Electronic Systems

Digital Systems & VHDL:

Key concepts & examples

Ing. Luca Zulberti – luca.zulberti@phd.unipi.it
Prof. Massimiliano Donati – massimiliano.donati@unipi.it
Prof. Luca Fanucci – luca.fanucci@unipi.it

Agenda

1. Levels of Abstraction
2. Digital Design Technology Map
3. VHDL Introduction
4. VHDL File:
   I. Library
   II. Entity
   III. Architecture
5. Writing VHDL: Rules & examples
6. Simulate VHDL

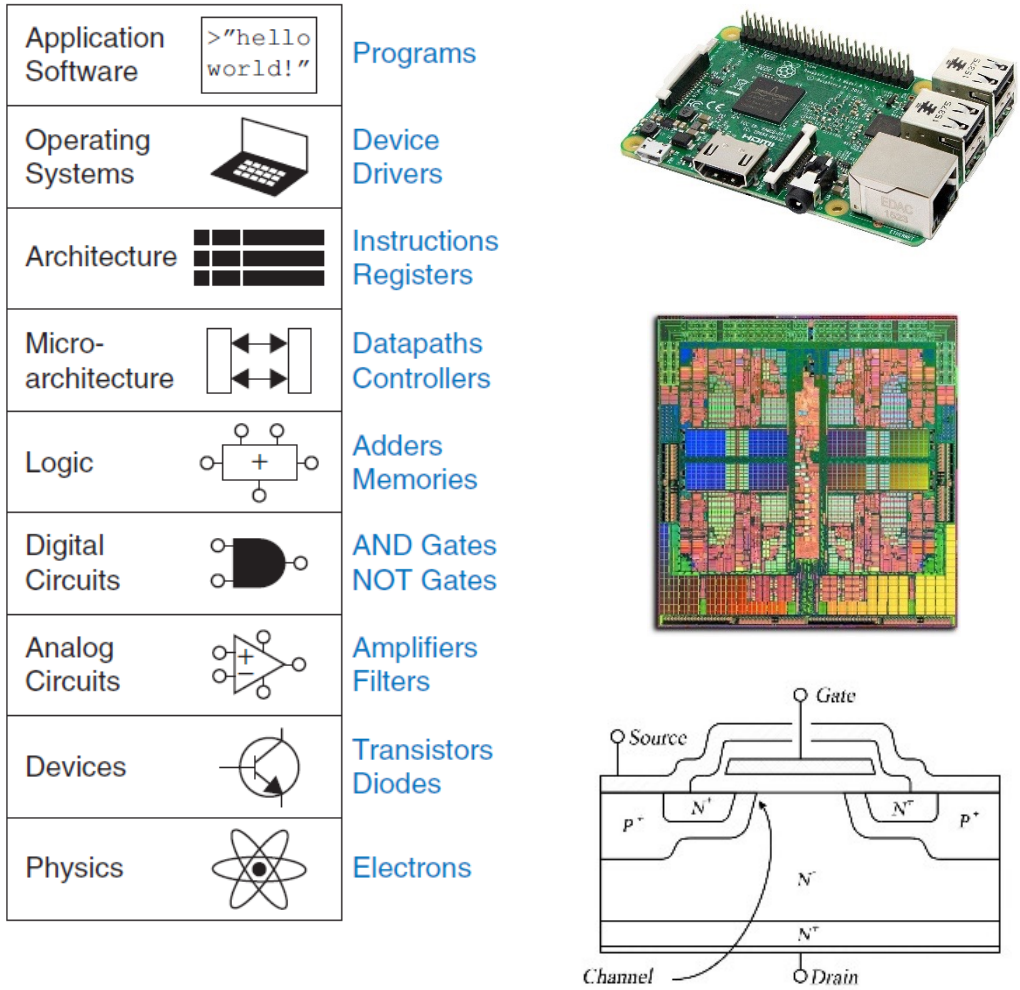
Agenda

1. Levels of Abstraction
2. Digital Design Technology Map
3. VHDL Introduction
4. VHDL File:
   I. Library
   II. Entity
   III. Architecture
5. Writing VHDL: Rules & examples
6. Simulate VHDL

Levels of Abstraction: Computer

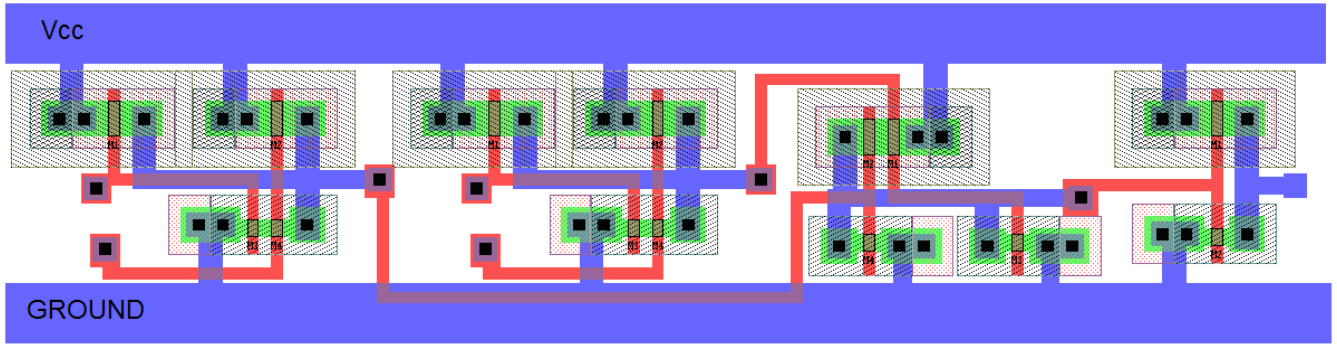
Software & Embedded engineering

Microelectronic engineering

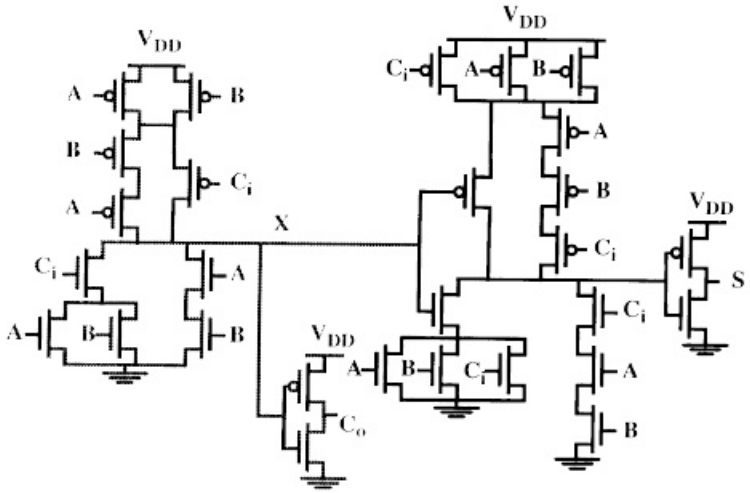


Levels of Abstraction: Digital Design

- Layout

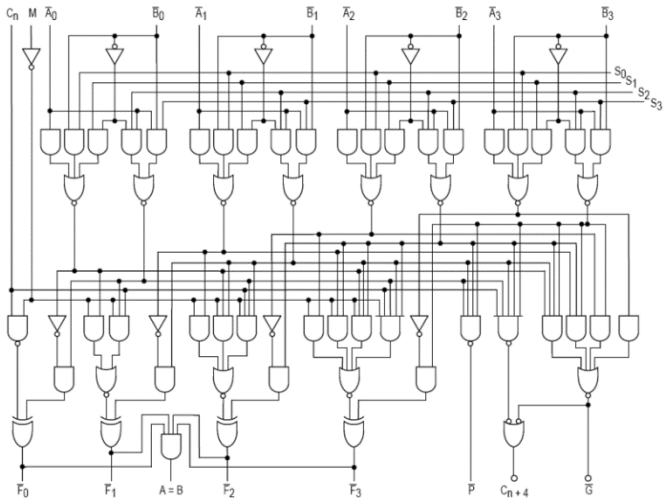


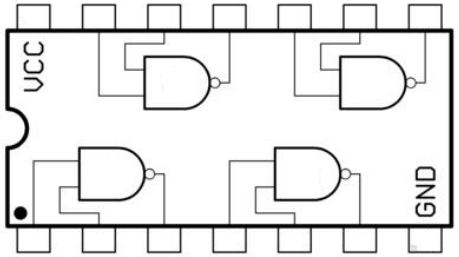
- Transistor



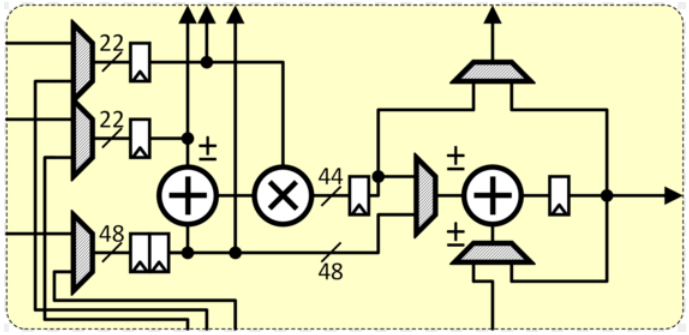
Levels of Abstraction: Digital Design

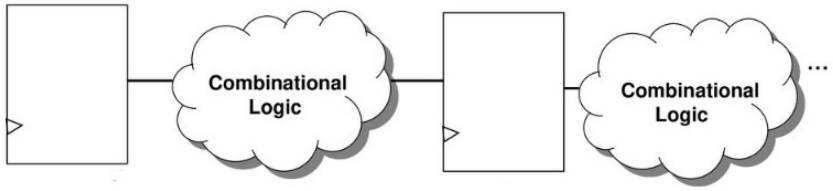
- Gate





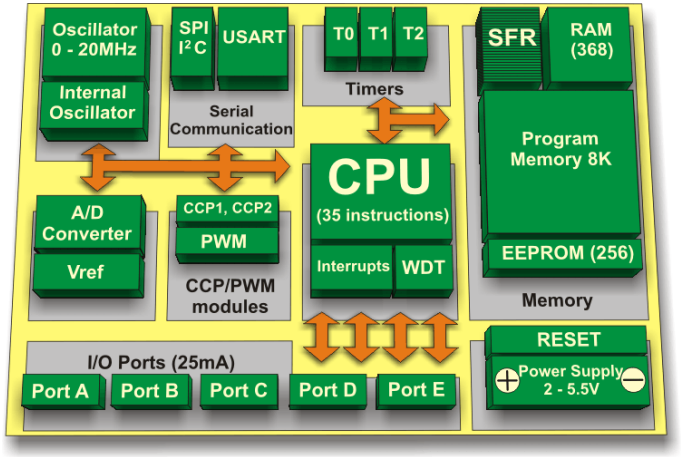
- RTL



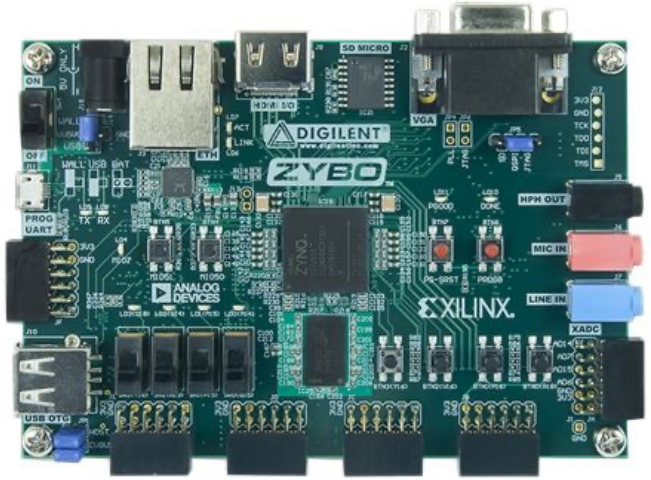


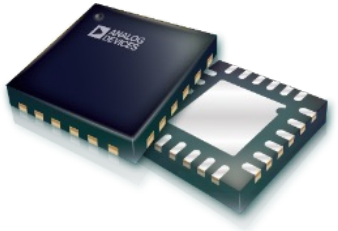
Levels of Abstraction: Digital Design

- Architecture (Partitioning)



