FPGA Design

Part III - Combinatorial VHDL

Thomas Lenzi

Vrije Universiteit Brussel

UNIVERSITÉ LIBRE DE BRUXELLES

ULB

Objective

- We will introduce you to VHDL and to combinatorial statements of VHDL using some examples and exercises.

VHDL Basics

Few Notes on VHDL

- VHDL is NOT case sensitive.

- VHDL is extremely talkative…

VHDL Entities

- In VHDL, every file you create is an entity.
- Entities can be seen as an electronic components which takes inputs, runs some logic on them, and set outputs.
- Creating a VHDL entity is similar to declaring an ICs. Once you have declared and defined it, you can use it wherever you want in your design.

Default VHDL File

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

--use IEEE.NUMERIC_STD.ALL;

--library UNISIM;
--use UNISIM.VComponents.all;

entity basic_module is
end basic_module;

architecture Behavioral of basic_module is

begin


end Behavioral;
```

- This is the default VHDL file that ISE creates for you (I removed some of the comments).
- Three sections are visible:
  - libraries
  - entity is
  - architecture of

Libraries

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

--use IEEE.NUMERIC_STD.ALL;

--library UNISIM;
--use UNISIM.VComponents.all;

entity basic_module is
end basic_module;

architecture Behavioral of basic_module is

begin


end Behavioral;
```

- Like in C, VHDL has libraries that you can include.
- The *library <name>* statement imports a library.
- The *use <name>* statement removes the namespace before the objects it imports.
- Through exercises we will see what each library is used for.

Entity is

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

--use IEEE.NUMERIC_STD.ALL;

--library UNISIM;
--use UNISIM.VComponents.all;

entity basic_module is
end basic_module;

architecture Behavioral of basic_module is

begin


end Behavioral;
```

- The *entity <name> is* statement declares an entity. It is its prototype.
- We will later on add statements in this region to tell the system what the inputs and what the outputs are.

Architecture of

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

--use IEEE.NUMERIC_STD.ALL;

--library UNISIM;
--use UNISIM.VComponents.all;

entity basic_module is
end basic_module;

architecture Behavioral of basic_module is

begin


end Behavioral;
```

- The *architecture of <name>* statement is where the logic that composes the entity sits.
- It is the body of the entity where the logic is described.

Adding Inputs and Outputs

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

--use IEEE.NUMERIC_STD.ALL;

--library UNISIM;
--use UNISIM.VComponents.all;

entity basic_module is
port(
    first_input      : in std_logic;
    second_input     : in std_logic;
    output           : out std_logic
);
end basic_module;

architecture Behavioral of basic_module is

begin


end Behavioral;
```

Name Direction Type

- For a module to do something interesting, you need to add inputs and outputs.
- The IOs are defined in a *port* statement of the *entity is*.
- In this exemple, you are telling the design that your entity has 3 ports:
  - 2 input wires (first_input, second_input)
  - 1 output wire (output)

VHDL Types

- When using types in VHDL, you have to imagine wires and not bits in memory.
- VHDL comes with two main types:
  - **std_logic**, which is a single wire/bit
  - **std_logic_vector**, which is a vector of wires/bits
- and with some helping types:
  - **integer**
  - **boolean**

std_logic

signal <name> : std_logic [:= <default>];

- An *std_logic* takes the following value: ‘0’, ‘1’, or ‘Z’ (single quotes!).

- ‘Z’ puts the signal in high-impedance mode so that other modules can drive it.

- In general, one signal is driven by one module.

std_logic_vector

signal <name> : std_logic_vector(<range>) [:= <default>];

- The *range* of an *std_logic_vector* defines the number of bits and the direction of the bits:
    - 7 downto 0 : means that the MSB of the 8 bits is placed on the left side;
    - 0 to 3 : means that the LSB of the 4 bits is placed on the left side.
- Values for vectors are given using double quotes:
    - vector_signal <= “00110011”;
    - vector_signal <= x”AB”; — in hexadecimal
- Or using *aggregates* that describe each bit:
    - vector_signal <= (0 => ‘1’, 3 => ‘0’, others => ‘0’);

integer

signal <name> : integer [range <from> to <to>] [:= <default>];

- An integer signal can be given a range of operation using the “range” statement.

- Values given to *integer* are… integer (no quotes).

boolean

signal <name> : boolean [:= <default>];

- Booleans take *true* or *false* as value.

Adding Logic

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

--use IEEE.NUMERIC_STD.ALL;

--library UNISIM;
--use UNISIM.VComponents.all;

entity basic_module is
port(
    first_input      : in std_logic;
    second_input     : in std_logic;
    output           : out std_logic
);
end basic_module;

architecture Behavioral of basic_module is

begin

    output <= first_input and second_input;

end Behavioral;
```

