Computer Structure

Unit 1: Digital design and VHDL

Escuela Politécnica Superior - UAM

Copyright © 2007 Elsevier, "Digital Design and Computer Architecture"

Outline

- **Introduction**
- Combinational logic
- Combinational circuits
- Sequential logic
- Structural modeling
- *Testbenches*

Introduction

- *Hardware description language* (HDL) allows for designing the functionality of a digital circuit without writing logic functions nor gates. Several software tools produce and **synthesize** the circuit that implements that functionality.
- Commercial circuits are designed with HDL languages.
- The two main HDL languages are:
  - **VHDL**
    - Designed in 1981 by the U.S Department of Defense.
    - It was defined as a standard IEEE-1076 in 1987
  - **Verilog**
    - Design in 1984 by Gateway Design Automation.
    - It was defined as a standard IEEE-1364 in 1995

Simulation and synthesis

- **Design and synthesis**
  - Behavioral description of the digital system, sometimes with internal components and their connections
  - It translates the HDL code in a circuit (*netlist*) which describes the hardware (a gate list and interconnecting nets)
- **Simulation**
  - Some values are driven as inputs
  - The outputs are checked
  - Much time/money is saved by debugging the circuit with simulations instead of using real hardware

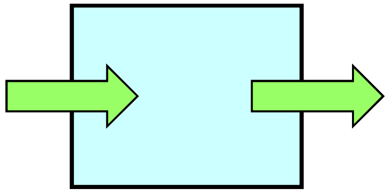
IMPORTANT:

✓ Not every design which can be simulated is also synthesizable.

✓ When describing HDL circuits, it is important to think which is the **hardware** that should be implemented (**different than programming software**).

Entity - Architecture

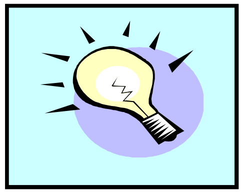
- The entity is used to describe the circuit like a black-box. It only describes its interface (the input and output **ports**)



<u>Equivalent to C</u>

```
float Average(float a, float b) {
    float c;
    c = (a+b) / 2;
    return c;
}
```

- The circuit content is modeled inside the architecture



```
float Average(float a, float b) {
    float c;
    c = (a+b) / 2;
    return c;
}
```

- An entity may have several architectures

PORTS: The interface

PORT

Name
Identifier

Mode
in = Input (it can only be read)
out = Output (it can only be written)
inout •Input port (read) and output port(write) •The read value (Input) is the value received in the port. Another different value may be assigned to its output.

Data type	
A set of values that have a name, so an Object (e.g. a Signal) of a certain Type (e.g. std_logic) may take any value of the set.	
std_logic (*std_logic_1164*)	Values ‘U’, ‘X’, ‘0’, ‘1’, ‘Z’, ‘W’, ‘L’, ‘H’, ‘-’
std_logic_vector (*std_logic_1164*)	Vector of std_logic
Integer (*standard*)	Integer numbers
Boolean (*standard*)	True or false

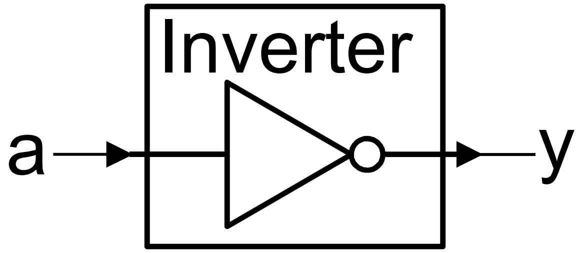
VHDL code example

LIBRARIES

```
library IEEE;                    -- similar to .h define
use IEEE.std_logic_1164.all;     -- To use std_logic
```

ENTITY

```
entity inverter is
   port (a : in std_logic;
         y : out std_logic);
end inverter;
```



ARCHITECTURE

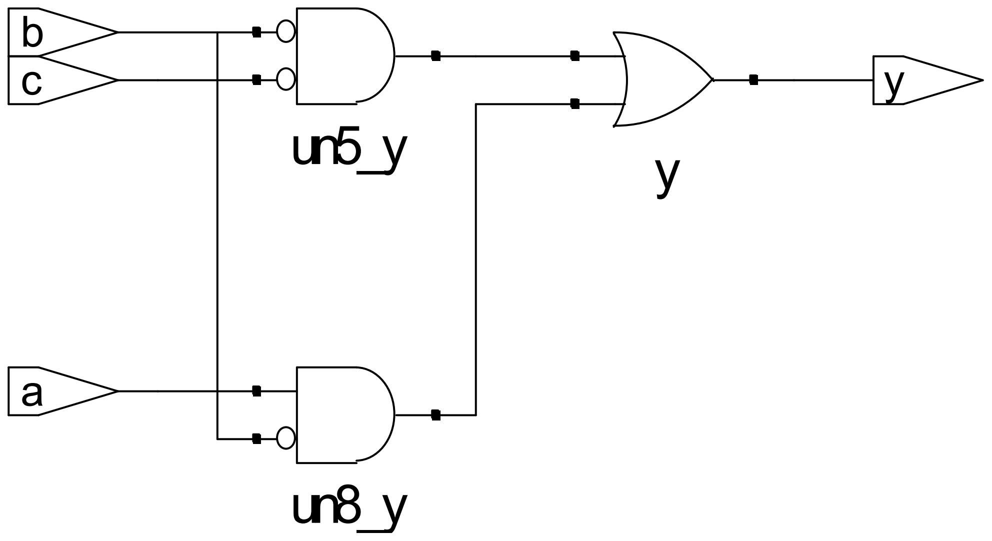
```
architecture behavioral of inverter is
begin
   y <= not a;                     -- assignment with arrow
end behavioral;
```

Synthesis of VHDL code

VHDL:

```
architecture behavioral of example is
begin
   y <= (not a and not b and not c) or (a and not b and not c) or
   (a and not b and c);
end behavioral;
```

Synthesis:



VHDL syntax

- It does not distinguish upper from lower case (*case insensitive*)
  - Example: `reset` and `Reset` are the same signal.
  - Recommendation: Write the same upper and lower-case letters to facilitate the reading and searching.
- The names cannot start with numbers
  - example: `2mux` is not a right name.
- Spaces, tabs and carriage returns are ignored
- Comments:

  -- From a double-hyphen until the end of the line.

  /* Block comment */

VHDL syntax: Identifiers

IDENTIFIER
Names and labels that are used to identiy: Constants, Signals, Processes, Entities, etc.
Length (Number of label characters): Without restrictions
Reserved word in VHDL cannot be identifers
In VHDL, an identifier in upper-case is the same as in lower-case
They are composed by numbers, letters (upper or lower-case) and underscore "_" with the following rules.

