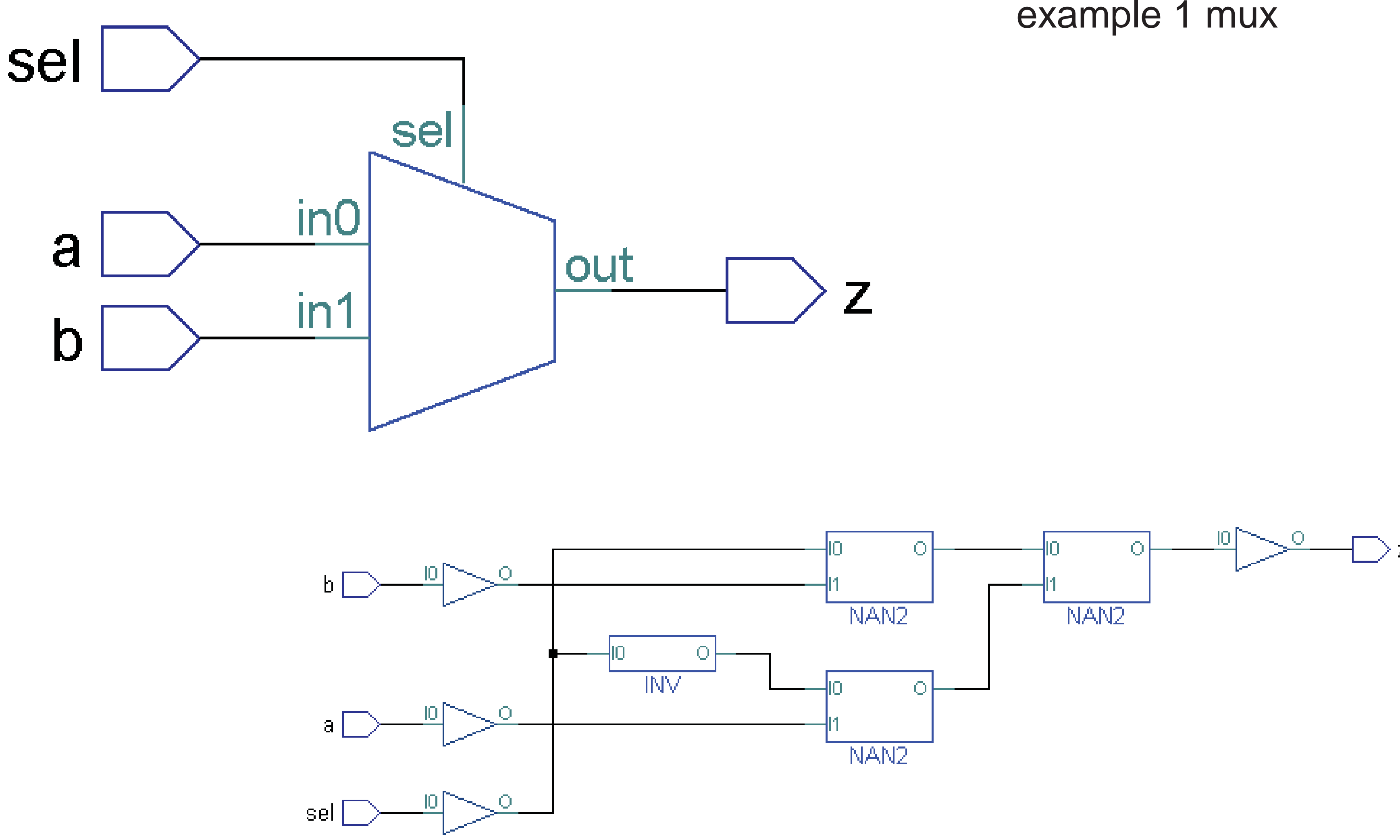
- The optional choice *[when <choice_n to/downto choice_m >* specify a discrete range of values. If the value of the <selector_expression> matches any of the values in the range, the statements in the branch are executed. This option can be used *only* if the <selector_expression> is of a *discrete type*.

Statement: *case*

```
entity mux is
        port( sel  :  in std_logic;
              a, b: in std_logic;
                z : out std_logic);
end entity;

architecture behavioral of mux is
begin
mux_proc: process(a,b,sel)
  begin
       case sel is
          when '0' =>
              z <= a;
          when '1' =>
              z <= b;
       end case;
    end process mux_proc;
end behavioral;
```

Statement: *case*

example 1 mux



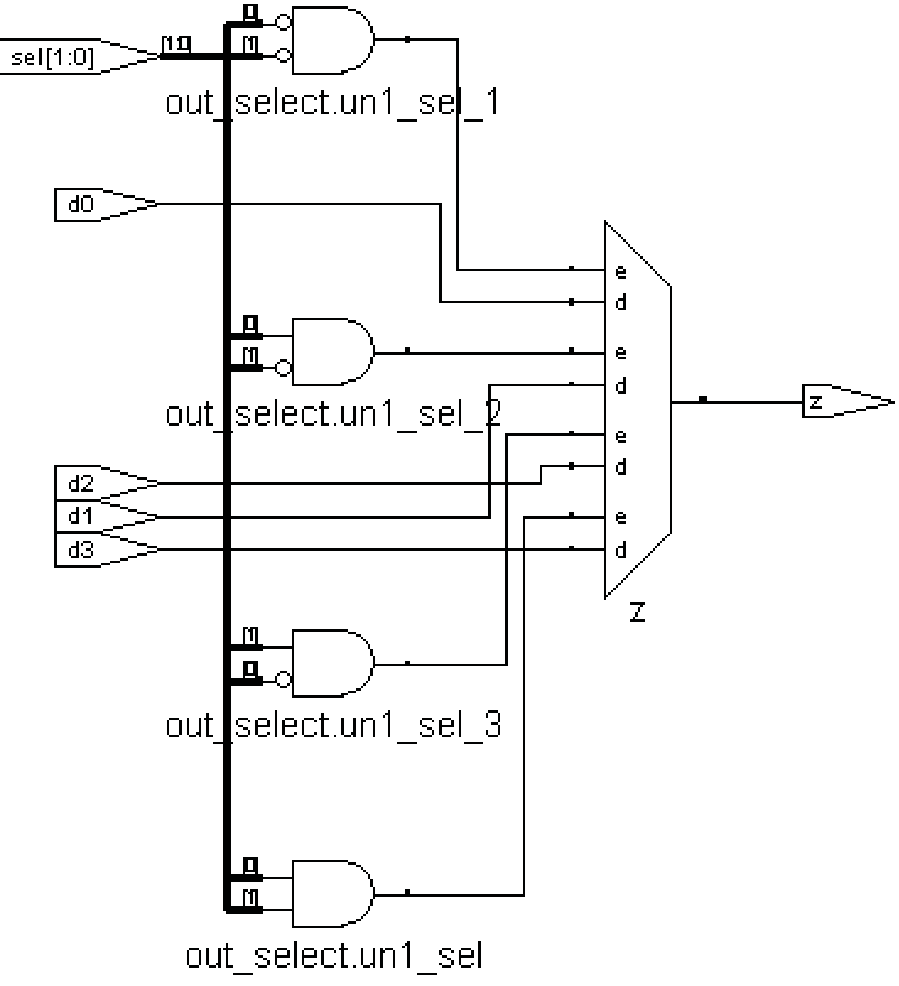
Statement: *case*

```
entity mux is
        port( sel :  in   std_logic;
              a, b:  in   std_logic;
                z :  out  std_logic);
end entity;
architecture behavioral of mux is
begin
  process(a,b,sel)
  begin
       case sel is
             when '0' =>
                    z <= a;
             when '1' =>
                    z <= b;
             --….. ??
       end case;
   end process;
end behavioral;
```

Statement: *case*

```
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;
```

Statement: *case*



Statement: *case*

```
type opcode is(nop, add, substract, load, store,
                        jump, jumsub, branch, halt);

. . .

   case opcode is
        when load | add | substract =>
                        operand <= memory_operand;
        when store | jump |jumpsub | branch =>
                        operand <= address_operand;
        when others =>
                operand <= '0';
   end case;
```

Statement: *case*

```
type opcode is(nop, add, substract, load, store,
                        jump, jumsub, branch, halt);

. . .

   case opcode is
        when add to load =>
                        operand <= memory_operand;
        when branch downto store =>
                        operand <= address_operand;
        when others =>
                operand <= '0';
   end case;
```

Statement: *case*

```
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;
       -- continue on next page . . .
```

Statement: *case*

```
        when "10" =>
           if(iq_bus = '0') then
               mux_out <= I_P1;
           else
               mux_out <= Q_P1;
           end if;
        when "11" =>
           if(iq_bus = '0') then
               mux_out <= I_A1;
           else
               mux_out <= Q_A1;--Q_P1;
          end if;
        when others =>
               mux_out <= I_P0;
   end case;
end process mux_mem_bus;
```

Statement: *case*

```
architecture no_good_def_out of case_example is
type my_fsm is(idle, run);
signal current_State, present_state: my_fsm;
begin
  process(current_state)
  begin
    case current_state is
        when idle =>
              q1 <= '1';
              q2 <= '0';
         when run =>
              q1 <= '0';
              q2 <= '1';
       when others =>
              q1 <= '0';
              q2 <= '0';
       end case;
  end process;
end no_good_def_out;
```

Statement: *case*

```
-- correct way of coding
...
process(current_state)
  begin
    q1 <= '0';
    q2 <= '0';
   case current_state is
        when idle =>
               q1 <= '1';
        when run  =>
               q2 <= '1';
        when others =>
               null;
        end case;
  end process;
end good;
```

Statement: *case*

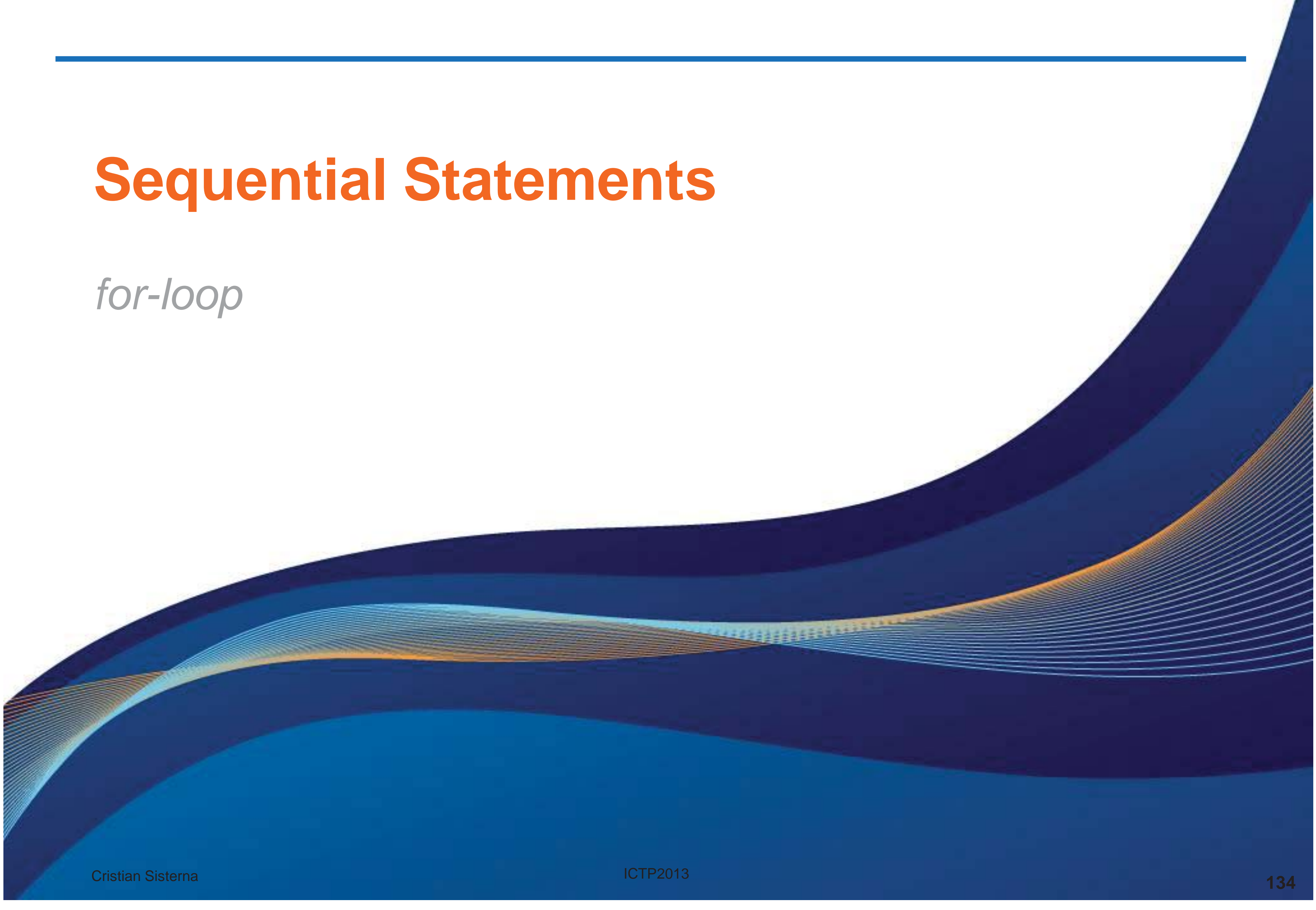
```
-- Example of using 'case' with vectors --
entity ex_vector is
   port( a: in  std_logic_vector(4 downto 0);
             q: out std_logic_vector(2 downto 0));
end ex_vector ;
architecture bad of ex_vector is
begin
process(a)
begin
   case a is
        when "00000" =>
                q <= "011";
        when "00001" to "11100" => -- error
                q <= "010";
        when others =>
                q <= "000";
        end case;
   end process;
end bad;
```

Statement: *case*

```
architecture good of ex_vector is
begin
process(a)
variable int: integer range 0 to 31;
   begin
         int := to_integer(unsigned(a));
         case int is
             when 0 =>
                   q <= "011";
             when 1 to 30 =>
                   q <= "010";
             when others =>
                   q <= "000";
         end case;
   end process;
end good;
```

Statement: *case*

- Use of case
  - Flat: do not induce unwanted prioritization
  - Device logic utilization is more predictable
  - Compiler force you to cover all cases
  - Easier to read
  - “truth-table-like”



Sequential Statements

for-loop

Statement: *loop*

```
[loop_label]:<iteration_scheme> loop
                <sequential_statements>
           end loop [loop_label];

-- first case of iteration scheme

           for <identifier> in <discrete_range>

-- second case of iteration scheme

           while <boolean_expresion | condition>

-- third case of iteration scheme
           loop
```

Statement: `loop`

- A loop is a mechanism for repeating a section of VHDL code
  - Simple `loop` continues looping indefinitely
  - `while-loop` continues looping an unspecified number of times until a condition becomes false
  - `for-loop` continues looping a specified number of times

The synthesis interpretation of loops is to *replicate* the hardware described by the contents of the loop statement once **for each pass** round of the loop.

The only synthesizable *loop* statement it is the *for-loop*

Statement: *for-loop* - Syntax

```
[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

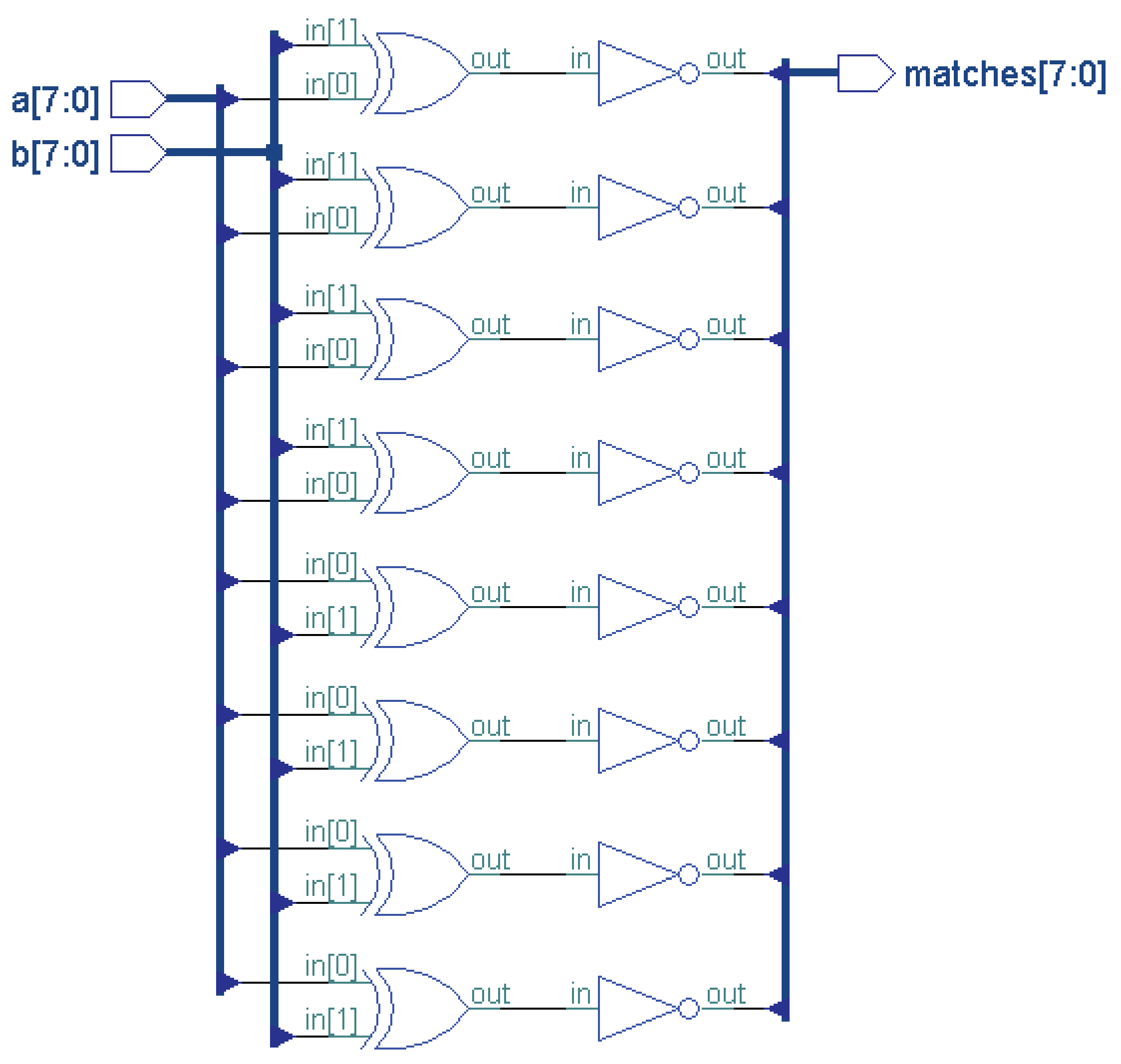
Statement: *for-loop*

```
--=========================================================--
-- Identifier declared in the loop is local to the loop
--==========================================================--

process
variable a, b: int;
begin
         a:=10;
         for a in 0 to 7 loop
                  b:=a;
         end loop;
         -- a = ?; b = ?
end process;
```

Statement: *for-loop* – Example 1

```
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;

-- process (a, b)
-- begin
-- matches(7) <= not (a(7) xor b(7));
-- matches(6) <= not (a(6) xor b(6));
-- ..
-- matches(0) <= not (a(0) xor b(0));
-- end process;
```

Synthesis Example 1

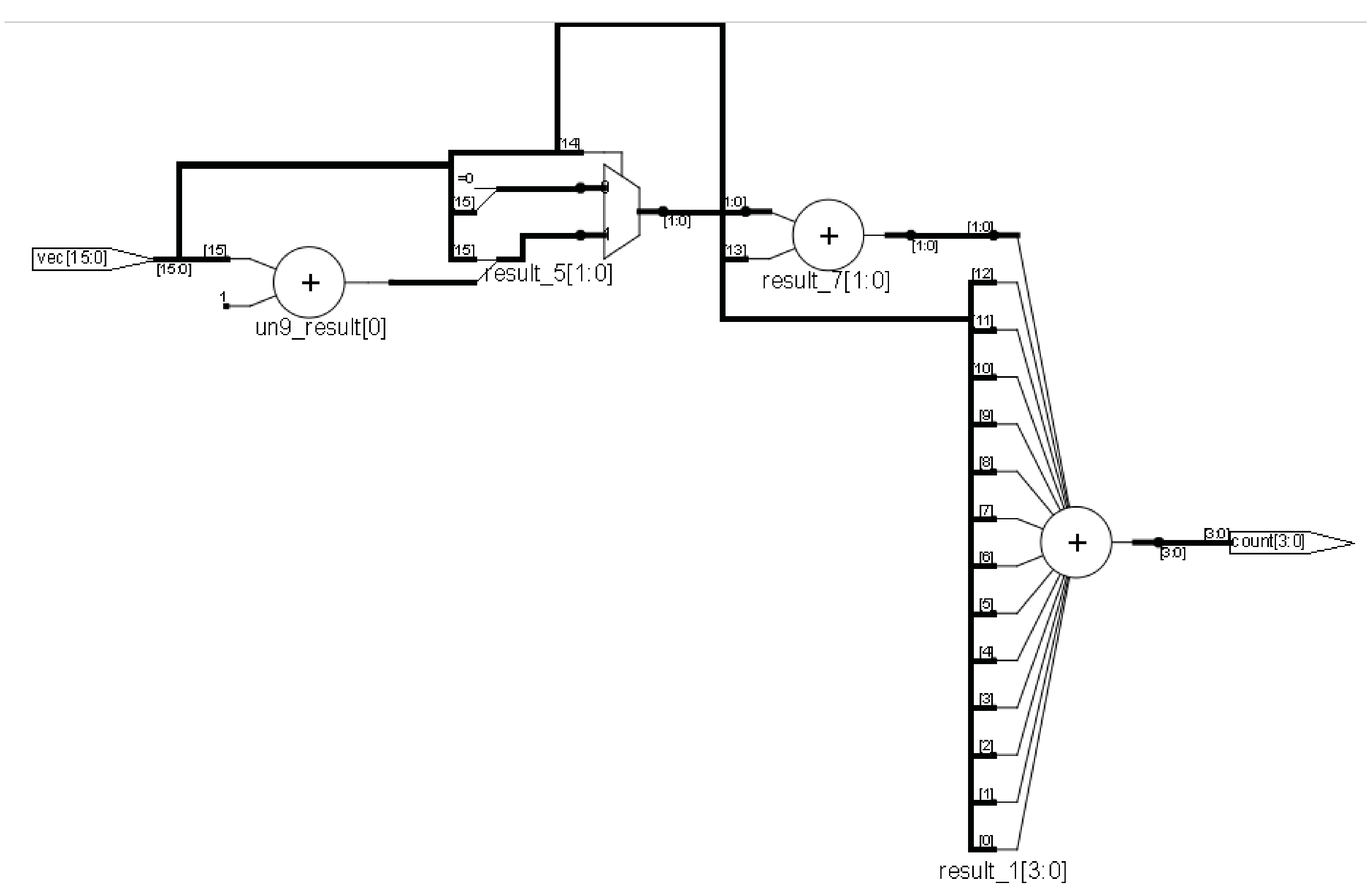


Example
for_loop_1.vhd:
match_bit

Statement: *for-loop* – Example 2

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

entity count_ones 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_ones 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;
```

Synthesis Example 2



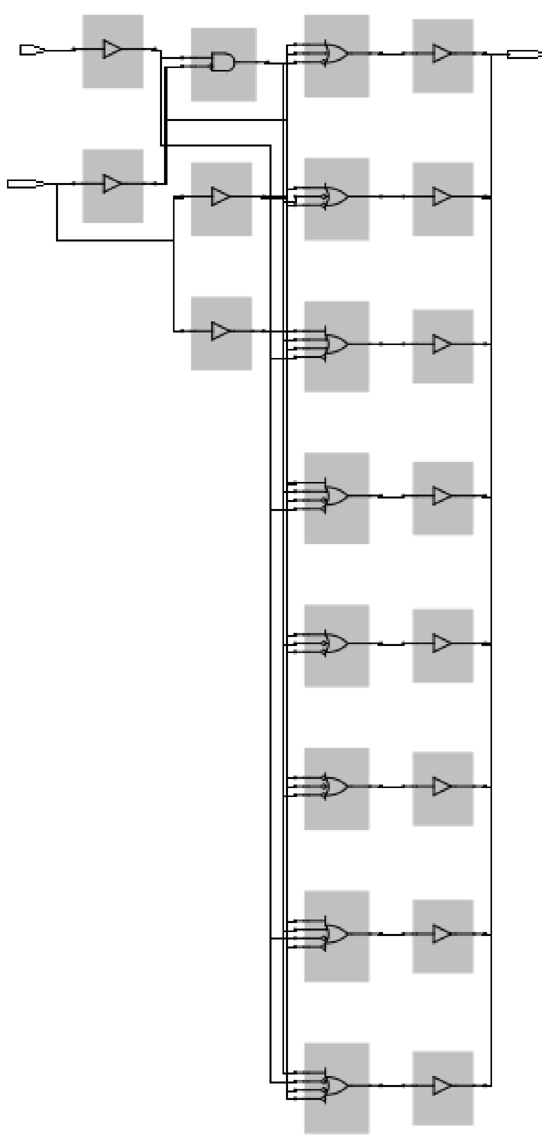
Statement: *for-loop* – Example 3

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

entity generic_??? is
  generic(width: positive := 3);
   port (
         sel  : in  std_logic_vector(width-1 downto 0);
         en   : in  std_logic;
         y_n  : out std_logic_vector((2**width)-1 downto 0)
        );
end generic_???;
```

Statement: *for-loop* – Example 3

```
architecture behavior of generic_??? is
begin
gen_proc: process (sel, en)
begin
    y_n <= (others => '1') ;
    dec_loop:for i in y_n'range loop
      if (en='1' and to_integer(unsigned(sel))= i) then
          y_n(i) <= '0' ;
      end if ;
    end loop dec_loop;
end process;
end behavior;
```

Synthesis – Example 3



Statement: *for-loop* -

- In practice, the array attributes are used to specify the loop bounds:

```
for i in vec'range loop
for i in vec'reverse_range loop
for i in vec'low to vec'high loop
for i in vec'high downto vec'low loop
```

Because the hardware interpretation of a for loop,
the bounds of the loop must be constant

Statement: *exit*

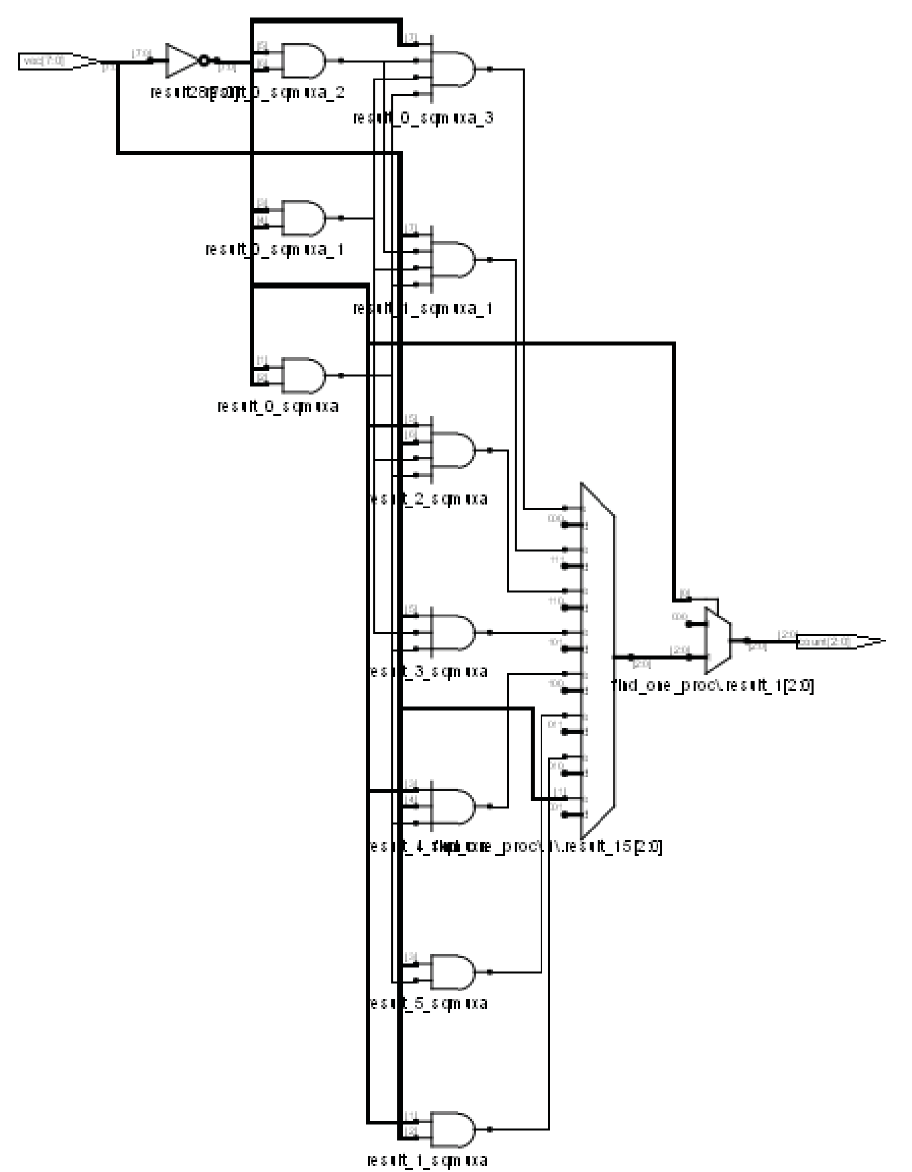
```
--========================================
-- exit syntax
--========================================
      exit [loop_label] [when <condition>]
```

Allows the execution of a *for loop* to be stopped, even tough it has not completed all its iterations

Statement: *exit* – Example 4