- Now that we have IOs, we can use them in the *architecture of* to run some logic.
- In this exemple, we set the output to the logic AND of the two inputs.

Signals and Assignation

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

--use IEEE.NUMERIC_STD.ALL;

--library UNISIM;
--use UNISIM.VComponents.all;

entity basic_module is
port(
    first_input     : in std_logic;
    second_input    : in std_logic;
    output          : out std_logic
);
end basic_module;
```

```
architecture Behavioral of basic_module is

    signal internal : std_logic := '0';

begin

    internal <= first_input and second_input;

    output <= internal;

end Behavioral;
```

- Apart from IOs, it is also possible to create local signals inside an entity.
- The declaration of local signals is done before the *begin* of the *architecture*.
- Signals are assigned using the “<=“ operator.

Back to IOs

- Input signals can only be found at the right-hand side of the assignment operator.

- Output signals can only be found at the left-hand side of the assignment operator.

- If you want to use the value of the output signal, you should create a temporary local signal to hold that value.

Top Level

- As previously stated, the Top Level is an entity like every other, except that its IOs are directly connected to the pins of the FPGA.

- To set a module as Top Level, right click the file and select “Set as Top Module…”.

- For a project to compile, we need to assign each signal of the Top Level with a pin. To do so, we will need to populate the UCF file.

UCF Pin Constraints

```
# Clock

NET "clk_50MHz_i"    LOC = B8;

# LEDS

NET "leds_o<7>"      LOC = G1;
NET "leds_o<6>"      LOC = P4;
NET "leds_o<5>"      LOC = N4;
NET "leds_o<4>"      LOC = N5;
NET "leds_o<3>"      LOC = P6;
NET "leds_o<2>"      LOC = P7;
NET "leds_o<1>"      LOC = M11;
NET "leds_o<0>"      LOC = M5;

# Switches

NET "sw_i<7>"        LOC = N3;
NET "sw_i<6>"        LOC = E2;
NET "sw_i<5>"        LOC = F3;
NET "sw_i<4>"        LOC = G3;
NET "sw_i<3>"        LOC = B4;
NET "sw_i<2>"        LOC = K3;
NET "sw_i<1>"        LOC = L3;
NET "sw_i<0>"        LOC = P11;

# Push button

NET "btn_i<3>"       LOC = A7;
NET "btn_i<2>"       LOC = M4;
NET "btn_i<1>"       LOC = C11;
NET "btn_i<0>"       LOC = G12;
```

- Each signal is mapped to a pin. The pin number can be found in the datasheet or on the development board next to each component.

Exercise

- Write a VHDL top level entity that takes an input signal and forwards it on an output bus of two elements.

- Implement this on the FPGA using a push button as source and two LEDs as output.

Combinatorial Logic

Logic Operators

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

--use IEEE.NUMERIC_STD.ALL;

--library UNISIM;
--use UNISIM.VComponents.all;

entity basic_module is
port(
    first_input     : in std_logic_vector(7 downto 0);
    second_input    : in std_logic_vector(7 downto 0);
    output          : out std_logic_vector(7 downto 0)
);
end basic_module;

architecture Behavioral of basic_module is

begin

    output(0) <= first_input(0) and second_input(0);

    output(1) <= first_input(1) nand second_input(1);

    output(2) <= first_input(2) or second_input(2);

    output(3) <= first_input(3) nor second_input(3);

    output(4) <= first_input(4) xor second_input(4);

    output(5) <= not (first_input(5) xor second_input(5));

    output(7 downto 6) <= first_input(7 downto 6);

end Behavioral;
```

- All logic operators can be used except for *nxor*.
- To select an element in a *std_logic_vector*, use parentheses:
  - with a range to make another *std_logic_vector*;
  - with an index to make a *std_logic*.