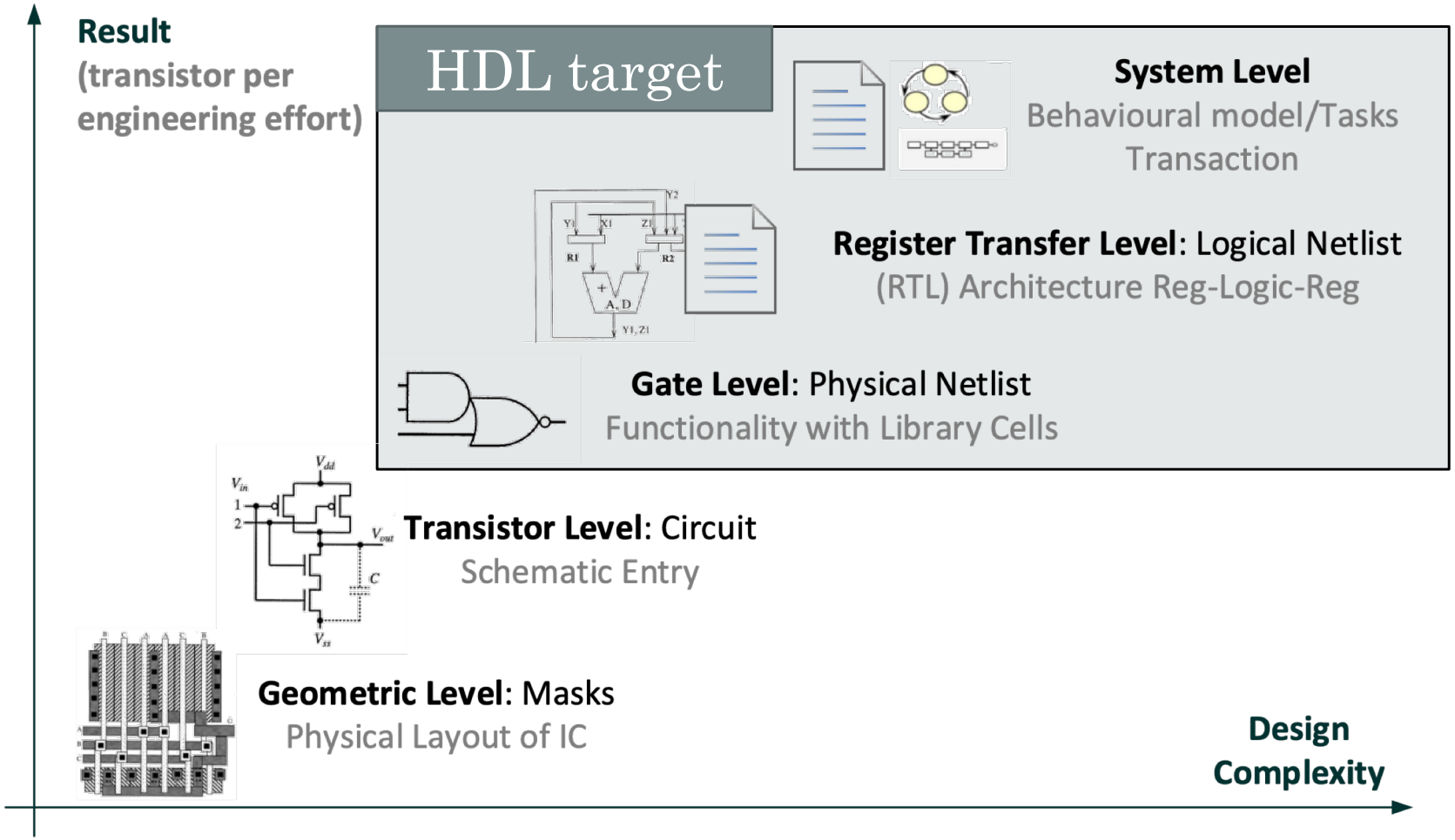
- System





Levels of Abstraction: Complexity

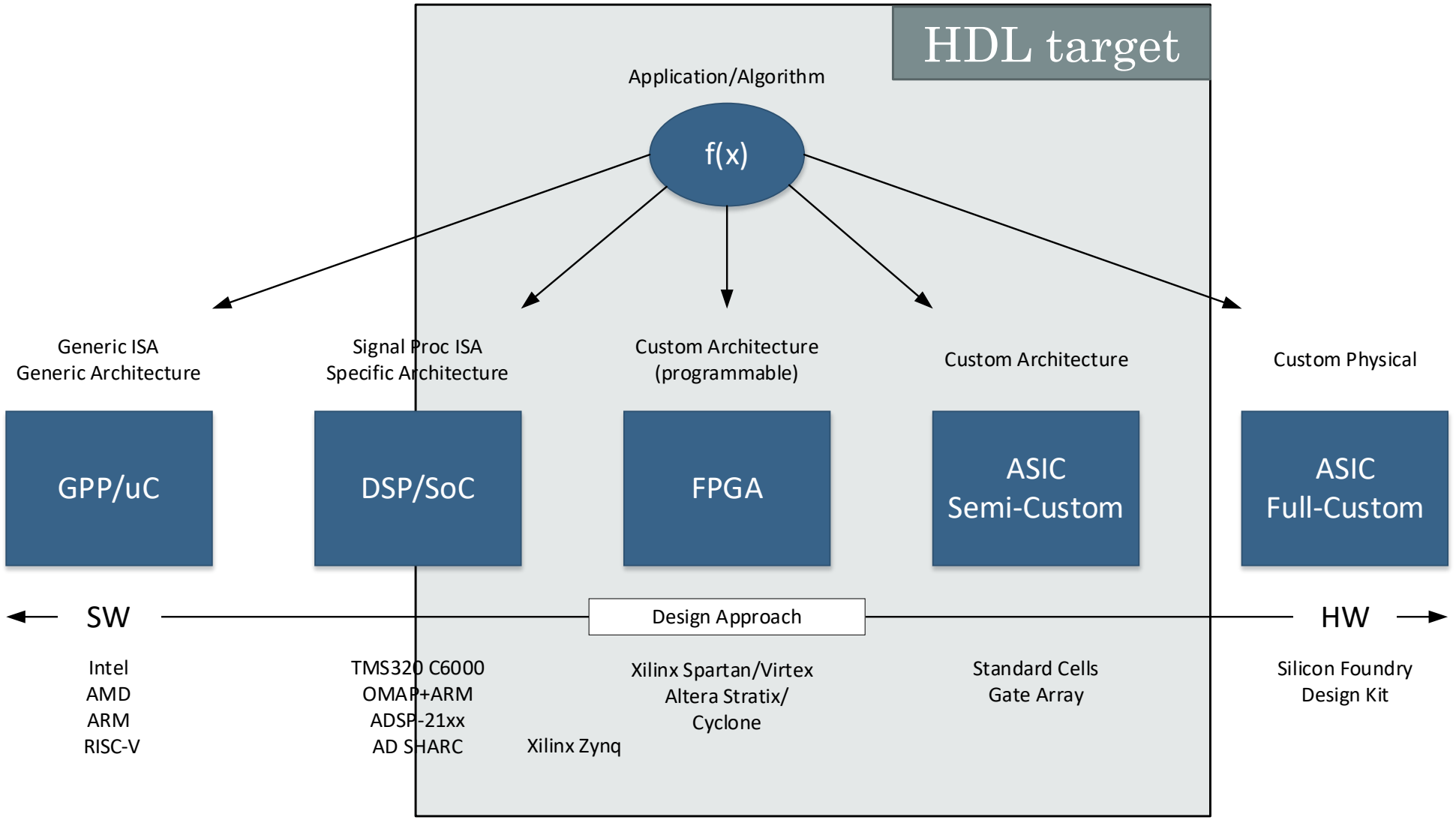
- A digital system can be viewed at different "scale" or levels of abstraction.



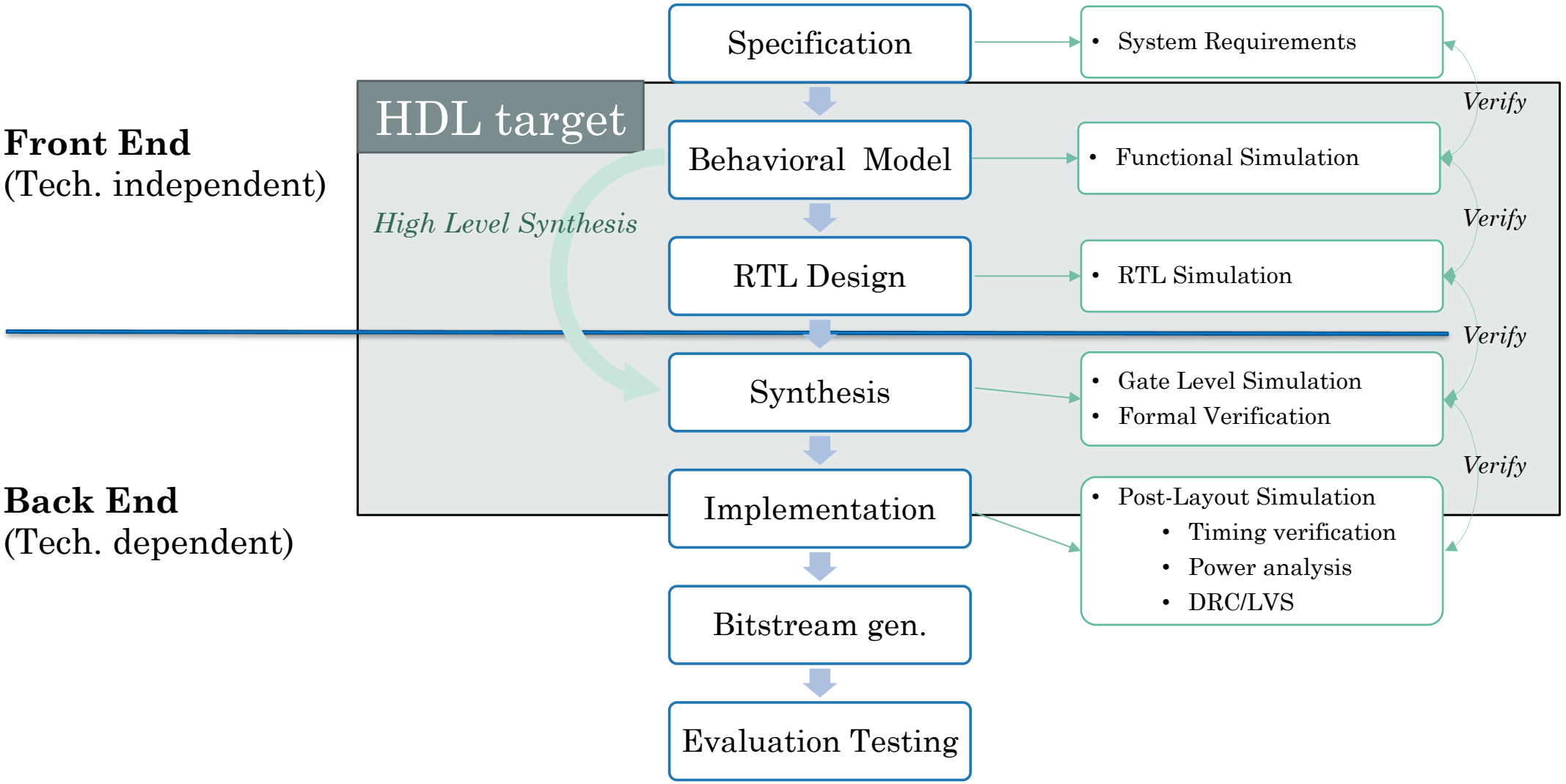
Agenda

1. Levels of Abstraction
2. Digital Design Technology Map
3. VHDL Introduction
4. VHDL File:
   I. Library
   II. Entity
   III. Architecture
5. Writing VHDL: Rules & examples
6. Simulate VHDL

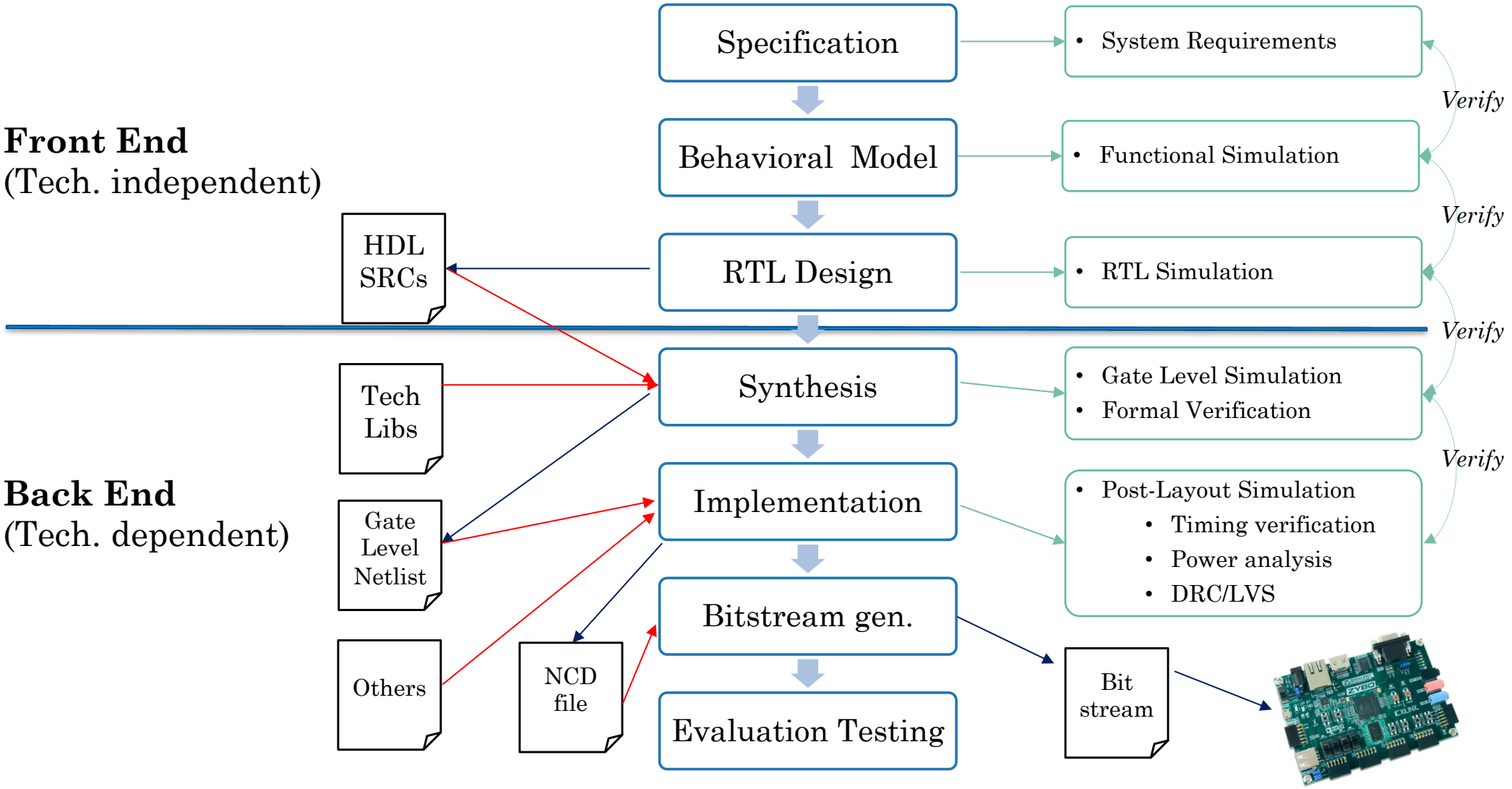
Digital Design Technology Map



D.D.Tech. Map: FPGA Design Flow



D.D.Tech. Map: FPGA Design Flow



D.D.Tech. Map: FPGA Synthesis

RTL Code
Logic description of the system
Logic gates have:
- An associated logic function
- No delay
- No area occupation
- …

Synthesis →

Gate Level Netlist
FPGA (family) library **cells**
Logic gates have:
- An associated logic function
- A delay (depending on the capacitive load)
- Area occupation
- …

Synthesis maps the RTL code with the Gate Level Netlist cells

The synthesis does NOT place the components on the FPGA!

- No interconnections are created
- Timing, power consumption, and area occupation estimates do not consider interconnections

D.D.Tech. Map: FPGA Implementation

Gate Level Netlist
FPGA (family) library cells
Gates have:

- An associated logic function
- A delay (depending on the capacitive load)
- Area occupation
- …

Implementation →

Native Circuit Description
File containing the routed circuit on the FPGA (which blocks are used and how they are linked)

Implementation maps the gates of the Gate Level Netlist with specific FPGA blocks (FPGA, LUT, DSP blocks,…)

- Interconnections are routed among the several blocks
- Timing, power consumption and area occupation extimations consider the interconnections

Agenda

1. Levels of Abstraction
2. Digital Design Technology Map
3. VHDL Introduction
4. VHDL File:
   I. Library
   II. Entity
   III. Architecture
5. Writing VHDL: Rules & examples
6. Simulate VHDL

Hardware Description Language

How to describe, model, design and verify a digital system?

HDL : Hardware Description Language

STANDARD IEEE

VHDL Verilog

SystemVerilog
SystemC
…

HDL features

- **Executable** (simulation)
- Provide several **levels of abstraction**
- Description of a **hardware system as a software** model
- **Independent** from technology, design methodology, design style, *design tool*

Hardware Description Language

Remember, you are not programming,

you are designing hardware!