```
entity find_one is
   port(vec   : in std_logic_vector(7 downto 0);
        count: out std_logic_vector(2 downto 0));
end find_one ;

architecture behavior of find_one is
begin
   find_one_proc: process(vec)
        variable result: unsigned(3 downto 0);
   begin
        result:= (others =>'0');
        for i in vec'reverse_range loop
             exit when vec(i) = '1'
             result := result + 1;
        end loop;
        count <= std_logic_vector(result);
   end process find_one_proc;
end behavior;
```

Synthesis Example 4



Statement: *next*

```
--========================================
-- next syntax
--========================================

      next [loop_label] [when <condition>]
```

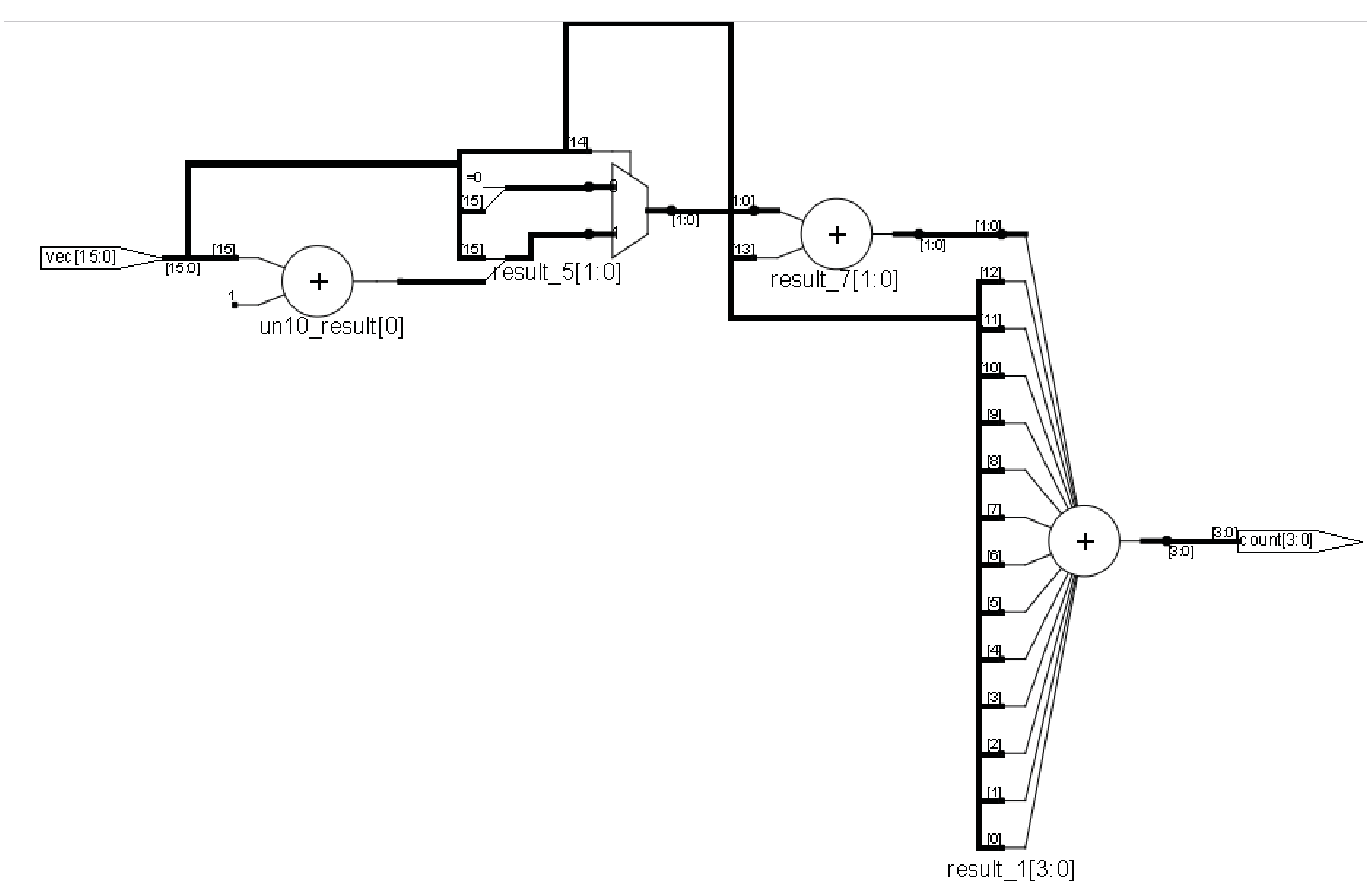
- Rather than exit the loop completely, the **next** statement skips any statements remaining in the current iteration of the loop and skips straight onto the next iteration

- A **next** statement can be a good substitute for an `if` statement for conditionally executing a group of statements. The hardware required to implement the **next** statement is the same as the hardware required to implement the equivalent `if` statement

Statement: *next* – Example 5

```
entity count_ones 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_ones is
begin
   process(vec)
         variable result: unsigned(3 downto 0);
   begin
         result:= (others =>'0');
         for i in vec'range loop
               next when vec(i) = '0';
               result := result + 1;
         end loop;
         count <= result;
   end process;
end behavior;
```

Synthesis - Example 5



Signals and Variables: loop

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

entity signal_variable_loop is
port(
       clk, data: in std_logic;
       ya, yb   : out std_logic_vector(3 downto 0));
end signal_variable_loop ;

architecture beh of signal_variable_loop is
 signal pipe_b: std_logic_vector(3 downto 0);
begin
. . . .
```

Signals and Variables: loop

```
var_loop: process(clk)
  variable pipe_a: std_logic_vector(3 downto 0);
begin
   if(rising_edge(clk)) then
        for i in 3 downto 1 loop
                pipe_a(i) := pipe_a(i-1);
        end loop;
        pipe_a(0) := data;
        ya  <= pipe_a;
    end if;
end process var_loop;
end beh;
```

Signals and Variables: loop

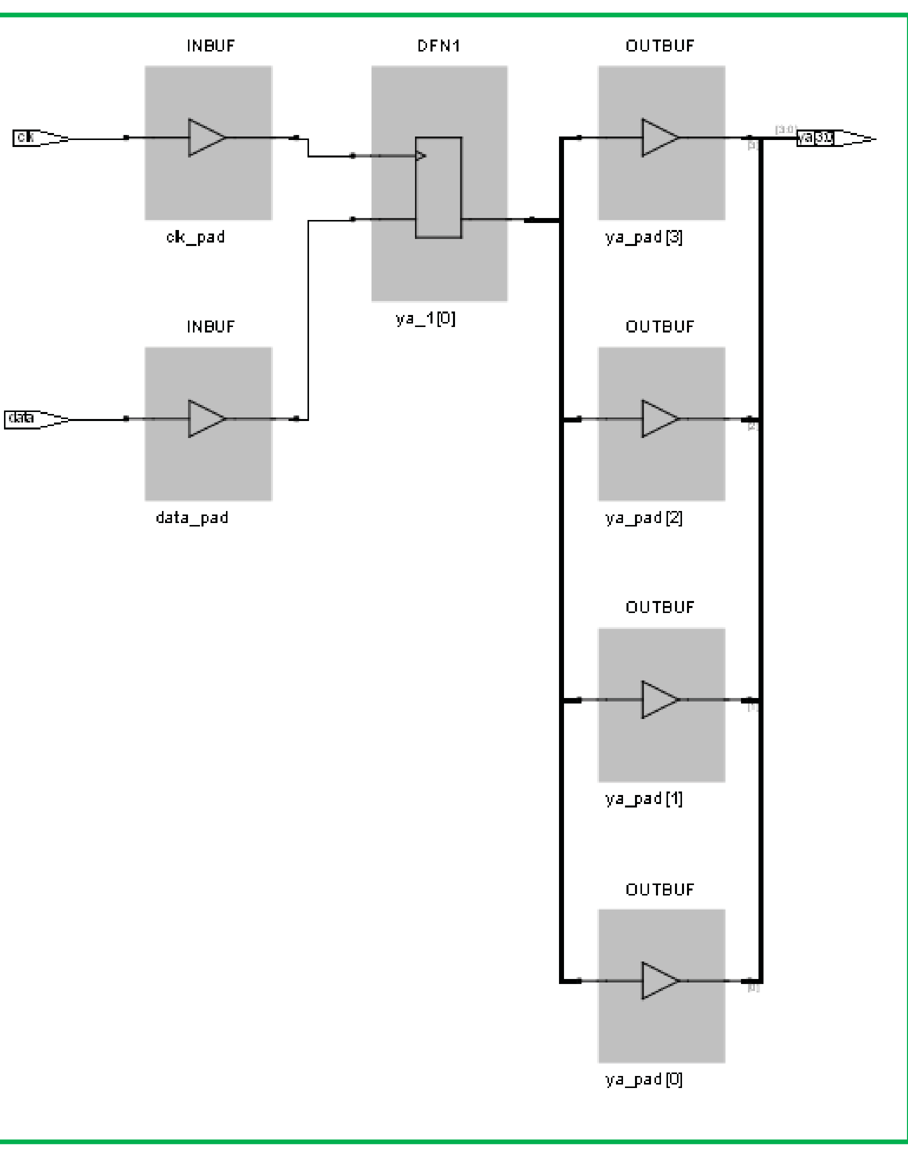
```
library ieee;
use ieee.std_logic_1164.all;

entity signal_variable_loop is
port(
       clk, data: in std_logic;
       ya, yb   : out std_logic_vector(3 downto 0));
end signal_variable_loop ;

architecture beh of signal_variable_loop is
 signal pipe_b: std_logic_vector(3 downto 0);
begin
. . . .
```

Signals and Variables: variable loop

```
var_loop: process(clk)
begin
   if(rising_edge(clk)) then
        pipe_a(0) := data;
        for i in 1 to 3 loop
                pipe_a(i) := pipe_a(i-1);
        end loop;
        ya  <= pipe_a;
   end if;
end process var_loop;
end beh;
```

```
pipe_a(0) := data;
pipe_a(1) := pipe_a(0);
pipe_a(2) := Pipe_a(1);
pipe_a(3) := pipe_a(2);
```

Signals and Variables: variable loop

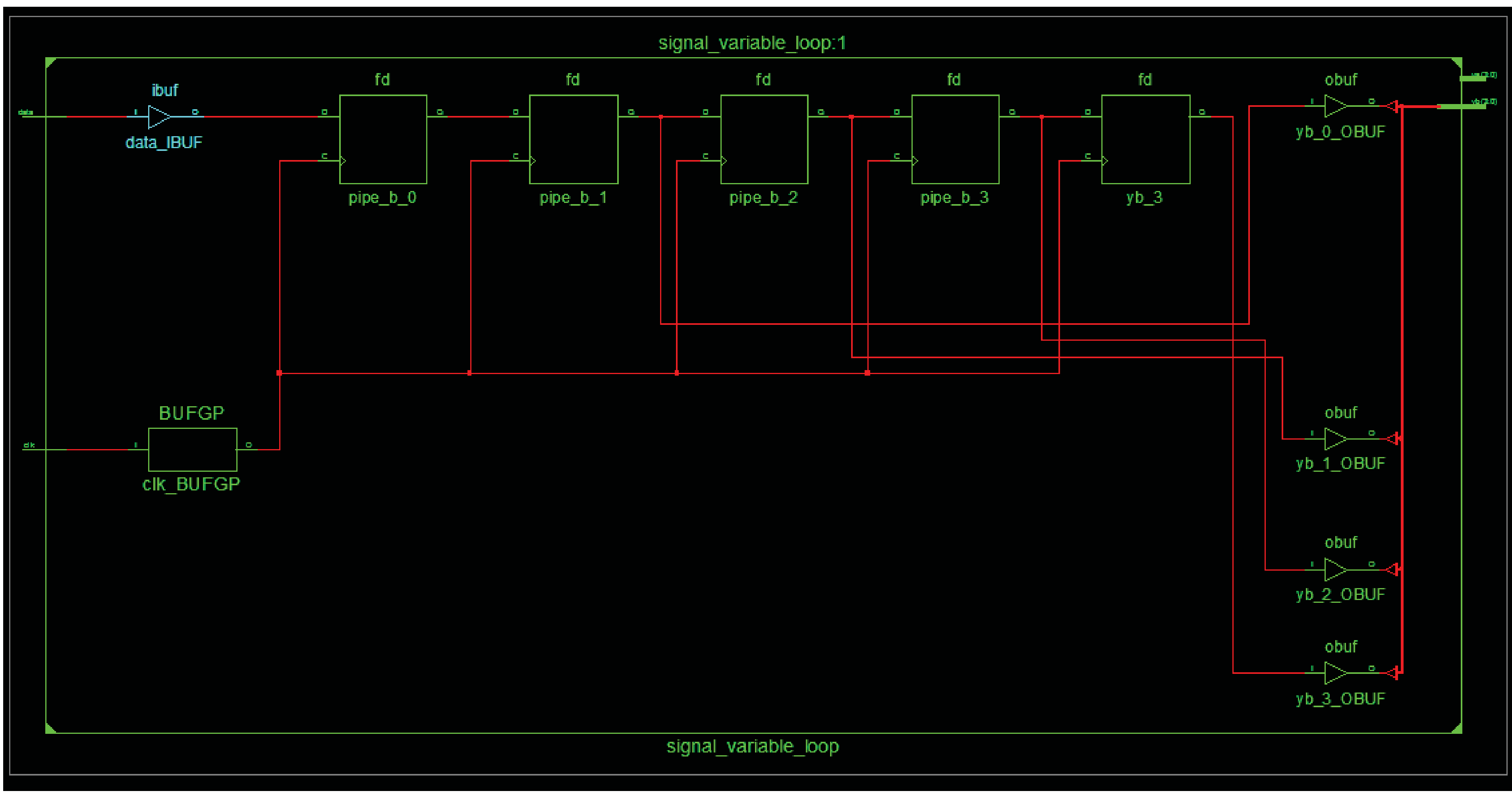


Signals and Variables: signal loop

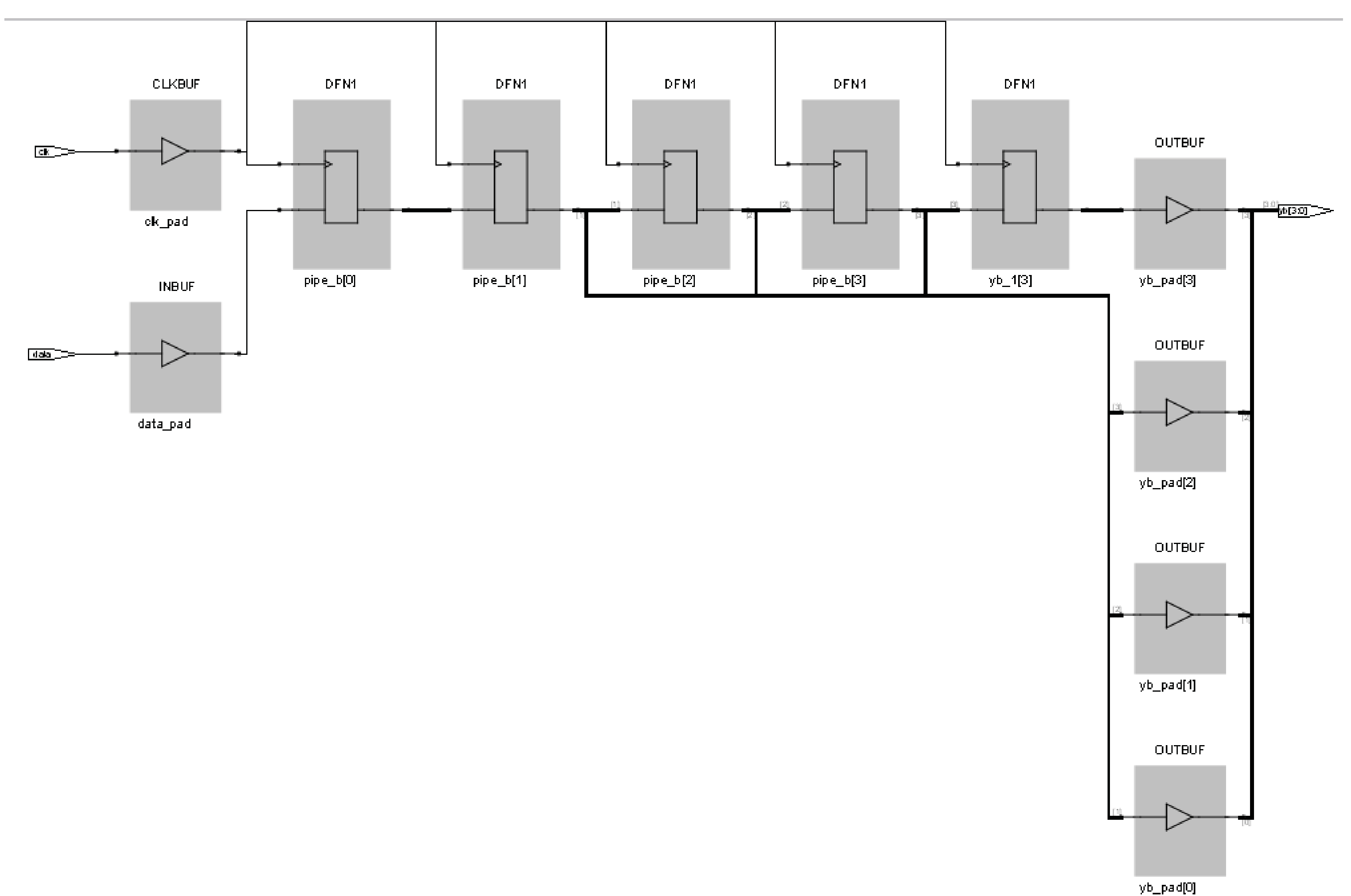
```
sig_loop: process(clk)
begin
   if(rising_edge(clk)) then
         pipe_b(0) <= data;
        for i in 1 to 3 loop
               pipe_b(i) <= pipe_b(i-1);
        end loop;
        yb  <= pipe_b;
    end if;
end process sig_loop;
end beh;
```

```
pipe_b(0) <= data;
pipe_b(1) <= pipe_b(0);
pipe_b(2) <= Pipe_b(1);
pipe_b(3) <= pipe_b(2);
```

Signal Implementation

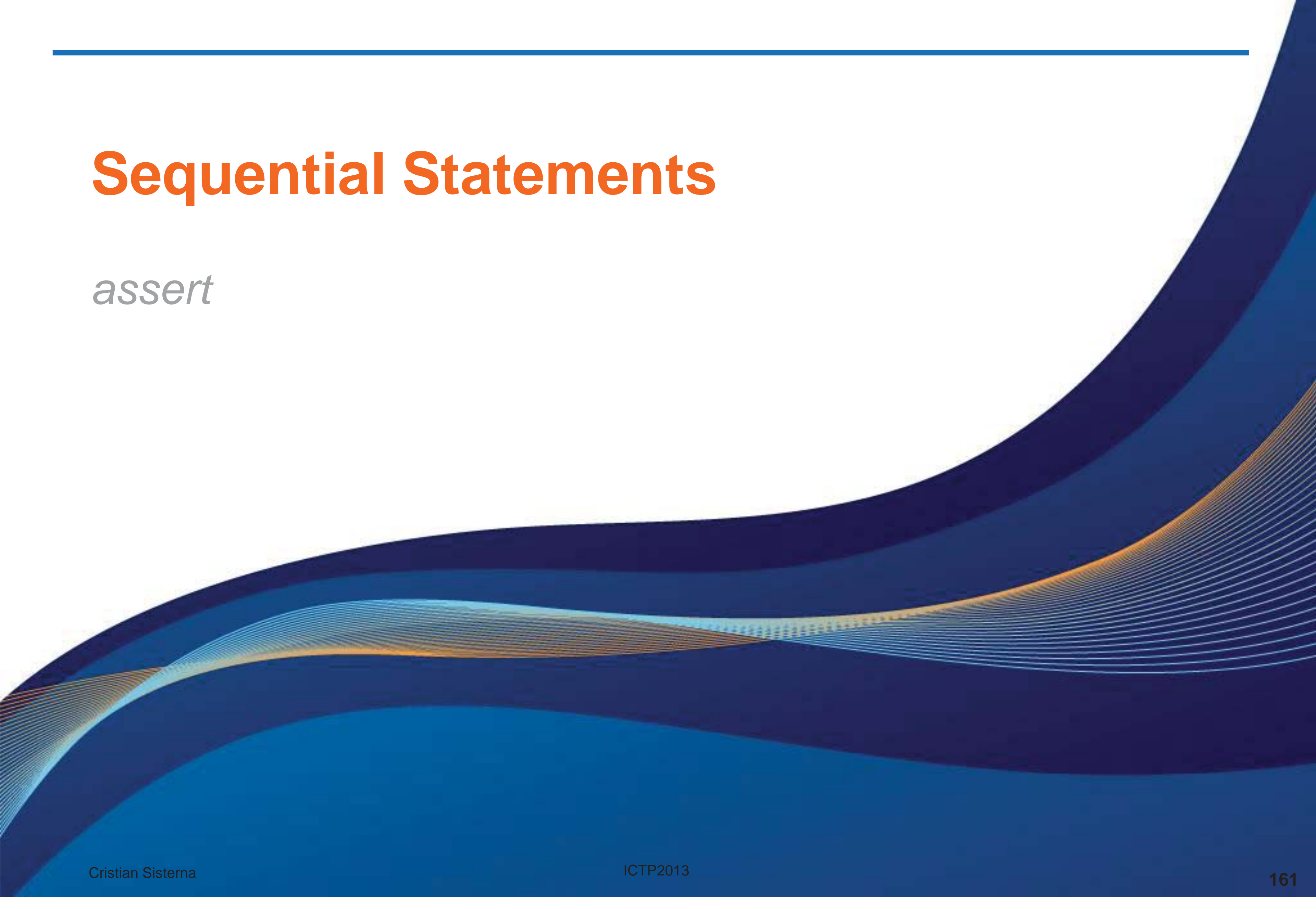


Signal Implementation



Sequential Statements

assert



Statement: `assert`

- One of the reasons for writing models of computer systems is to verify that a design functions correctly
- We can test a model by applying sample inputs and checking that the outputs meet our expectations
- *Assert statements* can check that the expected conditions are met within the model

Statement: *assert*

- Assert makes possible to test function and time constraints on a VHDL model
- Using assert it is possible to test prohibited signal combinations or whether a time constraints is not being met
- It is not synthesizable

Statement: *assert*

```
assert <boolean_expression>
      [report <string_expression>
            [severity <severity_level>];

-- severity must be a value of severity_level:
-- NOTE, WARNING, ERROR, FAILURE

-- report syntax --
      [report <string_expression>
            [severity <severity_level>];
```

Statement: `assert`

- If the boolean expression is **not meet** during simulation of a VHDL design, a message of a certain *severity* is sent to the designer

- There are four different severity (error) levels
  - `note`
  - `warning`
  - `error`
  - `failure`

- The message (from `report <string_expression>`) and error level are reported to the simulator and are usually displayed in the command window of the VHDL simulator

Statement: *assert*

- Severity (error) levels indicate the degree to which the violation of the assertion affects the operation of the model

  – ***note***: can be used to pass informative messages out

```
assert (free_memory => low_mem_limit)
   report "low in memory…!"
      severity note;
```

Statement: *assert*

– ***warning*** : can be used if an unusual situation arises in which the model can continue to execute, buy may produce unusual results

```
assert (packet_length /= 0)
   report "empty network packet received"
      severity warning;
```

Statement: *assert*

– ***error*** : can be used to indicate that something has definitely gone wrong and that corrective action should be taken

```
assert (clock_pulse_width => min_clock_width)
   report "clock width problems…!"
      severity error;
```

Statement: *assert*

– ***failure*** : can be used to detect inconsistency that should never arise

```
assert ((last_pos-first_pos)+1 = number_entries)
   report "inconsistency in buffer model!"
      severity failure;
```

Statement: *assert* – Example

Using assert to stop a simulation (test bench)

```
process (clk)
begin
    assert (now < 90 ns)
        report "-- Stopping simulator --"
         severity FAILURE;
end process;
```

`now`: is defined in the VHDL standard, it contains the simulator's internal absolute time

Component Declaration – Component Instantiation Hierarchical VHDL

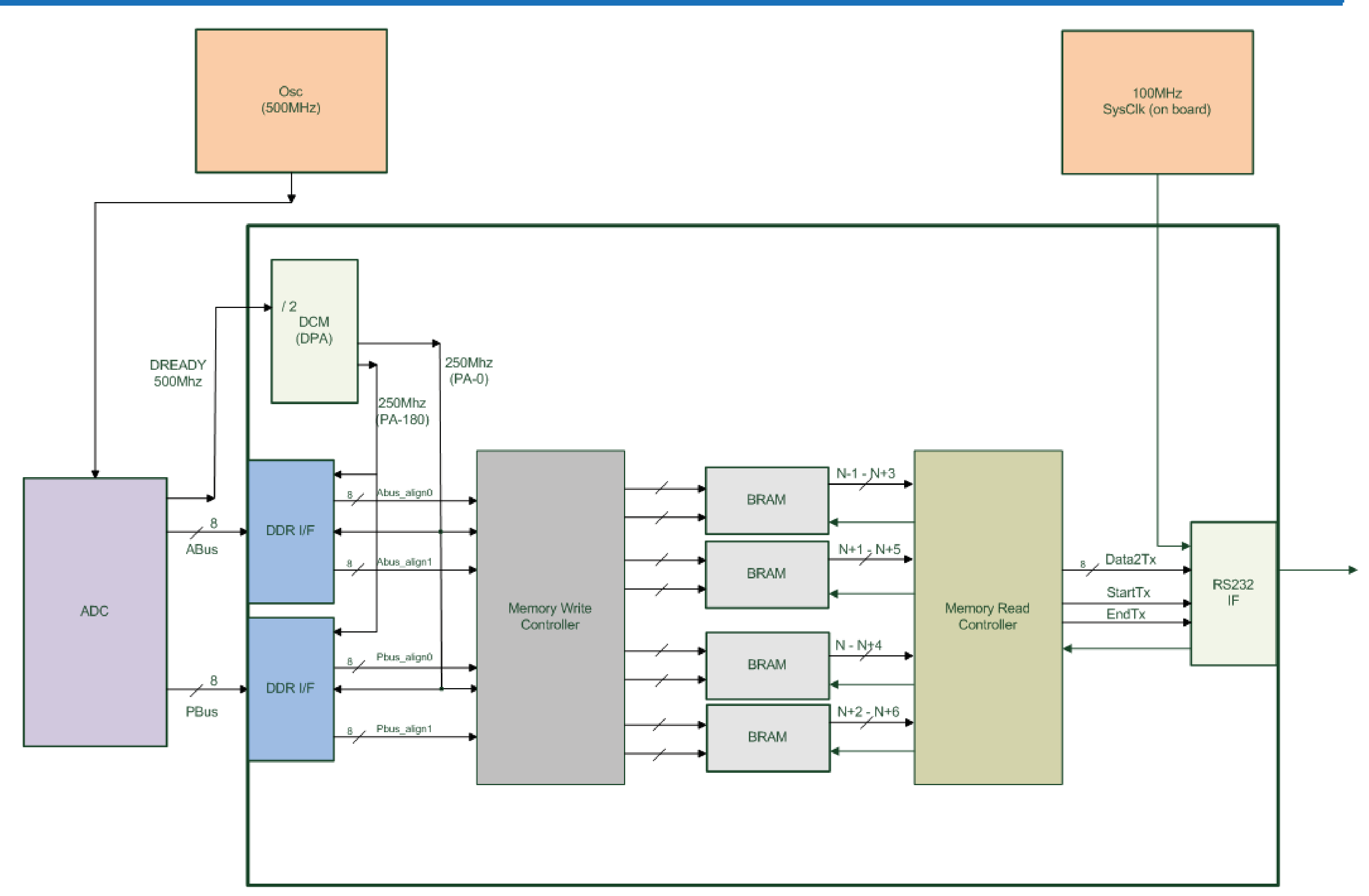
The Role of Components in RTL VHDL

Hierarchy in VHDL → **Components**

- 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

Hierarchy in VHDL - Components

Hierarchy in VHDL - Components



Components

- Components are design entities that are used in other design entities
- In order to use an entity within another entity, a *component declaration* is necessary for the entity to be used
- The interconnections between the component and the entity's signals is declared in the *component instantiation*

Component Declaration

- Define a sub-component within an entity (component)
- It can be declared in the architecture declarative part or in the package declaration (items declared in a package can be used in an entity-architecture pair by using the library and the package names)
- Specify the component external interface: ports, mode and type and also the component name (it looks like an entity declaration)