Rules to specify an indentifier	*wrong*	*right*
The first character must be always an upper or lower-case letter	4Add	Add4
The second character cannot be an underscore (_)	S_4bits	S4_bits
Two consecutive underscores are not allowed	Sub__4	Sub_4_
An identifier cannot use special symbols	Clear#8	Clear_8

Outline

- Introduction
- **Combinational logic**
- Combinational circuits
- Sequential logic
- Structural modeling
- *Testbenches*

std_logic type

- The values ‘0’ and ‘1’ of bit-type cannot model every state that a real digital signal can take.
- IEEE.std_logic_1164 package defines **std_logic type**, which defines every posible state:

U	Not initialized, default value
X	Unknown, multisource line (short circuit)
0	Logic 0
1	Logic 1
Z	High impedance (tri state)
W	Weak unknown, bus termination
L	Weak 0, pull-down resistor
H	Weak 1, pull-up resistor
–	Don’t care, used as a wildcard in synthesis

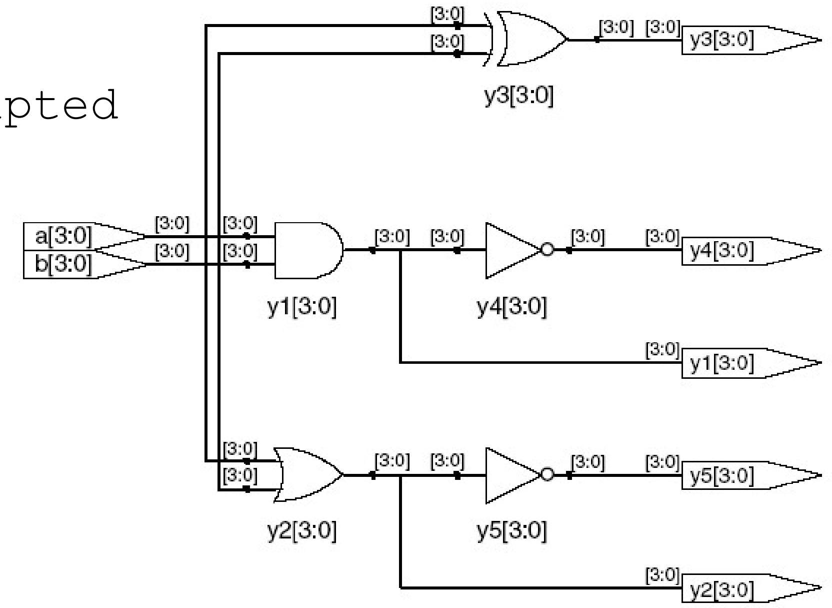
std_logic_vector type defines a *bit array* of **std_logic**.

Unsigned and signed types

- IEEE.numeric_std package define **unsigned** and **signed** types for representing unsigned and signed numbers. These types must be used for arithmetic operations. These types are vectors of **std_logic** equivalent to std_logic_vector but offering the following operations:
- Sign change operations: *“ - ”* and *“ abs ”*
- Arithmetic operations*: “ + ”, “ - ”* and *“ * ”* (multiply)
- Comparison operations: *“ > ”*, *“ < ”*, *“ <= ”*, *“ >= ”*, *“ = ”* and *“ /= ”* (different)
- Logic shifts: *“ sll ”* (left) y *“ srl ”* (right)
- Arithmetic shifts: *“ sla ”* (left) y *“ sra ”* (right)
- Rotations: *“ rol ”* (left) y *“ ror ”* (right)
- Change size (number of bits): *“ resize ”*

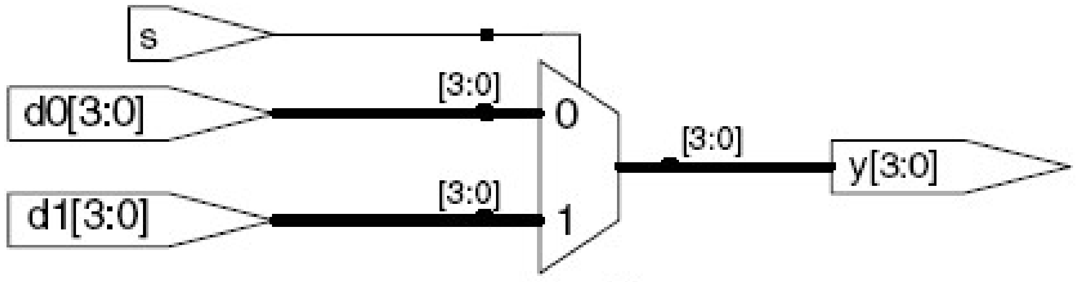
Bitwise operators

```
entity gates is
port (a, b: in std_logic_vector(3 downto 0);
   y1, y2, y3, y4, y5: out std_logic_vector(3 downto 0));
end gates;

architecture behav of gates is
begin
-- They are automatically adapted
-- to one bit or a bus
   y1 <= a and b; -- AND
   y2 <= a or b; -- OR
   y3 <= a xor b; -- XOR
   y4 <= a nand b; -- NAND
   y5 <= a nor b; -- NOR
end behav;
```



Conditional assignment: when … else

```
entity mux2a1_4bits is
port(d0,d1 : in std_logic_vector(3 downto 0);
      s : in std_logic;
      y : out std_logic_vector(3 downto 0));
end mux2a1_4bits;

architecture behav of mux2a1_4bits is
begin
  y <= d0 when s = '0' else d1;
end behav;
```



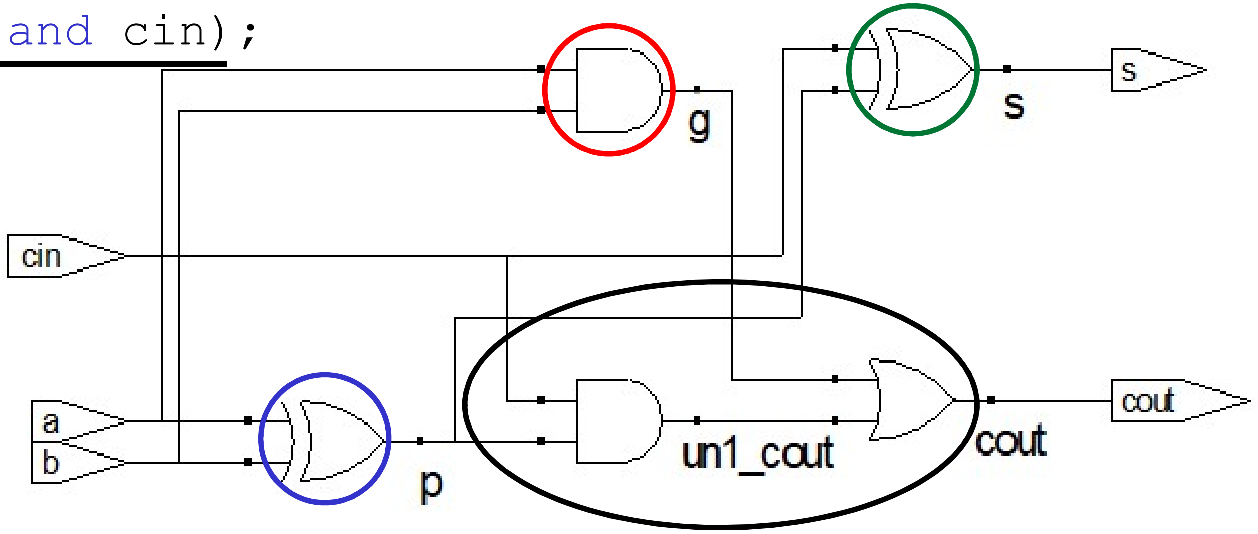
Conditional assignment: with … select

```
entity deco2a4 is
port(a : in std_logic_vector(1 downto 0);
      y : out std_logic_vector(3 downto 0));
end deco2a4;

architecture behav of deco2a4 is
begin

  with a select
        y <= "0001" when "00",
             "0010" when "01",
             "0100" when "10",
             "1000" when others;    -- last case, it should
                                    -- be included
end behav;
```