Agenda

1. Levels of Abstraction
2. Digital Design Technology Map
3. VHDL Introduction
4. VHDL File:
   I. Library
   II. Entity
   III. Architecture
5. Writing VHDL: Rules & examples
6. Simulate VHDL

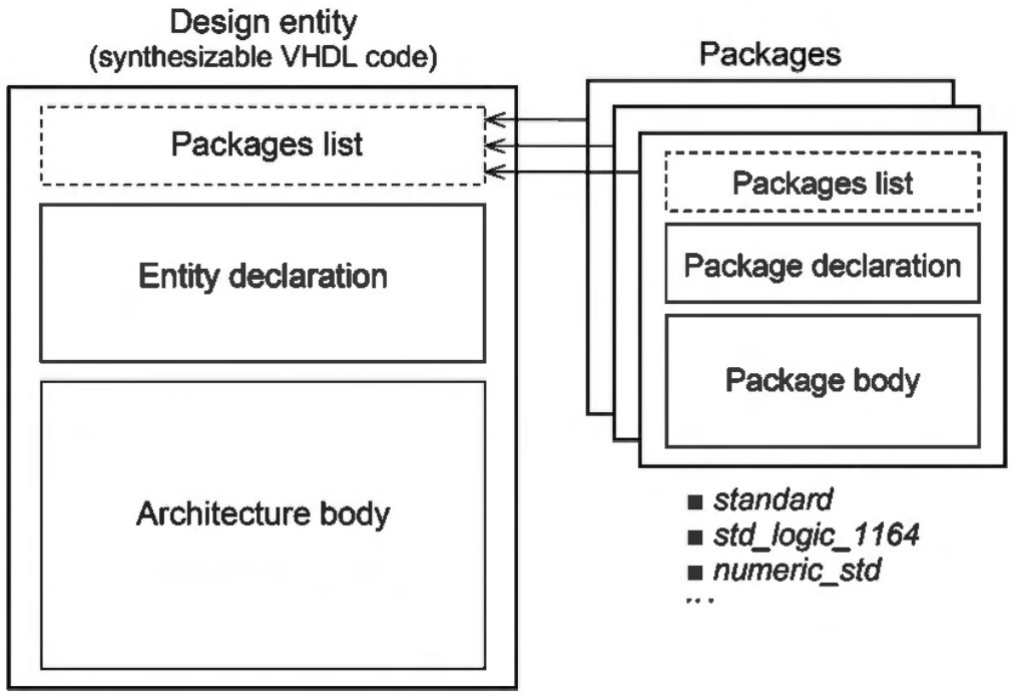
VHDL File: Design Units

A digital system (from µP to NAND gate) is modeled with 5 parts called **Design Units.**

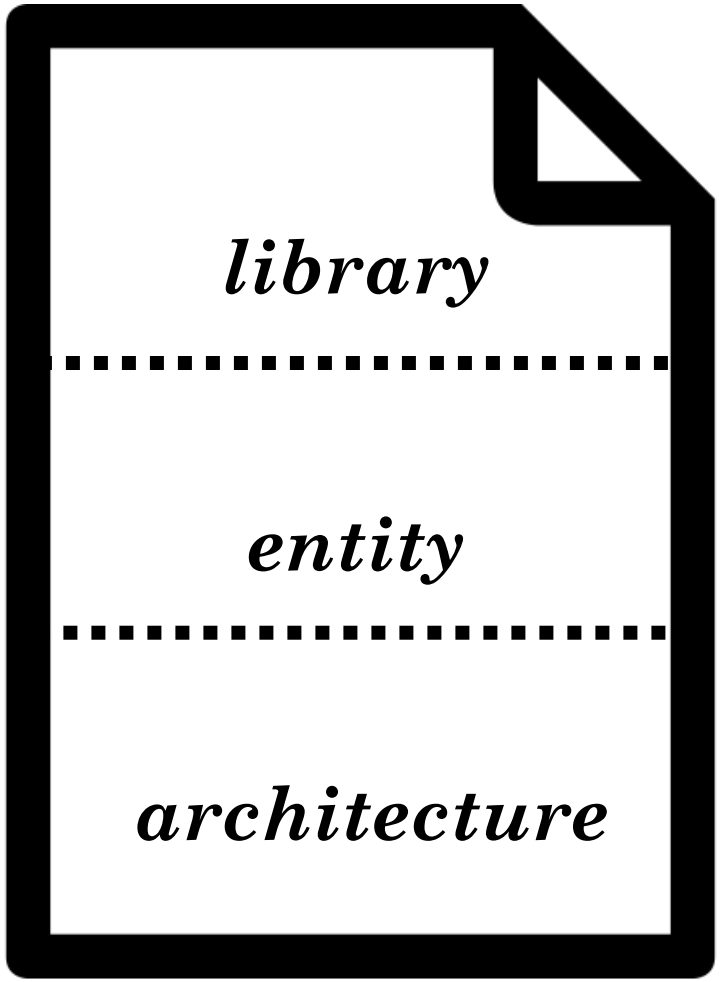
1. **Entity declaration**
2. **Architecture body**

Required

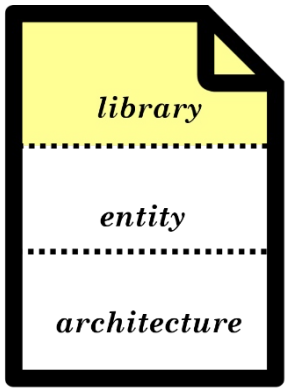
3. Package declaration
4. Package body
5. Configuration declaration



VHDL File: Structure



VHDL File: Library



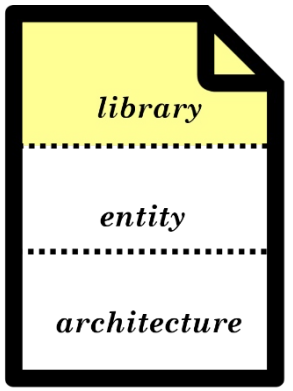
The main official VHDL packages are distributed in two libraries called ***std*** and ***ieee***

```
library std;
  use std.standard.all;
  use std.textio.all;

library ieee;
  use ieee.std_logic_1164.all;
  use ieee.numeric_std.all;
  use ieee.fixed_pkg.all;
  use ieee.math_real.all;

library work;
  use work.my_package.all;
```

If not specified, the default library where all user-defined entities are compiled is WORK.

- **std.standard**: basic VHDL types and some function.
- **std.textio**: Line and Text types, input output files and related functions.
- **ieee.std_logic_1164**: IEEE standard synthesizable types, functions, and operators. *(most used package)*
- **ieee.numeric_std**: used for integer mathematics, it defines **unsigned** and **signed** types with functions and operators.
- **ieee.fixed_pkg**: used for synthesis of fixed-point arithmetics.
- **ieee.math_real**: generally used to determine generics parameters using real values. (not used in synthesis)

VHDL File: Library & Types



Data Types

- Scalar
  - **Numeric (Integer, Floating, Physical)**
  - **Enumeration (Ordered symbols/chars)**
- Composite
  - **Array (also multidimensional)**
  - Record (composition of different types)
- Access (simulation only)
- File (simulation only)

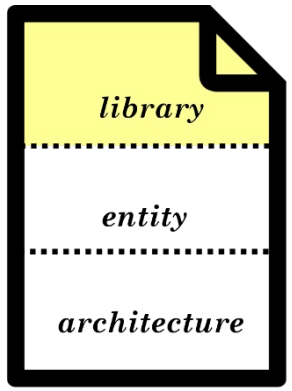
- Subtype is a constrained type (subset or range)

VHDL is a **strongly typed** language
(and is often **verbose** to write …)

$$'0' \neq 0 \neq 0.0$$

Std_logic/Character ≠ Integer ≠ Float

VHDL File: IEEE Types



Package **standard** has only **bit** and **bit_vector** types to represent the logic

- Ideal for high-level design: available states are '**1**' and '**0**'.
- NOT adequate for detailed designs: no '**–**' or '**Z**' states.

IEEE library provides the **std_logic_1164** package, defining a multi-value logic system.

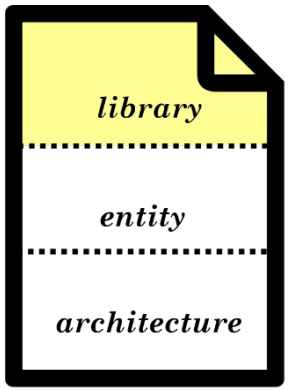
```
type std_logic is (
  'U',  -- Uninitialized
  'X',  -- Forcing Unknown
  '0',  -- Forcing 0
  '1',  -- Forcing 1
  'Z',  -- High Impedance
  'W',  -- Weak Unknown
  'L',  -- Weak 0
  'H',  -- Weak 1
  '-',  -- Don't care
);
```

Simulation behaviour is defined by the language but what about **their behaviour in synthesis?**

Synthesis tools can use particular primitives (*if available*) to implement the desired behaviour.

PRIMITIVES ARE TECHNOLOGY DEPENDENT

VHDL File: IEEE Types



Resolved type

```
subtype std_logic is resolved std_ulogic;
```

Vector/Array type

```
type std_logic_vector is
  array ( natural range <> ) of std_logic;
```

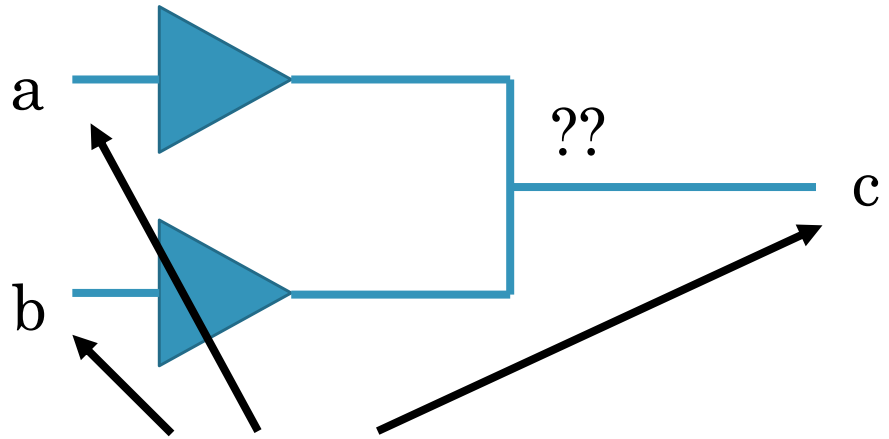
What is a resolved (sub)type?

Suppose to drive a signal ***c*** with two sources, ***a*** and ***b***:

If **c** is **std_ulogic → compile error**

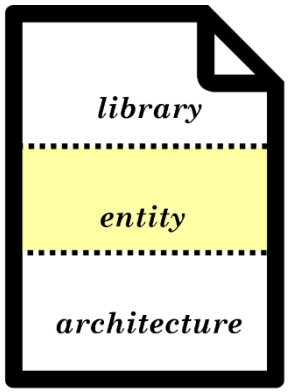
If **c** is **std_logic → OK resolved**

A *resolution map* is used to decide the value of ***c*** depending on ***a*** and ***b*** values.



Signals: VHDL objects that transport information

VHDL File: Entity

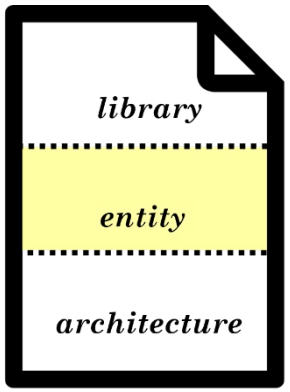



```
entity entity_name is
  generic (
    CONSTANT_NAME : constant_type := constant_def_value;
    CONSTANT_NAME : constant_type := constant_def_value;
    ...);
  port (
    port_name : port_mode port_type;
    port_name : port_mode port_type;
    ...);
end entity;
```

Used for entity generalization [optional]

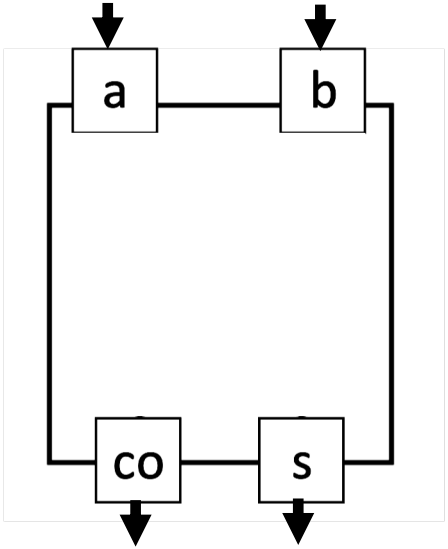
Define the entity interface

VHDL File: Entity



Entity Declaration

- The entity declaration describes the external view of the entity.
- The input and output signals define **the entity interface**.




```
entity HalfAdder is  -- Entity name
  port (
    a  : in  std_logic;
    b  : in  StD_LoGiC;
    co : out std_logic;
    s  : out std_logic  -- last port has no ';'
  );
end entity;
```

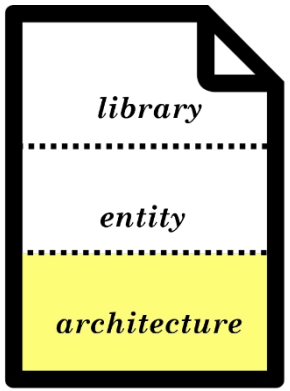
VHDL is case insensitive

Ports direction: **in, out, inout**

Ports signal **type**

Comments starts with '--' until end of line

VHDL File: Architecture

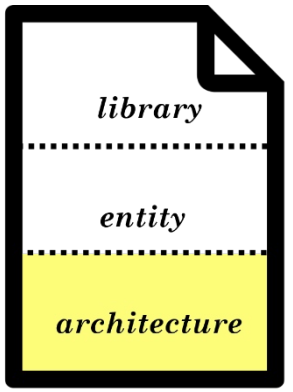


The architecture body contains **the internal description** of the entity.

A design can be expressed in any comb. of these **four descriptive styles**:

- **Behavioral**
  - algorithmic kind of description (system behavior, no hardware).
  - uses *sequential programming statements* (**if, for, while, case …**).
- **Data Flow**
  - boolean kind of description (combinational circuits between registers).
  - uses *concurrent programming statements* (**when**, **else**, **and, or, xor, not …**).
- **RTL**
  - collection of combinational blocks interconnected via registers.
- **Structural**
  - describes the systems with **component instantiation** and their interconnections.

VHDL File: Architecture




```
architecture architecture_name of entity_name is

  architecture_declarative_part

begin

  architecture_statement_part

end architecture;
```

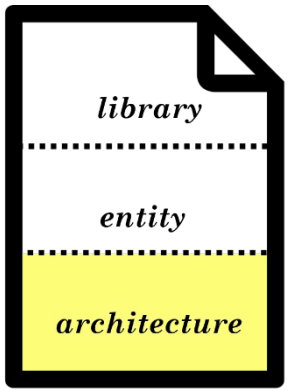
Declarative part can contain:

- declarations of components, types, signals, and constants.
- declarations of subprograms, packages, shared variables, files, aliases, attribute, and group.