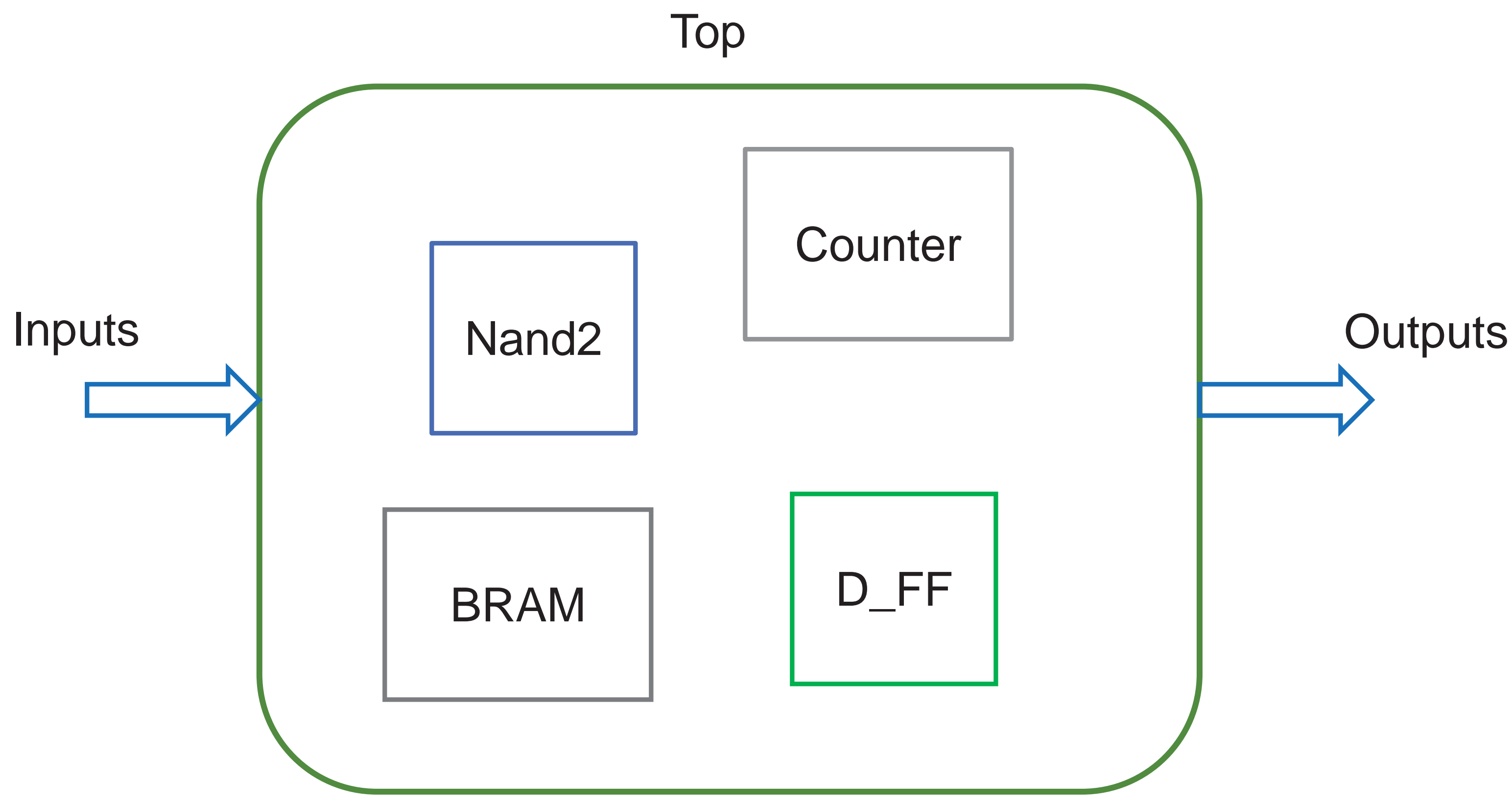
Component Declaration

Syntax:

```
component comp_name [is]
    [generic
   (generic_interface_list);]
      port (port_interface_list);
 end component [component_name];
```

- port_interface_list must be identical to that in the component's entity
- generic_interface_list do not need to be declared in the component declaration

Component Declaration



Component Declaration

```
entity nand2 is
       port (a, b: in std_logic,
                z: out std_logic);
end;
architecture rtl of nand2 is
…
end;
```

```
entity top is
     port(…
        );
end;
achitecture structural of top is
component nand2
      port (a, b: in  std_logic,
                 z   : out std_logic);
end component;
…
begin
….
end;
```

Component Instantiation

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

- `component_label` it labels the instance by giving a name to the component to be 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

Positional Association

In positional association, an association list is of the form

```
(actual1, actual2, actual3, … actualn);
```

- Each actual in the component instantiation is mapped *by position* with each port in the component declaration
- That is, the first port in the component declaration corresponds to the first *actual* in the component instantiation, the second with the second and so on
- The I/Os on the component declaration, are called *formals*

Positional Association

```
-- component declaration
component NAND2
        port (a, b: in std_logic,
              z: out std_logic);
end component;
```

formals

```
-- component instantiation
U1: NAND2 port map (S1, S2, S3);
-- S1 associated with a
-- S2 associated with b
-- S3 associated with z
```

actuals

Named Association

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: NAND2 port map (a=>S1, z=>S3, b=>S2);
-- S1 associated with a,S2 with b and S3 with z
```

Association Rules

- The type of the formal and the actual being associated must be the same
- The modes of the ports must conform the rule that if the formal is readable, so must the actual be. If the formal is writable so must the actual be
- If an actual is a port of mode in, it may no be associated with a formal of mode out or inout
- If the actual is a port of mode out, it may not be associated with a formal of mode in or inout
- If the actual is a port of mode inout, it may be associated with a formal of mode in, out or inout

Unconnected Outputs

- When a component is instanced, one of the outputs sometimes has to be unconnected
- This can be done using the keyword *open*

```
architecture rtl of top_level is
component ex4
    port (a, b   : in std_logic;
          q1, q2: out std_logic;
end component;
begin
        U1: ex4 port map(a=>a, b=>b, q1=>dout, q2=>open);
end;
```

Unconnected inputs

- Leaving floating inputs is **a very bad poor technique**
- If an input on a component is not to be used, the signal should be connected to VCC or GND.
- VHDL '87: It is not permissible to map the input directly in the port map, an internal signal must be used

Unconnected inputs

```
architecture rtl of top_level is
component ex4
   port (a, b: in std_logic;
          q1, q2: out std_logic;
end component;
signal gnd: std_logic;
begin
  gnd <= '0';
  U1: ex4 port map(a=>gnd, b=>b, q1=>dout, q2=>open);
end;
```

Unconnected inputs

```
architecture rtl of top_level is
component ex4
   port (a, b   : in std_logic;
         q1, q2: out std_logic;
end component;

begin
  U1: ex4 port map(a=>'0', b=>b, q1=>dout, q2=>open);

end rtl;
```

component - Example

```
entity GATING is
  port (A, CK, MR, DIN: in BIT;
        RDY, CTRLA: out BIT);
end GATING;

architecture STRUCT of GATING is
component AND2
  port( X, Y: in bit;
        Z: out bit);
end component;

component DFF
  port (D, CLOCK: in BIT;
        Q, QBAR: out BIT);
end component;
```



```
component NOR2
        port ( DA, DB: in BIT;
               DZ: out BIT);
end component;

signal S1, S2: BIT;
begin
D1: DFF port map (A, CK, S1, S2);
A1: AND2 port map (S2, DIN, CTRLA);
N1: NOR2 port map (S1, MR, RD1);
end STRUCT;
```

Generic Map

```
generic map (generic_assocation_list);
```

- If generic components have been specified in the component to be instanced, their value can be changed during instantiation using the command *generic map*
- By using generics, it is possible to design components which can be parameterized
- Positional and named association can be used

component - Example

```
ARCHITECTURE ejemplo OF regist_variable IS
COMPONENT dff
   GENERIC ( width: POSITIVE);
   PORT (rst, clk: IN std_LOGIC;
             d: IN STD_LOGIC_VECTOR(width-1 downto 0);
             q: OUT STD_LOGIC_VECTOR(width-1 downto 0));
END COMPONENT;
CONSTANT width_8: POSITIVE:= 8;
CONSTANT width_16: POSITIVE:= 16;
CONSTANT width_32: POSITIVE:= 32;
SIGNAL d8, q8: STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL d16, q16: STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL d32, q32: STD_LOGIC_VECTOR(31 DOWNTO 0);
```

component - Example (cont')

```
BEGIN
  FF8: dff GENERIC MAP(width_8)
           PORT MAP (rst, clk, d8, q8);
  FF16: dff GENERIC MAP(width_16)
            PORT MAP (rst, clk, d16, q16);
  FF32: dff GENERIC MAP(width_32)
            PORT MAP (rst=>rst,clk=>clk,d=>d32, q=>q32);
END ejemplo;
```

Concurrent Statements

generate

Generate Statements

- Concurrent statements can be conditionally selected or replicated using the *generate* statement

- Generate is a concurrent statement containing further concurrent statements that are to be replicated

- There are two forms of the generate statement:
  - `for-generate` scheme: concurrent statements can be replicated a predetermined number of times
  - `if-generate` scheme: concurrent statements can be conditionally elaborated

Generate Statements

- Generate statement resembles a macro expansion
- If the same component has to be instanced several times in the same architecture, it will be very effective to include the port map statement in a loop

for-generate

Concurrent statements are repeated a predetermined number of times

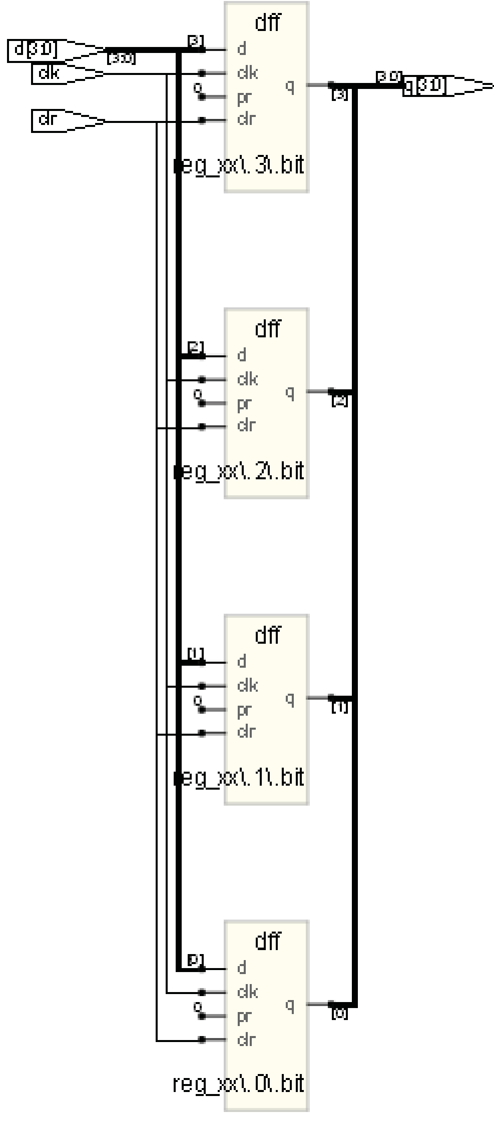
```
G_LABEL: FOR <identifier> IN <discrete_range> GENERATE
                [block_declarative_part]
                [begin]
                        concurrent_statements;
         END GENERATE [G_LABEL];
```

- The value in the `discrete range` must be globally static
- During the elaboration phase, the set of concurrent statements are replicated once for each value of the discrete range
- There is an implicit declaration for the generate `identifier`. No declaration is necessary for this identifier
- A label is required in the generate statement

for-generate

```
entity reg_xx is
        generic (bus_w:integer := 32);
         port(clk, clr:  in  std_logic;
                   d  :  in  std_logic_vector (bus_w-1 downto 0);
                   q  :  out std_logic_vector (bus_w-1 downto 0));
end reg_xx;
architecture estructural of reg_xx is
   -- component declaration
   component dff is
           port (clk, clr, d, pr: in std_logic;
                         q: out std_logic);
   end component;
   -- signal declaration
   signal gnd: std_logic;
begin
   gnd <= '0';
   -- component instantiation
   reg_xx: for i in d'range  generate
     bit: dff port map (clk=>clk, clr=>clr, d=>d(i), q=>q(i), pr=>gnd);
   end generate reg_xx;
end estructural;
```

for-generate



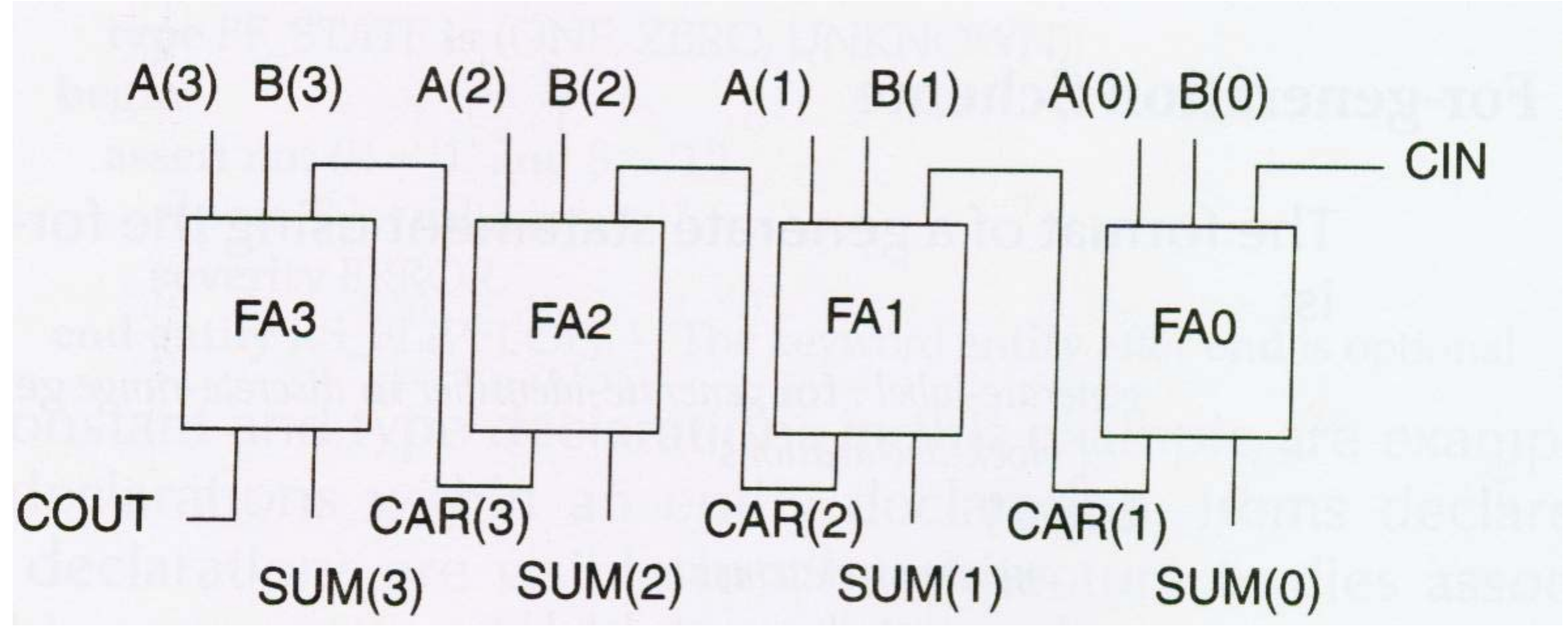
for-generate Scheme

```
entity FULL_ADD4 is
 port (A, B: in std_logic_vector(3 downto 0);
  CIN: in std_logic_vector;
  SUM: out std_logic_vector (3 downto 0); COUT: out std_logic);
end FULL_ADD4;
architecture FOR_GENERATE of FULL_ADD4 is
 component FULL_ADDER
  port (PA, PB, PC: in std_logic;
        PCOUT, PSUM: out std_logic);
 end component;
 signal CAR: std_logic_vector (4 downto 0);
 begin
  CAR(0) <= CIN;
  GK: for K in 0 to 3 generate
   FA:FULL_ADDER port map(CAR(K),A(K),B(K),CAR(K+1),SUM(K));
  end generate GK;
  COUT <= CAR(4);
end FOR_GENERATE;
```

for-generate Scheme

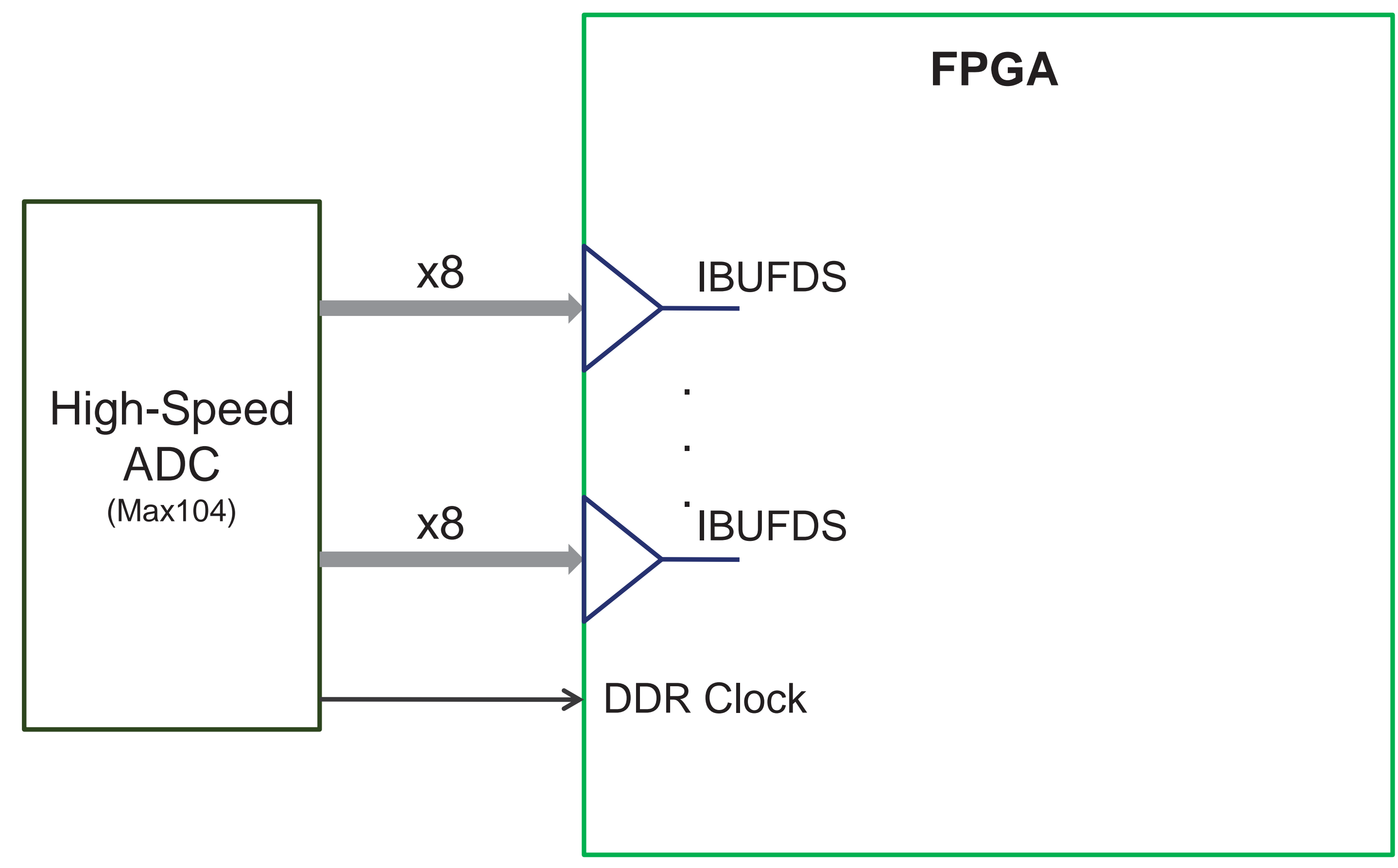
```
--- the previous generate statement is expanded to the
--- following four blocks
GK: block
constant K: INTEGER := 3;
begin
  FA: FULL_ADDER port map (CAR(K), A(K), B(K), CAR(K+1), SUM(K));
end block GK;

GK: block
constant K: INTEGER := 2;
begin
  FA: FULL_ADDER port map (CAR(K), A(K), B(K), CAR(K+1), SUM(K));
end block GK;


. . . .
GK: block
constant K: INTEGER := 0;
begin
  FA: FULL_ADDER port map (CAR(K), A(K), B(K), CAR(K+1), SUM(K));
end block GK
```

for-generate



Example 1

for-generate Scheme



for-generate Scheme

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
package VCOMPONENTS is

. . . .

-- IBUFDS: Differential Input Buffer
-- Virtex-II/II-Pro, Spartan-3
-- Xilinx HDL Libraries Guide version 11.1i

component IBUFDS
   generic map ( IOSTANDARD => "LVDS_25")
   port map (
            O => O,      -- buffer output
            I => I,      -- Diff_p clock buffer input
            IB => IB     -- Diff_n clock buffer input);
end component;
. . . .
end package VCOMPONENTS;
```

I
IB
O

for-generate Scheme

```
abus_diff_sstl2: for i in 0 to 7 generate
    u_abus: IBUFDS
      --generic map ( IOSTANDARD => "SSTL2_II")
      port map(
             O => a_bus(i),
             I => a_bus_p(i),
             IB=> a_bus_n(i)
             );
end generate abus_diff_sstl2;

pbus_diff_sstl2: for i in 0 to 7 generate
     u_pbus: IBUFDS
      --generic map ( IOSTANDARD => "SSTL2_II")

      port map(
             O => p_bus(i),
             I => p_bus_p(i),
             IB=> p_bus_n(i)
             );
end generate pbus_diff_sstl2;
```

if-generate

Syntax: ***if-generate***, concurrent statements can be condicionally elaborated

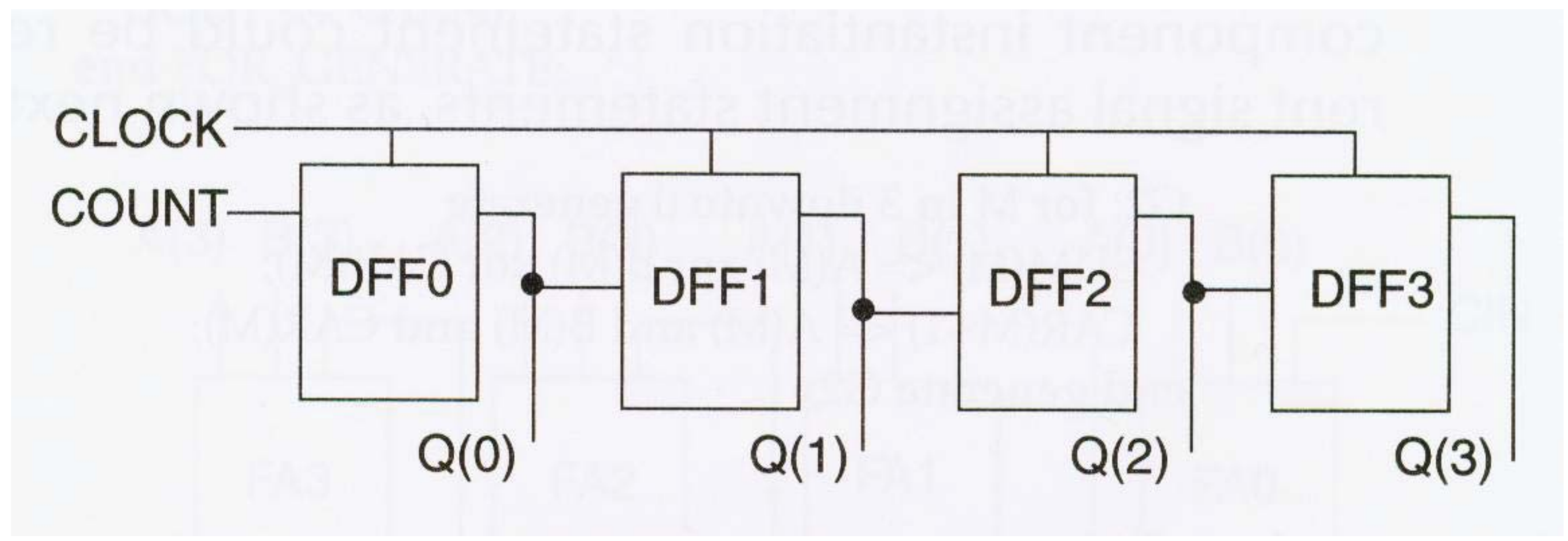
```
G_LABEL: IF <condition> GENERATE
            [begin]
                 concurrent_statements;
            END GENERATE [G_LABEL];
```

- The if-generate statement allows for conditional selection of concurrent statements based on the value of an expression
- The expression must be a globally static expression
- The if-generate statement does not have else, elsif, endif

if-generate

```
COMPONENT flip_flop IS
      PORT (clk, clr, d, pr: IN STD_LOGIC;
                      q, qb: OUT STD_LOGIC);
END COMPONENT;

-- ......
zzzzt_reg_32: FOR i IN 31 DOWNTO 0 GENERATE
   I31:IF (i=31) GENERATE
       BIT1: flip_flop PORT MAP (clk, clr, zzz_in, gnd, internal(i-1), qb=>open);
   END GENERATE I31;
   I30_1:IF (I<31 and I>0) GENERATE
       BIT301: flip_flop PORT MAP (clk, clr, interna(i), gnd, internal(i-1), qb=>open);
   END GENERATE I30_1;
   I0:IF (I=0) GENERATE
       BIT0: flip_flop PORT MAP (clk, clr, interna(i), gnd, zzz_out, qb=>open);
   END GENERATE I0;
END GENERATE zzzzt_reg_32;
```

if-generate



Example 2

if-generate

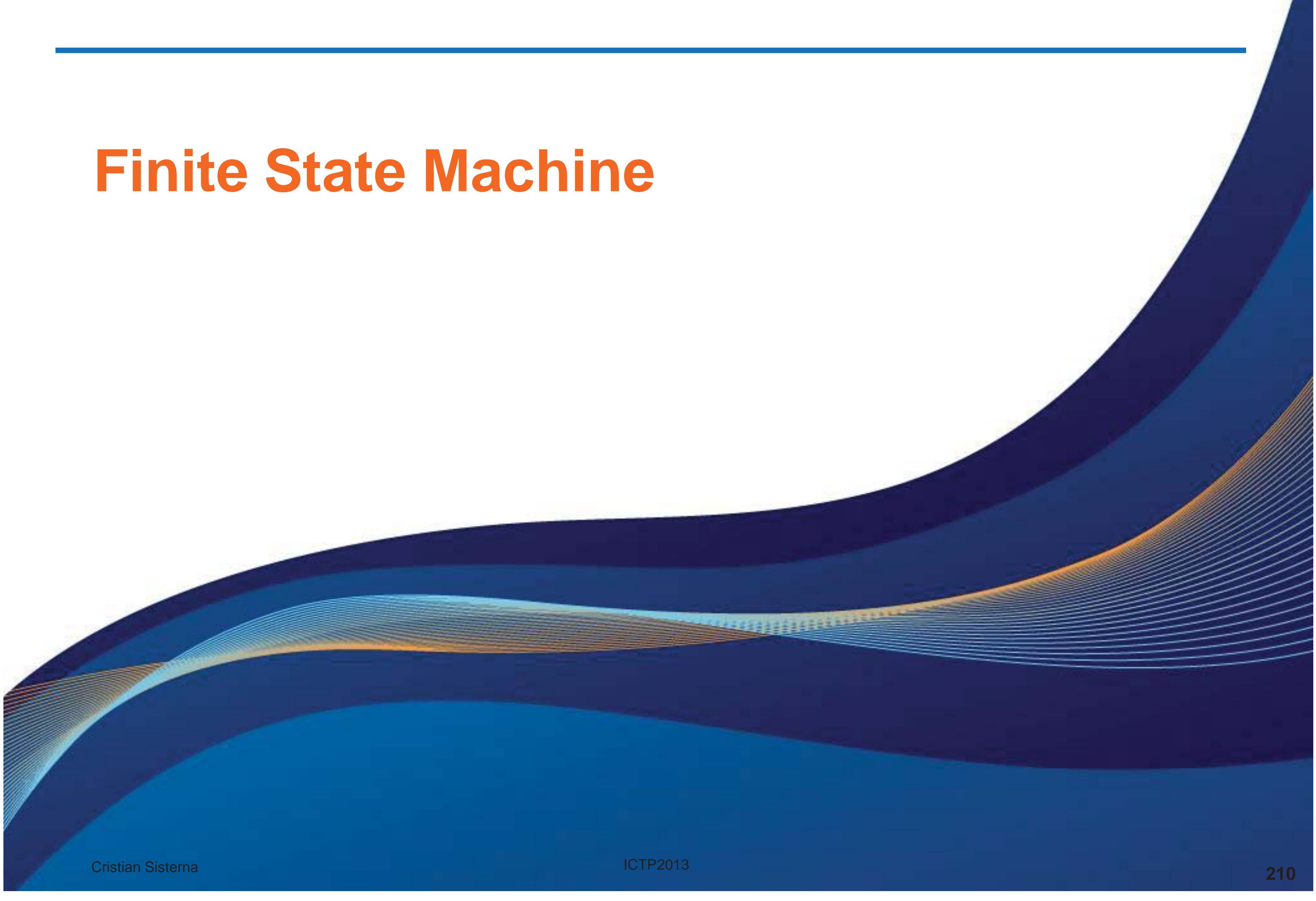
- Another important use of conditional generate statements is to conditionally include or omit part of the design. Usually depending on the value of a generic constant.