1.8

Internal signals

```
architecture behav of fulladder is

   signal p, g : std_logic;  -- Internal signal declaration
                             -- goes between architecture and begin
begin
   p <= a xor b;
   g <= a and b;
   s <= p xor cin;
   cout <= g or (p and cin);
end behav;
```



Concurrence

```
architecture behav of fulladder is
   signal p, g : std_logic;
begin
   p <= a xor b;             -- The line order is not
   g <= a and b;             -- important, all the hardware
   s <= p xor cin;           -- is executed simultaneously
   cout <= g or (p and cin);
end behav;
---------------------------------------------------------
architecture behav of fulladder is
   signal p, g : std_logic;
begin
   cout <= g or (p and cin); -- This is the same circuit
   s <= p xor cin;           -- P is assigned in the third
   p <= a xor b;             -- line but hardware
   g <= a and b;             -- is concurrent
end behav;
```

Operator precedence

- Order in which the operators in an expression are evaluated without parentheses
- Recommendation: use parentheses

	Operator	**Operation**
First	`not`	NOT
	`*   /   %`	mult div module
	`+   -`	add subtract
	`<   <=   >   >=`	compare
	`=   /=`	equal different
	`and   nand`	AND NAND
	`xor   xnor`	XOR XNOR
Last	`or   nor`	OR NOR

Number formats and bits

Format	Number of bits	Base	Memory
‘1’	1	Binary	1
“101”	>1	Binary	101
X“AF”	>1	Hexadecimal	10101111
1	It depends	Decimal	0…00001
-2	It depends	Decimal	1…11110 (C2)
1.5	It depends	Decimal	IEEE-754

The signal values are assigned with a left arrow:

```
a_bit <= ‘1’;
a_bus <= “101”;
a_int <= 1;
```

Bit manipulation

```
signal a : std_logic_vector(3 downto 0);
signal b : std_logic_vector(0 to 3);
...
a <= "0101";
b <= "0101";

-- it is equivalent to:
a(3) <= '0';   a(2) <= '1';   a(1) <= '0';    a(0) <= '1';
b(3) <= '1';   b(2) <= '0';   b(1) <= '1';    b(0) <= '0';
```

a

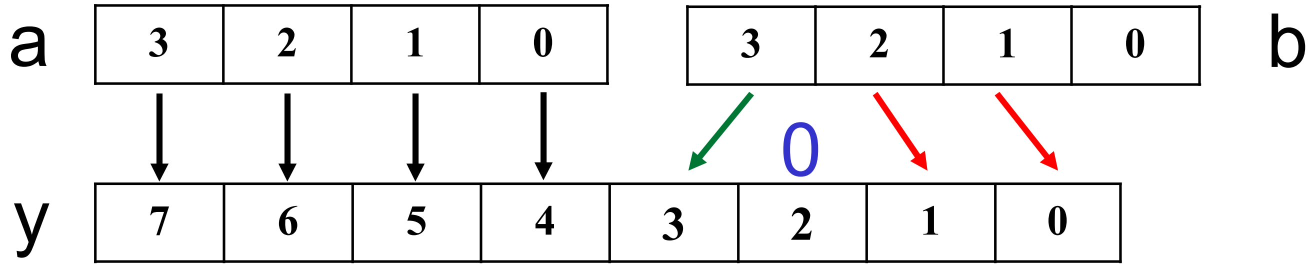
Position	3	2	1	0
Value	0	1	0	1

b

Position	0	1	2	3
Value	0	1	0	1

Bit manipulation

```
signal y : std_logic_vector(7 downto 0);
signal a, b : std_logic_vector(3 downto 0);
...
-- the & operator in VHDL means concatenate
y <= a & b(3) & '0' & b(2 downto 1);

-- it is equivalent to:

y(7) <= a(3);  y(6) <= a(2);  y(5) <= a(1);   y(4) <= a(0);

y(3) <= b(3);  y(2) <= '0';   y(1) <= b(2);   y(0) <= b(1);
```



Bit manipulation

```
signal y : std_logic_vector(7 downto 0);
signal a, b : std_logic_vector(3 downto 0);
...
-- Any partial manipulation can be done while the
-- destination and source lengths match.
y(7 downto 4) <= a;
y(3) <= b(3);
y(2) <= '0';
y(1 downto 0) <= b(2 downto 1);
```

`-- the signal "y" gets the same value as in the`
`-- previous slide`

```
y <= a & b(3) & '0' & b(2 downto 1);
```

Bit manipulation

```
signal y : std_logic_vector(7 downto 0);
signal a, b : std_logic_vector(3 downto 0);
...
-- in VHDL it is allowed to make index-based assignments
-- apart from order-based assignments (left to right).
--(aggregates)
   y <= (3 downto 2 => a(2), 4 => b(1), 5 => '1',
         others => '0');      -- others means "the rest of bits"

-- it is equivalent to:
   y(7) <= '0'; y(6) <= '0'; y(5) <= '1'; y(4) <= b(1);
   y(3) <= a(2); y(2) <= a(2); y(1) <= '0'; y(0) <= '0';

-- common in bus initialization:
   y <= (others => '0'); -- all the bits get '0' value
```

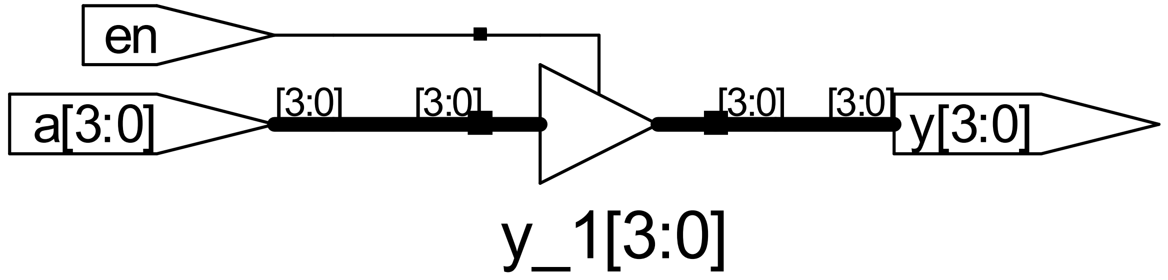
1.3, 1.9

Z: high impedance

VHDL:

```
signal y, a: std_logic_vector(3 downto 0);
signal en : std_logic;
...
y <= (others => 'Z') when en = '0' else a;
```

Synthesis:



Outline

- Introduction
- Combinational logic
- **Combinational circuits**
- Sequential logic
- Structural modeling
- *Testbenches*

Combinational circuits

- Most of combinational circuits can be defined with concurrent assignments (previously seen)
- There is also sequential code (it is executed line by line, like in software, without concurrence, so **the order of the code is important**) but is encapsulated in `processes` (apart from functions and procedures which are not explained in this course)
- Inside the processes (and only inside of them) you can use `if`, `case`, `for` and `while`

Process

“Process” structure:

```
architecture ArchName of EntityName is
   -- declarative part, internal signals of the architecture
begin
-- The processes must be implemented between the “begin” and
-- “end” of the architecture
[LABEL:] process(sensitivity list) -- The label is optional
   -- Declarative part, variables but not signals, for
   -- internal use
  begin
  -- code;
   end process [LABEL];

end ArchName;
```