Statement part contains:

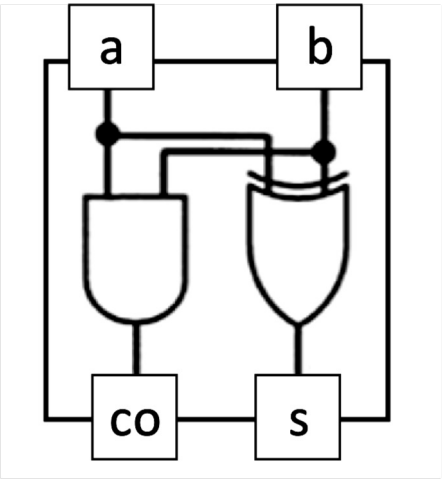
- VHDL statements. **Here is where the circuits are described.**

VHDL File: Architecture



Data Flow Example

```
architecture dataflow of HalfAdder is  -- Arch name
  -- empty declarative part
begin
  co <= a and b;  -- statement 1
  s  <= a xor b;  -- statement 2
end architecture;
```

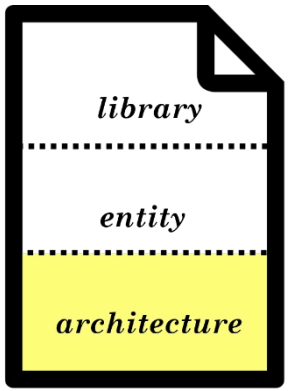


Concurrent statement, order is not relevant. VHDL is a description language not a programming language!!

Signal **assignment**

- '**<=**' delimiter is used to assign values to **signals**
- '**:=**' delimiter is used to assign values to **variables**, **constants**, and **initial values**

VHDL File: Architecture



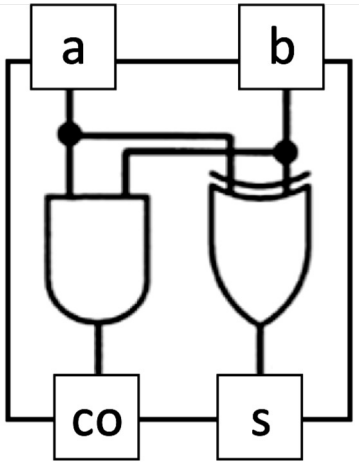
Behavioural Example

```
architecture beh of HalfAdder is
begin

  COMB: process(a, b)
  begin
    -- sequential statements
    if (a and b) = '1' then
      co <= '1';
    else
      co <= '0';
    end if;
  end process;

  s <= a xor b;  -- concurrent statement
end architecture;
```

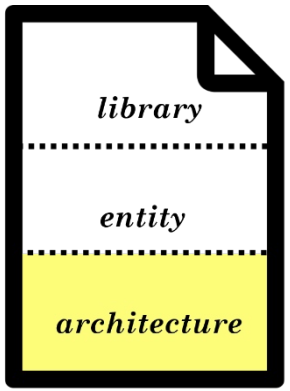
Process is "executed" (**activated**) when a signal in the **sensitivity list** produces an event.

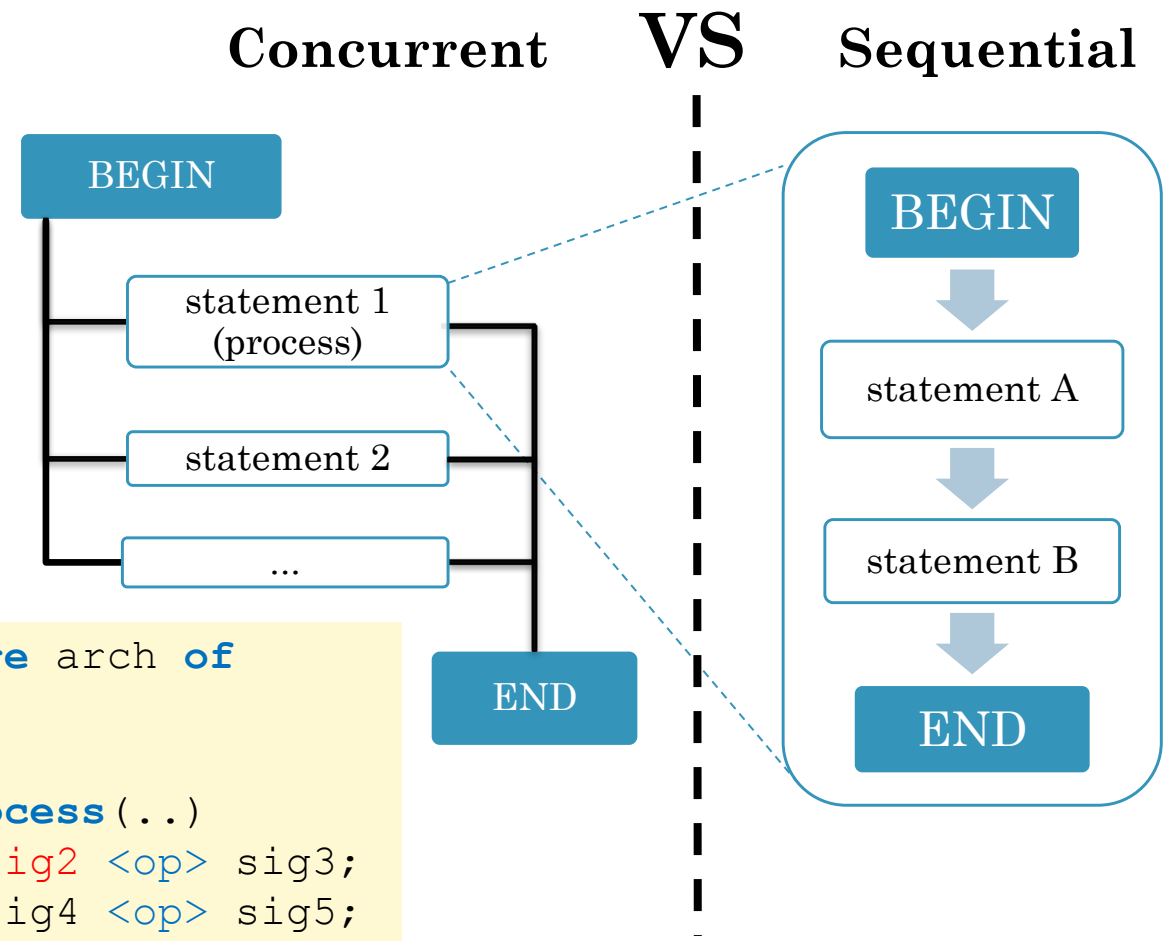
Comparison (relational) **operator**

Sequential statements into the process describe the algorithm.



VHDL File: Architecture

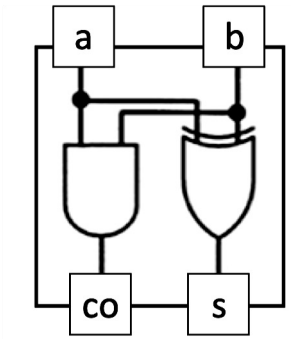


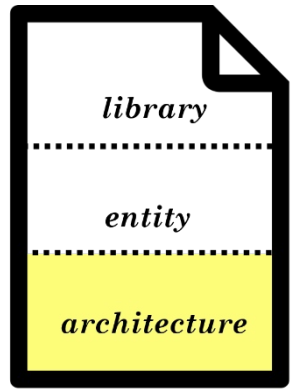



```
architecture arch of
entity is
begin
   PROC: process(..)
   sig1 <= sig2 <op> sig3;
   sig2 <= sig4 <op> sig5;
   ...
end architecture;
```

```
PROC: process(sig2,sig3)
begin
   sig6 <= default_value;

   if <cond1> then
      sig6 <= sig2;
   end if;

   sig6 <= sig3;

end process;
```

VHDL File: Architecture





Structural Example

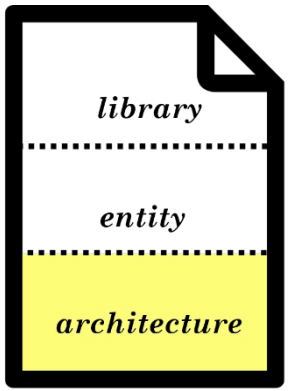
```
architecture Struct of HalfAdder  -- architecture name
  -- components declaration
  component My_AND is
    port (AND_in1 : in std_logic; AND_in2 : in std_logic; AND_out : out std_logic);
  end component;
  component My_XOR is
    port (XOR_in1 : in std_logic; XOR_in2 : in std_logic; XOR_out : out std_logic);
  end component;

begin -- architecture description
  i_AND : My_AND -- instance named/labelled "i_AND" of component "My_AND "
  port map (
    AND_in1 => a,          -- explicit port association
    AND_in2 => b,          -- !!! use "=>" and ","
    AND_out => co          -- last port has no ","
  );

  i_XOR : My_XOR -- instance named/labeled "i_XOR" of component "My_XOR"
  port map (a, b, s);      -- positional association

end Struct;
```

Component declaration to instantiate the entity it refers to

One instance of the component My_AND named i_AND. (possible to do i2_AND, i3_AND, i4_AND…)

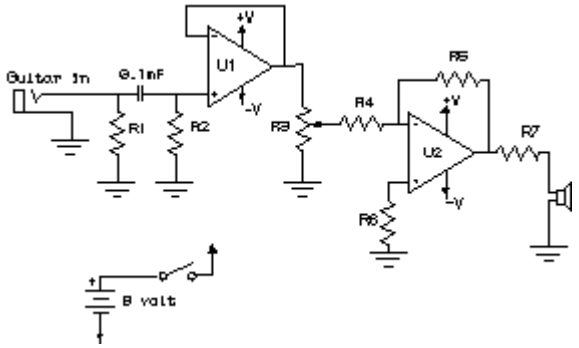
Positional association strongly discouraged

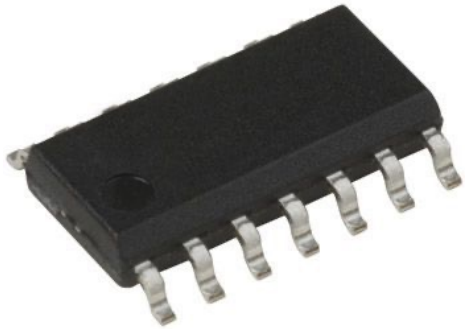
VHDL File: Architecture



Entity-Architecture Vs Component Instance

Differences between them may be tricky, think of it this way:
Consider the project of an integrated circuit:

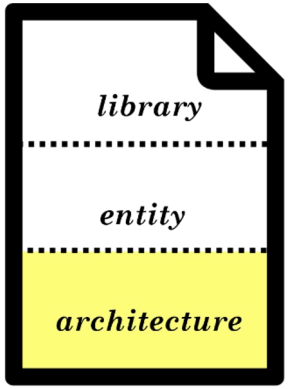




The IC's schematic will be its entity, the structure containing all the information about the IC.

The IC itself will be the component, the instance of what is described by the entity.

VHDL File: Architecture



Component Instantiation

```
architecture arch of MyEntityA is

  component MyEntityB is
    port (...)
  end component;

begin

  instance_1: MyEntityB port map (...);
  instance_2: MyEntityB port map (...);

end architecture;
```

Components must be declared before instantiation!

There can be **multiple instances** of the same component

Agenda

1. Levels of Abstraction
2. Digital Design Technology Map
3. VHDL Introduction
4. VHDL File:
   I. Library
   II. Entity
   III. Architecture
5. Writing VHDL: Rules & examples
6. Simulate VHDL

Writing VHDL: Rules

Operators (used in Expressions)

Logical

- `not  and  or  nand  nor  xor  xnor`

Relational

- `=  /=  <  <=  >  >=`

Arithmetic

- `+  -  *  /  mod  rem  abs  **`

Shift

- `sll  srl  rol  ror`

Concatenation

- `&`

```
"10" & '1' & sig2  =  "101_1010"   (sig2 = "1010")
```

Writing VHDL: Rules

Logical operators

- **Binary**: takes two inputs
  - `bit` **`<op>`** `bit`
  - `vector` **`<op>`** `vector` (same length)
  - *`bit <op> vector`*
- **Unary**: takes one input
  - **`not`** `bit`
  - *`<op> vector`* **:** *reduction* operation (apply the logical operator between bits of vector)

Operators (used in Expressions)

VHDL 2008

Data Aggregation

```
vect(4 downto 0) <= "11010";
vect(0 to 4)     <= "01011";
vect <= (0|2 => '0', others => '1');
```

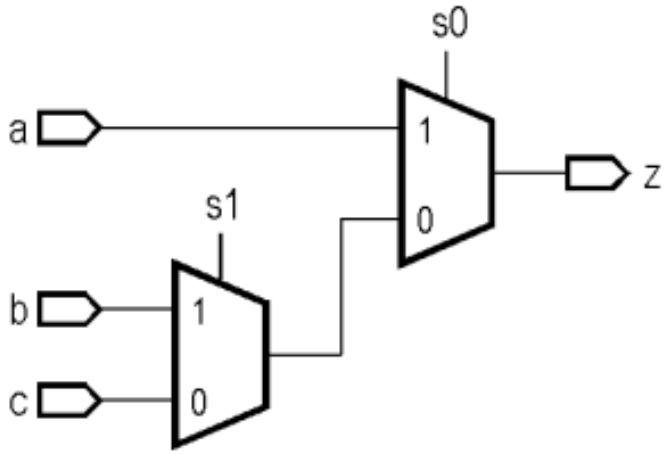
```
and vect(2 downto 0) → vect(2) and vect(1) and vect(0)

"0011" or not '0' → "0011" or ('1') → "1111"
xnor "1110" → not (xor "1110") → not ('1' xor '1' xor '1' xor '0') → '0'

bit <= and  vec_a or vec_b   -- Illegal! Reduction AND is evaluated first
bit <= and (vec_a or vec_b)  -- Legal ALWAYS USE PARENTHESIS
```

Writing VHDL: Rules

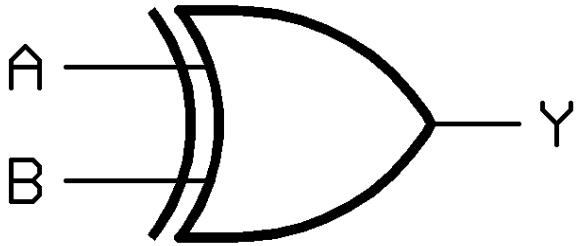
Signal assignment

Outside Processes -> Concurrent

```
-- conditional signal assignment
z <= a when s0 = '1' else
     b when s1 = '1' else
     c;
```




```
-- selected signal assignment
sig <= A & B;
with sig select
  Y <= '0' when "00",
  Y <= '1' when "01",
  Y <= '1' when "10",
  Y <= '0' when "11";
```



Writing VHDL: Rules

```
-- loop and exit statements
loop
  exit when value <= 0.0;
  value := value / 2.0;
end loop;
```

```
-- while and next statements
while value > 0 loop
  next when value = 5;
  value := value / 2;
end loop;
```

```
-- for statement
for I in 16 downto 0 loop
  for J in 0 to 8 loop
    tab(I, J) := I*J + 5;
  end loop;
end loop;
```