With

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

--use IEEE.NUMERIC_STD.ALL;

--library UNISIM;
--use UNISIM.VComponents.all;

entity basic_module is
port(
    first_input     : in std_logic_vector(3 downto 0);
    second_input    : in std_logic_vector(1 downto 0);
    output          : out std_logic
);
end basic_module;

architecture Behavioral of basic_module is

begin

    with second_input select output <=
        first_input(0) when "00",
        first_input(1) when "01",
        first_input(2) when "10",
        first_input(3) when "11",
        first_input(0) when others;

end Behavioral;
```

- The *with* statement assigns a signal according to the value of another signal.
- It can be seen as a multiplexer.

When

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

--use IEEE.NUMERIC_STD.ALL;

--library UNISIM;
--use UNISIM.VComponents.all;

entity basic_module is
port(
    first_input     : in std_logic_vector(3 downto 0);
    second_input    : in std_logic_vector(1 downto 0);
    output          : out std_logic
);
end basic_module;

architecture Behavioral of basic_module is

begin

    output <= first_input(0) when second_input = "00" else
              first_input(1) when second_input = "01" else
              first_input(2) when second_input = "10" else
              first_input(3) when second_input = "11" else
              first_input(0);

end Behavioral;
```

- With the *when* statement the assignment of a signal depends upon a condition and not only the value of a signal.

Conditions

- Conditions are combinations of one or more tests.
- Tests on signals use the following operators: = (equal), /= (not equal), <, >, <=, >=.
- Conditions are glued together using logic operators (and, or, not, …).

Example: Buttons to LEDs

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

--use IEEE.NUMERIC_STD.ALL;

--library UNISIM;
--use UNISIM.VComponents.all;

entity basic_module is
port(
    btn_i    : in std_logic_vector(3 downto 0);
    leds_o   : out std_logic_vector(7 downto 0)
);
end basic_module;

architecture Behavioral of basic_module is

    signal and_01    : std_logic := '0';

begin

    leds_o(1 downto 0) <= "10";

    leds_o(2) <= '1';

    leds_o(3) <= btn_i(3);

    leds_o(4) <= btn_i(2) or btn_i(3);

    and_01 <= btn_i(0) and btn_i(1);
    leds_o(5) <= and_01;

    leds_o(6) <= '1' when btn_i = "0000" else '0';

    with btn_i select leds_o(7) <=
        '1' when "0001" | "0010" | "0100" | "1000",
        '0' when "0000",
        '0' when others;