- Typical examples:
  - Logic added just for debugging purposes
  - Additional processes or component instances used only during simulation

if-generate

```
entity my_system is
       generic ( debug: boolean := true)
       port (
               . . .
             );
end entity my_system;

architecture rtl of my_system is
. . .
begin
  . . .
  debug_comp: if debug generate
        .. .
  end generate debug_comp;
  . . .
end architecture;
```

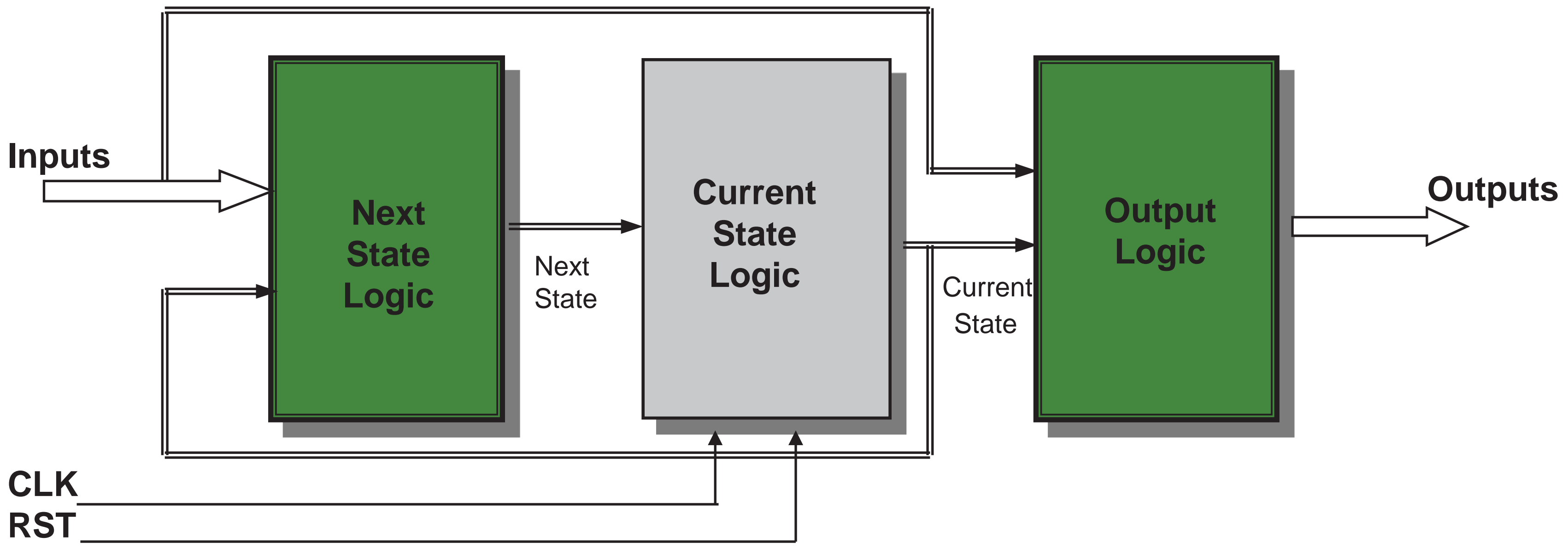
Finite State Machine



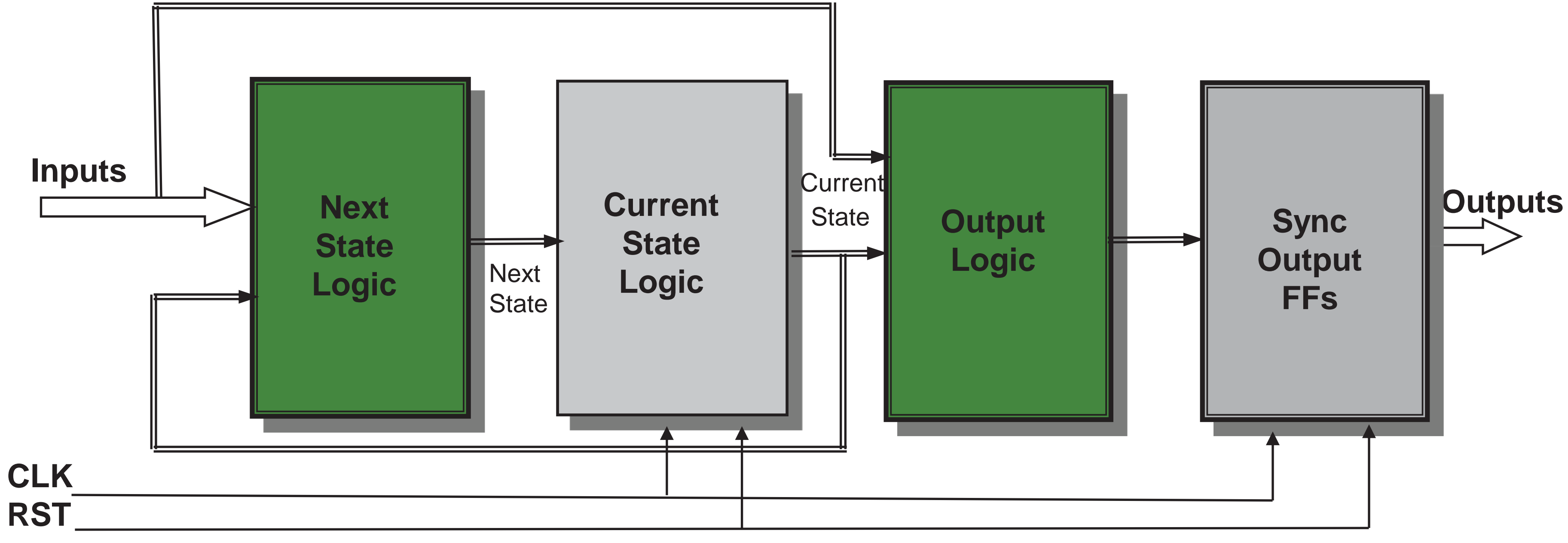
FSM Review

- A sequential circuit that is implemented in a fixed number of possible states is called a Finite State Machine (FSM).
- Finite state machines are critical for realizing the control and decision-making logic in a digital system.
  - Finite state machines have become an integral part of the system design.
- VHDL has no formal format for modeling finite state machines.
  - To model a finite state machine in VHDL certain guidelines must be followed.

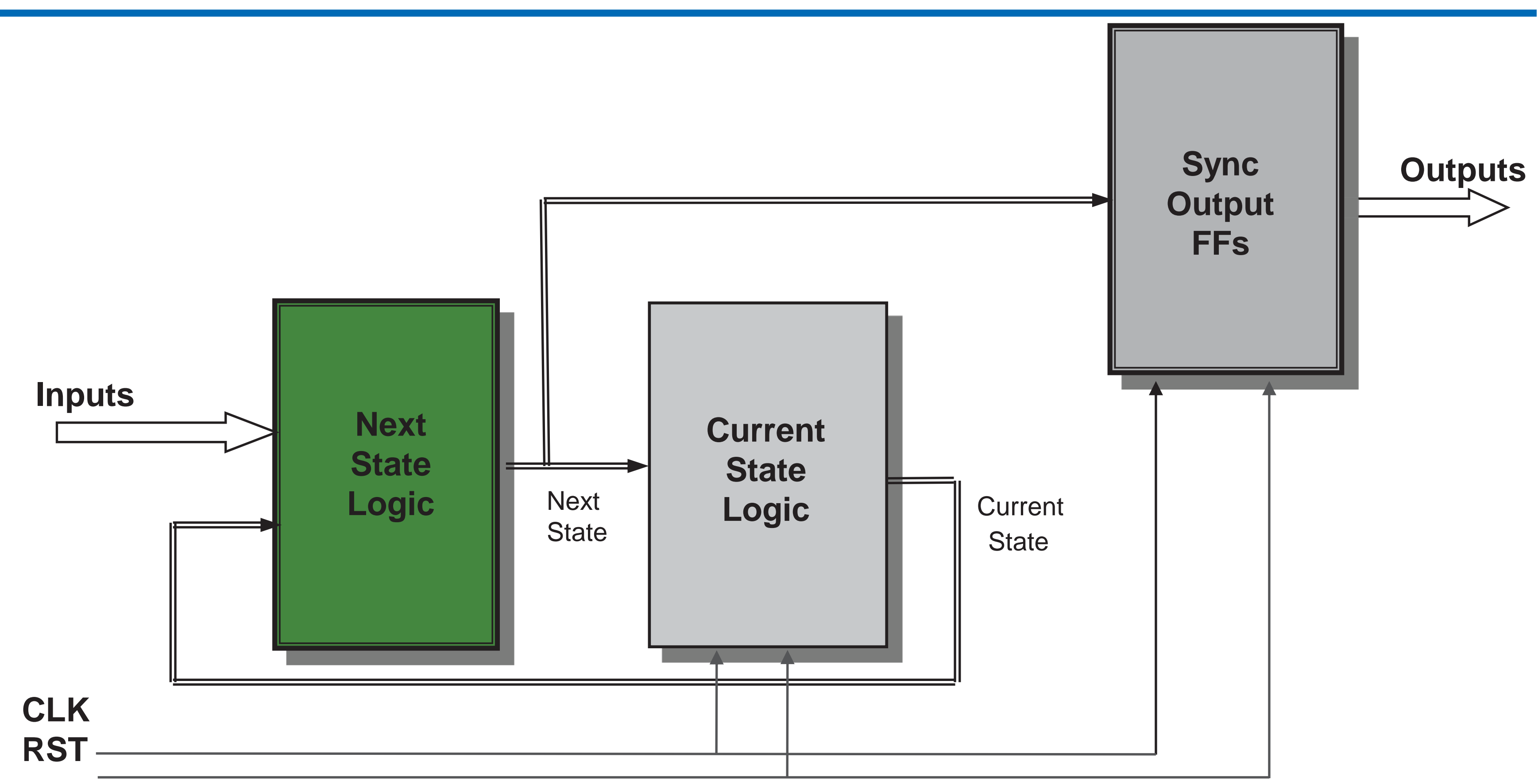
State Machine General Diagram 1



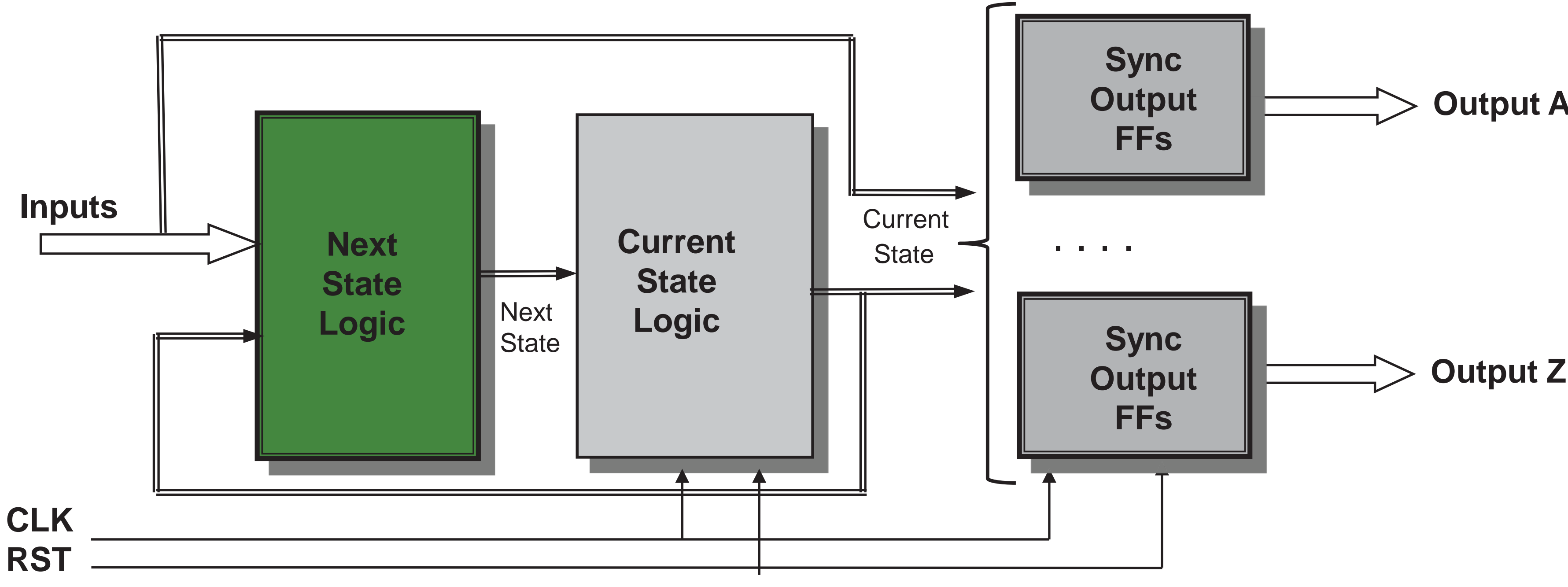
State Machine General Diagram 2



State Machine Diagram 3



State Machine General Diagram 4



State Machine VHDL General Flow

State Machine General Diagram

Coding Style	Current State Clocked Process	Next State Logic	Output Logic
Style A	○	○	○ (Comb)
Style B	●	●	●
Style C	●	●	○
Style D	○	●	●
Style E	○	○	○ (Seq)

○ **Different processes** ● **Combined processes**

State Machine: Declarative Part

Declare an enumerated data type with values the states of the state machine:

```
-- 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`

State Machine: Clocked Process

- The clocked process decides when the state machine should change state
- This process is activated by the state machine's clock signal
- Depending on the present state and the value of the input signals, the state machine can change state at every active clock edge
- Current state gets the value of the next state on the active edge of the clock
- Next state value is generated in the state transition process, depending on the values of current state and the inputs

State Machine: Combinatorial Process

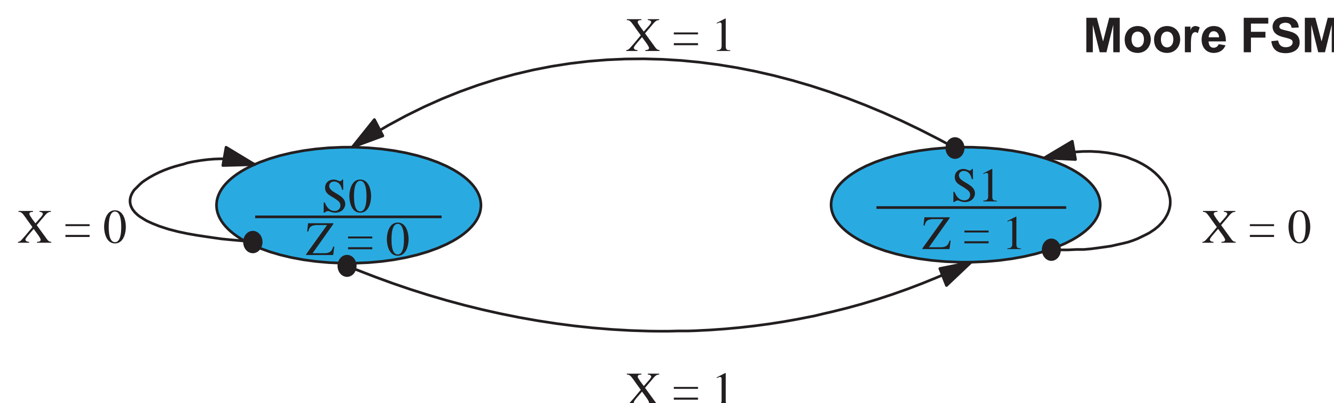
- Assigns the output signals their value depending on the present state
- Next state logic and output logic is best modeled using `case` statements are better for this process
- All the rules of combinatorial process have to be followed to avoid generating unwanted latches
- For Mealy Machines *`if-then-else`* statement is used to create the dependency between the current state, the input signal and output signal

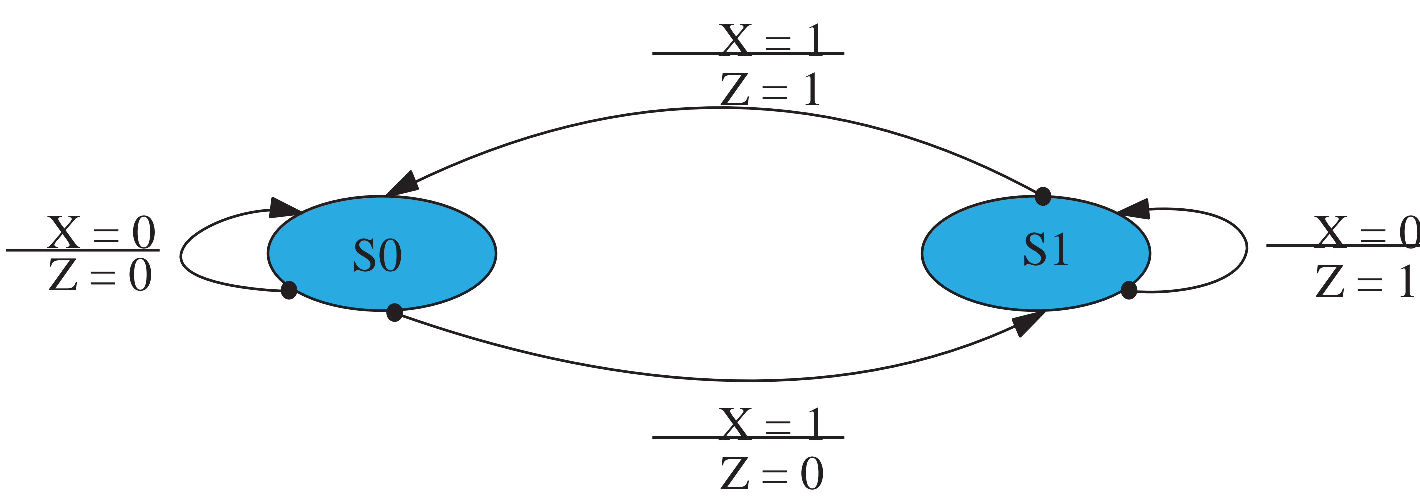
State Machine: Reset behavior

- Asynchronous reset: ensure that the state machine is always initialized to a known valid state, before the first active clock transition and normal operation commences

- No reset or a synchronous reset: there is no way to predict the initial value of the state register flip-flops. It could power up and become permanently stuck in an uncoded state.

FSM Style Descriptions - Example





Style A - Moore State Machine

Next State Logic

```
process (state, X)
begin
 case state is
  when S0 =>
     if(X='0') then
          next_state <= S0;
     else(X='1') then
          next_state <= S1;
    end if;
  when S1 =>
    if ….
          next_State <=..;
    ….
 end case;
end process;
```

Present State Logic

```
process (clk, rst)
begin
 if(rst = '1') then
      state <= S0;
 elsif (rising_edge(clk))
then
      state <= next_state;
 end if;
end process;
```

Output Logic