Every time one signal of the sensitivity list is changed, the ***code*** is executed. From VHDL-2008, the sensitivity list can be ***all***, avoiding to choose specific signals.

if

```
if condition_1 then
  -- sec instr 1
[elsif condition_2 then
  -- sec instr 2]
[elsif condition_3 then
  -- sec instr 3]
[else
  -- instr by default]
end if;
```

Similar to *when…else* conditional assignment

Example

```
CTRL: process(all)-- or (level)
begin
  if level > 60 then
    a <= "11";
  elsif level > 40 then
    a <= "10";
 elsif level > 20 then
    a <= "01";
 elsif level > 15 then
    a <= "00";
  end if;
end process CTRL;
```

Attention: In combinational logic, the "else" branch must be present

case

```
case expression is
  when case_1 =>
    -- instr 1
  when case_2 =>
    -- instr 2
  when others =>
    -- by default
end case;
```

Similar to *with…select* conditional assignment

Example

```
MUX: process(all)
          -- or (sel, a, b, c, d)
begin
  case sel is
    when "11" => y <= d;
    when "10" => y <= c;
    when "01" => y <= b;
    when others => y <= a;
  end case;
end process MUX;
```

Attention: In combinational logic, the "when others" branch must be present

1.2, 1.10

for

```
[LABEL:] for index in range loop
  -- sec instr
end loop [LABEL];
```

Example

```
AND8: process(all)
begin
  for i in 0 to 7 loop
    y(i) <= a(i) and b(i);
  end loop;
end process AND8;
```

```
AND8: process(all)
begin
  for i in 7 downto 0 loop
    y(i) <= a(i) and b(i);
  end loop;
end process AND8;
```

```
                 -- This solution is easier
y <= a and b;    -- because and is a bitwise operator
                 -- Besides, a process is not mandatory
```

while

```
[LABEL:] while condition loop
  -- sec instr
end loop [LABEL];
```

Example

```
process(…)
begin
    while a = '1' loop
       …
       …
       …
    end loop;
end process;
```

When/else and with/select inside a process

```
process(all)
begin
    y <= a and b when sel = ‘0’ else c;
      …
end process;
```

```
process(all)
begin
    with sel select
    y <= “0001” when “00”,
         “0010” when “01”,
         “0100” when “10”,
         “1000” when others;
    …
end process;
```

Concurrent or sequential?

- Hardware is concurrent. What's the point of writing a sequential code?

```
…
a <= '0';
…
a <= '1';
…
```

a=? a=X

```
process(all)
begin
    …
    a <= '0';
    …
    a <= '1';
end process;
```

a=? a='1'

Signal update inside processes

- The time is "**paused**" when a process is executed. The signals that are written receive the new value after finishing the process or after a *wait*.

```
process(all)
begin
    …
    a <= '0';
    b <= not a;
    …
end process;
```

Signal	Before the process	After the process
a	'1'	'0'
b	?	'0'

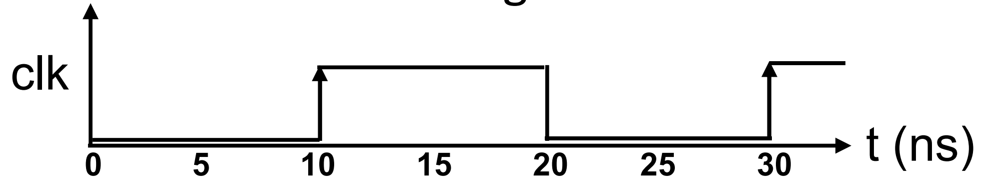
Signal update inside processes

- The time is "**paused**" when a process is executed. The signals that are written receive the new value after finishing the process or after a *wait*.

```
Process                              -- without sensitivity list
begin
     while simulating = '1' loop
        clk <= '0';              ← '?'
        wait for 10 ns;                      ← '0'
        clk <= '1';                                      ← '0'
        wait for 10 ns;
     end loop;                                                      ← '1'
     wait ;   -- it prevents the continuos execution of the
              -- while structure when simulating = '0'
end process;
```

Signal	Before the process	Before 1st wait	After 1st wait	Before 2nd wait	After 2nd wait
clk	?	?	'0'	'0'	'1'

Time "**does continue**" during the execution of a *wait*



Attention: The *wait sentence cannot be used for real hardware*.
It can only be simulated.

Conditional assignments and loops: Summary

- Inside and uutside processes
  - ✓ When – else …..else – ;
  - ✓ With – select ……when others ;
- Inside processes
  - ✓ If ** then – ; elsif – ; else – ; end if ;
  - ✓ Case ** is when – ; ….when others – ; end case;
  - ✓ For ** in ** downto/to ** loop – ; end loop ;
  - ✓ While ** loop – ; end loop;

1.16

Combinational logic: Inside or outside the process?

- Except in the use of loops, any combinational code written in a process can be written outside the process (concurrent sentences) and vice versa.

- If a process is used, longer code required but more flexibility (using if / case, overwritting signals, etc).

Outline

- Introduction
- Combinational logic
- Combinational circuits
- **Sequential logic**
- Structural modeling
- *Testbenches*

Sequential logic

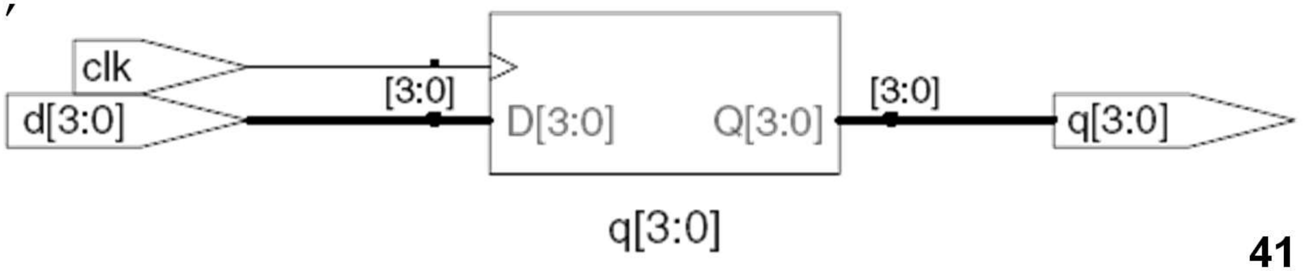
- In VHDL, flip-flops (or registered signals) are described using always this template:

  - **Using a `process`**

- Oher descriptions may reach equivalent simulations, but they do not synthesize the same hardware