end Behavioral;
```

- This example summarises what we learned until now:
- Types
- Assignation
- Logic & Test operators
- With / when statements

Exercises

- Write a code that connects the status of each push button (pressed or released) to an LED.

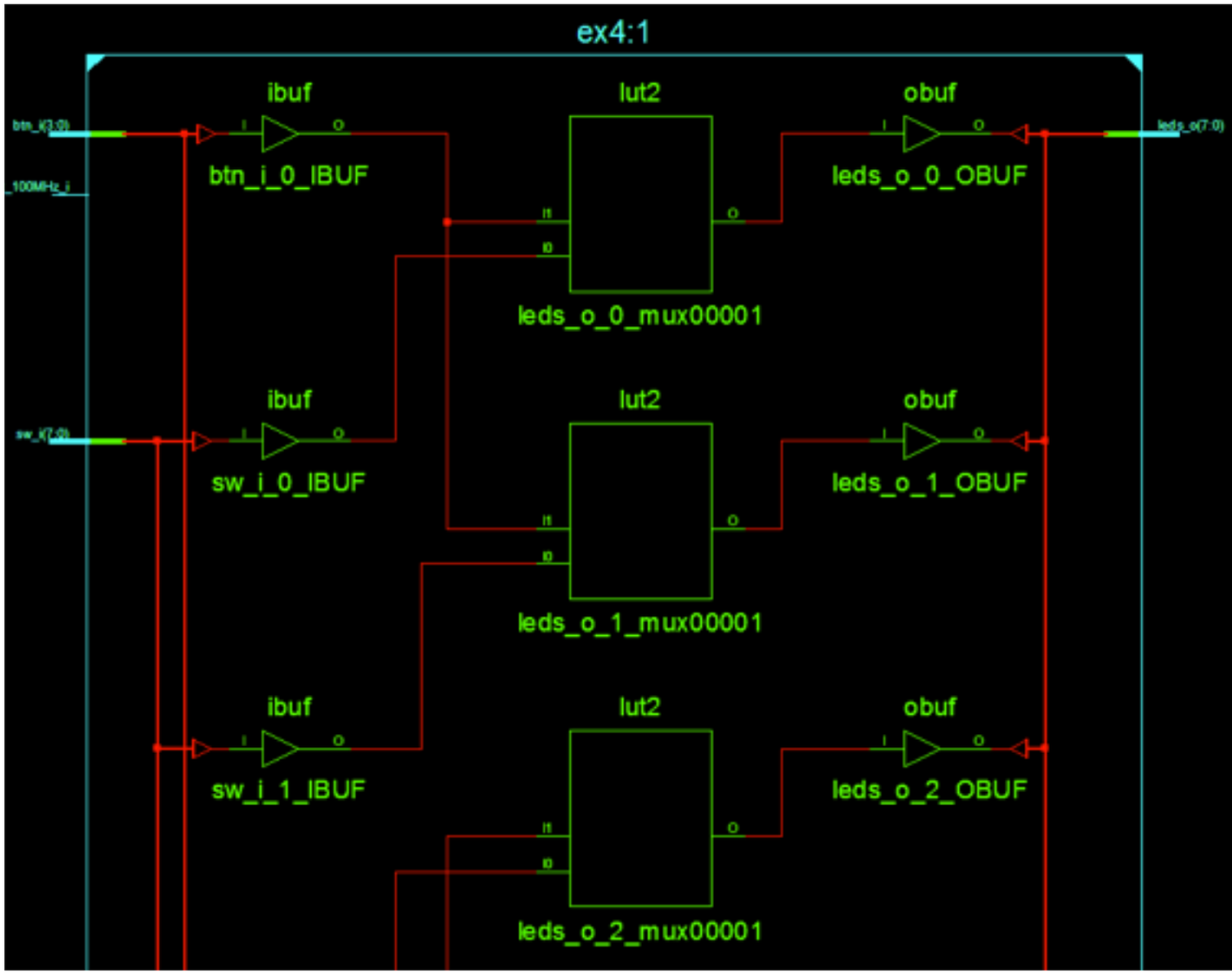
- For each LED, create the following logic: the LED is controlled by a push button, but only if a switch is ON. Otherwise it stays OFF.

- Write a script that turns on the LEDs if more than 4 switches are turned ON.

- For each exercice, write the schematic diagram using fundamental logic blocks.

Solution

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

--use IEEE.NUMERIC_STD.ALL;

--library UNISIM;
--use UNISIM.VComponents.all;

entity ex4 is
port(
    clk_50MHz_i      : in std_logic;
    sw_i             : in std_logic_vector(7 downto 0);
    btn_i            : in std_logic_vector(3 downto 0);
    leds_o           : out std_logic_vector(7 downto 0)
);
end ex4;

architecture Behavioral of ex4 is
begin

    leds_o(0) <= btn_i(0) when sw_i(0) = '1' else '0';
    leds_o(1) <= btn_i(0) when sw_i(1) = '1' else '0';
    leds_o(2) <= btn_i(1) when sw_i(2) = '1' else '0';
    leds_o(3) <= btn_i(1) when sw_i(3) = '1' else '0';
    leds_o(4) <= btn_i(2) when sw_i(4) = '1' else '0';
    leds_o(5) <= btn_i(2) when sw_i(5) = '1' else '0';
    leds_o(6) <= btn_i(3) when sw_i(6) = '1' else '0';
    leds_o(7) <= btn_i(3) when sw_i(7) = '1' else '0';

end Behavioral;
```