```
process (state)
begin
   case state is
      when S0 => Z <='0';
      when S1 => Z <='1';
end case;
end process;
```

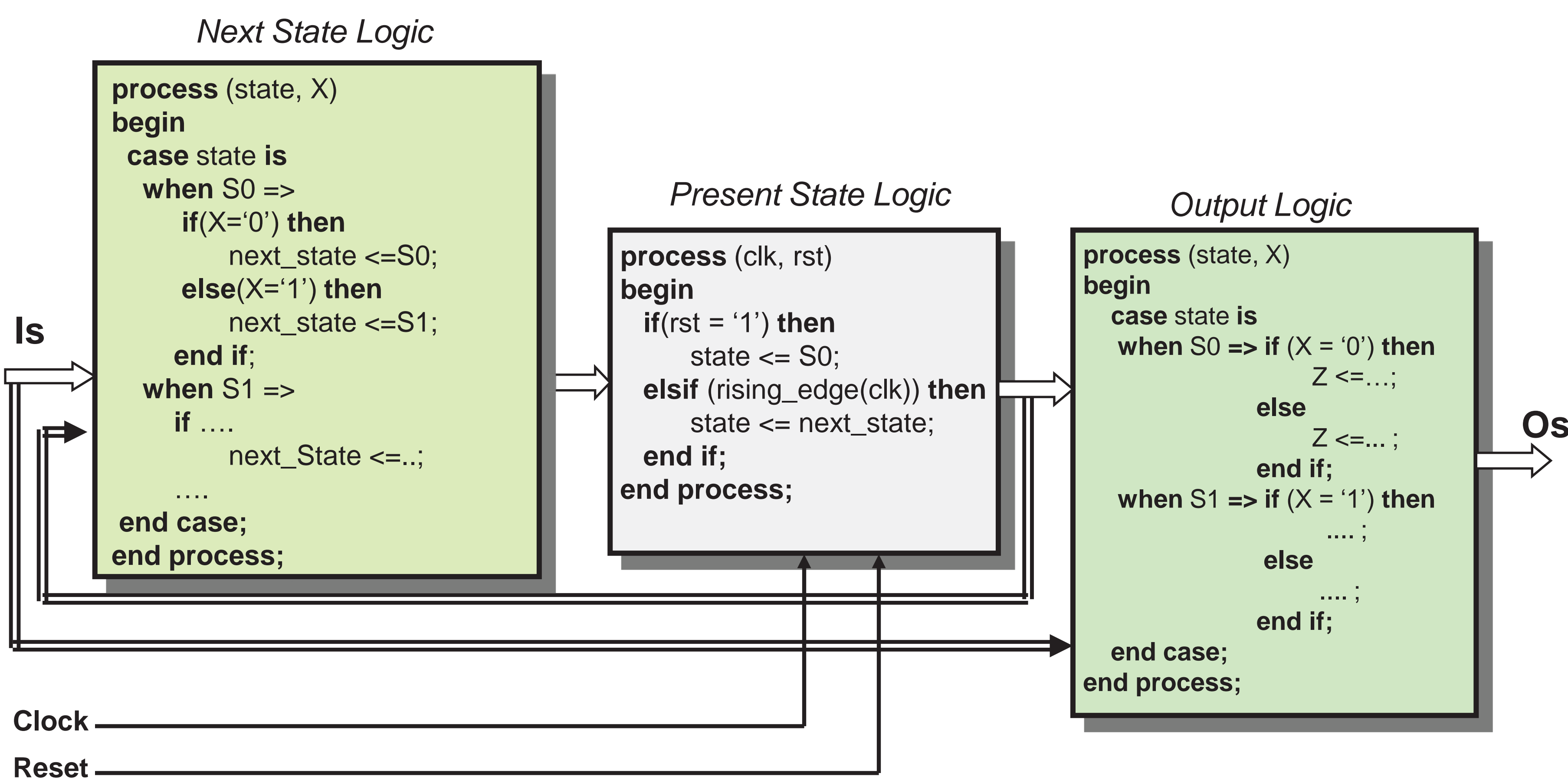
Is

Os

Clock

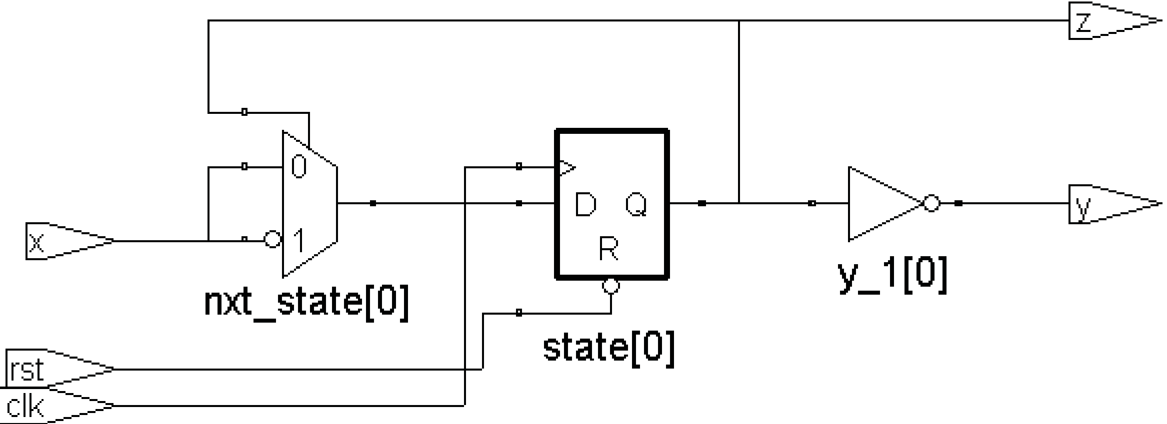
Reset

Style A - Mealy State Machine

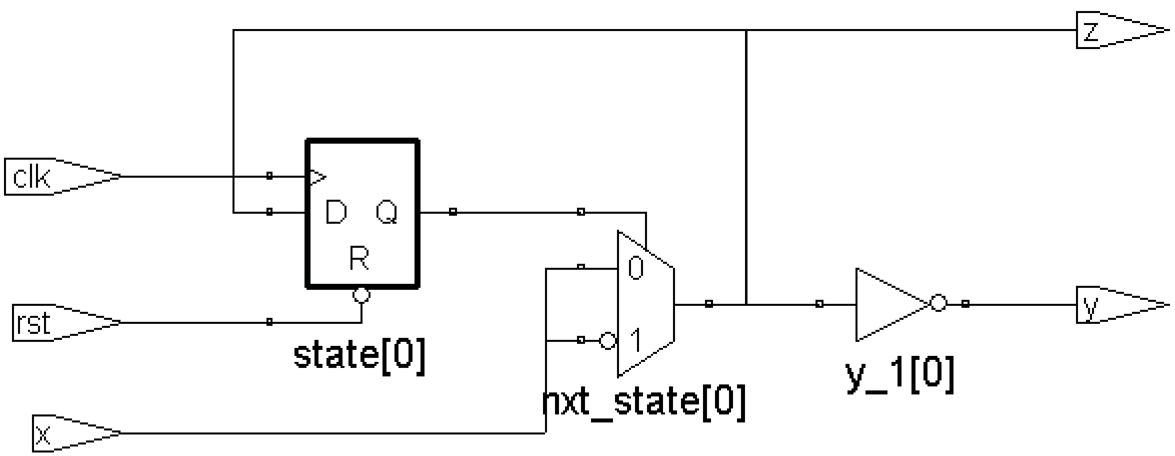


Style A - Synthesis

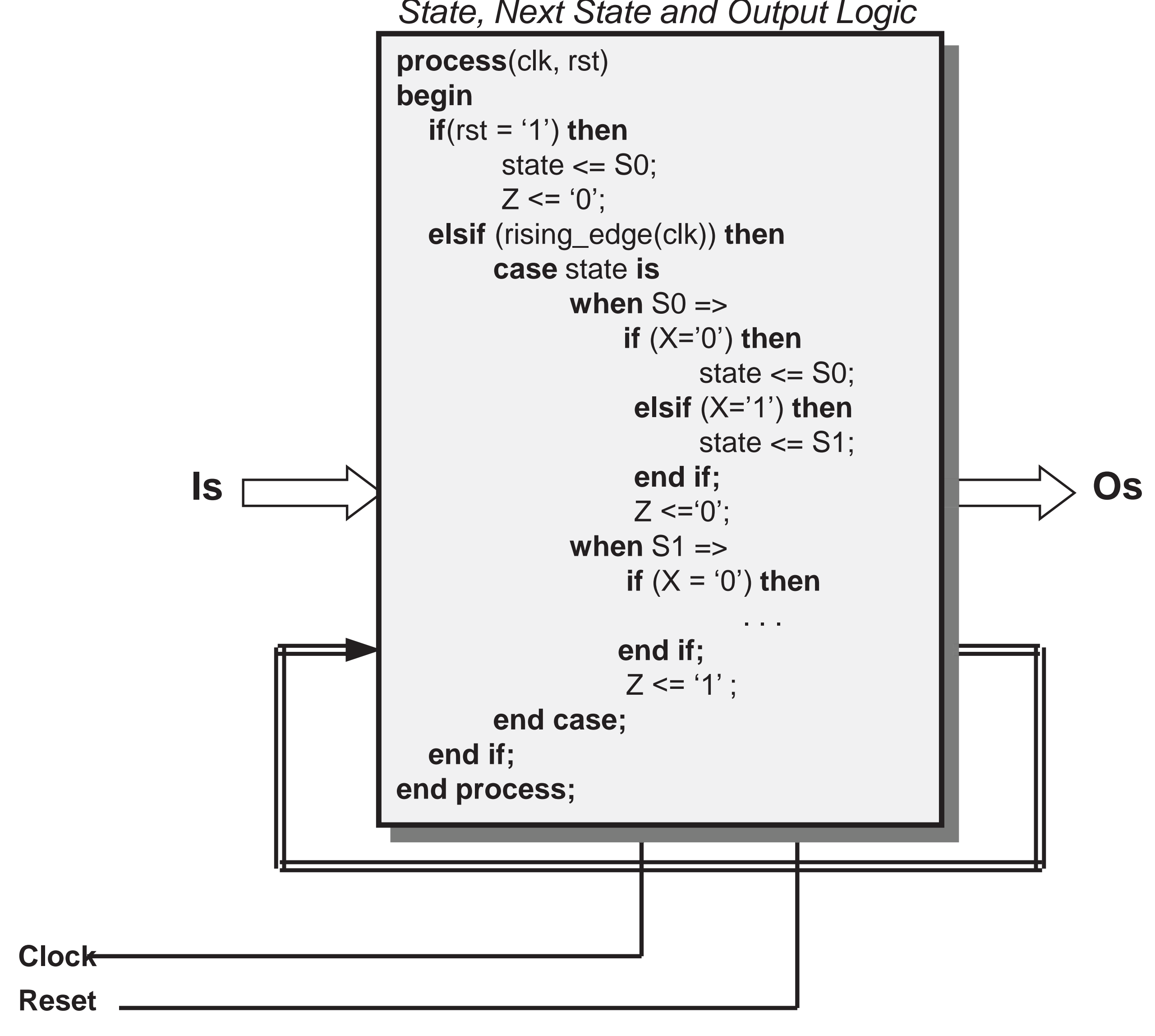
Moore



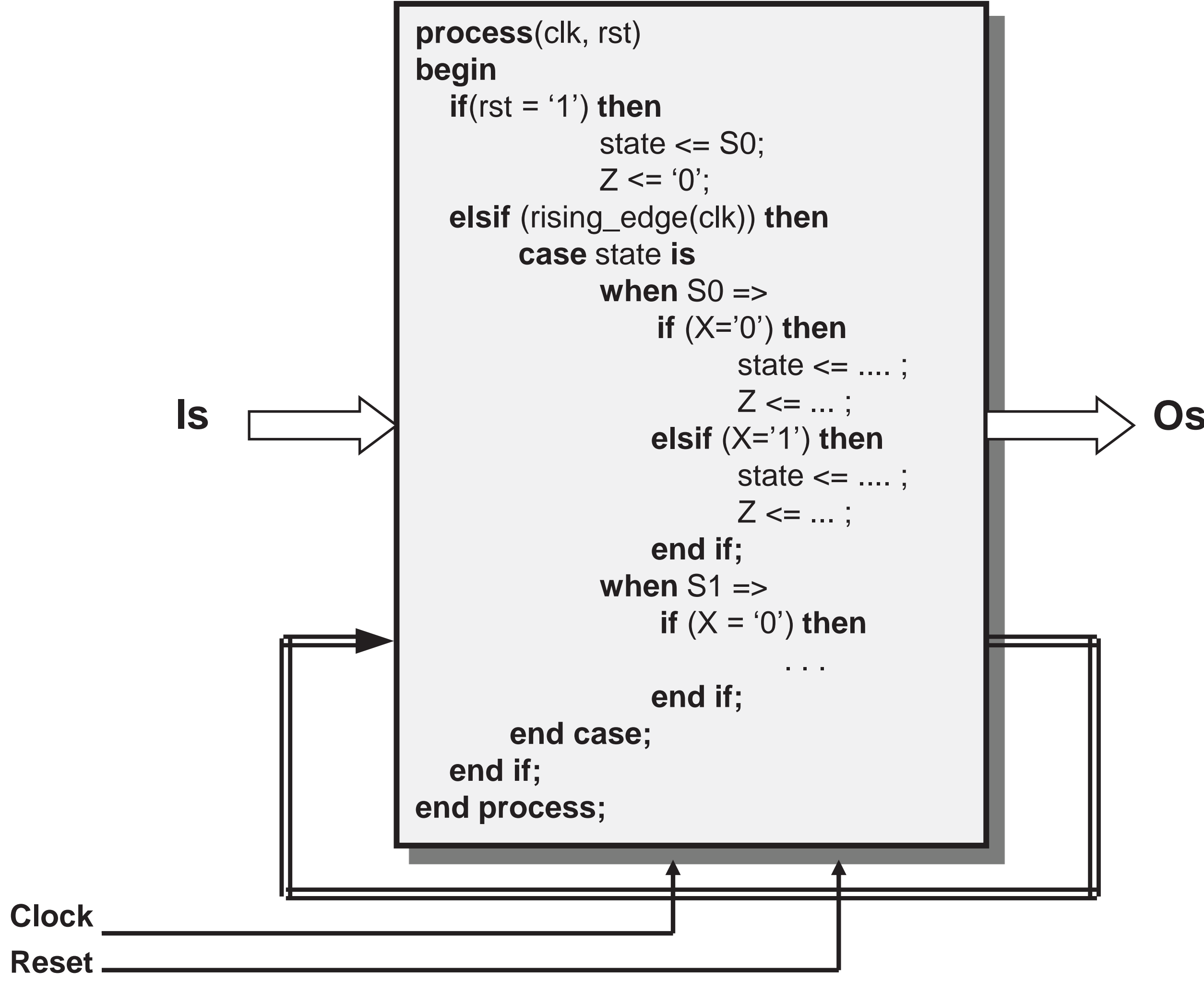
Mealy



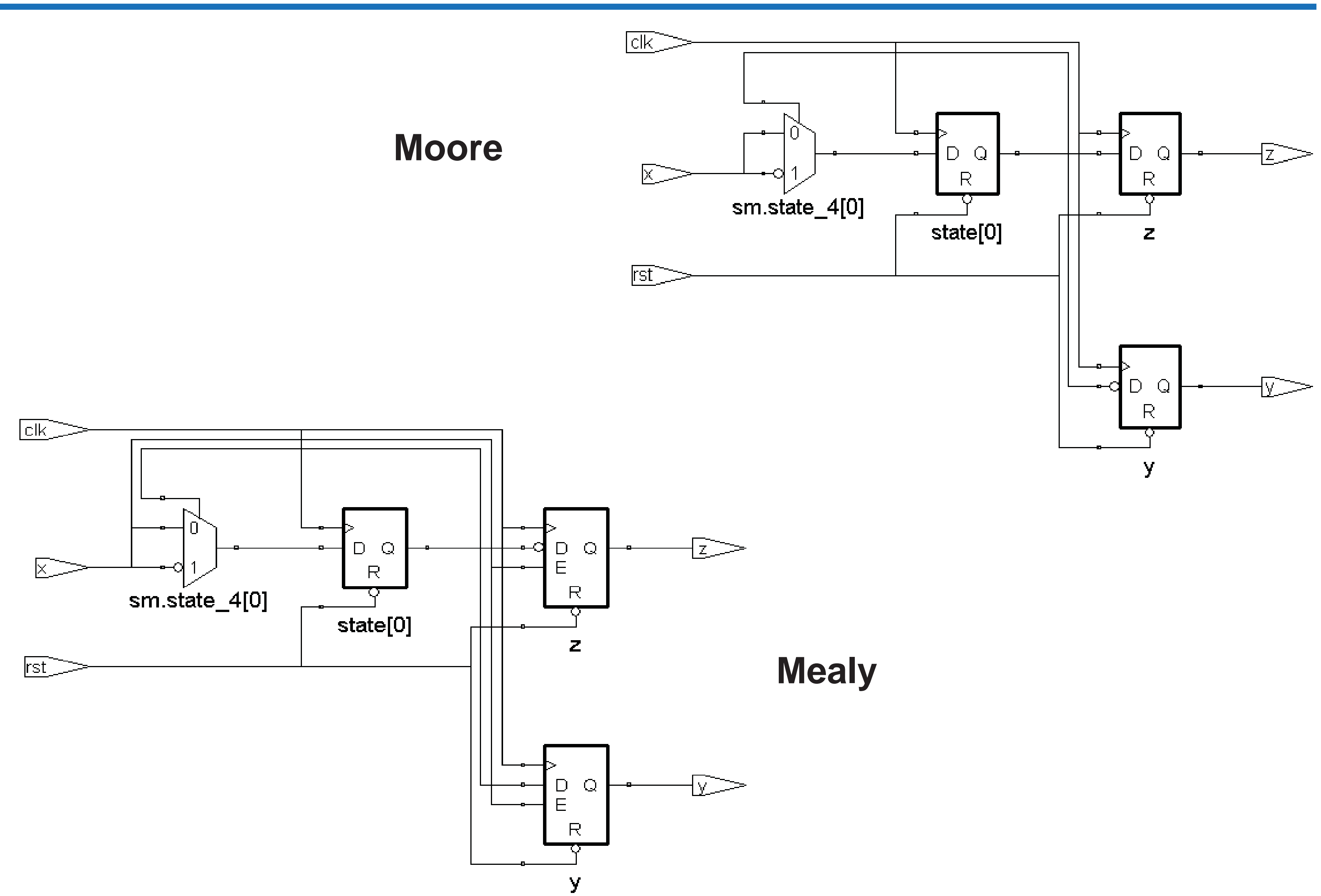
Style B - Moore State Machine



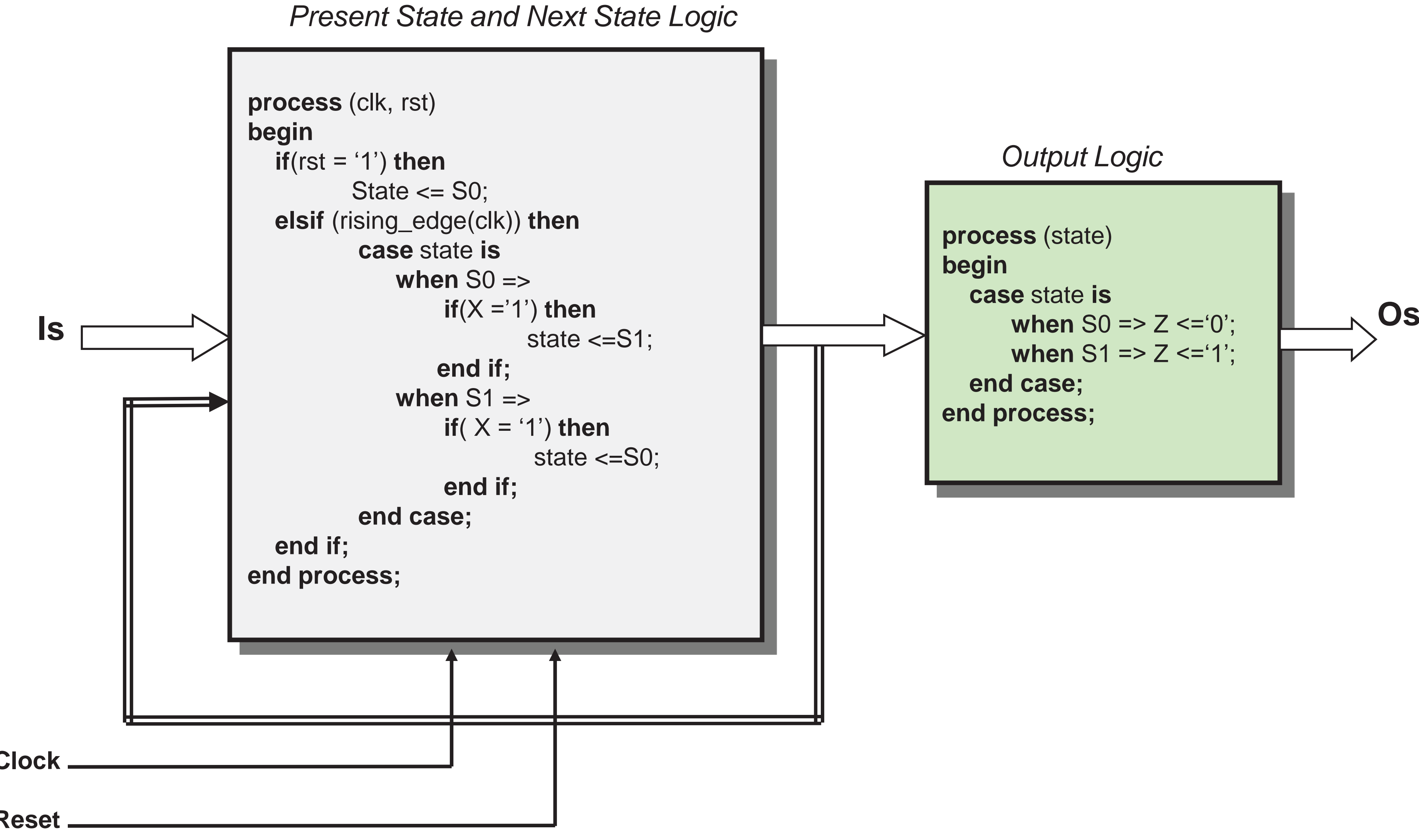
Style B - Mealy State Machine



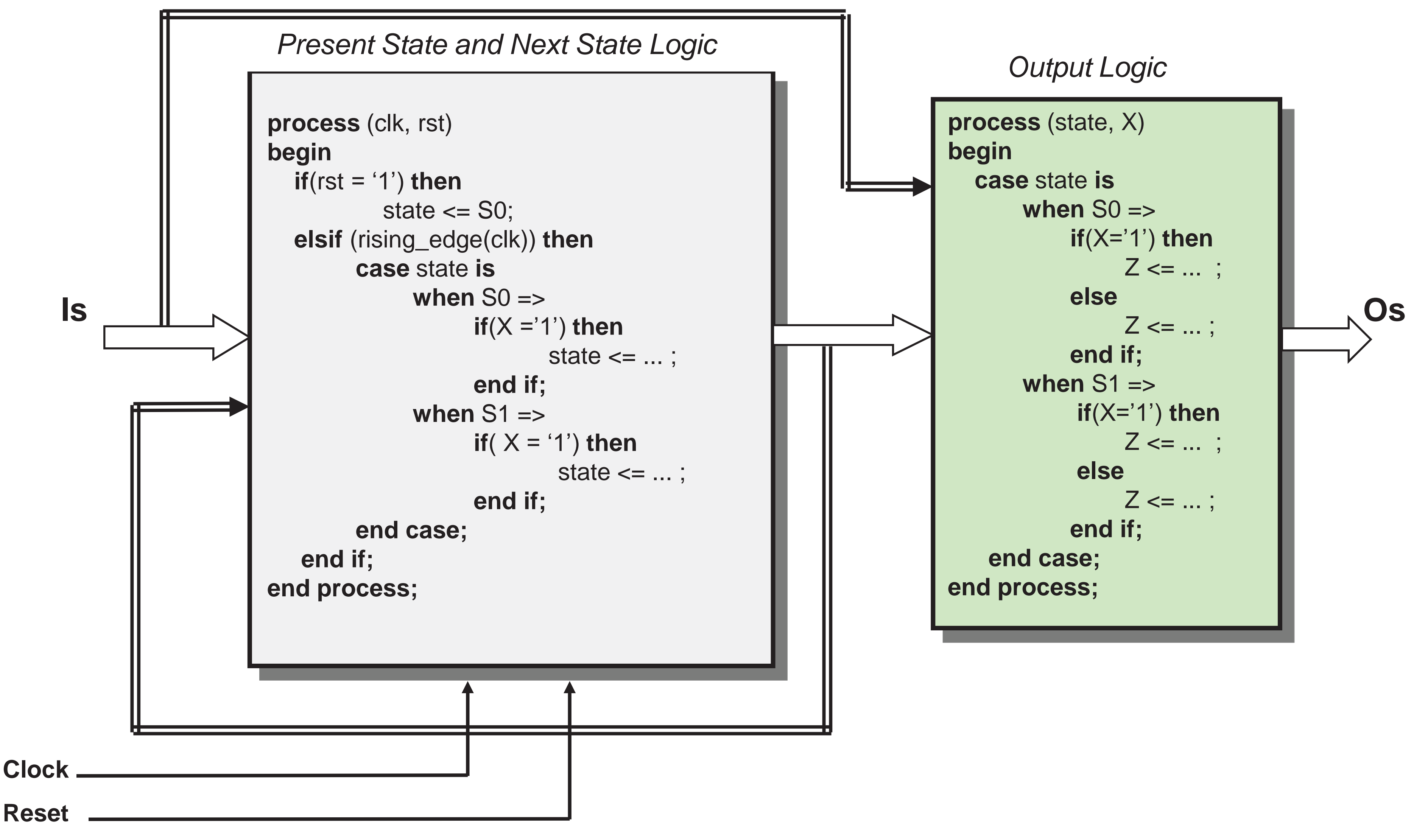
Style B - Synthesis



Style C - Moore State Machine

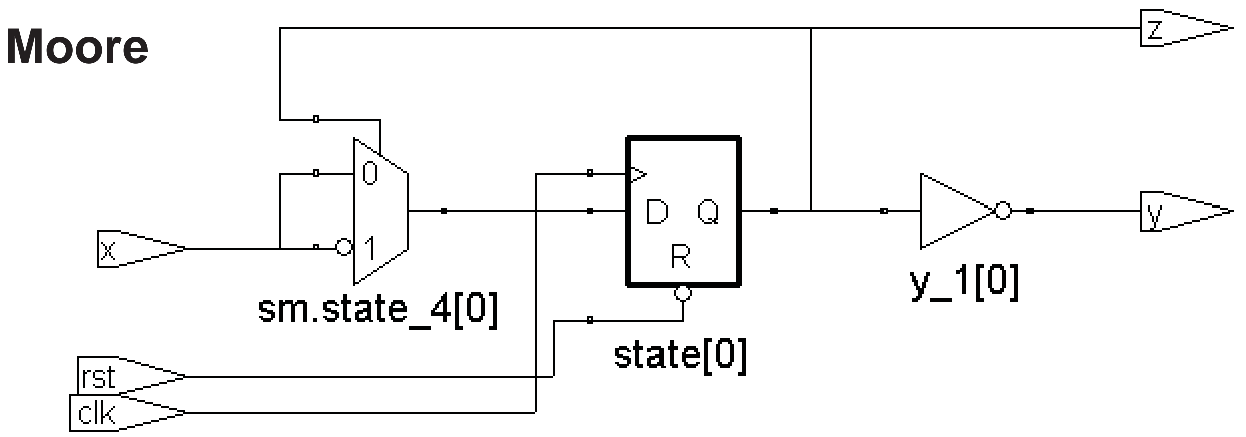


Style C - Mealy State Machine

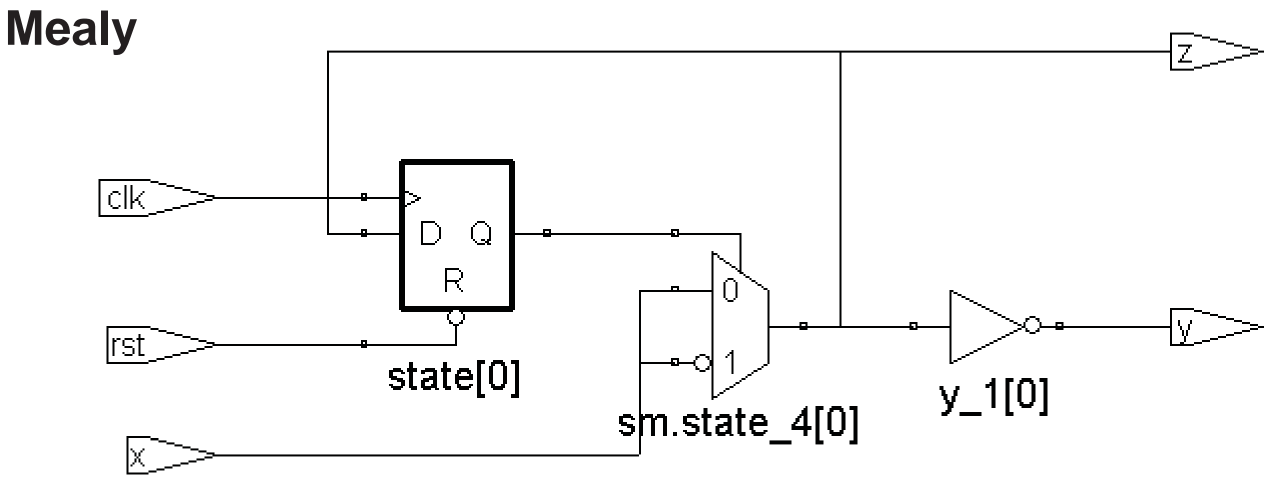


Style C - Mealy State Machine

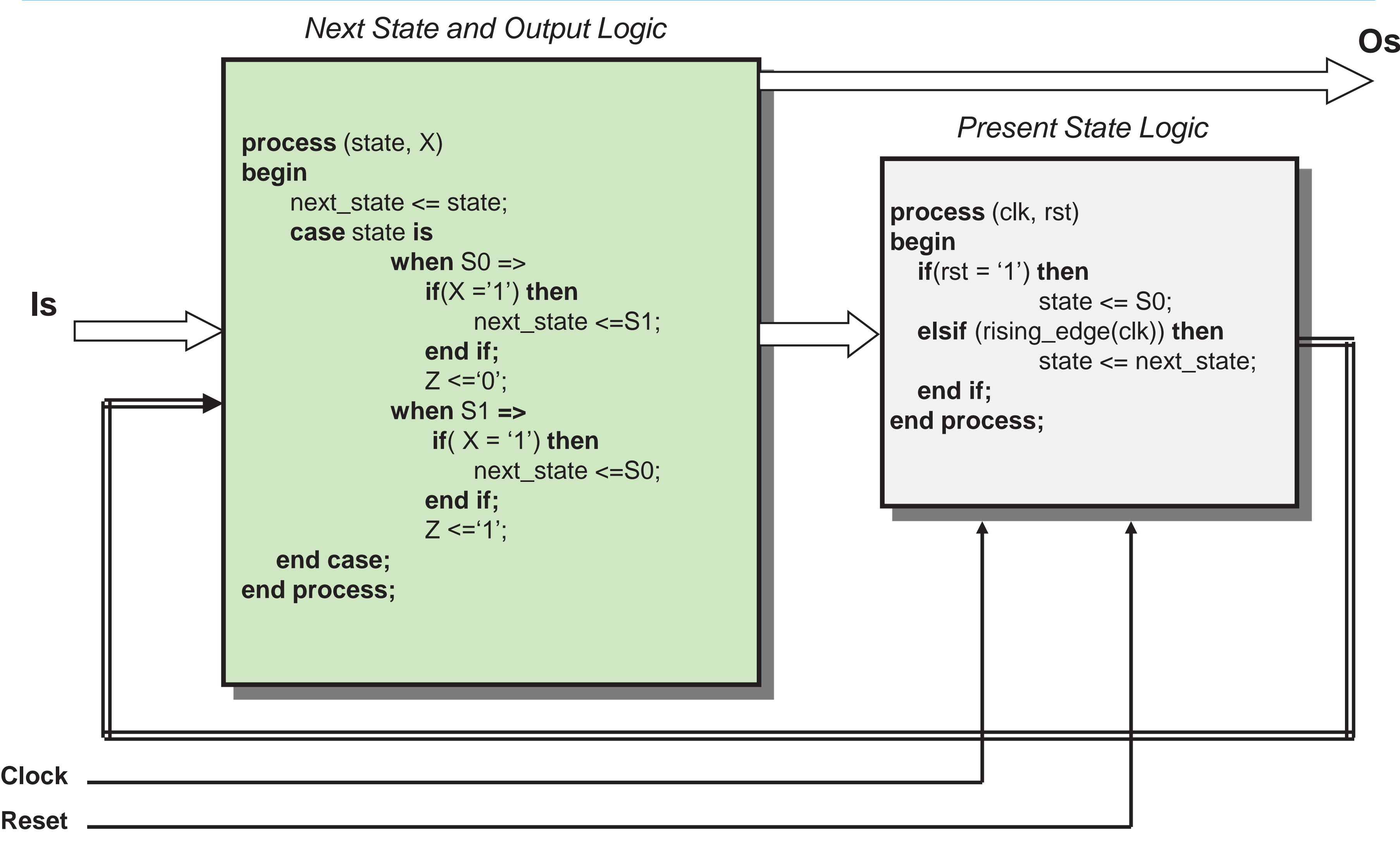
Moore



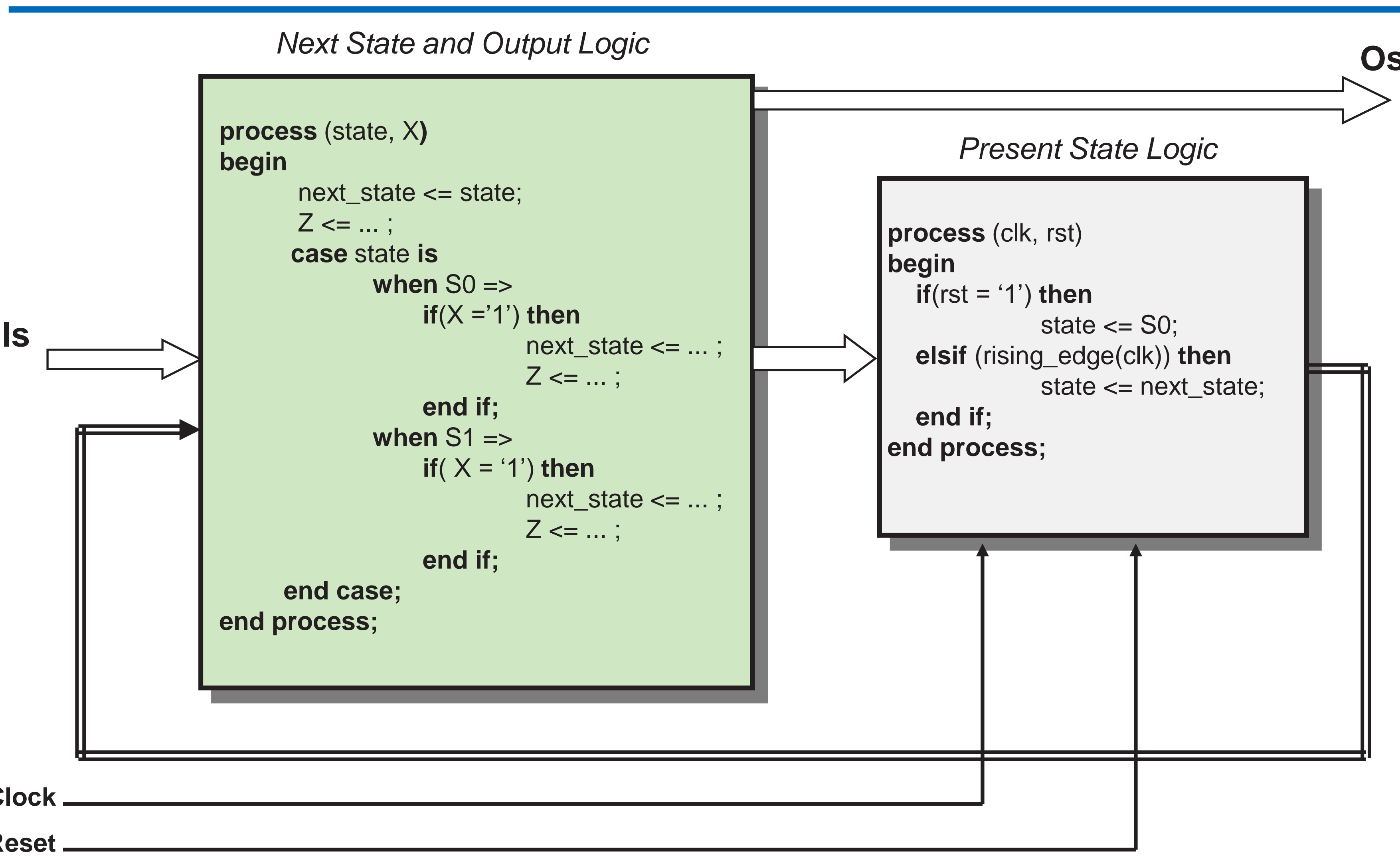
Mealy



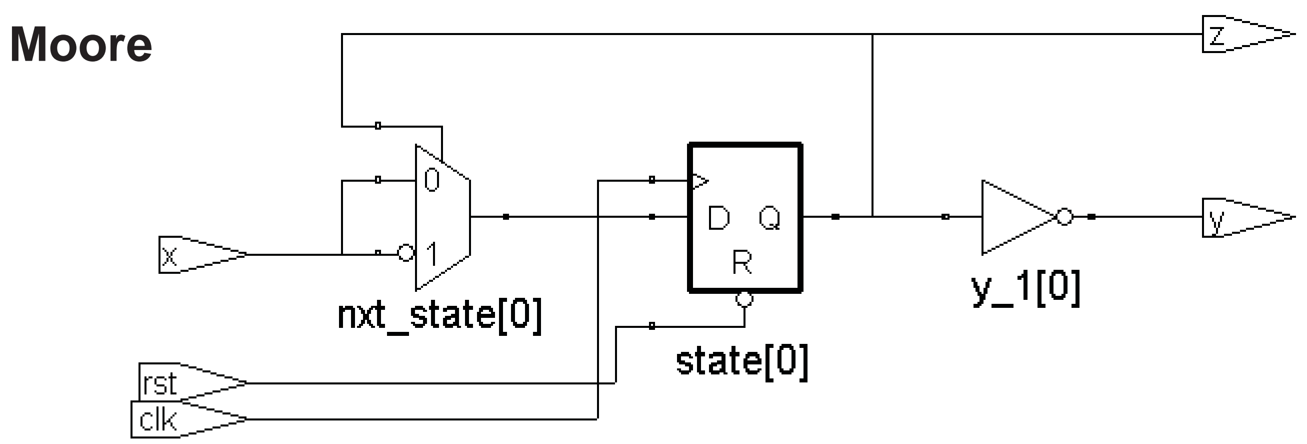
Style D - Moore State Machine

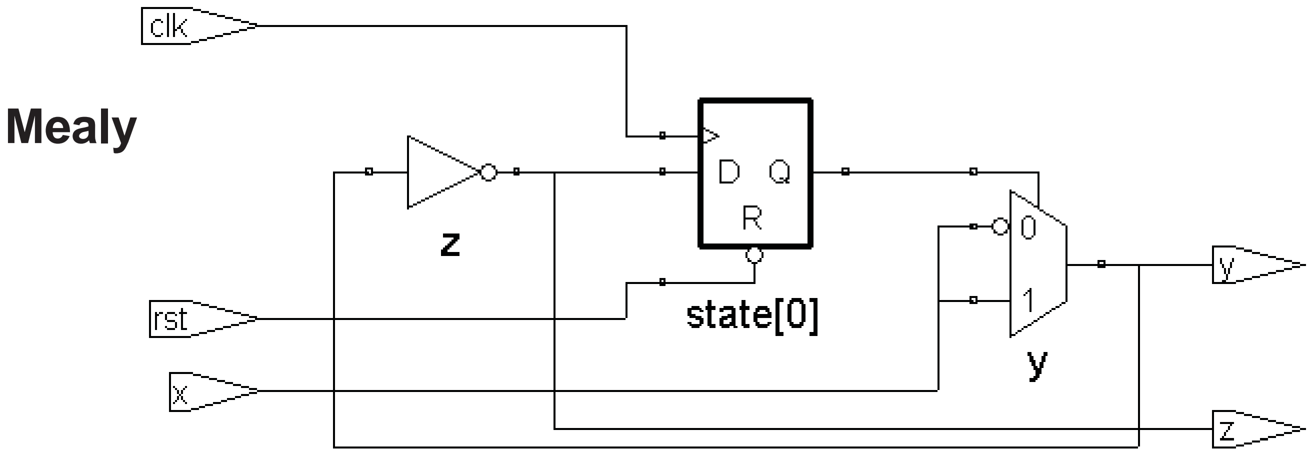


Style D - Mealy State Machine

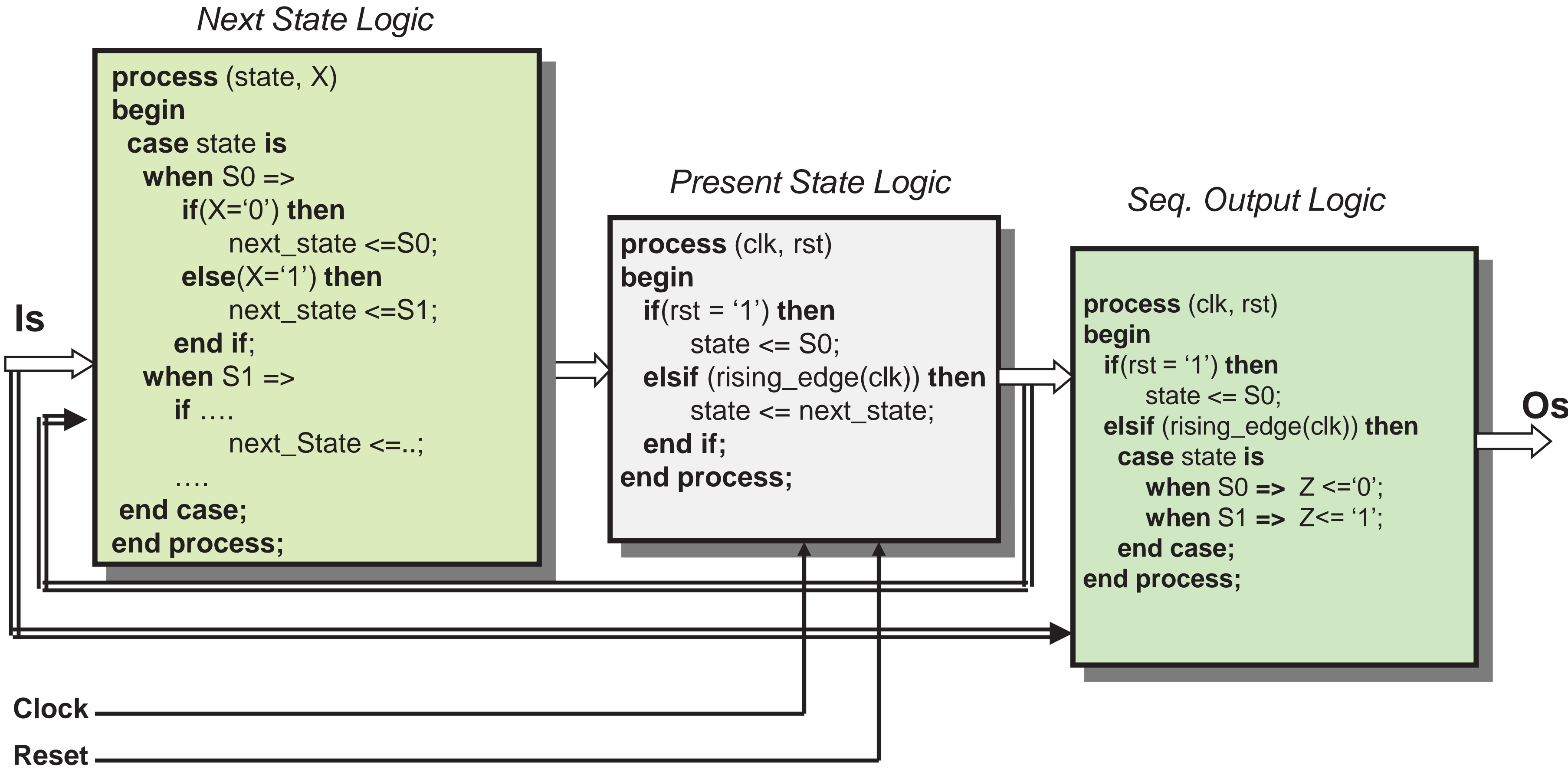


Style D - Mealy State Machine





Style E - Moore State Machine



Xilinx XST State Encoding Techniques

XST supports the following state encoding techniques:

- Auto-State Encoding
- One-Hot Encoding
- Gray State Encoding
- Compact State Encoding
- Johnson State Encoding
- Sequential State Encoding
- Speed1 State Encoding
- User State Encoding

XST State Encoding Techniques

FSM_ENCODING (FSM Encoding Algorithm) Algorithm VHDL Syntax

Declare the attribute as follows:

```
attribute fsm_encoding: string;
```

Specify as follows:

```
attribute fsm_encoding of {entity_name|signal_name }: {entity
|signal} is "{auto|one-hot|compact|sequential|gray|johnson|
speed1|user}";
```

The default is **auto**

syn_encoding - Synplify

Override the default FSM Compiler encoding for a FSM

Possible values:

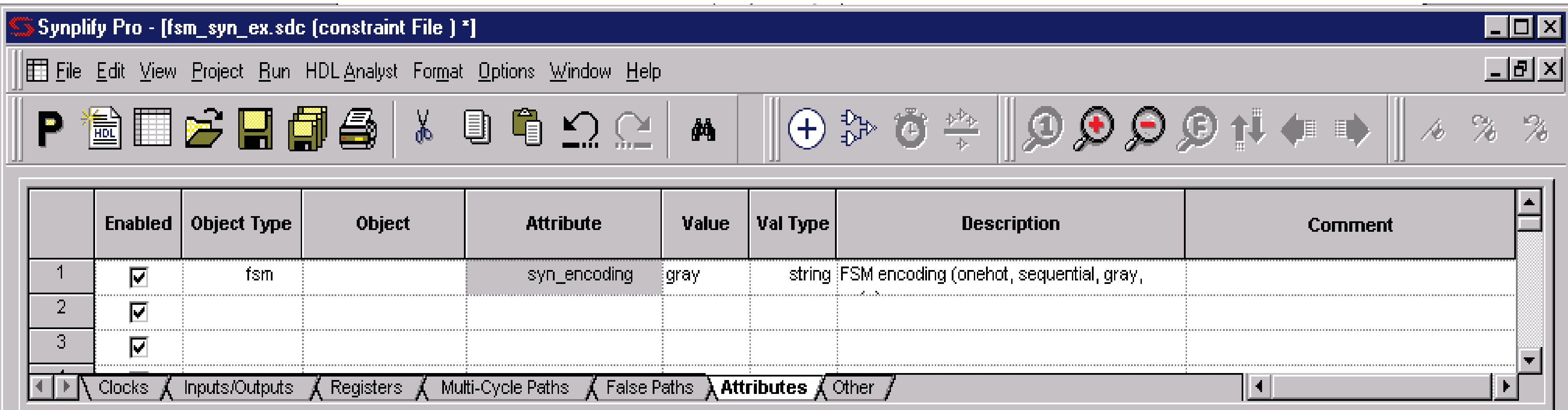
- **Default**: assign an encoding style based on the number of states:
  - Sequential for 0 - 4 enumerated types
  - One-hot for 5 - 24 enumerated types
  - Gray for > 24 enumerated types
- **Sequential**: 000 001 010 011 100 101 110 111
- **One-hot**: 00000001 00000010 00000100 . . .
- **Gray**: 000 001 011 010 110 111 101 100
- **Safe**: default encoding + reset logic to a known state

syn_encoding - Synplify

Syntax (source code)

```
-- declare the (state-machine) enumerated type
type my_state_type is (SEND, RECEIVE, IGNORE, HOLD, IDLE);
-- declare signals as my_state_type type
signal nxt_state, current_state: my_state_type;
-- set the style encoding
Attribute syn_encoding: string;
attribute syn_encoding of current_state: signal is one-hot;
```

syn_encoding in SCOPE



type_encoding_style – Precision

Possible values:

- Binary
- Onehot
- Twohot
- Gray
- Random

```
-- Declare the (state-machine) enumerated type
type my_state_type is (SEND, RECEIVE, IGNORE, HOLD, IDLE);
-- Set the TYPE_ENCODING_STYLE of the state type
attribute TYPE_ENCODING_STYLE of my_state_type:type is ONEHOT;
```

type_encoding - Precision

type_encoding allows ***to fully*** control the state encoding hard code the state code in the source code

```
-- Declare the (state-machine) enumerated type
type my_state_type is (SEND, RECEIVE, IGNORE, HOLD, IDLE);
-- Set the type_encoding attribute
attribute type_encoding of my_state_type:type is
                    ("0001","01--","0000","11--","0010");
```

State Table

	bit3	bit2	bit1	bit0
SEND	0	0	0	1
RECEIVE	0	1	-	-
IGNORE	0	0	0	0
HOLD	1	1	-	-
IDLE	0	0	1	0

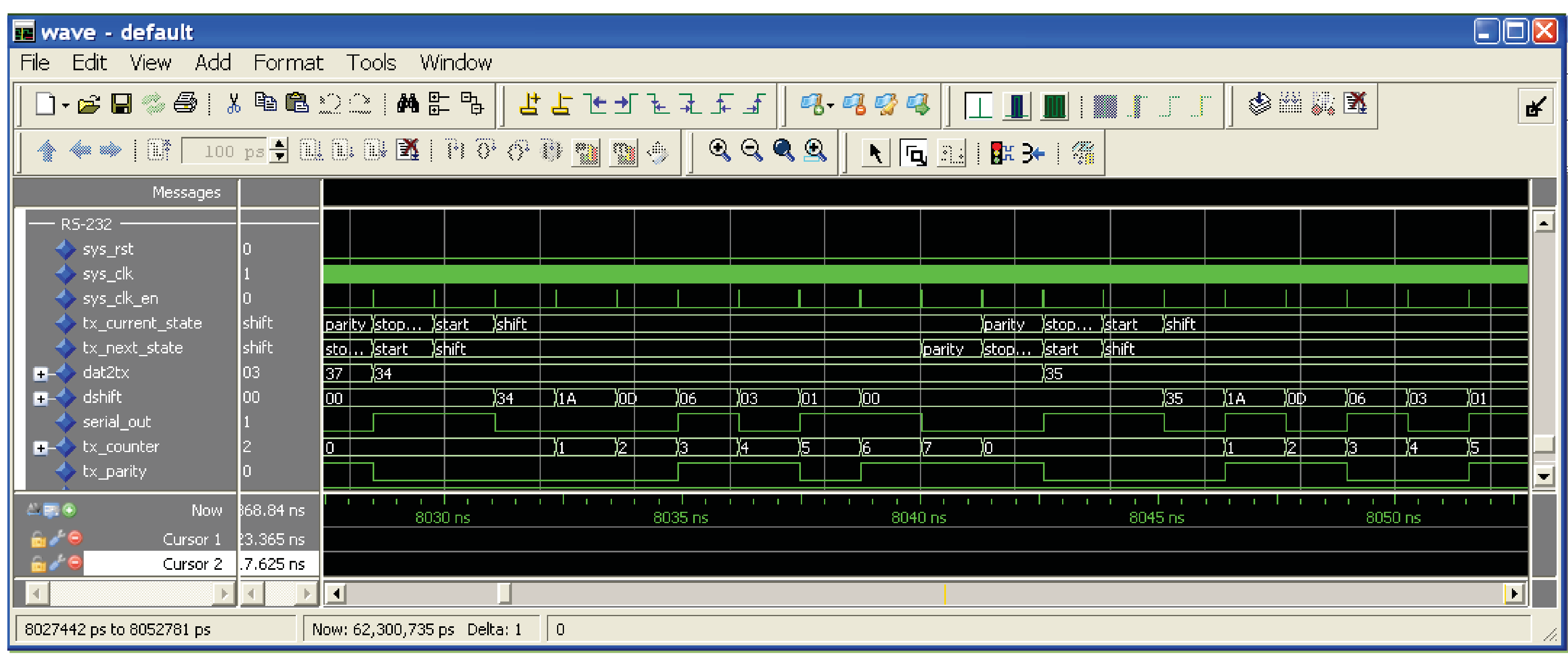
Note: LeonardoSpectrum allows to use '-'. It can be used to reduce the size of the circuit

State Machine Coding: Residual States

- If one-hot state coding is not used, the maximum number of states is equal to 2**N, where N is the vector length of the state vector

- In state machines where not all the states are used, there are three options:
  - Let chance decide what is to happen if the machine goes to an undefined state
  - Define what is to happen if the state machine goes to an undefined state by ending the case statement with `when others`
  - Define all the possible states in the VHDL code

State Machine Coding: Simulation



State Machine Coding: Synthesis

Synplify FSM Report:

```
25 Extracted state machine for register tx_current_state
26 State machine has 5 reachable states with original encodings of:
27    00001
28    00010
29    00100
30    01000
31    10000
32 @END
```

State	tx_current_state[0]	tx_current_state[1]	tx_current_state[2]	tx_current_state[3]	tx_current_state[4]
idle	0	0	0	0	1
start	0	0	0	1	0
shift	0	0	1	0	0
parity	0	1	0	0	0
stop_1bit	1	0	0	0	0

Transitions | RTL Encodings | Mapped Encodings

State Machine Coding: Synthesis

XST FSM Report:

```
=============================================
*              Advanced HDL Synthesis               *
=============================================
Analyzing FSM <FSM_0> for best encoding.
Optimizing FSM <rs232_tx/tx_current_state/FSM> on signal
<tx_current_state[1:3]> with sequential encoding.
----------------------------
 State        | Encoding
----------------------------
 idle         | 000
 start        | 001
 shift        | 010
 parity       | 011
 stop_1bit | 100
----------------------------
```

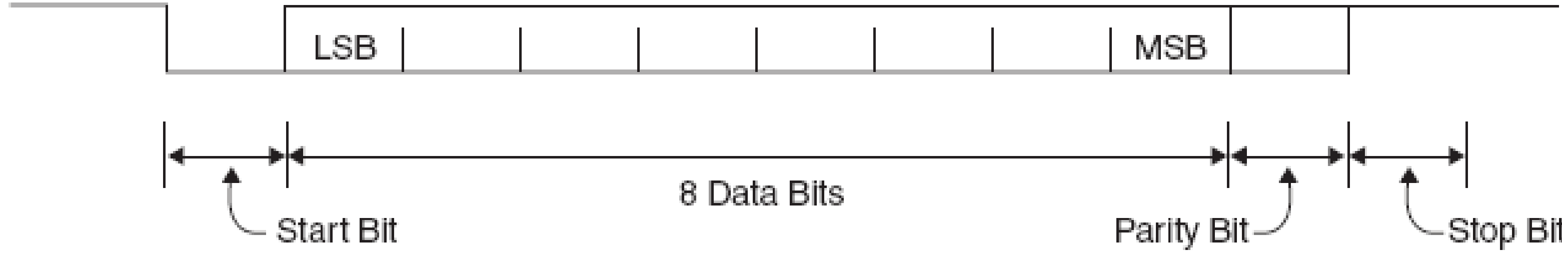
FSM Example

Realizar la descripción en VHDL de un receptor y transmisor serie tipo RS-232. El dato recibido y el dato a transmitir debe tener el siguiente formato:

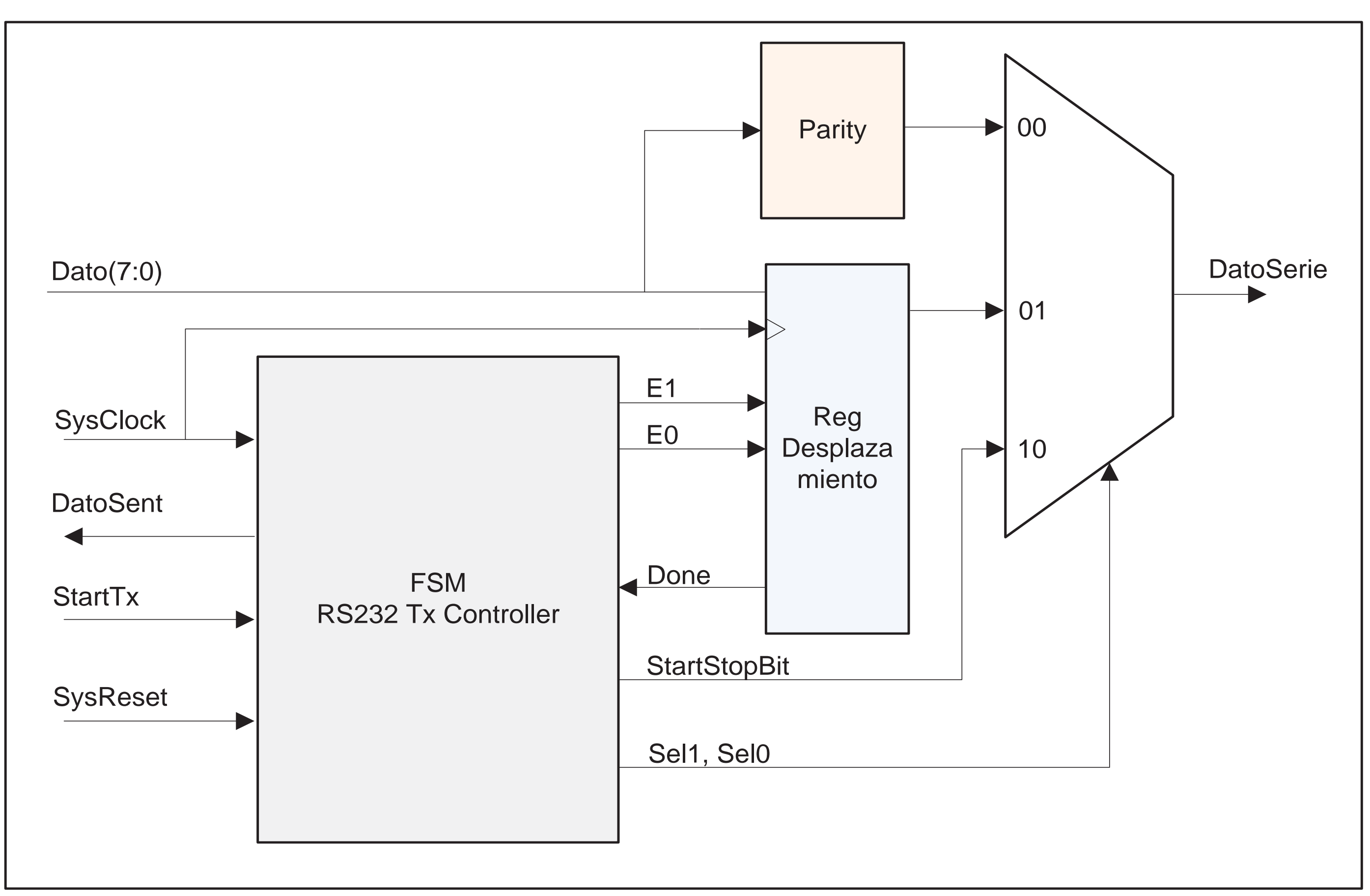
- 1 bit de start
- 8 bits de datos
- Paridad programable: paridad/no paridad, par/impar
- 1 bit de stop
- Frecuencia de transmisión por defecto es de 9600 Bauds, pero el código debe permitir también otras frecuencias como: 4800, 38400, 115200 Bauds.

FSM Example

RS-232 Tx format



FSM Example



Subprogramas

Subprograms

- Sequence of statements that can be executed from multiple locations in the design
- Provide a method for breaking large segments of code into smaller segments
- Types of subprograms:
  - Functions: used to calculate and return one value
  - Procedures: used to define an algorithm which affects many values or no values

Subprograms

Function

- Return only one value
- Executed in zero simulation time
- Can be invoked from an expression

Procedure

- Return zero or several values
- Can or can not be executed in zero simulation time (wait statement)

Function

- Provides a method of performing simple algorithms
- Produce a single return value
- Invoked by an expression
- Can not modify parameters. Only mode allowed:

  IN (read only)
- `wait` statements are NOT permitted in a function

Function Syntax

```
function function_name(parameters_list)return return_type is
        function_declarations;
begin
        sequential_statements;
        return simple_value;
end [function] [function_name];
```

Behavioral part

Function Parts - Definitions

```
[function_declarations]
-- the only class declarations allowed are:
```

Variables	Constants

```
<parameters_list>
```

- mode *in* only
- signals only

```
<sequential_statements>
```

- sequential instructions only
- always has to end with a *return* statement

Function Key Concepts

- ⇨ Variables can be used to accumulate results or hold intermediate values. But, variables declared within the function do not keep the value between executions
- ⇨ Initial values assigned to variables are synthesizable
- ⇨ Function can have more than one return, however just one is executed
- ⇨ It is useful to declare the parameters as undefined for a general use of the function

Function Example 1

```
package my_pack is
function max(a,b: in std_logic_vector)
                              return std_logic_vector;
end package;

package body of my_pack is
function max(a,b: in std_logic_vector)
                                return std_logic_vector is
  begin
      if a> b then
           return a;
      else
           return b;
       end if;
   end function;
end package body;
```

Function Example 1

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

package my_pack is
function max(a,b: in std_logic_vector)
                                     return std_logic_vector;
end package;

package body my_pack is
function max(a,b: in std_logic_vector)
                                   return std_logic_vector is
  begin
      if a> b then
           return a;
      else
           return b;
       end if;
   end function;
end package body;
```

Function Usage - Example 1

```
-- use of the function declared in the package
use work.my_pack.all;
...
entity example is
        port(...);
end;
architecture beh of example is
begin
     ...
     q_d1d2 <= max(d1,d2);                    -- concurrent call
process(d3,d4)
begin
     q_d3d4 <= max(b=>d3, a=>d4); -- sequential call
end process;
end;
```

Package holding function declaration and behavior

Parameter passing list: by position

Parameter passing list: by name

Note: d1, d2, data and g have to be std_logic_vector type

Function Example 2

```
-- convert Boolean to Bit
  function bool_to_bit(b:Boolean) return bit is
    begin
      if b then
        return '1';
      else
        return '0';
      end if;
  end function bool_to_bit;
```

Function Example 3

```
function sehctam_tnuoc (a, b: bit_vector) return natural is
  variable va : bit_vector(a'length-1 downto 0):= a;
  variable vb : bit_vector(b'length-1 downto 0):= b;
  variable cnt: natural := 0;
begin
  assert va'length = vb'length
    report ”the vectors have different size”
       severity failure;
  for i in va'range loop
      if va(i) = vb(i) then
          cnt := cta + 1;
      end if;
  end loop;
  return cnt;
end funciton;
```

Function Example 4

```
-- FUNCTION BIN2GRAY(VALUE)
-- Used to convert Binary  into Gray-Code
function bin2gray(value:std_logic_vector)return
                                            std_logic_vector is
variable grayval:std_logic_vector((value'length-1) downto 0);
variable result: std_logic_vector((value'length-1) downto 0);
 begin
    l_oop:for i in 0 to (value'length-1) loop
       if(i=value'length-1) then
          grayval(i) := value(i);
       else
          grayval(i) := value(i+1) xor value(i);
       end if;
    end loop l_oop;
    result := grayval;
  return result;
end function  bin2gray;
```

Function Example 5

```
-- integer to bit_vector conversion
-- result'low is the lsb
function to_vector (nbits: positive; int_val: natural)
                                         return bit_vector is
variable m1: natural := 0;
variable result:bit_vector(nbits-1 downto 0):=(others => '0');
  begin
    m1 := int_val;
    for j in result'reverse_range loop
      if (m1/mod2) = 1 then
        result(j) := '1';
      else
        result(j) := '0';
      end if;
      m1 := m1/2;
    end loop m1;
    return result;
end function to_vector;
```

Procedure

- Provides a method of performing complex algorithms
- May produce multiple return (output) values
- Invoked by a statement
- Can affect input parameters
- Parameter's mode:
  - IN: read only
  - OUT: write only
  - INOUT: read/write

Procedure Syntax

```
procedure proced_name (parameter_list) is
        [procedure_declarations]
begin
        sequential_statements;
end [procedure] [proced_name];
```

Behavioral part

Procedure Definitions

```
<procedure_declarations>
-- the only class declarations allowed are:
```

Variables	Constants
Functions	Types

<parameter_list>

- *in* mode: to receive values
- *out* mode: to return values
- *inout* mode: to receive/return values
- Parameters can be signals, variables and constants. By default *in* = constant, *out* = variable
- Can return as many values as are needed using the *out* parameters

Procedure Calls

- Can be used in sequential or concurrent statements areas
  - Concurrent calls are activated by changes in any signal associated with a parameter of mode IN or INOUT
  - Sequential calls is executed whenever the procedure is encounter during the execution of the sequential code
- Passing parameter list:
  - Association by name
  - Association by position

Procedure Example 1

```
procedure calc(a,b: in integer;
          avg, max: out integer) is
begin
        avg<= (a+b)/2;
        if a>b then
                max<=a;
        else
                max<=b;
        end if;
end calc;
```

Procedure Example 2

```
-- sum, cout as variables
procedure full_adder (a,b,c: in bit; sum, cout: out bit) is
  begin
        sum := a xor b xor c;
        cout := (a and b) or (a and c) or (b and c);
end full_adder;
```

Procedure Example 3

```
-- sum, cout as variables
procedure full_adder(a,b,c:in bit; signal sum, cout: out bit)is
  begin
        sum <= a xor b xor c;
        cout<= (a and b) or (a and c) or (b and c);
end full_adder;
```

Procedure Example 3

```
entity adder4 is
        port (  a,b : in bit_vector(3 downto 0);
                cin : in bit;
                sum : out bit_vector(3 donwto 0);
                cout: out bit);
end entity;
architecture behavioral of adder4 is
begin
   process (a, b, cin)
        variable result: bit vector(3 downto 0);
        variable carry : bit:
   begin
       full_adder(a(0), b(0), cin, result(0), carry);
       full_adder(a(1), b(1), carry, result(1), carry);
       full_adder(a(2), b(2), carry, result(2), carry);
       full_adder(a(3), b(3), carry, result(3), carry);
       sum <= result;
       cout <= carry;
   end process;
end behavioral;
```

Procedure Example 4

```
architecture rtl of ex2 is
procedure calc (a,b: in integer; avg, max: out integer) is
begin
        avg:= (a+b)/2;
        if a>b then
            max:=a;
        else
            max:=b;
        end if;
end calc;
begin
        calc(d1,d2,d3,d4);        -- concurrent call. ok?
        process(d3,d4)
         variable a,b: integer;
        begin
            calc(d3,d4,a,b);      -- sequential call. ok?
            q3<=a;
            q4<=b;
                ....
        end process;
end;
```

Procedure Example 4

```
architecture rtl of ex3 is
procedure calc(a,b:in integer; signal avg, max:out integer)is
begin
        avg<= (a+b)/2;
        if a>b then
             max<=a;
        else
             max<=b;
        end if;
end calc;
begin
        calc(d1,d2,d3,d4);       -- concurrent call. ok?
        process(d3,d4)
         variable a,b: integer;
        begin
              calc(d3,d4,a,b); -- sequential call. ok?
              q3<=a;
              q4<=b;
              ....
        end process;
end;
```

Procedure Example 5

```
procedure echo(constant str : in string) is
        variable l: line;
  begin
       write(l, string'("<"));
       write(l, time'(now));
       write(l, string '("> "));
       write(l, str);
       writeline(output,l);
end echo;
```

Procedure Example 6

```
------------------------------------------------------
 --
------------------------------------------------------
  procedure echo_now(constant str : in string) is
    variable l : line;
  begin
    write(l, str);
    write(l, string'(" @ <--"));
    write(l, time'(now));
    write(l, string'("--> "));
    writeline(output, l);
  end echo_now;;
```

Subprograms Overloading

When two or more subprograms have the same name but different types in the list of parameters

```
function max (a, b: in std_logic_vector)
             return std_logic_vector);
function max (a, b: in bit_vector)
             return bit_vector);
function max (a, b: in integer)
             return integer;
```

Package

Package Syntax

```
package <package_name> is
        [subprograma_declarations];
        [constant_declarations];
        [type_declarations];
        [component_declarations];
        [attribute_declarations];
        [attribute_specifications];
end <package_name>;

package body <package_name> is
        [subprogram_bodies];
        [internal_subprogram_declarations];
        [internal_constant_declaration];
        [internal_type_declaration];
end <package_name>;
```

Package

- A package declaration contains a set of declarations that may possible be shared by many design units. Common declarations are constants, functions, procedures and types
- A package body contains the hidden details of a package, such as the behavior of the subprograms
- Items declared in a package can be accessed by other design unit by using the library and use clauses

Declaring Subprograms in Packages

```
-- package declaration
package my_package is
        constant word_size: positive:=16;
        subtype word is bit_vector (word_size-1 downto 0);
        subtype address is bit_vector(address_size-1 downto 0);
        type status_value is
               (halted, idle, fetch, mem_read, mem_wr, io_read);
        function int_to_bit_vector (int_valor: integer)
                                         return bit_vector;
end my_package;

-- package body
package body my_package is
        function int_to_bit_vector (int_vector: integer)
                                   return bit_vector is
                begin
                ...
                ...
        end int_to_bit_vector;
end my_package;
```

Function Declaration

Function Body

Package .vhd File

```
library ieee;
use ieee.std_logic_1164.all;
-- package declaration
package my_pack is
      constant width: integer := 8;
      function max(a, b: in std_logic_vector)
                              return std_logic_vector;
end;
-- package body
package body my_pack is
function max(a, b: in std_logic_vector)
                        return std_logic_vector is
       begin
           if a> b then
                 return a;
           else
                 return b;
           end if;
       end;
end;
```

Package Usage

```
library ieee;
use ieee.std_logic_1164.all;
use work.my_pack.all;        -- make the package available

entity package_test_2 is
  port(d1,d2,d3: in std_logic_vector(width-1 downto 0);
              z: out std_logic_vector(width-1 downto 0);
              q: out std_logic_vector(width-1 downto 0));
end;

architecture beh of package_test_2 is
begin
  q <= max(d1,d2);      -- concurrent call of the function

 process(d2,d3)
 begin
   z <= max(d3, d2);    -- sequential call of the function
 end process;
end;
```

Package Common Usage

```
package my_pack is
-- constants declarations
constant bus_width : natural:=2;
-- component declaration
component bcd_7segm is
      port(  led_outs : out std_logic_vector(6 downto 0);
             bcd_inps : in  std_logic_vector(3 downto 0)
             );
end component;

component mux is
      port(  inf_inputs: in  std_logic_vector(3 downto 0);
             sel       : in  std_logic_vector(1 downto 0);
             output    : out std_logic);
end component;
end package my_pack;
```

Package Common Usage

```
library ieee;
use ieee.std_logic_1164.all;
use work.my_pack.all;       -- make the package available

entity bcd_7seg_top is
 port ( . . .
       . . . );
end;

architecture beh of package_test_2 is
begin

U1: bcd_7segm
       port map ( . . . );
U2: mux
       port map ( . . . );
 . . .
end beh;
```

Package Common Usage

```
library ieee;
use ieee.std_logic_1164.all;
use work.my_pack.all;      -- make the package available

entity bcd_7seg_top is
 port ( . . .
       . . . );
end;

architecture beh of package_test_2 is
begin

U1: bcd_7segm
       port map ( . . . );
U2: mux
       port map ( . . . );
 . . .
end beh;
```

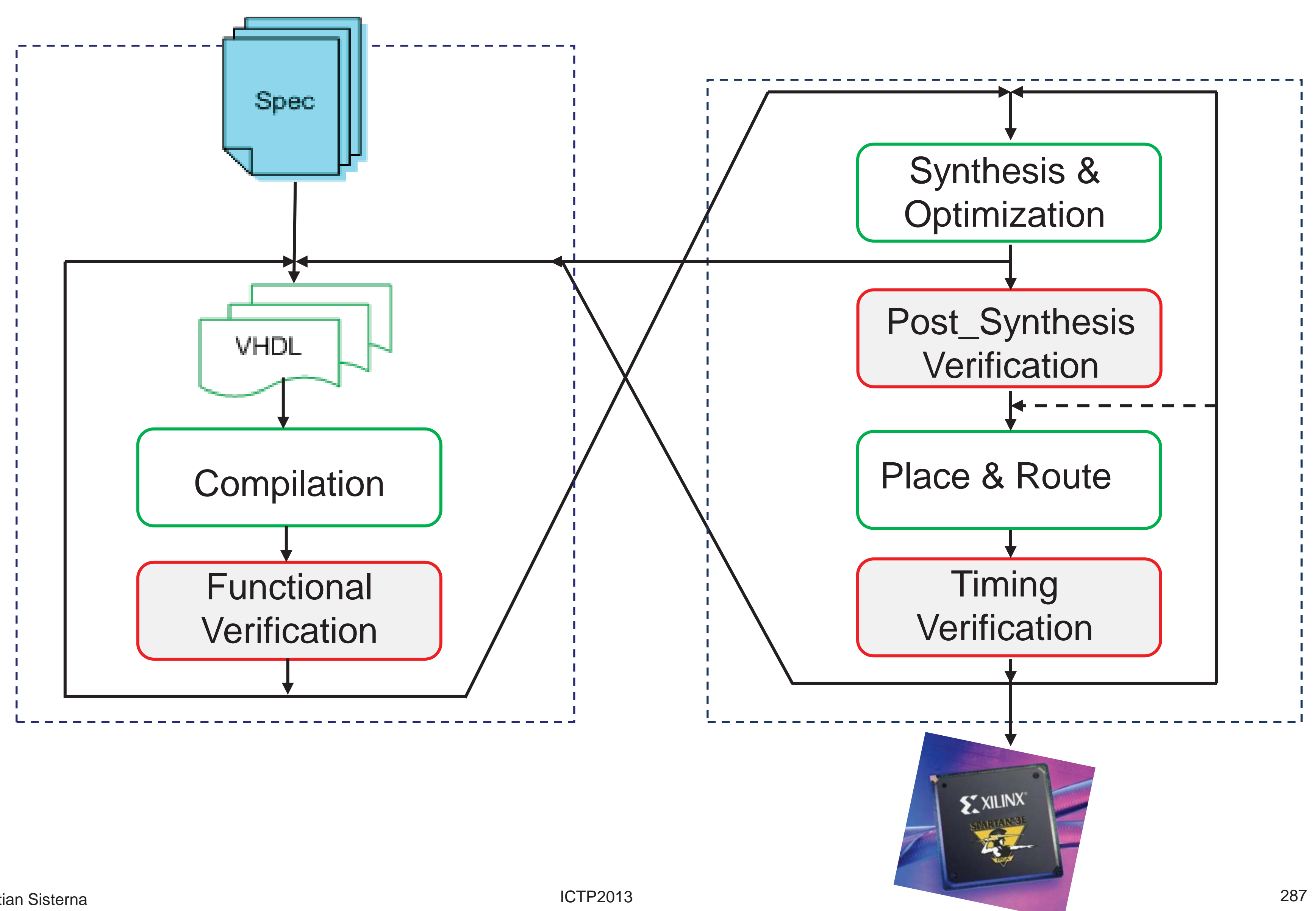
Validation

Verification

- It’s an IMPORTANT part of the design process: how de I know that the design works as expected?