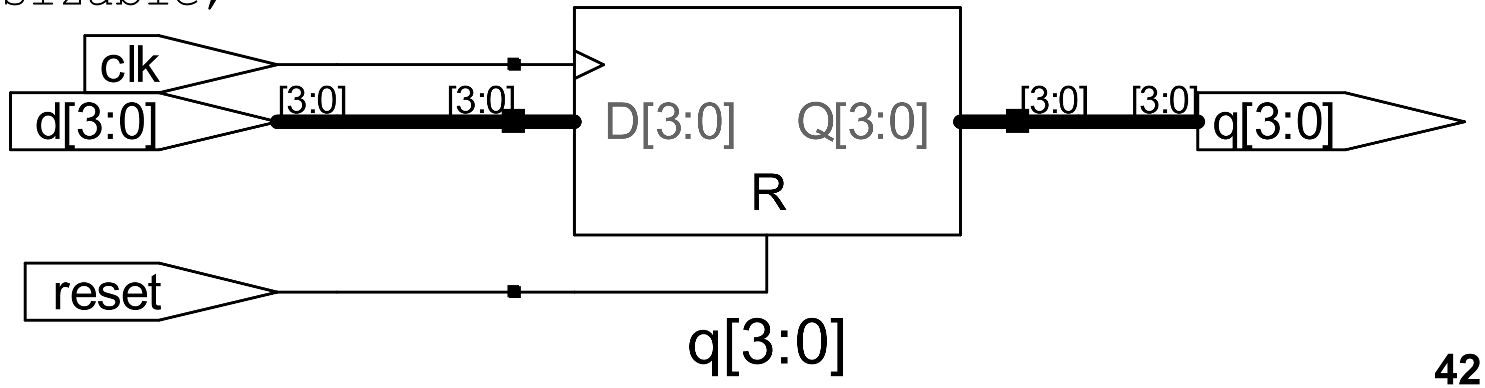
D-type Flip-Flop

```
entity flop is
port(Clk : in std_logic;
     D : in std_logic_vector(3 downto 0);
     Q : out std_logic_vector(3 downto 0));
end flop;

architecture synthesizable of flop is
begin

REG: process(all)
  begin
    if Clk = ‘1’ and Clk’event then  --if rising_edge(Clk) then
      Q <= D;
    end if;
  end process;

end synthesizable;
```



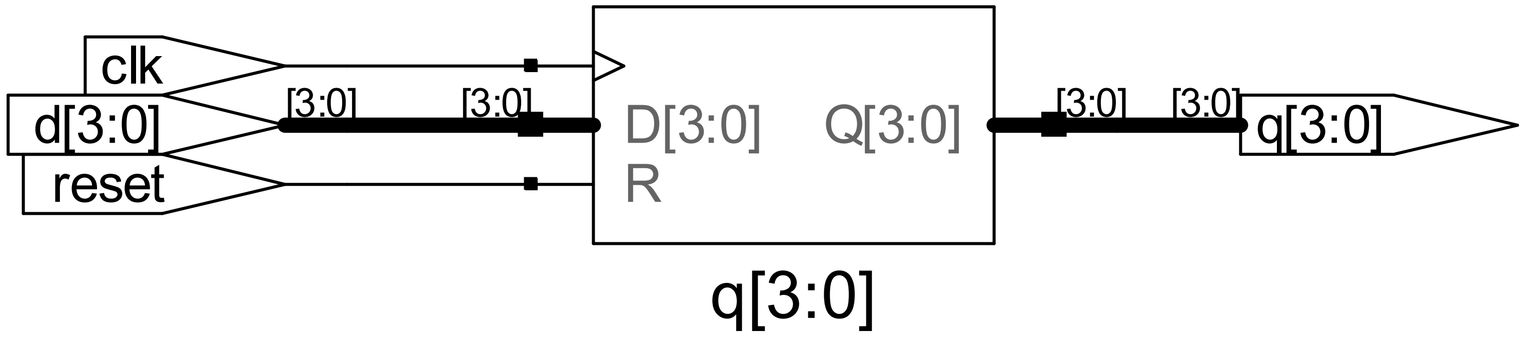
D-type Flip-Flop with asynchronous Reset

```
-- Reset is added in the entity

architecture synthesizable of flop is
begin

  process(all)
    begin
      if Reset = '1' then
        Q <= (others => '0');
      elsif Clk = '1' and Clk'event then
        Q <= D;
      end if;
  end process;

end synthesizable;
```



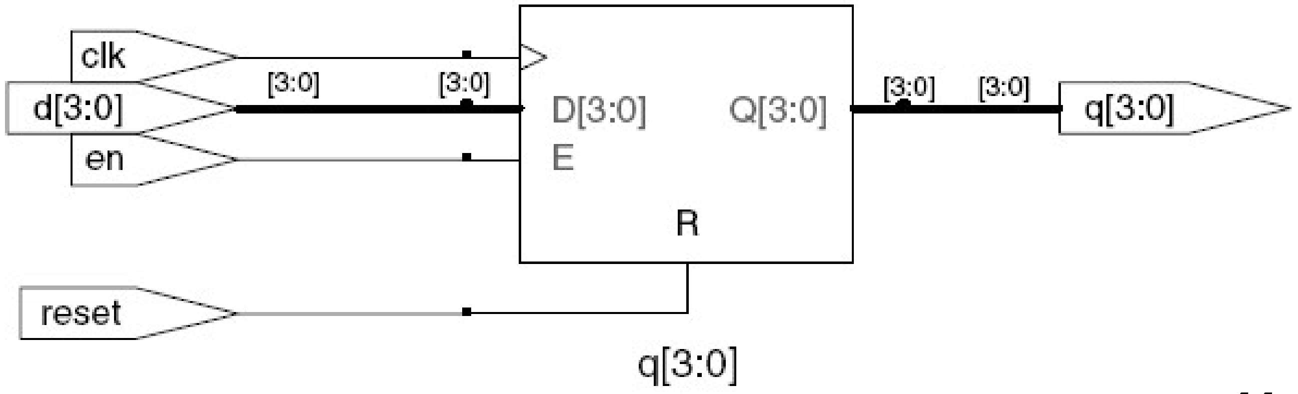
D-type Flip-Flop with synchronous Reset

```
architecture synthesizable of flop is
begin
  process(all)
  begin
    if Clk = '1' and Clk'event then
      if Reset = '1' then
        Q <= (others => '0');
      else
        Q <= D;
      end if;
    end if;
  end process;
end synthesizable;
```



D-type Flip-Flop with Enable

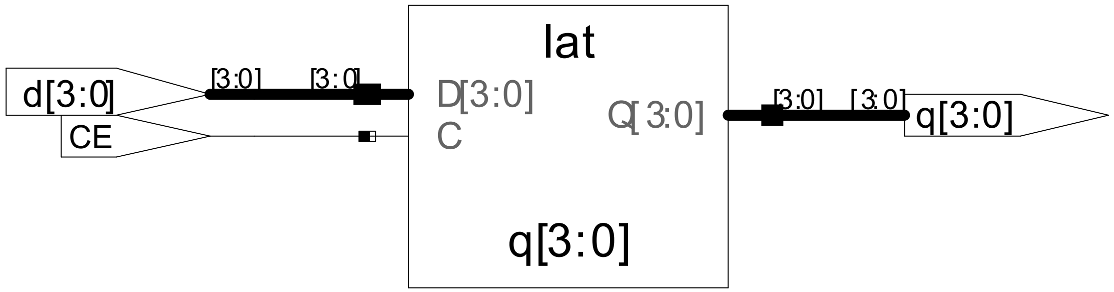
```
-- En is added in the entity

architecture synthesizable of flop is
begin
  process(all)
  begin
    if Reset = '1' then
      Q <= (others => '0');
    elsif Clk = '1' and Clk'event then
      if En = '1' then
        Q <= D;
      end if;
    end if;
  end process;
end synthesizable;
```



1.11, 1.14

Latch

```
architecture synthesizable of latch is
begin
  process(all)
  begin
    if CE = ‘1’ then -- without edge, only by level
      Q <= D;
    end if;
  end process;
end synthesizable;
```



Unwanted latches

```
process(all)
begin
       if a = "00" then
              b <= "11";
       elsif a = "01" then
              b <= "10";
       elsif a = "10" then
              b <= "00";
       end if;
end process;
```

Which is the value of *b* when a="11"?

b keeps the previous value => Latch

Attention: unwanted latches are synthesized when the destination value does no receive values in every path. In this course the latches are not used, but a code can generate them unintentionally. If the synthesized hardware include latches, it will be considered as an error, especially if the code should be combinational.