Mathematics

Signed & Unsigned

- The *use ieee.numeric_std.all* adds a new type to VHDL:
  - **signed** / **unsigned**, which is an std_logic_vector that acts like an integer

- signed / unsigned makes it easier to add and compare std_logic_vectors to integers.

signed / unsigned

signal <name> : unsigned(<range>) [:= <default>];

- Signed / unsigned are *std_logic_vectors* that behave like integers.

- They are assigned like *std_logic_vectors* (double quotes), but can be incremented with *integers* are compared to them.

  - unsigned_signal <= unsigned_signal + 5;

Mathematical Operators

- The available mathematical operators are: +, -, *, /, abs, mod, rem.

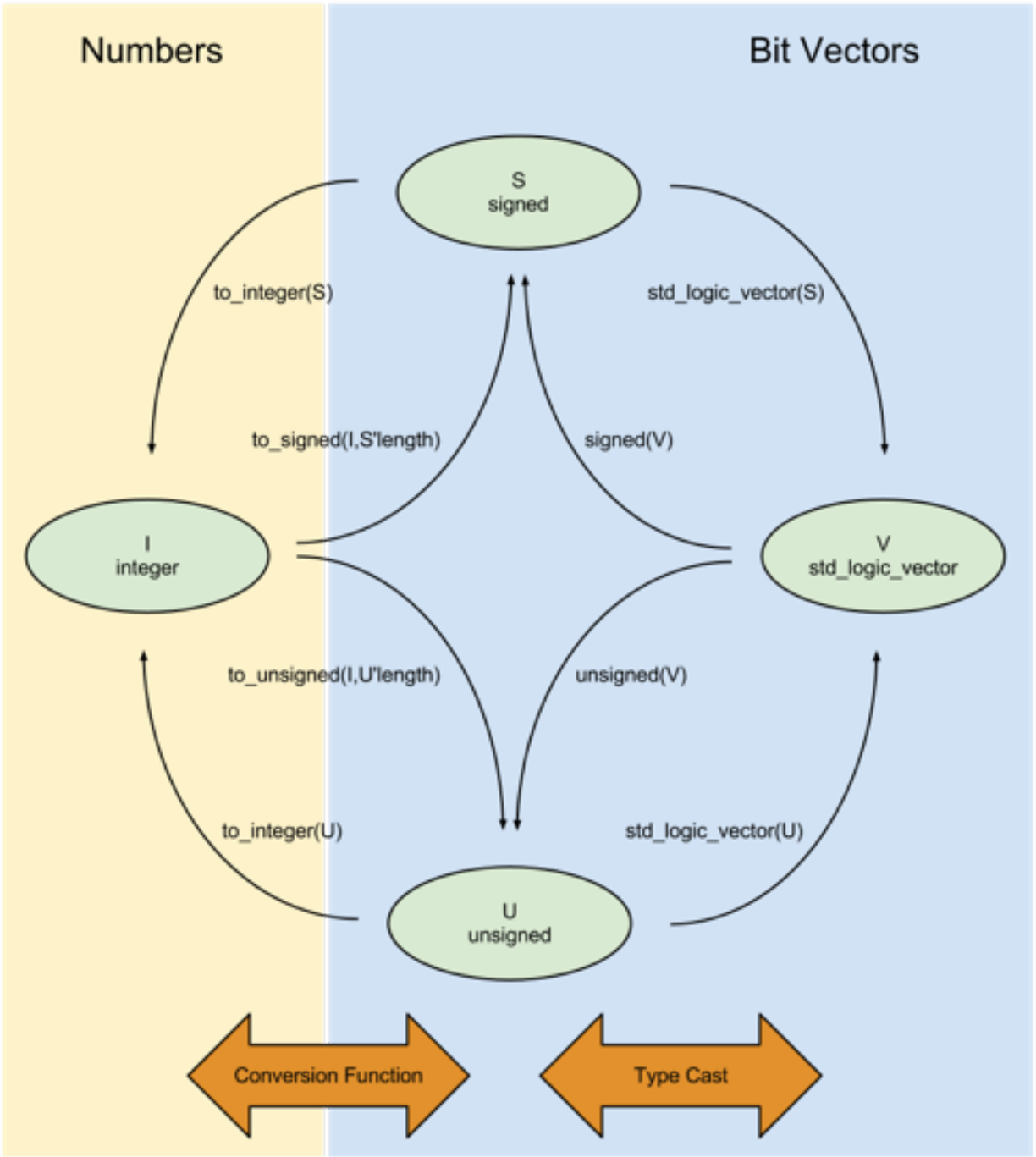
- But you should be careful using them!