- It’s IMPORTANT/NECESSARY to verify the behavior of the design in **at least one** of following parts of the design process:
  - Functional Verification / Pre-synthesis Verification
  - Post-synthesis
  - Timing Verification / Post-Place & Route Verification

Verification – Where?



Verification – Where??

- *Functional Verification / Pre-synthesis Verification:*
  - Where most errors can and should be found
  - Use this verification ALWAYS
- *Post-synthesis Verification*:
  - Not necessary unless several synthesis attributes were used and it's necessary to verify the behavior after synthesis
- *Timing Verification / Post-Place & Route Verification:*
  - Necessary in high frequency designs, in the order of >100MHz
  - Very slow process (very long verification time)

Verification – Test Bench

Test bench

a VHDL model which generates stimulus waveforms to be used to test a digital circuit described in VHDL

Test Bench - Overview

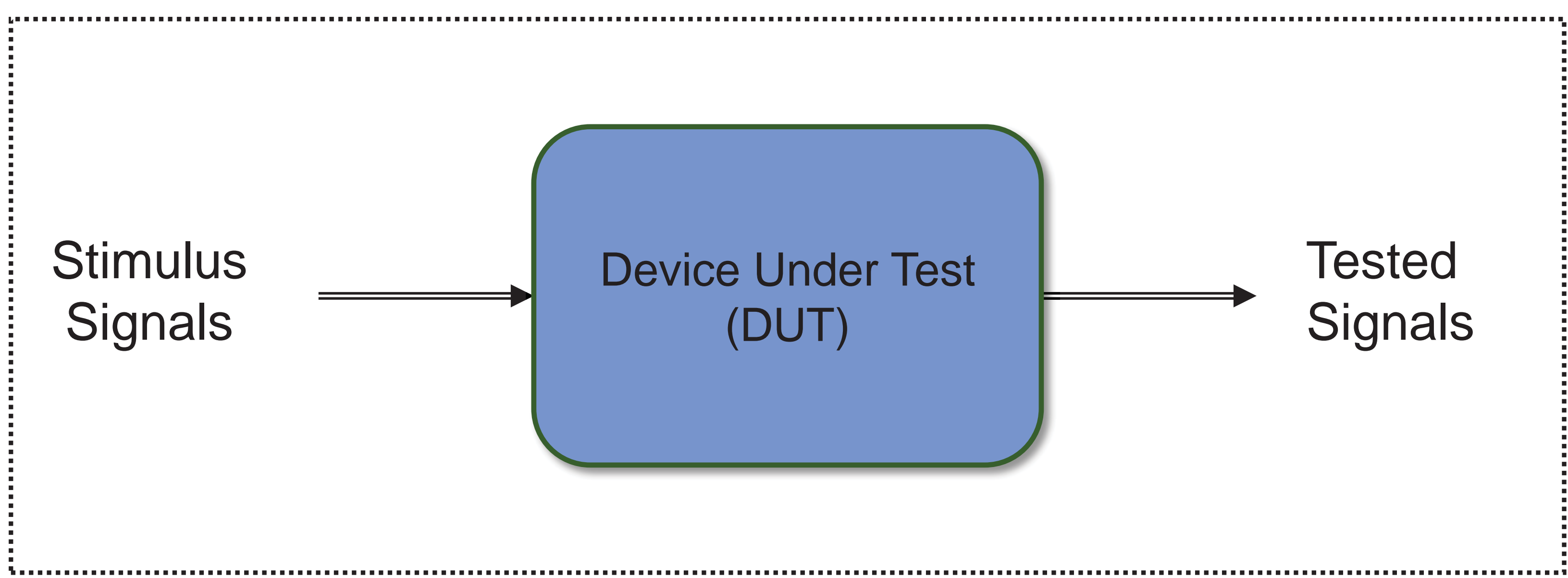
- A model used to exercise and verify the correctness of a hardware design
- Besides synthesis, VHDL can also be used as a Test Language
- Very important to conduct comprehensive verification on your design
- To simulate your design you need to produce an additional entity and architecture design

Test Bench

- It has three main purposes:
  - To generate stimulus for simulation
  - To apply this stimulus to the entity under test and to collect output responses
  - To compare output responses with expected values

- Test bench should be created by a DIFFERENT engineer than the one who created the module under test

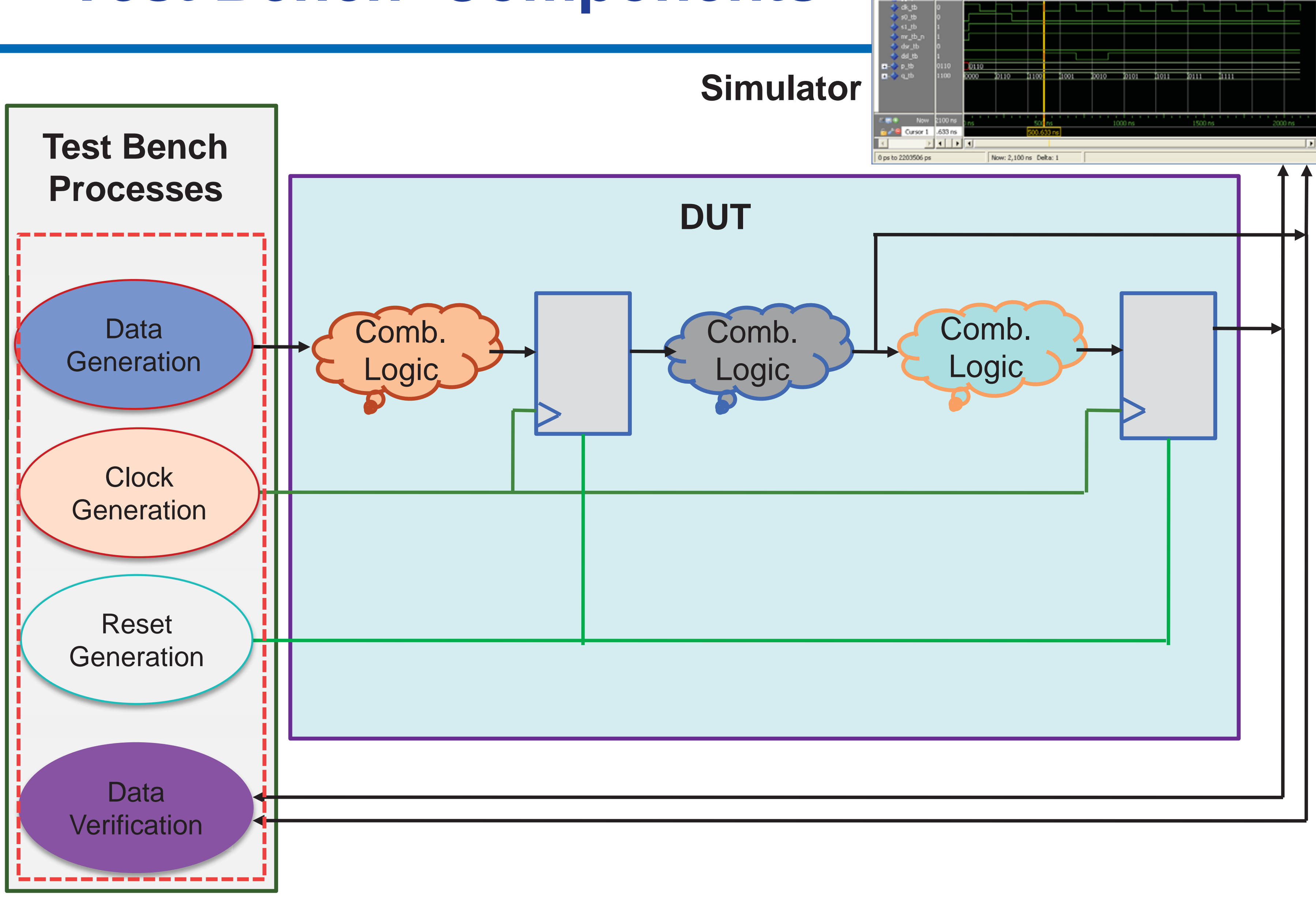
Test Bench



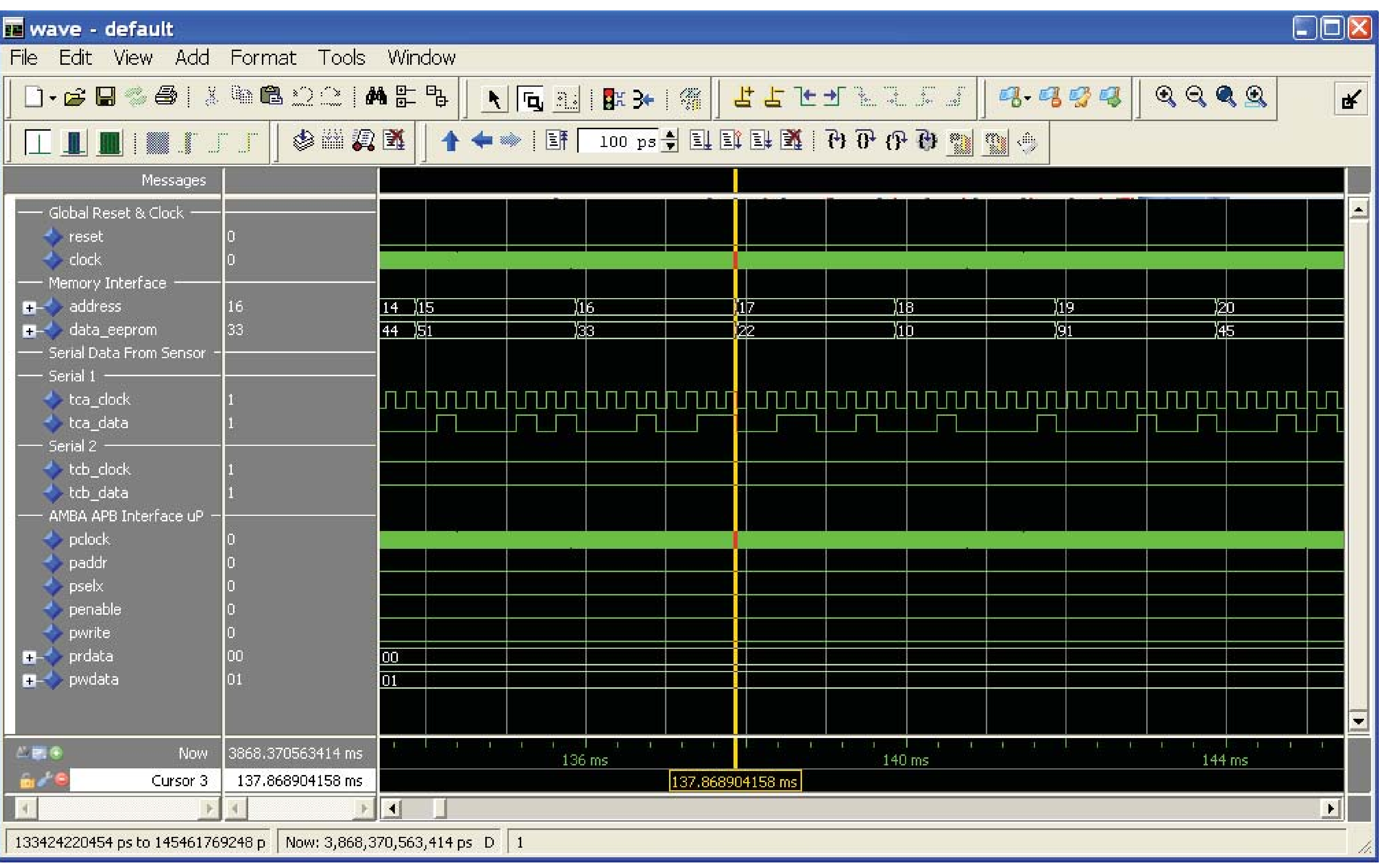
Test Bench - Components

- **TB entity:**
  - Empty declaration
- **TB architecture**:
  - Component declaration
  - Local signal declaration
  - Component instantiation
  - Stimulus statements
  - Check values statements

Test Bench–Components



Test Bench – Simulation Results



Test Bench - Components

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

entity test_my_design is
end test_my_design;

architecture testbench of test_my_design is
  component my_design is
     port (a,b : in std_logic;
           x,y : out std_logic);
  end component;

  signal as,bs : std_logic:='1';
  signal xs,ys : std_logic;

begin
  uut : my_design port map
        (a=>as, b=>bs, x=>xs, y=>ys);

    as <= not(as) after 50 ns;

  process begin
    bs <= '1'; wait for 75 ns;
    bs <= '0'; wait for 25 ns;
  end process;
end testbench;
```

- Define library, same as in VHDL source code
- VHDL model without entity interface
- Component declaration of the device to test
- Define signal names
- Instantiated UUT in test bench
- Define the stimulus of the test

Test Bench – VHDL Template

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

-- TB entity declaration del TB (empty entity)
entity testbench is
end testbench;

-- architecture declaration
architecture tb of testbench is
-- component declaration: component to test
      component device_under_test
            port(list_of_ports);
      end component;

-- local signal declarations. Used to:
-- stimulate the DUT's inputs
-- test the DUT's outputs
      <local_signal_declarations;>
```

Test Bench – VHDL Template (cont)

```
begin
-- component instantiation:
-- associate the top-level (TB)
-- signals to their equivalent DUT's signals

  DUT: entity_under_test port map( list_of_ports);

-- stimulus statements for the input signals.
-- values are assigned at different times per each input

   generate_input_waveforms;

-- ouptut signals' check

   monitor_output_statements;

end tb;
```

Test Bench – Use of *wait*

The wait statement can be located anywhere between begin and end process

- Basic Usages:

```
wait for time;

wait until condition;

wait on signal_list;

wait;
```

Test Bench – Use of *wait* 1

```
process
   . . .
  J <= '1';
  wait for 50 ns;-- process is suspended for 50 ns after J is
  . . .                -- assigned to '1'
end process;
```

```
process
   . . .
  wait until CLK = '1';-- sync with CLK rising edge before
                        -- continuing of simulation
  . . .
end process;
```

Test Bench – Use of *wait 2*

```
process
  . . .
  wait on A until CLK = '1';-- the process is resumed
                            -- after a change on A signal,
                            -- but only when the value of
                            -- the signal CLK is equal to '
  . . .
end process;
```

```
process
    rst <= '1';
    wait for 444 ns;
    rst <= '0';
    wait;          -- used without any condition,
                   -- the process will be suspended
                   -- forever
end process;
```

Test Bench – Clock Generation 1

```
architecture testbench of test_my_design is
  signal clk_50 : std_logic := '1';
  signal clk_75 : std_logic := '0';
  constant clk_period  : time := 100 ns;
  constant h_clk_period: time := 50 ns;

begin

-- case 1: concurrent statement
clk_50 <= not(clk_50) after h_clk_period;-- 50% duty

-- case 2: sequential statement
clk_75_proc: process
begin
    clk_75 <= '1';
    wait for 75 ns; -- 75% duty
    clk_75 <= '0';
    wait for 25 ns;
end process clk_75_proc;
. . .
. . .
end testbench;
```

Test Bench – Clock Generation 2

```
architecture testbench of test_my_design is

signal clk: std_logic := '0';
constant period: time := 40 ns;

begin

diff_duty_clk_cycle: process
begin
   clk <= '0';
    wait for period * 0.60;
   clk <= '1';
    wait for period * 0.40; -- 60% duty cycle

end process diff_duty_clk_cycle;

. . .
end architecture testbench;
```

Test Bench – Clock Generation 3

```
architecture testbench of test_my_design is

   signal clk: std_logic:= '0';
   constant half_period: time := 20 ns;

begin

clk_cycle: process
begin
   clk <= '0';
    wait for half_period; -- the clock will toggle
   clk <= '1';
    wait for half_period; -- as long as the simulation
                          -- is running
end process clk_cycle;
. . .

end architecture testbench;
```

Test Bench – Data Generation

- Avoid race conditions between data and clock
  - Applying the data and the active edge of the clock simultaneously might cause a race condition

- To keep data synchronized with the clock while avoiding race condition, apply the data at a different point in the clock period that at the active edge of clock

Test Bench – Data Generation 1

Example of Data generation on inactive clock edge

Clock generation process

```
clk_gen_proc: process
begin
Clk <= '0';
wait for 25 ns;
clk<= '1';
wait for 25 ns;
end process clk_gen_proc;
```

Data generation process

```
data_gen_proc: process
while not (data_done) loop
   DATA1 <= X1;
   DATA2 <= X2;
   ...
   wait until falling_edge(clk);
 end loop;
end process data_gen_proc;
```

Test Bench – Data Generation 2

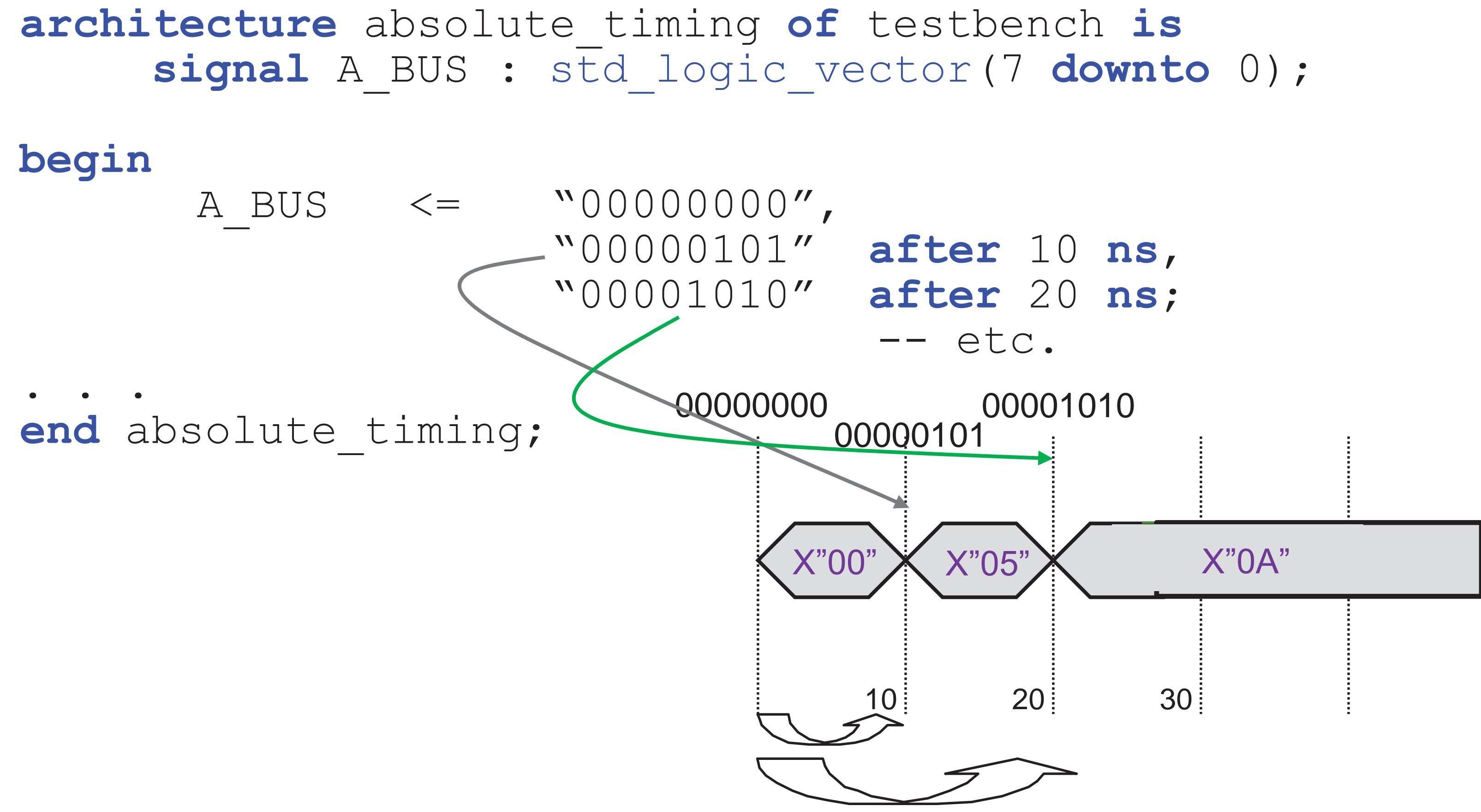
Relative time: signal waveforms that are specified to change at simulation times relative to the previous time, in a time accumulated manner

```
architecture relative_timing of myTest is

    signal Add_Bus : std_logic_vector(7 downto 0);

begin
   patt_gen_proc: process
     begin
       Add_Bus <= "00000000";
       wait for 10 ns;

       Add_Bus <= "00000101";
       wait for 10 ns;

       Add_Bus <= "10101010";
       wait for 10 ns;
   end process patt_gen_proc;
. . .
end relative_timing;
```

Test Bench – Data Generation 3

Absolute time: signal waveforms that are specified to change at simulation times absolute since the moment that the simulation begin

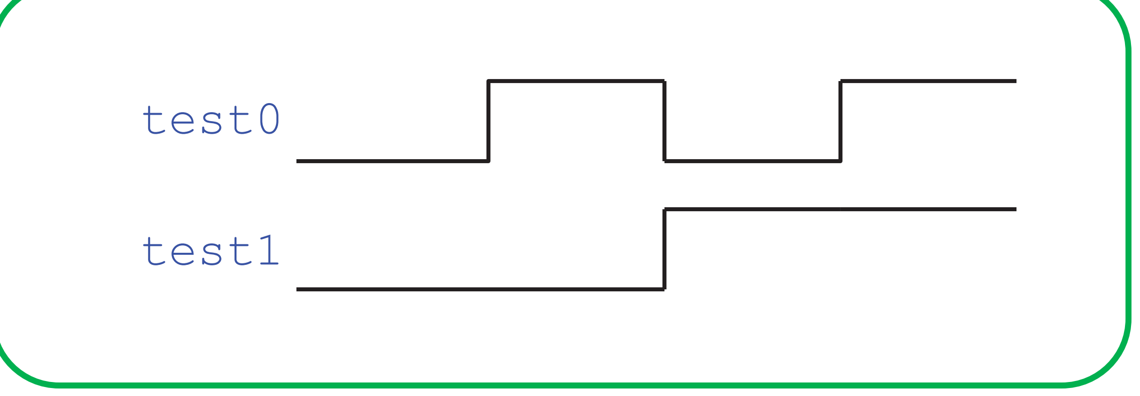
```
architecture absolute_timing of testbench is
     signal A_BUS : std_logic_vector(7 downto 0);

begin
      A_BUS   <=    "00000000",
                    "00000101"  after 10 ns,
                    "00001010"  after 20 ns;
                                -- etc.
. . .
end absolute_timing;
```



Test Bench – Data Generation 3

Absolute time: signal waveforms that are specified to change at simulation times absolute since the moment that the simulation begin

```
-- 2 bits test pattern
begin
        test0 <= '0', '1' after 10 ns, '0' after 20 ns,
                 '1' after 30 ns;
        test1 <= '0',
                 '1' after 20 ns;
. . .
```



Test Bench – Data Generation 4

```
architecture array_usage of in_test_benches is
   signal Add_Bus : std_logic_vector(7 downto 0);
   -- type & signal declarations: 5 data for 8 bits
   type stimulus is array (0 to 4) of
                               std_logic_vector (7 downto 0);

   constant DATA : stimulus :=
             ("00000000",          -- declare the stimulus
              "00000001",          -- as an array.
              "00000010",          -- these values will be
              "00000011",          -- used to stimulate the
              "00000100");         -- inputs
begin
     stim_proc: process
     begin
       for i in 0 to 4 loop     -- for loop that assign
           Add_BUS <= DATA(i); -- to Add_Bus a new value
           wait for 10 ns;     -- from stimulus every 10ns
       end loop;
     end process stim_proc;
. . .
end array_usage;
```

Test Bench – Data Generation 5

```
architecture array_usage of in_test_benches is

-- same declarations as previous example

begin
      process
      begin
        for i in 0 to 4 loop
              Add_BUS <= DATA(i);
                for k in 1 to 7 loop
                        wait until rising_edge(clk);
                end loop;
                wait until falling_edge(clk);
        end loop;
      end process;
. . .
end array_usage;
```

In this case each pattern in the sequence is held for how many clock cycles???