Variable assignment

Sequential statements: Only inside "process"!

```
-- if statement
if A > B then
  Max <= A;
elsif A < B then
  Max <= B;
else
  Max <= 0;
end if;
```

```
-- case statement
case Int is
  when 0       => B <=  4;
  when 1|2|7   => B <= 10;
  when 3 to 6  => B <=  5;
  when 9       => null;
  when others  => B <=  0;
end case;
```

```
process (SEL, A, B, C, D)
  case SEL is
    when "00" =>   MUX_OUT <= A;
    when "01" =>   MUX_OUT <= B;
    when "10" =>   MUX_OUT <= C;
    when "11" =>   MUX_OUT <= D;
    when others => MUX_OUT <= (others => '0');
  end case;
end process;
```

Writing VHDL: Rules

Useful functions defined in `std_logic_1164`

- **Conversion functions:**

```
to_bit(s : std_ulogic)
to_bitvector(s : std_ulogic_vector)
to_stdlogic(s : bit)
to_stdlogicvector(s : bit_vector)
```

- `std_logic_vector` is not compatible with `std_ulogic_vector`
  use **type casting**:

```
A <= std_logic_vector(B);
B <= std_ulogic_vector(A);
```

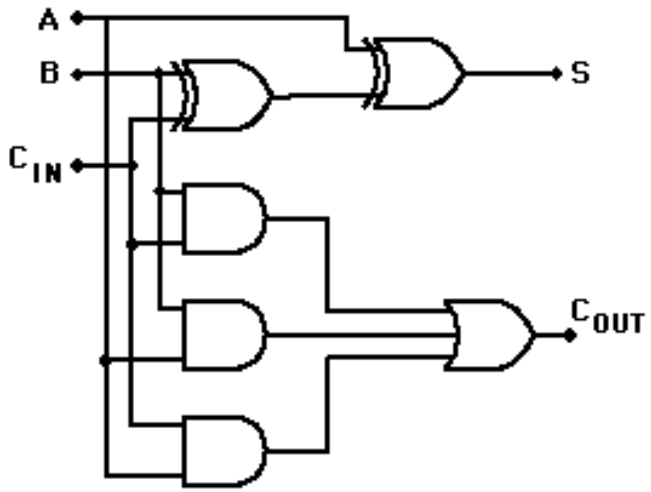
- Also in the package, **edge detecting** functions: `rising_edge()`, `falling_edge()`

The `std_logic_1164` package does not contain any functions for arithmetic operations

Writing VHDL: Examples

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

entity FullAdder is
  port (
    a   : in  std_logic;
    b   : in  std_logic;
    cin : in  std_logic;
    co  : out std_logic;
    s   : out std_logic
  );
end entity;

architecture dataflow of FullAdder is
begin
  s    <= a xor b xor cin;
  cout <= (a and b) or (a and cin) or
          (b and cin);
end architecture;
```

VHDL example: Full Adder



Writing VHDL: Examples

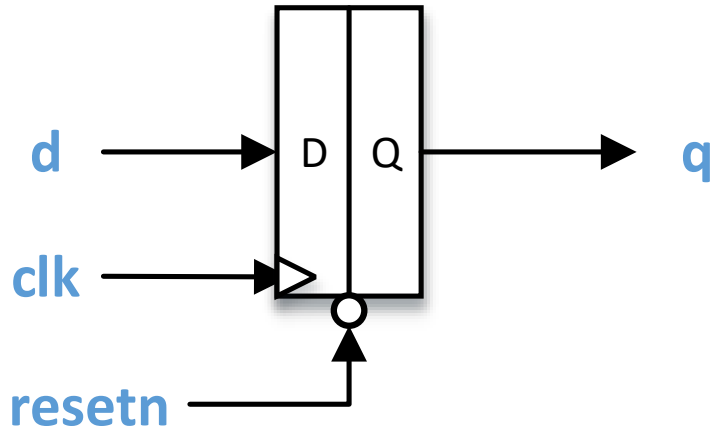
```vhdl
entity DFC is
  port (
    clk    : in  std_logic;
    resetn : in  std_logic;
    d      : in  std_logic;
    q      : out std_logic
  );
end entity;

architecture rtl of DFC is
begin

  p_DFC: process(clk, resetn)
  begin
    if resetn = '0' then
      q <= '0';
    elsif (clk'event and clk = '1') then
      q <= d;
    end if;
  end process;

end architecture;
```

Sequential process

Asynchronous negative reset

Synchronous section, on **positive edge** of `clk`

VHDL example: DFC




```vhdl
if (clk'event and clk = '1') then
```

```vhdl
if (clk'event and clk = '0') then
```

```vhdl
if rising_edge(clk) then
```

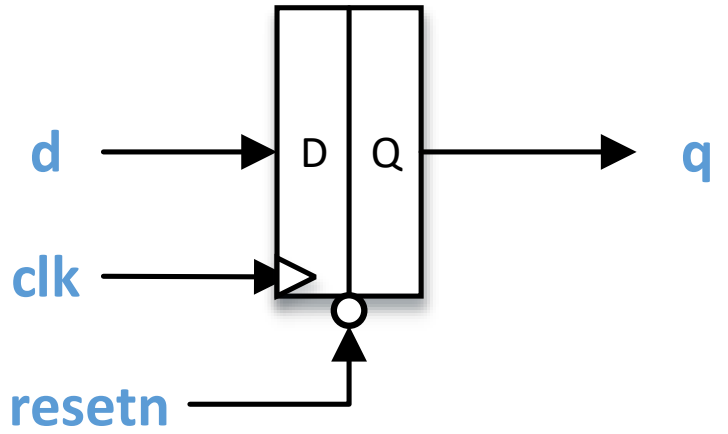
```vhdl
if falling_edge(clk) then
```

Writing VHDL: Examples

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

architecture rtl of DFC is
begin

  p_DFC: process(clk)
  begin
    if rising_edge(clk) then
      if resetn = '0' then
        q <= '0';
      else
        q <= d;
      end if;
    end if;
  end process;

end architecture;
```

Synchronous negative reset

VHDL example: DFC SynRst

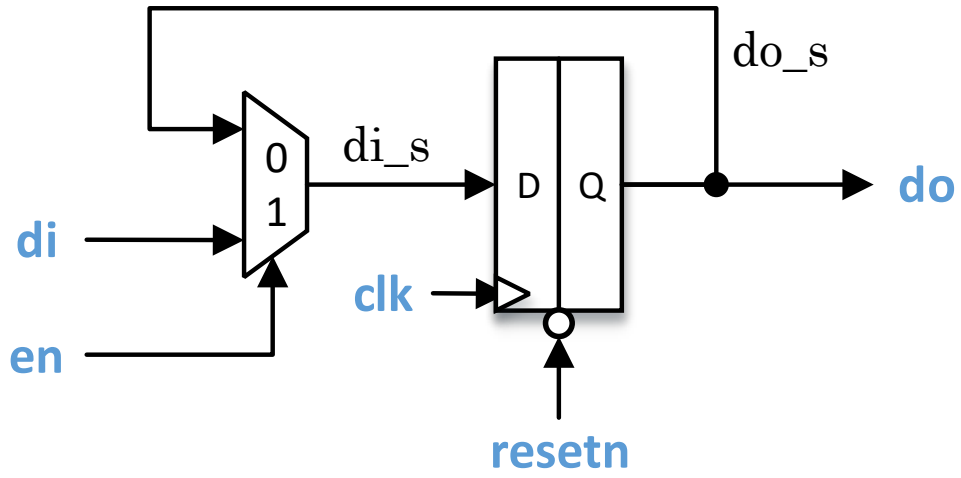


Writing VHDL: Examples

```
entity DFCE is
  port (
    clk    : in  std_logic;
    resetn : in  std_logic; en : in  std_logic;
    di     : out std_logic; do : out std_logic
  );
end entity;

architecture rtl of DFCE is
  -- internal signals
  signal di_s : std_logic;
  signal do_s : std_logic;
begin
  p_DFCE: process(clk, resetn)
  begin
    if resetn = '0' then
      do_s <= '0';
    elsif rising_edge(clk) then
      do_s <= di_s;
    end if;
  end process;

  di_s <= di when en = '1' else do_s;
  do   <= do_s;
end architecture
```

Data Flow MUX

VHDL example: DFCE
positive edge-triggered with enable



Port of type `out` cannot be read (stay on the right of the assignment).

Use internal net (`signal`).

Writing VHDL: Examples

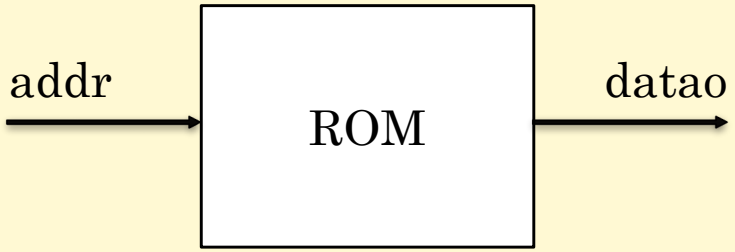
```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity LUT_32x8 is
  port (
    addr  : in  std_logic_vector(4 downto 0);
    datao : out std_logic_vector(7 downto 0)
  );
end entity;

architecture dataflow of LUT_32x8 is
  signal   addr_int : integer range 0 to 31;
  type     lut_t is array (0 to 31) of std_logic_vector(7 downto 0);
  constant lut      : lut_t :=
  ("00011000","00011001","00011010","00011100","00111000","01011001","10011000","11011001",
   "00010000","00010001","00010010","00010100","00110000","01010001","10010000","11010001",
   "00001000","00001001","00001010","00001100","00101000","01001001","10001000","11001001",
   "00000000","00000001","00000010","00000100","00100000","01000001","10000000","11000001");
begin
  addr_int <= to_integer(unsigned(addr));
  datao    <= lut(addr_int);
end architecture;
```

Needed for **unsigned** and **signed** types

Always advisable to specify the **range** for integer signals

New type definition

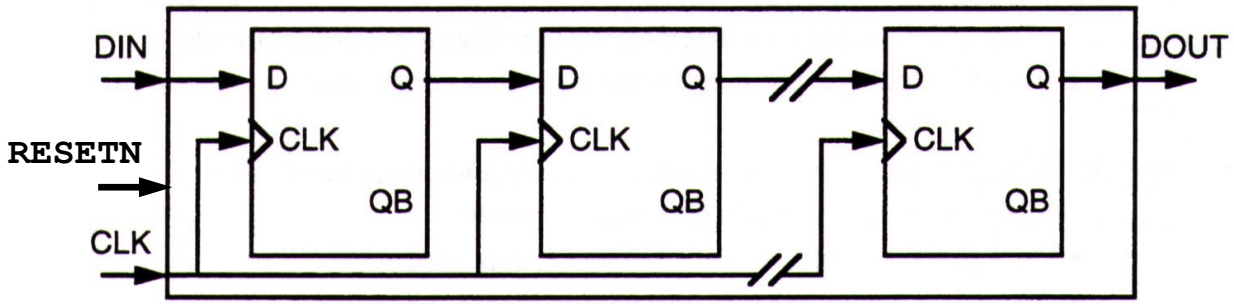
VHDL example: LUT/ROM



Writing VHDL: Examples

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

entity ShiftReg is
  generic (
    Nbit : positive := 8
  );
  port (
    clk    : in  std_logic;
    resetn : in  std_logic;
    din    : in  std_logic;
    dout   : out std_logic
 );
end entity;
```

VHDL example: Shift Register (1/2)



Generic Parameter

Attribute

Integer Subtypes

```
subtype natural  is integer range 0 to integer'high;
subtype positive is integer range 1 to integer'high;
```

Writing VHDL: Examples

```
architecture structural of ShiftReg is
  component
    port (
      clk    : in  std_logic;
      resetn : in  std_logic;
      d      : out std_logic;
      q      : out std_logic
    );
  end component;

  signal q_s : std_logic_vector(Nbit-1 downto 1);

begin
  -- generation of N instances of the D flip-flop
  g_DFC: for i in 1 to Nbit generate
    g_FIRST: if i = 1 generate -- first cell
      i_FF1: DFC port map (clk, resetn, din, q_s(i));
    end generate;

    g_INTERNAL: if i > 1 and i < Nbit generate
      i_FF1: DFC port map (clk, resetn, q_s(i-1), q_s(i));
    end generate;

    g_LAST: if i = Nbit generate -- last cell
      i_FF1: DFC port map (clk, resetn, q_s(i-1), dout);
    end generate;
  end generate;

end architecture;
```

VHDL example: Shift Register (2/2)

Generate Statement:
used to **instantiate** components by sequence or static **conditions**.

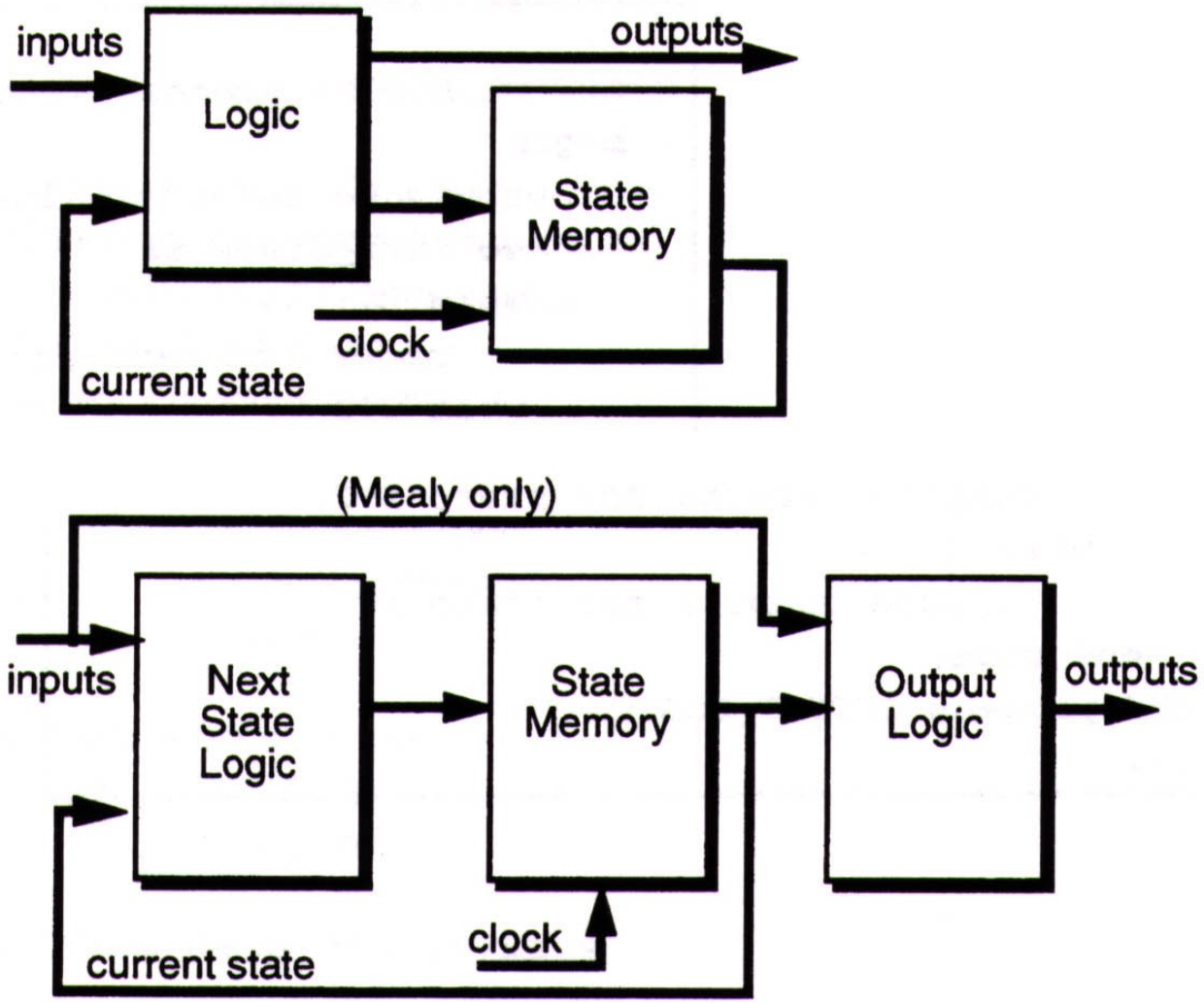
Writing VHDL: Examples

VHDL example: Shift Register (2/2)

LABELS are needed for:
- processes
- generate
- component instances