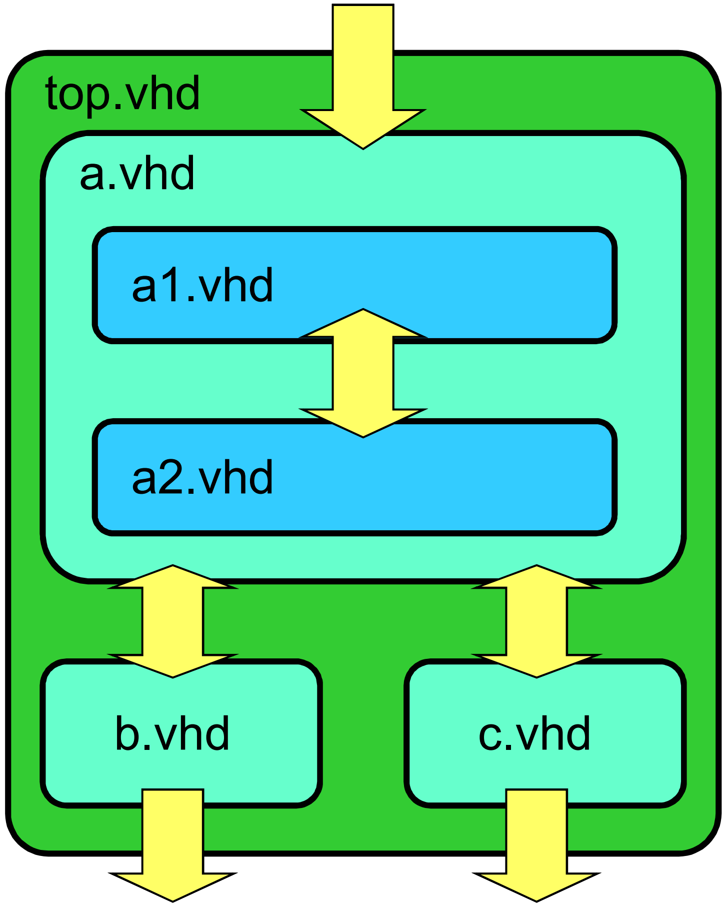
1.18

Outline

- Introduction
- Combinational logic
- Combinational circuits
- Sequential logic
- **Structural modeling**
- *Testbenches*

Structural model

- The basic components are used as elements of other bigger elements.
- They are essential for code reuse.
- It allows the designer to mix components design with different methods:
  - Schematics
  - VHDL, Verilog
- It generates more legible and portable designs.
- It is necessary for *top-bottom* o *bottom-up designs.*



How to instantiate a component

```
entity top is
port
( ... );
end top;
```

```
architecture hierarchical of top is
  signal s1, s2 : std_logic;
```

```
component a
port
(entrada: in std_logic;
  salida: out std_logic);
end component;
```

```
begin
```

```
u1: a
port map
(entrada => s1,
 salida => s2);
```

```
end hierarchical;
```

Example of hierarchical design: Internal component

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY myand2 IS PORT (
  x, y: IN std_logic;
  z: OUT std_logic);
END myand2;

ARCHITECTURE archand2 OF myand2 IS
BEGIN
    z <= x AND y;
END archand2;
```

Internal component description:
"2-input AND gate"

Example of hierarchical design: top-level component

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY myand4 IS PORT (
    a, b, c, d: IN std_logic;
         z: OUT std_logic);
END myand4;

ARCHITECTURE archmyand4 OF myand4 IS

COMPONENT myand2 PORT (
  x, y: IN std_logic;
  z: OUT std_logic);
END COMPONENT;

SIGNAL s1, s2: std_logic;

BEGIN
 a1: myand2 PORT MAP (x=>a, y=>b, z=>s1);
 a2: myand2 PORT MAP (z=>s2, y=>c, x=>d); -- The order can be varied
 a3: myand2 PORT MAP (x=>s1, y=>s2, z=>z);
-- Apart from the instantiations, more code can be added in the architecture
END archmyand4;
```

Top-level component:
"4-input AND gate"

Component declaration

Component instantiation. Associated by name

Outline

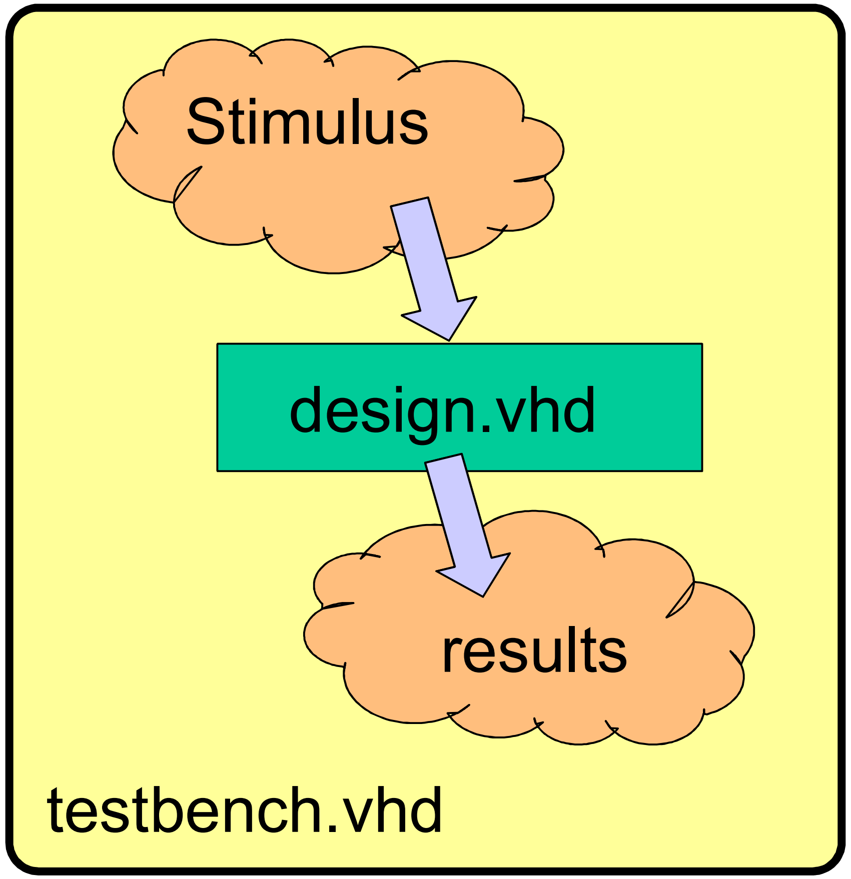
- Introduction
- Combinational logic
- Combinational circuits
- Sequential logic
- Structural modeling
- ***Testbenches***

Testbenches

- HDL code written to check if an HDL component works properly: the *device under test* (dut), or *unit under test* (uut)
- No synthesizable
- *Testbenches* types:
  - Simple
  - Self-checking