Test Bench – Reset Generation

```
-- asynchronous desassert reset
reset: process
begin
  rst <= '1';
  wait for 23 ns;
  rst <= '0';
  wait for 1933 ns;
  rst <= '1';
  wait for 250 ns;
  rst <= '1';
  wait;
end process;
```

Test Bench – Reset Generation

```
-- synchronous desassert reset
sreset: process
begin
   rst <= '1';
   for i in 1 to 5 loop
      wait until clk = '1';
   end loop;
   rst <= '0';
end process;
```

Test Bench Simple – Ex. Decoder 2:4

```
entity dcd_2_4 is
  port (in1   : in  std_logic_vector (1 downto 0);
        out1  : out std_logic_vector (3 downto 0));
end dcd_2_4;
--
architecture dataflow of dcd_2_4 is
begin
  with in1 select
    out1 <= "0001" when "00",
            "0010" when "01",
            "0100" when "10",
            "1000" when "11",
            "0000" when others;
end dataflow;
```

Test Bench Simple – Ex. Decoder 2:4

```
-- Test Bench to exercise and verify
-- correctness of DECODE entity
entity tb2_decode is
end tb2_decode;

architecture test_bench of tb2_decode is

type input_array is array(0 to 3) of
                          std_logic_vector(1 downto 0);
constant input_vectors: input_array :=
                          ("00", "01", "10", "11");

signal in1  : std_logic_vector (1 downto 0);
signal out1 : std_logic_vector (3 downto 0);

component decode
      port (
             in1 : in  std_logic_vector(1 downto 0);
             out1: out std_logic_vector(3 downto 0));
end component;
```

Test Bench Simple – Ex. Decoder 2:4

Stimulus to the Inputs and Component Instantiation:

```
begin
decode_1: decode port map(
                   in1   => in1,
                   out1  => out1);

apply_inputs: process
begin
  for j in input_vectors'range loop
      in1 <= input_vectors(j);
      wait for 50 ns;
  end loop;
end process apply_inputs;
```

Component Instantiation Inputs Stimulus and Outputs port map

Data generation

Test Bench Simple – Ex. Decoder 2:4 (1)

Data verification:

```
test_outputs: process
begin
   wait until (in1 = "01");
   wait for 25 ns;
   assert (out1 = "0110")
         report"Output not equal to 0110"
                severity ERROR;
  …         -- check the other outputs
end process test_outputs;
```

Input stimulus

Wait on certain time

Check the output

Test Bench Elaborated: Decoder 2:4 (2)

```
-- Test Bench to exercise and verify
-- correctness of DECODE entity
entity tb3_decode is
end tb3_decode;

architecture test_bench of tb3_decode is
type decoder_test is record
  in1: std_logic_vector(1 downto 0);
  out: std_logic_vector(3 downto 0);
end record;
type test_array is array(natural range <>) of decoder_test;
constant test_data: test_array :=
    (("00", "0001"),
     ("01", "0010"),
     ("10", "0100"),
     ("11", "1000"));
-- same component declaration as before
signal in1_tb : std_logic_vector(1 downto 0);
signal out1_tb: std_logic_vector(3 downto 0);
```

Test Bench Elaborated: Decoder 2:4 (3)

```
begin
decode_1: decode port map(
                       in1  => in1_tb,
                       out1 => out1_tb);
apply_in_check_outs: process
begin
   for j in test_data'range loop
        in1_tb <= test_data(j).in1);
        wait for 50 ns;
        assert (out1_tb = test_data(j).out1)
            report "Output not equal to the expected value,
                  error en indice " & integer'image(j);
               severity ERROR;
   end loop;
end process apply_in_check_outs;
```

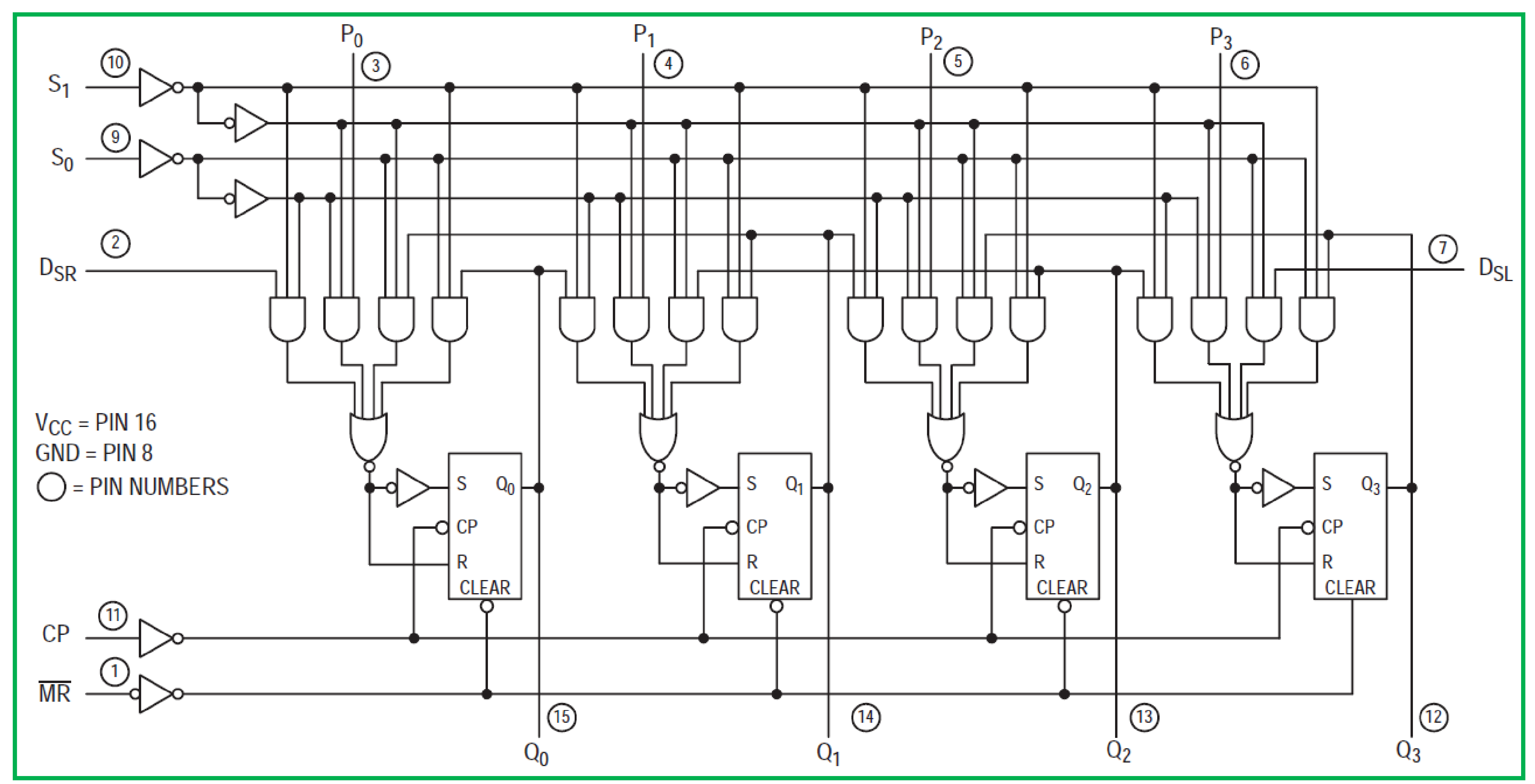
Test Bench Elaborated: Decoder 2:4 (4)

```
begin
…
apply_inputs: process
begin
   for j in test_data'range loop
        in1 <= test_data(j).in1);
        wait for 50 ns;
   end loop;
end process apply_inputs;
data_verif: process
begin
  wait for 25 ns;
  assert (out1 = test_data(j).out1)
            report "Output not equal to the expected
   value"
                severity ERROR;
  wait for 50 ns;
end process data_verif;
```

Test Bench Elaborated 2: Decoder 2:4 (5)

```
entity dcd_2_4 is
  port (in1 : in  std_logic_vector(1 downto 0);
         clk : in  std_logic ;
        out1 : out std_logic_vector(3 downto 0));
end dcd_2_4;
architecture dataflow of dcd_2_4 is
signal out1_i: std_logic_vector (3 downto 0);
begin
  with in1 select
    out1_i <= "0001" when "00",
              "0010" when "01",
              "0100" when "10",
              "1000" when "11",
              "0000" when others;
reg_proc: process(clk)
 begin
    if(rising_edge(clk)) then
          out1 <= out1_i;
    end if;
 end process reg_proc;
end dataflow;
```

Test Bench – Example 74LS194



Test Bench – Example 74LS194

MODE SELECT — TRUTH TABLE

OPERATING MODE	INPUTS						OUTPUTS			
	$\overline{MR}$	S_1	S_0	D_{SR}	D_{SL}	P_n	Q_0	Q_1	Q_2	Q_3
Reset	L	X	X	X	X	X	L	L	L	L
Hold	H	l	l	X	X	X	q_0	q_1	q_2	q_3
Shift Left	H	h	l	X	l	X	q_1	q_2	q_3	L
	H	h	l	X	h	X	q_1	q_2	q_3	H
Shift Right	H	l	h	l	X	X	L	q_0	q_1	q_2
	H	l	h	h	X	X	H	q_0	q_1	q_2
Parallel Load	H	h	h	X	X	P_n	P_0	P_1	P_2	P_3

L = LOW Voltage Level
H = HIGH Voltage Level
X = Don't Care
l = LOW voltage level one set-up time prior to the LOW to HIGH clock transition
h = HIGH voltage level one set-up time prior to the LOW to HIGH clock transition
p_n (q_n) = Lower case letters indicate the state of the referenced input (or output) one set-up time prior to the LOW to HIGH clock transition.

Test Bench – Example 74LS194 (1)

```
-- example of a 4-bit shift register
-- LS194 4-bit bidirectional universal shift register

library ieee;
use ieee.std_logic_1164.all;

entity ls194 is
   port(
       clk,mr_n,s0,s1,dsr,dsl_ser: in std_logic;
       p       : in  std_logic_vector(3 downto 0);
       q       : out std_logic_vector(3 downto 0)
        );
end ls194;
```

Test Bench – Example 74LS194 (2)

```
architecture behav of ls194 is
signal temp: std_logic_vector(3 downto 0);
signal ctrl: std_logic_vector (1 downto 0);
begin
ctrl <= s0 & s1;
shift_proc: process(clk, mr_n )
begin
    if (mr_n = '0') then
       temp <= (others => '0');
    elsif (rising_edge(clk)) then
      case ctrl is
        when "11" => temp <= p;
        when "10" => temp <= dsr & temp(3 downto 1);
        when "01"=>  temp <= temp(2 downto 0) & dsl;
        when others => temp <= temp;
      end case;
    end if;
end process;
q <= temp; end behav;
```

Test Bench – Example 74LS194 (3)

```
-- example of test bench to test the ls194
library ieee;
use ieee.std_logic_1164.all;

entity test_bench is
end test_bench;
architecture tb of test_bench is
component ls194 is
   port(
       clk,mr_n,s0,s1,dsr,ds: in std_logic;
       p        : in  std_logic_vector(3 downto 0);
       q        : out std_logic_vector(3 downto 0));
end component;
-- internal signals
signal clk_tb: std_logic:= '1';
signal s0_tb, s1_tb, mr_tb_n, dsr_tb, dsl_tb: std_logic:=
  '0';
signal p_tb, q_tb : std_logic_vector (3 downto 0);
```

Test Bench – Example 74LS194 (4)

```
-- constant declarations
constant clk_period: time := 200 ns;

begin
-- component instantiation
U1: ls194 port map(
      clk  => clk_tb,
      mr_n => mr_tb_n,
      s0   => s0_tb,
      s1   => s1_tb,
      dsr  => dsr_tb,
      dsl  => dsl_tb,
      p    => p_tb,
      q    => q_tb);

-- clock generation
clk_tb <= not clk_tb after clk_period/2;
```

Test Bench – Example 74LS194 (5)

```
main_proc: process
begin
    -- check_init_proc
   wait for 10 ns;
   assert q_tb = "0000"
       report " Initialization Error "
         severity ERROR;

   wait for 20 ns;
   mr_tb_n <= '1';

    -- check synchronous load
    s0_tb <= '1';
    s1_tb <= '1';
    p_tb  <= "0110";
```

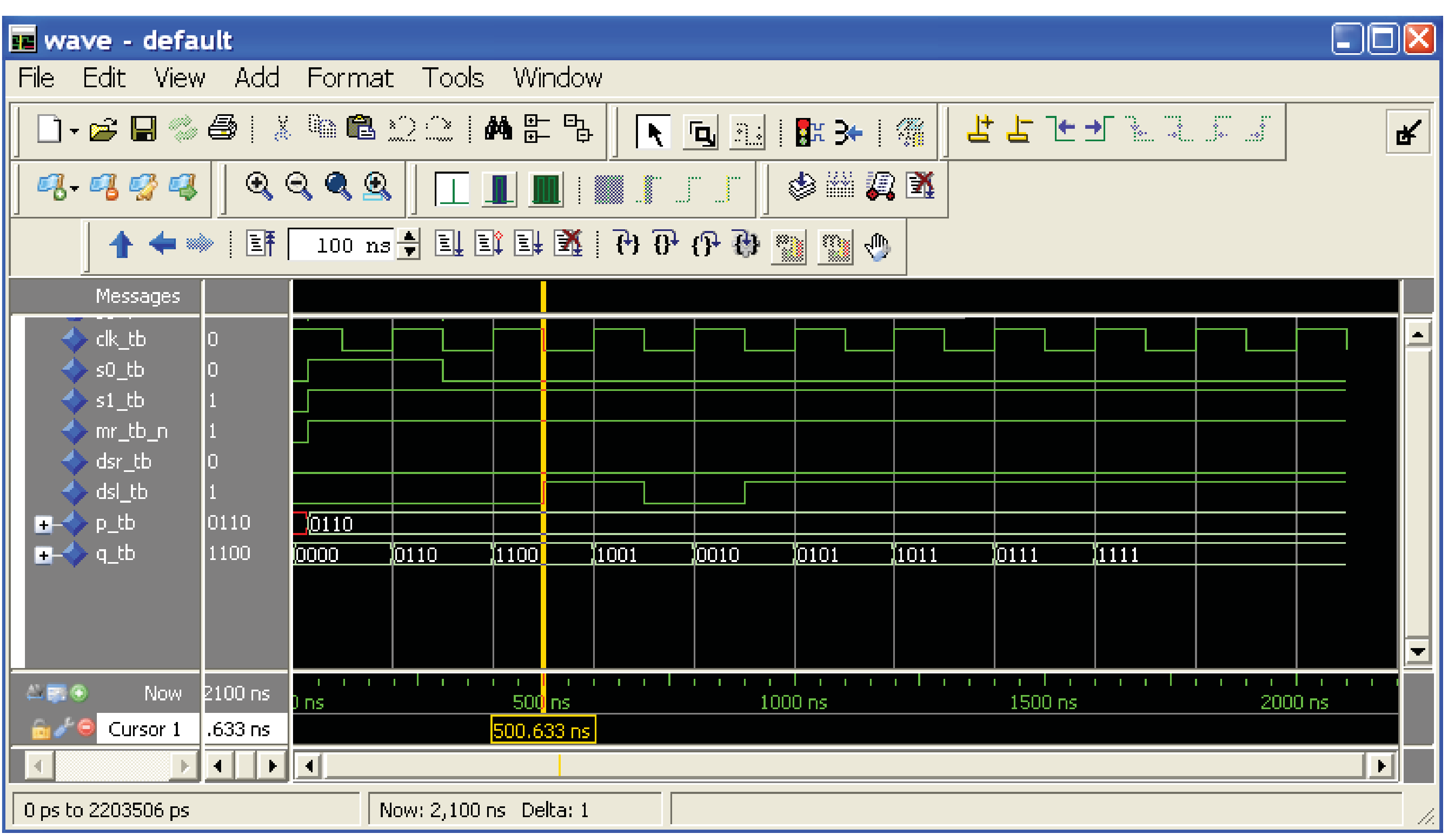
Test Bench – Example 74LS194 (6)

```
wait for clk_period;
-- wait until falling edge clk_tb
wait until clk_tb = '0';

assert q_tb = "0110"
    report " Load Error "
      severity ERROR;

-- check shift left
s0_tb <= '0';
-- wait until falling edge clk_tb
wait until clk_tb = '0';

assert q_tb = "1100"
    report " Error: Shift left Failed "
      severity ERROR;
```

Test Bench – Example 74LS194 (7)

```
  -- three more shift left
  for i in 0 to 2 loop
     if i = 1 then
        dsl_tb <= '0';
      else
       dsl_tb <= '1';
     end if;
     wait until clk_tb = '0';
   end loop;

  assert q_tb = "0101"
      report " Error: serial left shift failed "
        severity ERROR;

  wait;
end process;
end tb;
```

Test Bench – Example 74LS194 (8)



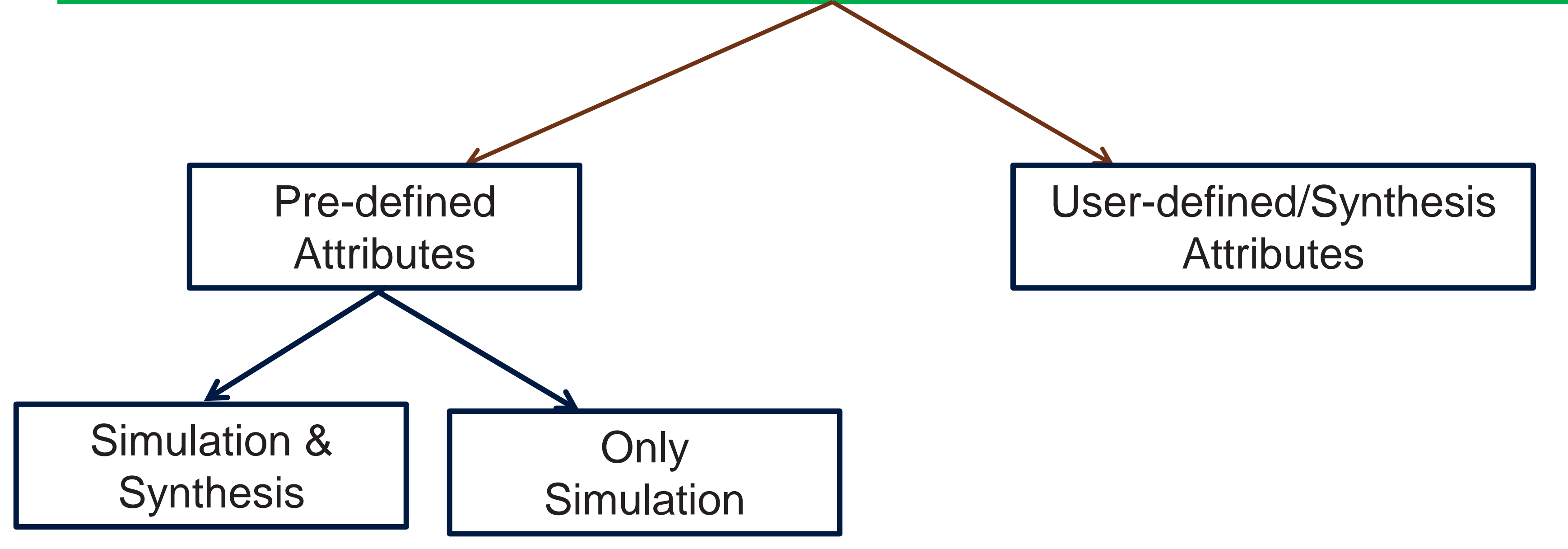
Test Bench - Conclusion

- TB is done in VHDL code
- Emulate the Hardware
- Use all the power of VHLD
- There is no size limit
- TB is the top-level unit
- Usually is more complex than the design itself

Attributes

Attributes

- It's a 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)



Predefined Attributes

Syntax:

```
<data_object/type/block>'attribute_identifier;
```

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

Array Attributes – Value Related	
A'left	Returns the left-most bound of a given type or subtype
A'right	Returns the left-most bound of a given type or subtype
A' high	Returns the upper bound of a given type or subtype
A'low	Returns the lower bound of a given type or subtype
A'length	Return the length (number of elements of an array)
A'ascending	Return a boolean true value of the type or subtype declared with an ascending range

Array Attributes

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 - Examples

```
type bit_array_1  is array (1 to 5) of bit;
variable L: integer:= bit_array_1'left;         -- L = 1
variable R: integer:= bit_array_1'right;        -- R = 5
type bit_array_2  is array (0 to 15) of bit;
variable H: integer:= bit_array_2'high;         -- H = 15
type bit_array_3  is array (15 downto 0) of bit;
variable L: integer:= bit_array_3'low;          -- L = 0
variable LEN: integer:= bit_array_3'length;    -- L = 16
variable A1: boolean := bit_array_1'ascending;-- A1 = true
variable A2: boolean := bit_array_3'ascending;-- A2 = false
type states is (init, reset, cnt, end);
signal Z, Y: states;
Z <= estados'left;           -- Z = init
Y <= estados'right;          -- Y = end
```

Array Attributes

`range` and `reverse_range` usage:

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

then:

   w_bus'range        will return:    7 downto 0

and:

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

Array Attributes

```
function parity(D: std_logic_vector) return std_logic is
      variable result: std_logic := '0';
  begin
      for i in D'range loop
            result := result xor D(i);
      end loop;
      return result;
  end parity;
```

Array Attributes

```
for i in D'range loop

for i in D‘reverse_range loop

for i in D‘low to D’high loop

for i in D‘high to D’low loop
```

Attributes of Signals

Attributes that return information about signals such as previous value, value change, etc.

Attributes of Signals	
S’event	True if there is an event on S in the current simulation cycle, false otherwise
S’active	True if there is an transaction S in the current simulation cycle, false otherwise
S’ last_event	The time interval since the last event on S
S’last_active	The time interval since the last transaction on S
S’last_value	The value of S just before the last event on S

Attributes of Signals - Example

```
process
constant setup_time: time :=1.2 ns;
  begin
        wait until clk = '0';
                assert (data'stable(setup_time))
                         report "Setup Violation"
                     severity warning;
end process;
```

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 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;
```

Ejemplos

```
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";
```

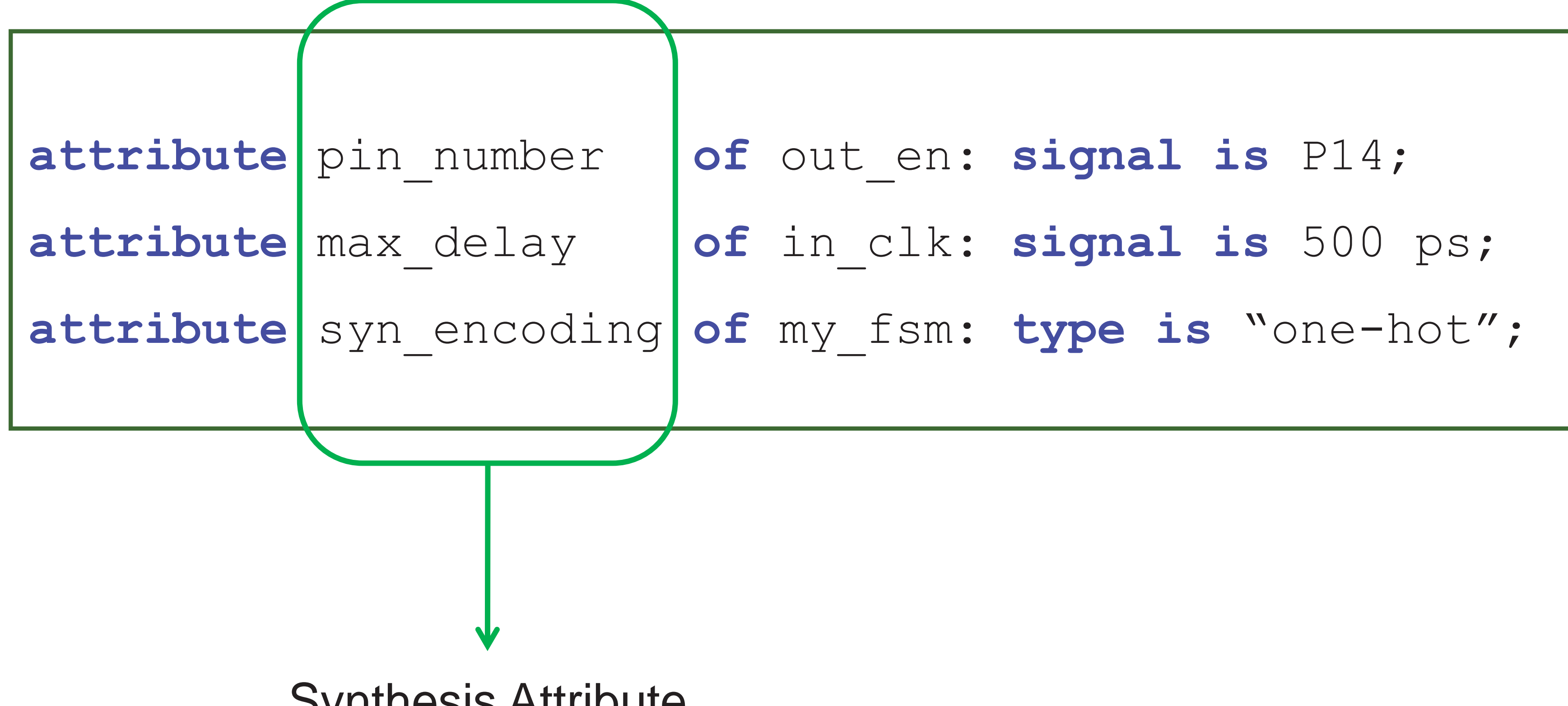
User-defined/Synthesis Attributes

Example:

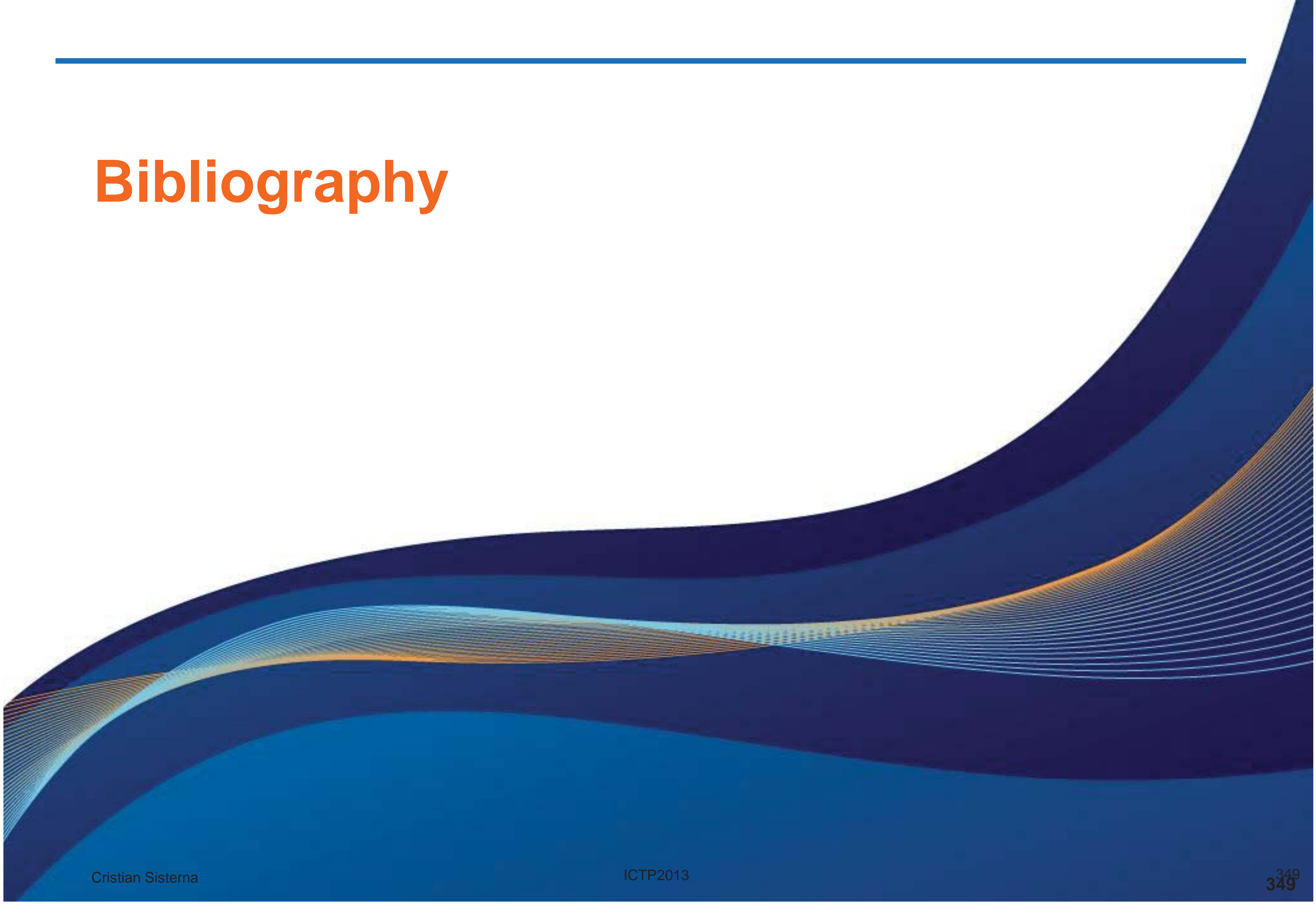
```
type ram_type is array (63 downto 0) of
                                std_logic_vector (15 downto 0);
signal ram: ram_type;
attribute syn_ramstyle: string;
attribute syn_ramstyle of ram: signal is “block_ram”;
```

User-defined/Synthesis Attributes

Example:



Bibliography



349

Books

1. *VHDL for Logic Synthesis*, Andy Rushton. John Wiley & Sons, 2011.
2. *RTL Hardware Design Using VHDL*, Chu, IEEE Wiley Interscience, 2006
3. *Advanced FPGA Design*, Steve Kilts, IEEE John Wiley & Sons, 2007.
4. *A VHDL Primer*, J. Bhasker, Prentice Hall, 1995.
5. *VHDL Made Easy*, D. Pellerin and D. Taylor, Prentice Hall, 1997.
6. *Digital Design and Modeling with VHLD and Synthesis*, K. C. Chang, IEEE Computer Society Press, 1997.
7. *A VHDL Modeling and Synthesis Methodology for FPGAs*, C. Sisterna, Arizona State University, 1998.
8. *Digital Design using Field Programmable Gate Arrays*, Pak Chan and Samiha Mourad, Prentice Hall, 1994.
9. *VHDL for Desginers*, Stefan Sjoholm and Lennert Lindh, Prentice Hall, 1997.
10. *Digital Design Principles and Practices*, Third Edition, John Wakerly, Prentice Hall, 2000.
11. *Digital Design with FPGAs using VHDL*, C. Sisterna, Arizona State University, 1997.
12. *The Designer's Guide to VHDL*, Peter Ashenden, Morgan Kaufman, 1996.
13. *Synthesis and Optimization of Digital Circuits*, Giovanni De Micheli, Mc Graw-Hill, 1994.
14. *HDL Chip Design, A Practical Guide for Designing, Synthesizing and Simulationg ASICs and FPGAs using VHDL or Verilog*, Douglas Smith, Doome Publications, 1998.

Others

- Altera Corporation (www.altera.com)
  - Application Notes
  - Datasheets
- C7 Technology (www.c7t-hdl.com)
  - Application Notes
  - Technical Notes
- Actel Corporation (www.actel.com)
- Xilinx Corporation (www.xilinx.com)
- Intel Corporation (www.intel.com)
- Lattice Semiconductors (www.latticesemi.com)
- Blog: http://hdl-fpga.blogspot.com.ar/ (spanish)
- Blog: http://fpga-hdl.blogspot.com.ar/ (english)