```
...
begin
  -- generation of N instances of the D flip-flop
  g_DFC: for i in 1 to Nbit generate
    g_IRST: if i = 1 generate -- first cell
      i_FF1: DFC port map (...);
    end generate;

    g_INTERNAL: if i > 1 and i < Nbit generate
      i_FF1: DFC port map (...);
    end generate;

    g_LAST: if i = Nbit generate -- last cell
      i_FF1: DFC port map (...);
    end generate;
  end generate;
end architecture;
```

Writing VHDL: Examples



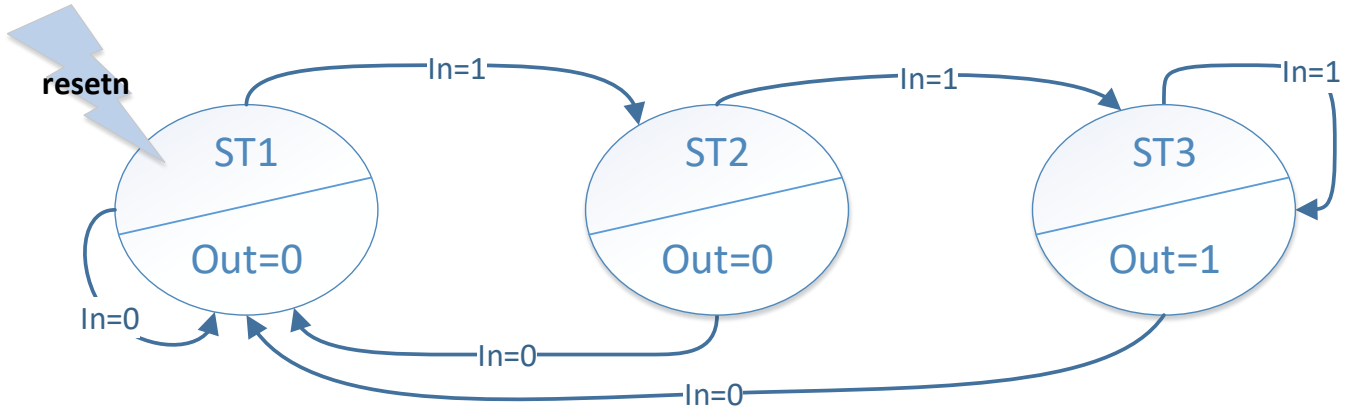
VHDL example:
Finite State Machine (FSM)

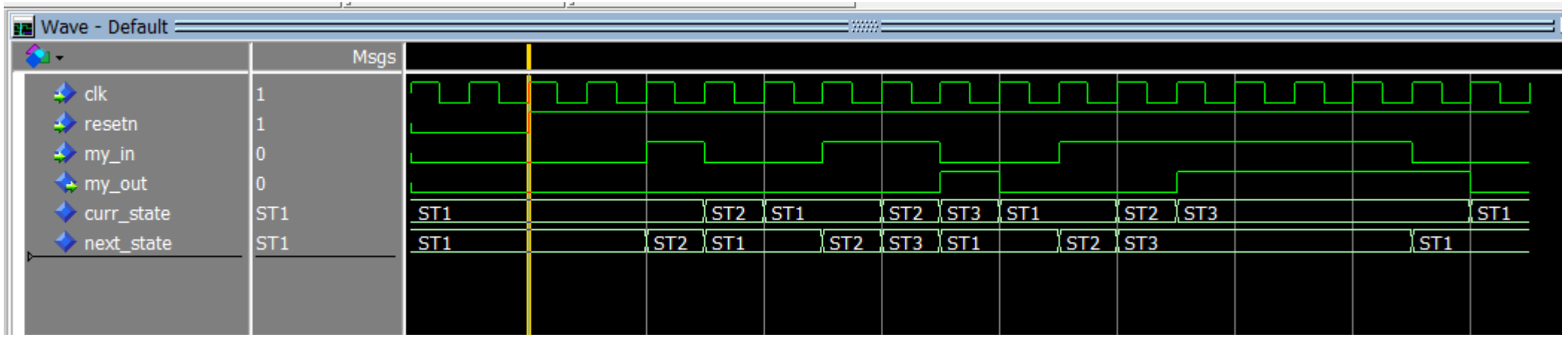
Different **processes** can be used for:
- Combinational logic
- State memory update

Writing VHDL: Examples

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

entity FSM is
  port (
    clk    : in  std_logic;
    resetn : in  std_logic;
    my_in  : in  std_logic;
    my_out : out std_logic
  );
end entity;
```

VHDL example:
Finite State Machine (FSM) (1/4)





Writing VHDL: Examples

```
architecture three_proc of FSM is

  type state is (ST1, ST2, ST3);
  signal curr_state, next_state : state;

begin

  p_STATE_REG: process (clk, resetn)
  begin
    if resetn = '0' then
      -- initial state, asynch resetn
      curr_state <= ST1;
    elsif rising_edge(clk) then
      curr_state <= next_state;
    end if;
  end process;

  ...
```

VHDL example:
Finite State Machine (FSM) (2/4)

1st process (sequential):
State memory update

Writing VHDL: Examples

```
p_NEXT_STATE_LOGIC: process (curr_state, my_in)
begin

  -- default, hold in current state
  next_state <= curr_state;

  case curr_state is
    when ST1 =>
      if my_in = '1' then
        next_state <= ST2;
      end if;
    when ST2 =>
      if my_in = '1' then
        next_state <= ST3;
      else
        next_state <= ST1;
      end if;
    when ST3 =>
      if my_in = '0' then
        next_state <= ST1;
      end if;
  end case;
end process;
```

VHDL example:
Finite State Machine (FSM) (3/4)

Always create a *default condition* for combinational processes to **avoid** inferred **latches**

2nd process (combinational):
Next State logic

Writing VHDL: Examples

```
  p_OUTPUT_LOGIC: process (curr_state)
  -- this is moore, mealy:   (curr_state, my_in)
  begin

    -- default
    my_out <= '0';

    case curr_state is
      when ST1 => null;
      when ST2 => null;
      when ST3 => null;
        my_out <= '1';
    end case;
  end process;

end architecture;
```

VHDL example:
Finite State Machine (FSM) (4/4)

3rd process (combinational):
Output logic

Writing VHDL: Examples

```
architecture dataflow of MyEntity is

  signal my_vec   : std_logic_vector(15 downto 0);
  signal num_zero : std_logic_vector(3 downto 0);

begin

  PROC: process (my_vec)
    variable zero_count : integer;
  begin
    zeros_count := 0;
    for i in 0 to my_vec'length loop
      if my_vec(i) = '0' then
        zero_count := zero_count + 1;
      end if;
    end loop;

    num_zero <=
     std_logic_vector(
      to_unsigned(zero_count, num_zero'length));
  end process;

end architecture;
```

VHDL example:
Signals vs Variables

Variables:

- *Immediately* take the value of their assignment

Signals:

- *Immediately*, if used in combinatorial code
- *After process elaboration*, if used in sequential code

Agenda

1. Levels of Abstraction
2. Digital Design Technology Map
3. VHDL Introduction
4. VHDL File:
   I. Library
   II. Entity
   III. Architecture
5. Writing VHDL: Rules & examples
6. Simulate VHDL

VHDL timing model

Time model

VHDL time is an integer multiple of the selected **resolution limit.**
The minimum resolution limit of 1 fs of physical type TIME.
Secondary units are possible for a longer period of simulation time.

unit	resolution	Tmax (32 bits)	Tmax (64 bits)
fs	1 e-15	2 microseconds	3 hours
ps	1 e-12	2 milliseconds	3 years
ns	1 e-9	2 seconds	300 years
…	…	…	…

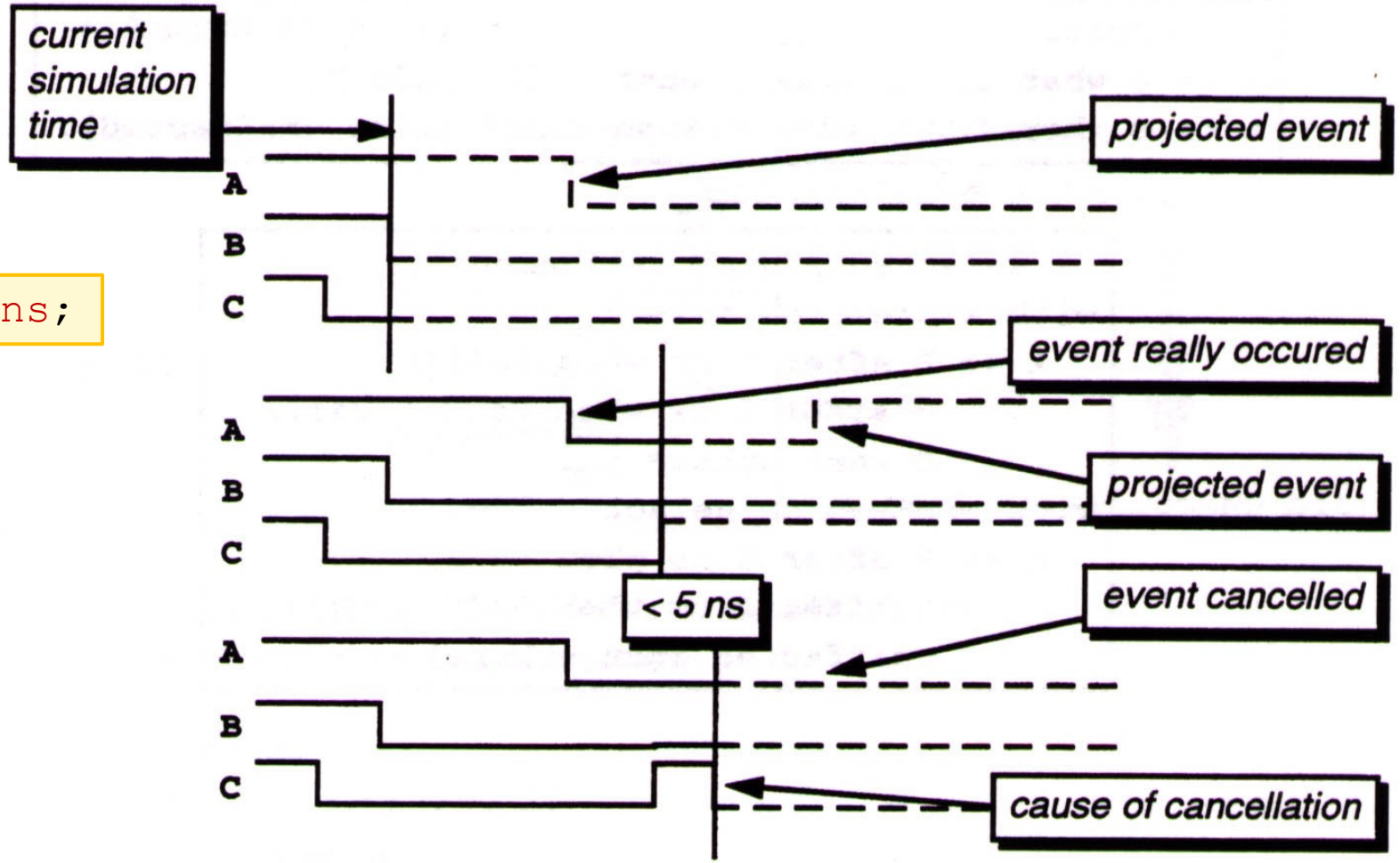
A signal assignment may occur:

- Without any delay or with a delay of 0 fs --> **delta delay**
- With a delay that models a switching time --> **inertial delay**
- With a delay that models a transmission line --> **transport delay**