How to create a *testbench*

1. Instantiate the design that we are going to verify
   - The *testbench* will be the new top-level component.
   - The entity won't have any ports
2. Write the code to:
   - Generate the stimulus
   - Check the results
   - Report the results to the user



Example

We are going to verify that this component works properly:

$$y = a \cdot b$$

VHDL

```
entity MyAnd is
  port (a, b : in std_logic;
        y : out std_logic);
end MyAnd;

architecture behavioral of MyAnd is
begin
  y <= a and b;
end behavioral;
```

Testbench (part 1, instantiation)

```
entity testbench1 is -- There are neither inputs
end;                 -- nor outputs (ports)

architecture test of testbench1 is
   component MyAnd          -- UUT declaration
    port (a, b: in std_logic;
         y: out std_logic);
    end component;
   -- Signals to connect the ports of the uut component
   signal a, b, y: std_logic;    -- Their names can be different
begin
   uut: MyAnd port map(
        a => a,
        b => b,
        y => y);
```

Testbench (part 2, gen. stimulus)

```
-- the stimulus are generated inside a process
process        -- There is no sensitivity list but there
begin          -- are wait sentences
   a <= ‘0’; b <= ‘0’;
   wait for 10 ns;
   a <= ‘0’; b <= ‘1’;
   wait for 10 ns;
   a <= ‘1’; b <= ‘0’;
   wait for 10 ns;
   a <= ‘1’; b <= ‘1’;
   wait for 10 ns;
   wait;        -- It “kills” this process. The process would
               -- be restarted otherwise
end process;
end; -- end of the architecture
```

The simulation analyzes the value of ‘y’ signal in every case and check if it works properly

Self-checking

An *assert* sentence can be used to check the results

```
assert condition report “Message” severity level;
```

It verifies that `condition` is met. If not, it prints `Message` in the simulator Console and it generates an exception with the specified `level`.

Depending on the `level` (note, warning, error, failure), the simulator stops or not (it can be configured by the user).

MyAnd self-checking example

```
process -- There is no sensitivity list but there
begin   -- are wait sentences
   a <= '0'; b <= '0';
   wait for 10 ns;
   assert y = '0' report "Fails when 00" severity error;
   a <= '0'; b <= '1';
   wait for 10 ns;
   assert y = '0' report "Fails when 01" severity error;
   a <= '1'; b <= '0';
   wait for 10 ns;
   assert y = '0' report "Fails when 10" severity error;
   a <= '1'; b <= '1';
   wait for 10 ns;
   assert y = '1' report "Fails when 11" severity error;
   wait; -- It "kills" this process. The process would
        -- be restarted otherwise
end process;
```

1.5

Testbenches for sequential circuits

The clock is usually generated in a separate process:

```
process
begin
   Clk <= ‘0’;
   wait for CYCLE/2; -- CYCLE is a constant
   Clk <= ‘1’;
   wait for CYCLE/2;
end process; -- There is no wait, so the process
             -- will be restarted automatically
```

It is better to generate the clock only while one auxiliary signal is on, so the clock generation can stop when necessary. See the example of `while` loop.

Example of a counter

```
process -- There is no sensitivity list but there
begin   -- are wait sentences
  Reset <= '1'; -- the reset is always activated at the
                -- beginning
  wait for CYCLE;
  Reset <= '0';     -- It is deasserted so the counter starts
                    -- to work
  for i in 0 to 255 loop
   assert to_integer(Count) = i
     report "Fails at " & to_string(i)
       severity error; -- Count is of type unsigned
   wait for CYCLE;          -- The clock is generated in parallel
                            -- with another process, see the
                            -- previous slide
  end loop;                 -- It finishes the loop
  wait;                     -- It "kills" this process
end process;
```

Sensitivity lists: *testbenches*

- The processes in *testbenches* can have no sensitivity lists
- In that case, it is necessary to write some *wait* sentence/s
- If there were no wait sentences, the simulator will get stuck, because the *time* wouldn't be able to advance (infinite loop)

1.17

Constants

- Like with any other language, you can use *constants* in VHDL.
- They are also declared in the declarative part, between *architecture* and *begin*, and the value must be provided in the declaration:

```
architecture example of test is
    constant C1 : std_logic_vector(3 downto 0) := “0101”;
    constant C2 : integer := 5;
    constant CYCLE : time := 10 ns;
begin
```

- There is no restriction about the type of constant (std_logic, std_logic_vector, integer, time, etc).

Type conversion

- VHDL is *strongly typed* and there is no automatic cast:

```
architecture example of test is
    signal s1 : std_logic_vector(3 downto 0);
    signal s2 : integer;
    signal s3 : signed(3 downto 0);
    signal s2 : unsigned(3 downto 0);
begin
    s1 <= s2; -- Wrong in VHDL
    s2 <= s1; -- Also wrong
    s3 <= s1; -- Also wrong
    s4 <= s1; -- Also wrong
    s2 <= s3; -- Also wrong
    s2 <= s4; -- Also wrong
```

Type conversions

- Different functions are used for conversion in each case:

		TO			
		integer	signed	unsigned	std_logic_vector
FROM	integer		to_signed()	to_unsigned()	No direct conversion
	signed	to_integer()		unsigned()	std_logic_vector()
	unsigned	to_integer()	signed()		std_logic_vector()
	std_logic_vector	No direct conversion	signed()	unsigned()	

Computer Structure

Unit 1: Digital design and VHDL

Escuela Politécnica Superior - UAM

Copyright © 2007 Elsevier, “Digital Design and Computer Architecture”

Exercises U1

1.8.- Write the VHDL code of a multiplexer whose output is z, whose data inputs are a0, a1, a2 y a3, all of the type std_logic_vector(31 downto 0). The control input is sel, of the type std_logic_vector(1 downto 0). It is not necessary to include the entity or the architecture, only the behavioral part of the code (including process if necessary).

T

1.3.- “fuente” is std_logic_vector(3 downto 0) and “dest” is std_logic_vector(6 downto 0). Write the code so that “dest” is equal to “fuente” multiplied by 8, but not using the any multiplication or addition.

1.9.- “dato4” is std_logic_vector(3 downto 0). Write the VHDL code for getting that “salida8”, which is an std_logic_vector(7 downto 0), is equal to dato4 but sign extended.

T

Exercises U1

1.2.- Write the equivalent code to the shown process, but using a concurrent assignment of the type when – else.

```
process(all)
begin
  if a < b then
    s <= b;
  elsif a < c then
    s <= c;
  else
    s <= a;
  end if;
end process;
```

1.10.- Write the equivalent code to the one given, but using a single case sentence. Add a process and its sensitivity list if necessary.

```
with s select
     z <= a when "00",
          b when "01",
          c when others;
```

T

1.16. Design a 4 to 1 multiplexer with 8 bits inputs using the following four sentences:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
entity Mux4a1 is
        port (
                    D0, D1, D2, D3 : in unsigned(7 downto 0);
                    Sel: in std_logic_vector(1 downto 0);
                    Y: out unsigned(7 downto 0)
         );
end Mux4a1;
```

```
architecture ArchWhen of Mux4a1 is       -- a) Architecture based on sentence WHEN ELSE

begin
        Y <=     D0 when Sel = "00" else
                 D1 when Sel = "01" else
                 D2 when Sel = "10" else
                 D3;                              -- Sel="11"
```