- The only ones I recommend using are +/- and abs as they can be easily translated to logic. * is also acceptable.

- +/-, * can only be used with integers and signed/ unsigned.

- abs can only be used with signed/unsigned.

Type Casting



- Going from std_logic_vector to signed/unsigned is a simple cast operation (no cost).
- Going from integer to any other type is a conversion function (cost).

http://www.bitweenie.com/

Exercise

- Use the switches as input, convert it to an *unsigned*, add 5 to the value, and show the result on the LEDs.

Solution

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

use IEEE.NUMERIC_STD.ALL;

--library UNISIM;
--use UNISIM.VComponents.all;

entity ex4 is
port(
    clk_50MHz_i     : in std_logic;
    sw_i            : in std_logic_vector(7 downto 0);
    btn_i           : in std_logic_vector(3 downto 0);
    leds_o          : out std_logic_vector(7 downto 0)
);
end ex4;

architecture Behavioral of ex4 is

    signal math : unsigned(7 downto 0) := (others => '0');

begin

    math <= unsigned(sw_i) + 5;

    leds_o <= std_logic_vector(math);

end Behavioral;
```

Generics & Generates

Generics

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

--use IEEE.NUMERIC_STD.ALL;

--library UNISIM;
--use UNISIM.VComponents.all;

entity ex4 is
generic(
    DEFAULT_LED     : std_logic := '0';
    my_other_param  : integer := 255
);
port(
    clk_50MHz_i     : in std_logic;
    sw_i            : in std_logic_vector(7 downto 0);
    btn_i           : in std_logic_vector(3 downto 0);
    leds_o          : out std_logic_vector(7 downto 0)
);
end ex4;

architecture Behavioral of ex4 is
begin

    leds_o <= "00000000" & DEFAULT_LED;

end Behavioral;
```

- Like C's #define, VHDL has parameters that can be set before the generation of the design.
- They are called *generics*.
- Generics take a default value which can then be overwritten when the entity is used.

For Generate

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

--use IEEE.NUMERIC_STD.ALL;

--library UNISIM;
--use UNISIM.VComponents.all;

entity ex4 is
port(
    clk_50MHz_i     : in std_logic;
    sw_i            : in std_logic_vector(7 downto 0);
    btn_i           : in std_logic_vector(3 downto 0);
    leds_o          : out std_logic_vector(7 downto 0)
);
end ex4;

architecture Behavioral of ex4 is
begin

    loop_leds : for I in 0 to 7 generate
    begin

        leds_o(I) <= btn_i(I / 2) when sw_i(I) = '1' else '0';

    end generate;