VHDL timing model

```
A <= B or C after 5 ns;
```

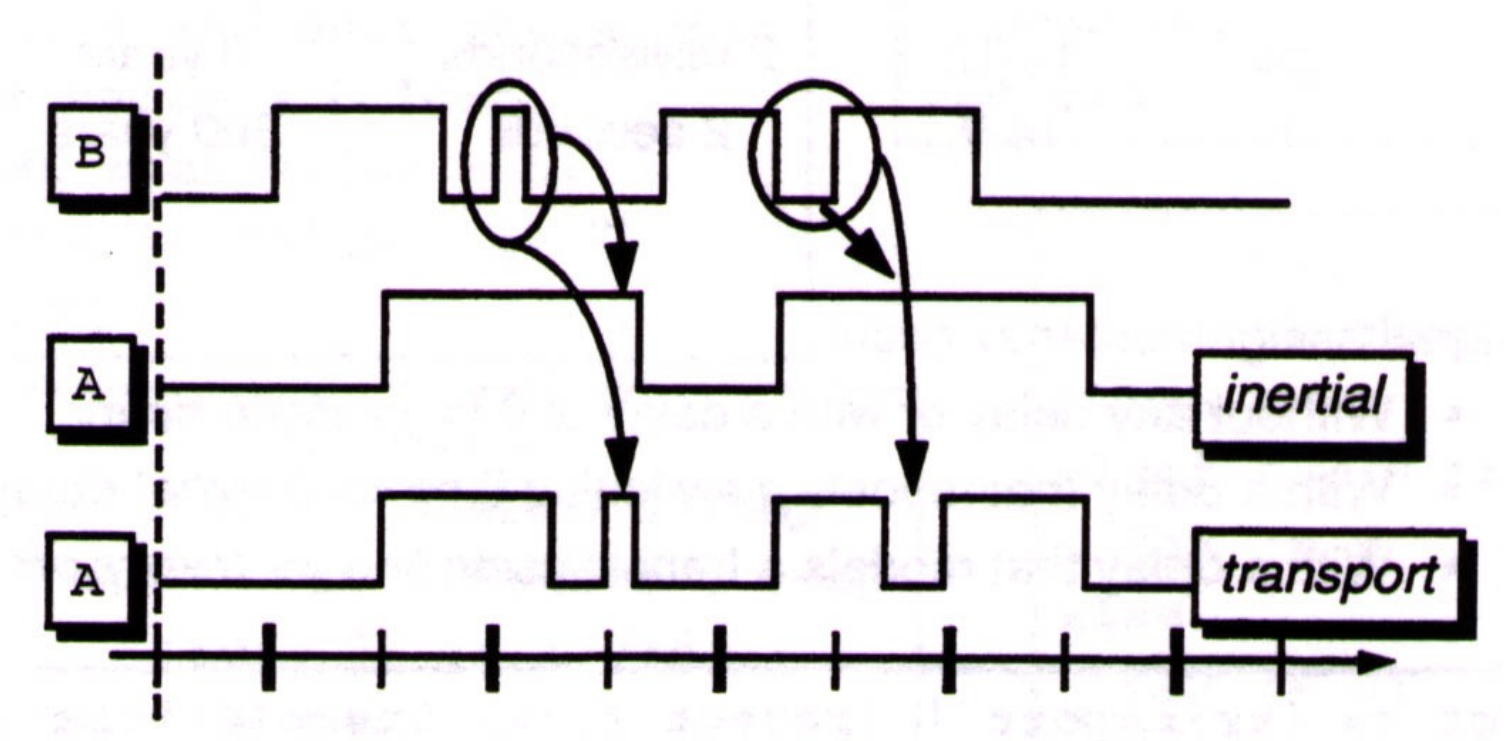
Inertial delay
if not specified



VHDL timing model

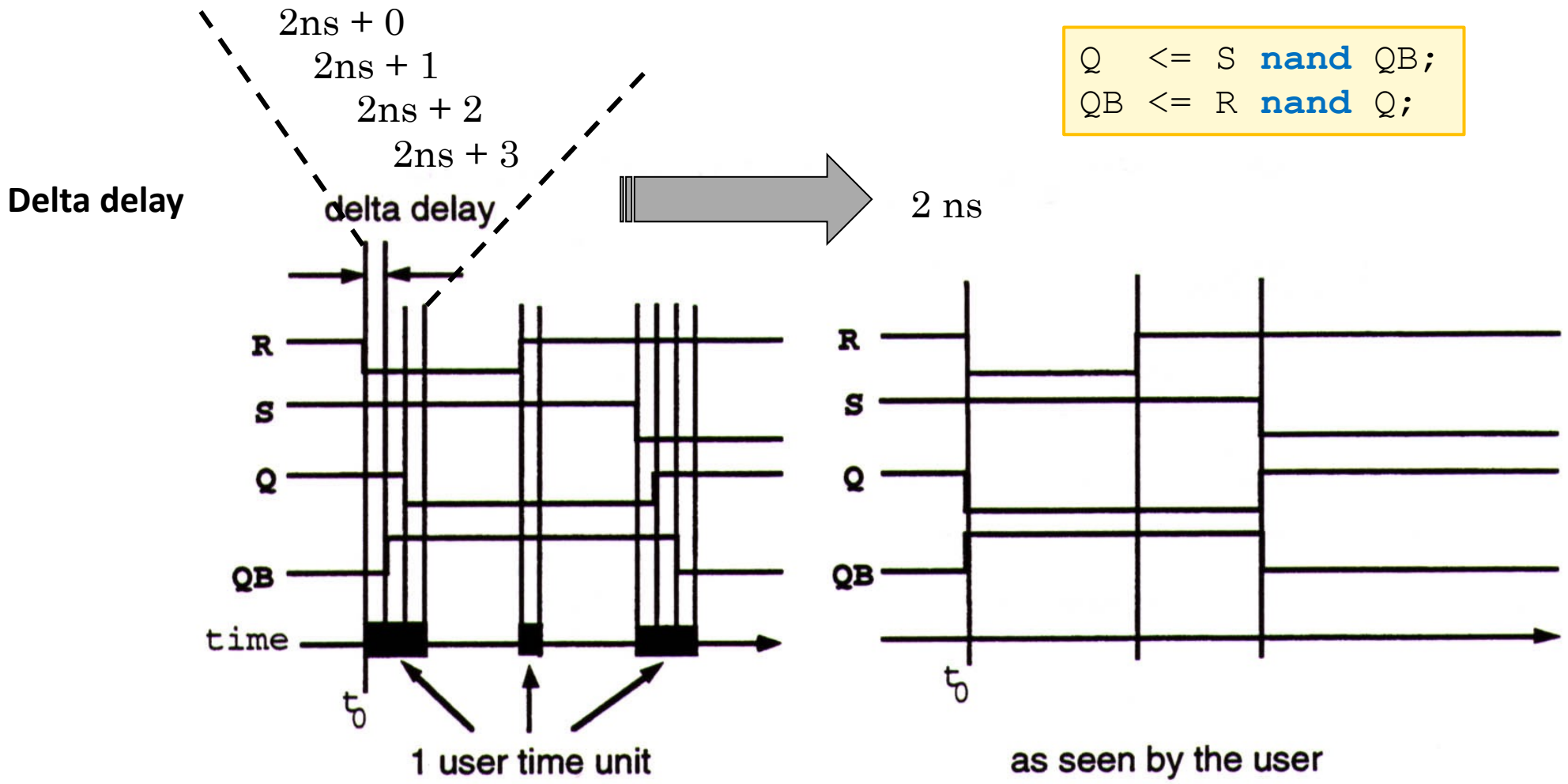
```
A <= inertial B after 10 ns;
```

```
A <= transport B after 10 ns;
```



VHDL timing model

```
Q  <= S nand QB;
QB <= R nand Q;
```



VHDL timing model: Signals Vs Variables

Variables

- Used for local storage in processes, procedures and functions
- Must be declared within the process in which they are used (local)

Signals

- Must be declared outside processes
- Can be used anywhere within the architecture

Assignment

```
variable_name := expression;
```

- The expression is evaluated, and the variable instantly updated

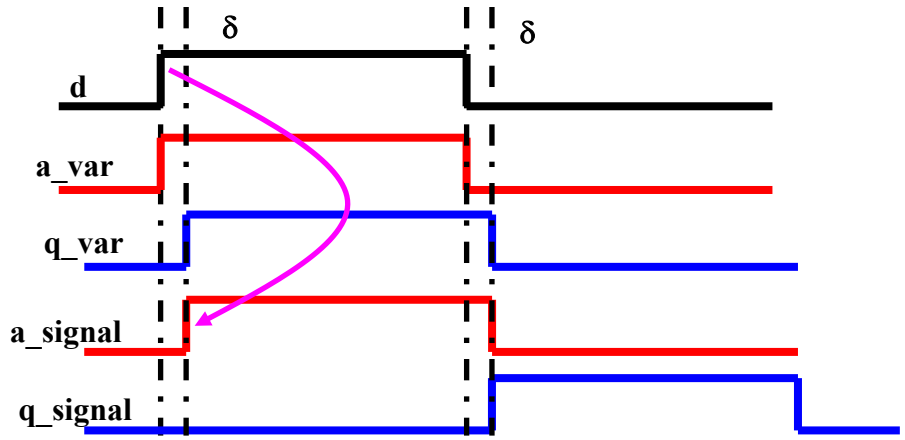
```
signal_name <= expression [after delay];
```

- The expression is evaluated, and the signal is ***scheduled*** to change after delay (**delta delay** if not specified)

VHDL timing model: Signals Vs Variables

```
entity sv is
   port( d        : in  std_logic;
         q_signal : out std_logic;
         q_var    : out std_logic);
end entity;

architecture behav of sv is
  signal a_signal : std_logic := '0';
begin
  P1: process(d)
    variable a_var : std_logic;
  begin
    a_signal <= d;          -- after δ cycle
    a_var    := d;
    q_signal <= a_signal; -- after δ cycle
    q_var    <= a_var;    -- after δ cycle
  end process;
end architecture;
```

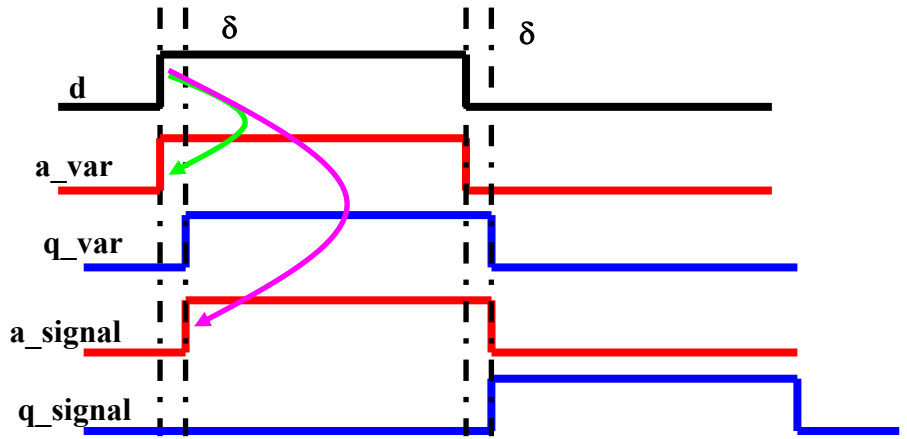
Assignments made to signals inside a process are scheduled only when the same process is completed.
The actual assignment is not made until after the process terminates (delta-cycle).
Assignments made to variables are instantly resolved.



VHDL timing model: Signals Vs Variables

```
entity sv is
   port( d        : in  std_logic;
         q_signal : out std_logic;
         q_var    : out std_logic);
end entity;

architecture behav of sv is
  signal a_signal : std_logic := '0';
begin
  P1: process(d)
    variable a_var : std_logic;
  begin
    a_signal <= d;          -- after δ cycle
    a_var    := d;
    q_signal <= a_signal;   -- after δ cycle
    q_var    <= a_var;      -- after δ cycle
  end process;
end architecture;
```

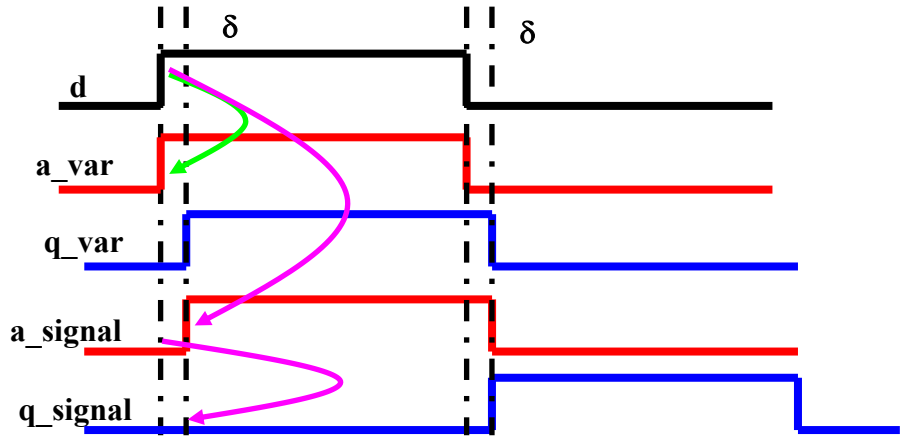
Assignments made to signals inside a process are scheduled only when the same process is completed.
The actual assignment is not made until after the process terminates (delta-cycle).
Assignments made to variables are instantly resolved.



VHDL timing model: Signals Vs Variables

```
entity sv is
   port( d         : in  std_logic;
         q_signal  : out std_logic;
         q_var     : out std_logic);
end entity;

architecture behav of sv is
  signal a_signal : std_logic := '0';
begin
  P1: process(d)
    variable a_var : std_logic;
  begin
    a_signal <= d;          -- after δ cycle
    a_var    := d;
    q_signal <= a_signal;   -- after δ cycle
    q_var    <= a_var;      -- after δ cycle
  end process;
end architecture;
```

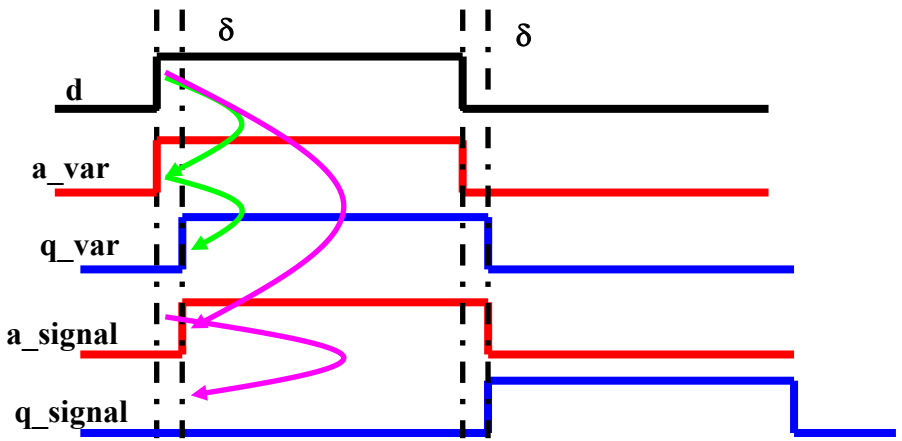
Assignments made to signals inside a process are scheduled only when the same process is completed.
The actual assignment is not made until after the process terminates (delta-cycle).
Assignments made to variables are instantly resolved.



VHDL timing model: Signals Vs Variables

```
entity sv is
   port( d        : in  std_logic;
         q_signal : out std_logic;
         q_var    : out std_logic);
end entity;

architecture behav of sv is
  signal a_signal : std_logic := '0';
begin
  P1: process(d)
    variable a_var : std_logic;
  begin
    a_signal <= d;        -- after δ cycle
    a_var    := d;
    q_signal <= a_signal; -- after δ cycle
    q_var    <= a_var;    -- after δ cycle
  end process;
end architecture;
```

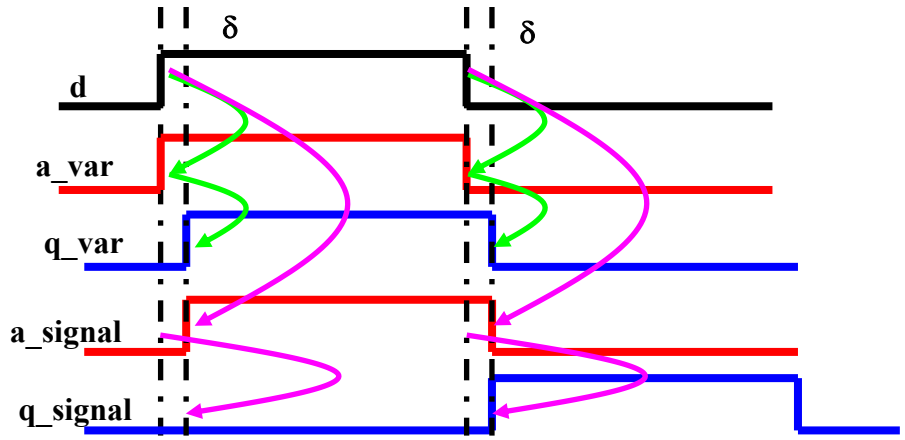
Assignments made to signals inside a process are scheduled only when the same process is completed.
The actual assignment is not made until after the process terminates (delta-cycle).
Assignments made to variables are instantly resolved.