```
architecture ArchIf of Mux4a1 is         -- b)  Architecture based on sentence IF

begin
        process (all)
        begin
                if Sel = "00" then  Y <= D0;
                elsif Sel = "01" then  Y <= D1;
                elsif Sel = "10" then  Y <= D2;
                else   Y <= D3;                -- Used instead of Sel = "11" to avoid latches
                end if;
        end process;
end ArchIf;
```

1.16. Design a 4 to 1 multiplexer with 8 bits inputs using the following four sentences:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
entity Mux4a1 is
        port (
                D0, D1, D2, D3 : in unsigned(7 downto 0);
                Sel: in std_logic_vector(1 downto 0);
                Y: out unsigned(7 downto 0)
         );
end Mux4a1;
```

```
architecture ArchWith of Mux4a1 is        -- c) Architecture based on sentence WITH-SELECT

begin
        with Sel select
                Y <=        D0 when "00",
                            D1 when "01",
                            D2 when "10",
                            D3 when others;          -- "11"
end ArchWith;
```

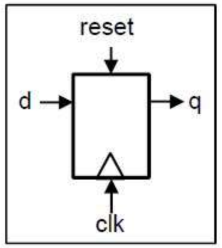
```
architecture ArchCase of Mux4a1 is        -- d) Architecture based on sentence CASE

begin
                process (all)
                begin
                        case Sel is
                                when "00" => Y <= D0;
                                when "01" => Y <= D1;
                                when "10" => Y <= D2;
                                when others => Y <= D3;        -- Closed with when others to avoid latches
                        end case;
                end process;
end ArchCase;
```

T

Exercises U1

1.11.- clk, reset, d and q are std_logic signals. Write the code so that q is a D flip-flop active at falling edge and with asynchronous reset active at high level.

1.14.- Write the VHDL code of the given circuit, which is a D flip-flop active at rising edge and with synchronous reset active at high level.



T

1.18. Write the code of an ascending 8 bits counters with asynchronous reset and parallel load.

T

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

entity Cont8Load is
   port (
         Clk , Reset, Load : in std_logic;
         Data : in unsigned(7 downto 0);
         Q : out unsigned(7 downto 0)
         );
end Cont8Load;
```

```
architecture Practica of Cont8Load is

 begin
         PrCont: process(all)
         begin
            if Reset = '1' then
                     Q <= (others => '0');
            elsif rising_edge(Clk) then
                     if Load = '1' then
                           Q <= Data;
                     else
                            Q <= Q + 1; -- normal behavior
                     end if;
            end if;
         end process;

end Practica;
```

1.19. Write the code of an ascending 8 bits counters with asynchronous reset and parallel load whose data is chosen among 4 possibilities. Use the previous designed modules (multiplexer and counter).

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

entity Cont8Mux is
   port (
                  Clk, Reset, Load : in std_logic;
                  Sel : in std_logic_vector(1 downto 0);
                  D0, D1, D2, D3 : in unsigned(7 downto 0);
                  Q : out unsigned(7 downto 0)
          );
end Cont8Mux;
```

```
architecture Practica of Cont8Mux is

              component Mux4a1                          -- mux declaration
              port (
                     Sel: in std_logic_vector(1 downto 0);
                     D0, D1, D2, D3 : in unsigned(7 downto 0);
                     Y : out unsigned(7 downto 0)
                     );
              end component;

              component Cont8Load                       -- counter declaration
              port (
                     Clk, Reset, Load : in std_logic;
                     Data : in unsigned(7 downto 0);
                     Q : out unsigned(7 downto 0)
                     );
              end component;

              signal Data : unsigned(7 downto 0);  -- It is only necessary to declare as internal signals those that go
                                                   -- from one entity to the other, but are not ports (inputs or outputs)
```

1.19. Write the code of an ascending 8 bits counters with asynchronous reset and parallel load whose data is chosen among 4 possibilities. Use the previous designed modules (multiplexer and counter).

```
begin                         -- Instantiation and port mapping (by name)

            ElMux: Mux4a1 port map(            -- Instantiation of Mux4a1
                        D0 => D0,
                        D1 => D1,
                        D2 => D2,
                        D3 => D3,
                        Sel => Sel,
                        Y => Data
            );

            ElContador: Cont8Load port map(     -- Instantiation of Cont8Load
                        Clk => Clk,
                        Reset => Reset,
                        Load => Load,
                        Data => Data,
                        Q => Q
            );

end Practica;
```

T

1.5.- Write an assert in order to test that the signal “s” is at ‘1’. If not, the simulation should output the text “Signal s is not at 1” and abort the simulation.

T

1.22.- Draw the schematic of the following VHDL code. All the connections must be labeled in the schematic. A is std_logic_vector(7 downto 0). All the components must be correctly identified with their standard symbol or an explanation of the functionality in text.

```
process(all)
begin
   if rising_edge(Clk) then
      if CE = ‘1’ then
         if X = ‘1’ then
            Y <= resize(A,16);
         else
            Y <= A & “00000000”;
         end if;
      end if;
   end if;
end process;
```

1.17.- Design a testbench for the 4 to 1 multiplexer of the exercise 1.16.

1.17. Design a testbench for the 4 to 1 multiplexer of the exercise 1.18:

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

entity Mux4a1Tb is
end Mux4a1Tb;
```

```
entity Mux4a1 is
        port ( D0, D1, D2, D3 : in unsigned(7 downto 0);
                  Sel: in std_logic_vector(1 downto 0);
                  Y: out unsigned(7 downto 0) );
end Mux4a1;
```

```
architecture Test of Mux4a1Tb  is



   component Mux4a1
          port  ( D0, D1, D2, D3 : in unsigned(7 downto 0);
                    Sel: in std_logic_vector(1 downto 0);
                    Y: out unsigned(7 downto 0) );          -- Declaring the entity to be tested
   end component;
-- One signal for each port of the previous component. The names could be the
-- same as in the Mux4a1 entity, but different names can also be used.
                signal sD0, sD1, sD2, sD3, sY : unsigned(7 downto 0);
                signal sSel : std_logic_vector(1 downto 0);

                constant CICLO : time := 10 ns;            -- Time constant

begin
                uut: Mux4a1 port map (               -- Instantiation of the tested circuit (Unit Under Test)
                               D0 => sD0,
                               D1 => sD1,
                               D2 => sD2,
                               D3 => sD3,
                               Sel => sSel,
                               Y => sY
                );
```

1.17. Design a testbench for the 4 to 1 multiplexer of the exercise 1.18:

T

```
                                    -- Continuation of the architecture
      ProcPrinc: process            -- Generate values for the inputs and check the outputs
      begin
          sD0 <= (others => '0');
          sD1 <= "00000001";
          sD2 <= X"02";
          sD3 <= (1 downto 0 => '1', others => '0');

          sSel <= "00";
          wait for CICLO;
          assert sY = sD0 report “Error in the 00 selection” severity failure;

                                    -- A signal that is not updated keeps its previous value.
          sSel <= "01";
          wait for CICLO;
          assert sY = sD1 report “Error in the 01 selection" severity failure;

          sSel <= "10";
          wait for CICLO;
          assert sY = sD2 report " Error in the 10 selection " severity failure;

          sSel <= "11";
          wait for CICLO;
          assert sY = sD3 report " Error in the 11 selection " severity failure;

          report "Always enters as false is not true”  severity note;
          wait;        -- Kills the process. As this is the only process, the simulation ends here
      end process;

end Test;
```