VHDL timing model: Signals Vs Variables

```
entity sv is
   port( d        : in  std_logic;
         q_signal : out std_logic;
         q_var    : out std_logic);
end entity;

architecture behav of sv is
  signal a_signal : std_logic := '0';
begin
  P1: process(d)
    variable a_var : std_logic;
  begin
    a_signal <= d;          -- after δ cycle
    a_var    := d;
    q_signal <= a_signal;   -- after δ cycle
    q_var    <= a_var;      -- after δ cycle
  end process;
end architecture;
```

Assignments made to signals inside a process are scheduled only when the same process is completed.
The actual assignment is not made until after the process terminates (delta-cycle).
Assignments made to variables are instantly resolved.



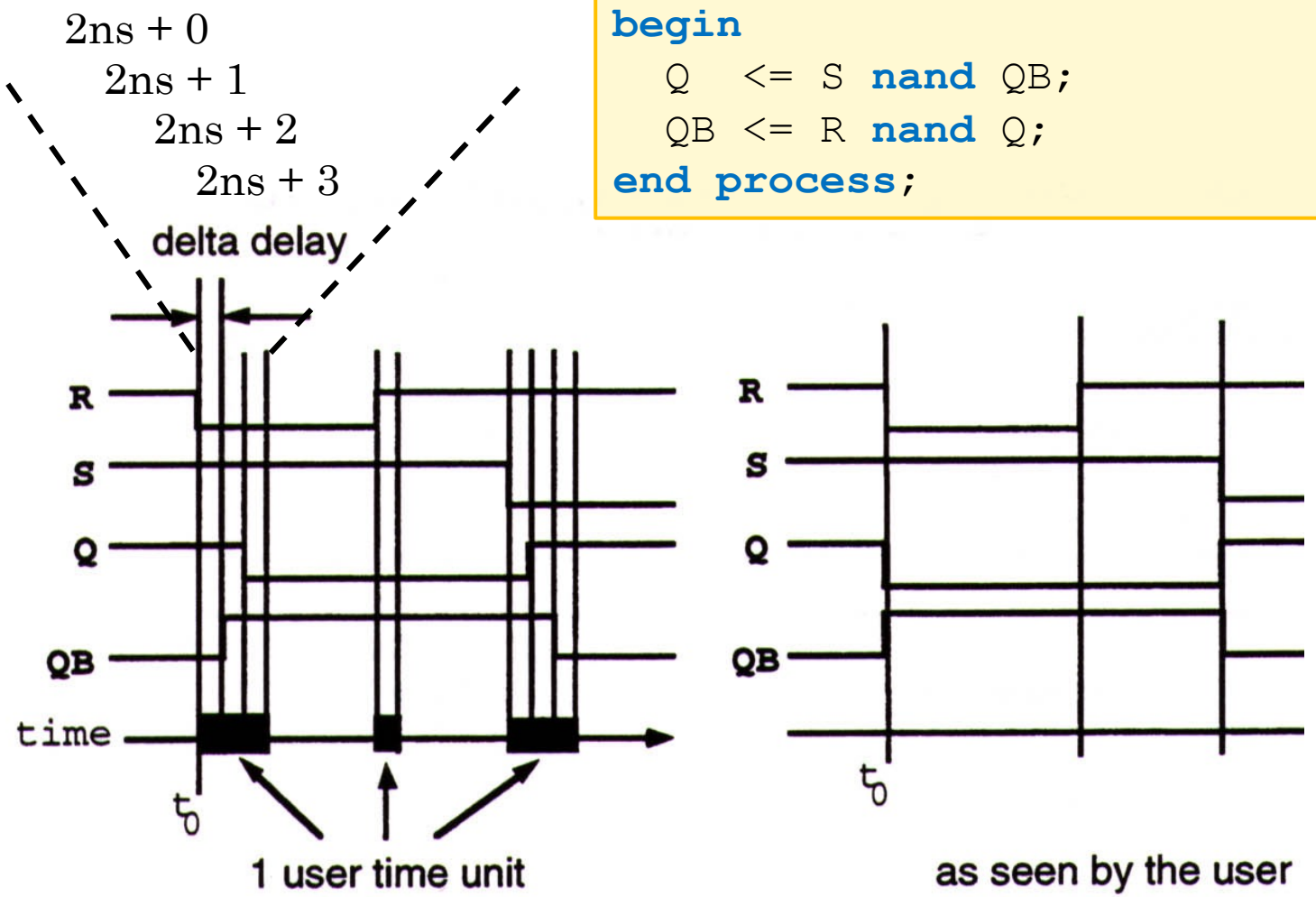
VHDL timing model: Signals Vs Variables

Beware of:

- Variables are often hard to synthesize by your tool (i.e. Vivado). Try to avoid using them unless it is necessary.
- You will not notice (by default) on your simulator (Modelsim) the delta delay proper of signal assignment.
- Overusing combinational logic with feedback paths may create **combinational loops**. 
- Signals are updated after a number of delta delays, dependent on the assignment we are making. Anyway at the end of the execution of the process delta delays are flattened. See the example on the next slide.

VHDL timing model: Signals Vs Variables

```
PROC: process(S, R, Q, QB)
begin
   Q  <= S nand QB;
   QB <= R nand Q;
end process;
```



Note:

- The process is executed 3 times at t_0, each execution correspond to a delta delay
- The two assignment, made outside a process, would be traduced by the compiler to the same process described here
- W/o Q and QB in the sensitivity list, only the first variation of QB would be present!

VHDL timing model: Combinational loop

Combinational loop example

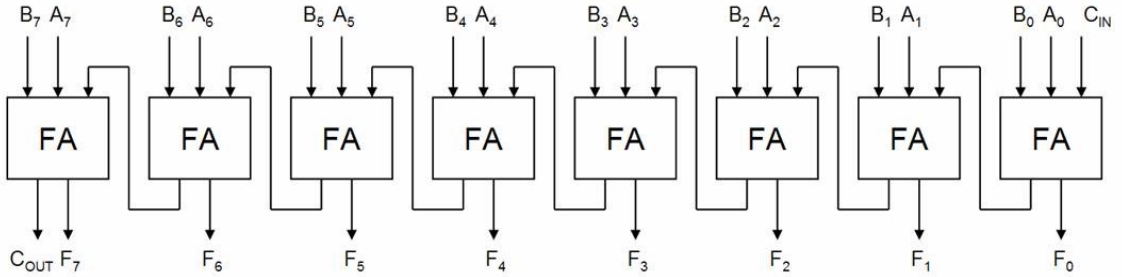
```
library ieee;
use ieee.std_logic_1164.all;

entity loop_tb is
end entity;

architecture beh of loop_tb is
  signal clk : std_logic := '0';
begin
  clk <= not clk;
end architecture;
```

**** Error (suppressible): (vsim-3601) Iteration limit 10000000 reached at time 0 ns.**

VHDL timing model: variable example

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

entity ripple_carry_adder is
  generic (
    Nbit : positive := 8
  );
  port (
    a    : in  std_logic_vector(Nbit - 1 downto 0);
    b    : in  std_logic_vector(Nbit - 1 downto 0);
    cin  : in  std_logic;
    s    : out std_logic_vector(Nbit - 1 downto 0);
    cout : out std_logic
  );
end entity;

architecture beh of ripple_carry_adder is
begin

  p_ADDER: process(a, b, cin)
    variable c : std_logic;
  begin
    c := cin;
    for I in 0 to Nbit - 1 loop
      s(i) <= a(i) xor b(i) xor c;
      c    := (a(i) and b(i)) or (a(i) and c) or (b(i) and c);
    end loop;

    cout <= c;
  end process;

end architecture;
```

Ripple Carry Adder

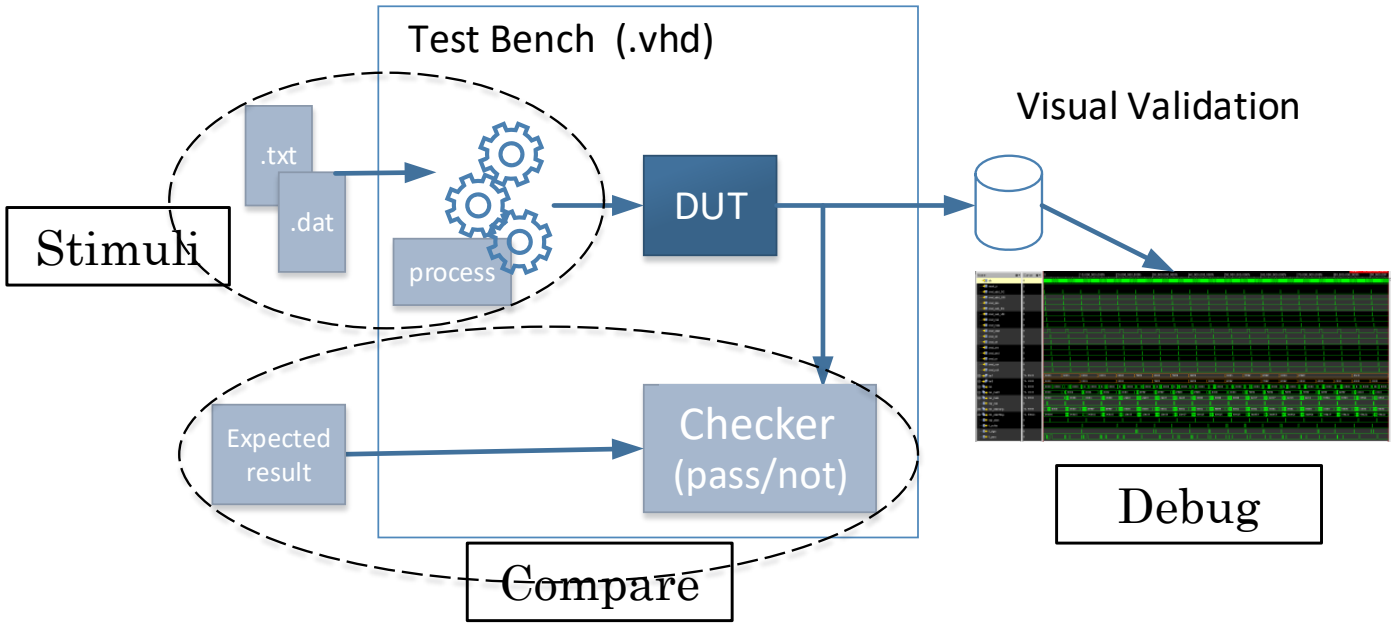


Simulate VHDL

Test Bench

A **test bench** is a model that is used to exercise and verify the correctness of a hardware model. The expressive power of the VHDL language provides us with the capability of writing test bench models also in the same language. A test bench has three main purposes:

1) to ***generate*** stimuli for simulation (waveforms, test vector, stimuli file).
2) to ***apply*** these stimuli to the Design Under Test (DUT) and to monitor the output responses.
3) to ***compare*** output responses with expected known values.



Again, the language provides many ways to write a test bench.

Simulate VHDL

Data Objects

A data object holds a value of a specified data type. Every data object belongs to one of this class:

- *Constant*

```
constant RISE_TIME : TIME := 10 ns;
constant BUS_WIDTH : INTEGER := 8;
constant PI : REAL := 3.1416;
```

- *Variable*

(used in processes, values are immediately updated)

```
variable CTRL_STATUS : BIT_VECTOR(10 downto 0);
variable SUM : INTEGER range 0 to 100 := 10;
variable FOUND, DONE : BOOLEAN;
```

- *Signal*

(models wires)

```
signal CLOCK : BIT;
signal DATA_BUS : BIT_VECTOR(0 to 7);
signal GATE_DELAY : TIME := 10 ns;
```

Simulate VHDL: Test bench

Analysis of the example

The structure is standard, only the implementation of the stimuli process may vary

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

entity ripple_carry_adder_tb is
  generic (...);
end entity;
```

- Libraries
- Empty entity (only generic)

Simulate VHDL: Test bench

```
architecture beh of ripple_carry_adder_tb is

  constant clk_period : time     := 100 ns;
  constant N          : positive := 8;

  component ripple_carry_adder
    generic (
      Nbit : positive
    );
    port (
      a    : in  std_logic_vector(Nbit-1 downto 0);
      b    : in  std_logic_vector(Nbit-1 downto 0);
      cin  : in  std_logic;
      s    : out std_logic_vector(Nbit-1 downto 0);
      cout : out std_logic
    );
  end component;

  signal clk      : std_logic := '0';
  signal a_ext    : std_logic_vector(N-1 downto 0) := (others => '0');
  signal b_ext    : std_logic_vector(N-1 downto 0) := (others => '0');
  signal cin_ext  : std_logic := '0';
  signal s_ext    : std_logic_vector(N-1 downto 0) := (others => '0');
  signal cout_ext : std_logic;
  signal testing  : boolean := true;
```

Architecture:

- Top level component declaration
- Signals & constant declaration
- **Input signals** shall be initialised

Simulate VHDL: Test bench

Architecture:

- Clock signal “continuous” assignment
- Component instance

```
begin
  clk <= not clk after clk_period/2 when testing else '0';

  DUT: ripple_carry_adder
    generic map (
      Nbit => N
    )
    port map (
      a    => a_ext,
      b    => b_ext,
      cin  => cin_ext,
      s    => s_ext,
      cout => cout_ext
    );
```

Simulate VHDL: Test bench

Architecture:

- Process without sensitivity list (executed continuously w/o control)
- `wait for` / `wait until`
- When the test is finished, assign *false* to testing variable

Note:
This is just a possible structure for the stimuli process, you can use your own if it works!

```
  STIMULUS: process begin
    a_ext   <= (others => '0');
    b_ext   <= (others => '0');
    cin_ext <= '0';
    wait for 200 ns;
    a_ext   <= "00000110";
    b_ext   <= "00100110";
    cin_ext <= '0';
    wait until rising_edge(clk);
    a_ext   <= x"76";  -- means "0111_0110"
    b_ext   <= x"14";  -- means "0001_0100"
    cin_ext <= '1';
    wait until rising_edge(clk);
    a_ext   <= (others => '0');
    b_ext   <= (others => '0');
    cin_ext <= '0';
    wait for 1008 ns;
    a_ext   <= "11111111";
    b_ext   <= "11111111";
    cin_ext <= '0';
    wait for 500 ns;
    testing  <= false;
  end process;
end architecture;
```

End, Questions?

1) Levels of Abstraction
2) Digital Design Technology Map
3) VHDL Introduction
4) VHDL File
5) Writing VHDL: Rules & examples
6) Simulate VHDL