end Behavioral;
```

- VHDL also offer the possibility to duplicate a code using a for loop. For loops in VHDL are not iterative processes like in C. They simply duplicate code that will be run in parallel.
- The example on the left is identical to the previous example with the LEDs

If Generate

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

--use IEEE.NUMERIC_STD.ALL;

--library UNISIM;
--use UNISIM.VComponents.all;

entity ex4 is
port(
    clk_50MHz_i     : in std_logic;
    sw_i            : in std_logic_vector(7 downto 0);
    btn_i           : in std_logic_vector(3 downto 0);
    leds_o          : out std_logic_vector(7 downto 0)
);
end ex4;

architecture Behavioral of ex4 is
begin

    loop_leds: for I in 0 to 7 generate
    begin

        cond_leds: if (I mod 2 = 0) generate

            leds_o(I) <= btn_i(I / 2) when sw_i(I) = '1' else '0';

        end generate;

    end generate;

end Behavioral;
```

- An *if generate* statement also exists which generates the block only if the condition is fulfilled.

Exercise

- Do the following exercise using a For Generate loop.

- For each LED, create the following logic: the LED is controlled by a push button OR by a switch (logic OR of the two).

Solution

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

--use IEEE.NUMERIC_STD.ALL;

--library UNISIM;
--use UNISIM.VComponents.all;

entity ex4 is
port(
    clk_50MHz_i : in std_logic;
    sw_i        : in std_logic_vector(7 downto 0);
    btn_i       : in std_logic_vector(3 downto 0);
    leds_o      : out std_logic_vector(7 downto 0);
);
end ex4;

architecture Behavioral of ex4 is
begin

    buttons_gen: for I in 0 to 7 generate
    begin

        leds_o(I) <= btn_i(I / 2) or sw_i(I);

    end generate;

end Behavioral;
```

Using Entities

Using Entities

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

--use IEEE.NUMERIC_STD.ALL;

--library UNISIM;
--use UNISIM.VComponents.all;

entity LogicalAND is
port(
    a_i : in std_logic;
    b_i : in std_logic;
    o_o : out std_logic

);
end LogicalAND;

architecture Behavioral of LogicalAND is
begin

    o_o <= a_i and b_i;

end Behavioral;
```

- Let's say we have an entity as shown on the left which implements a logical and of two signals.
- This entity can be used in other entities all over the design.
- To do so, we will instantiate the entity.

Instances

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

library work;

--use IEEE.NUMERIC_STD.ALL;

--library UNISIM;
--use UNISIM.VComponents.all;

entity UsingAND is
port(
    a   : in std_logic;
    b   : in std_logic;
    c   : in std_logic;
    f   : out std_logic
);
end UsingAND;

architecture Behavioral of UsingAND is

    signal tmp  : std_logic := '0';

begin

    first_instance: entity work.LogicalAND
    port map(
        a_i => a,
        b_i => b,
        o_o => tmp
    );

    second_instance: entity work.LogicalAND
    port map(
        a_i => tmp,
        b_i => c,
        o_o => f
    );

end Behavioral;
```

- When instantiating an entity, a mapping between signals in the caller and the called module should be done.
- In this example, we implement f = a AND b AND c
- Note that the instances are totally distinct from one another.

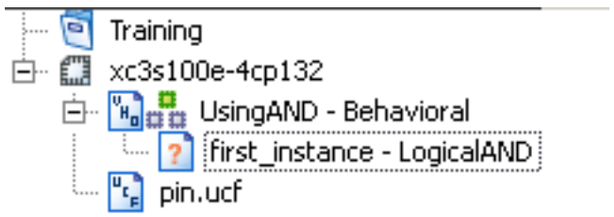
Instances and Generics

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

library work;

--use IEEE.NUMERIC_STD.ALL;

--library UNISIM;
--use UNISIM.VComponents.all;

entity UsingAND is
port(
    a   : in std_logic;
    b   : in std_logic;
    c   : in std_logic;
    f   : out std_logic
);
end UsingAND;

architecture Behavioral of UsingAND is

    signal tmp  : std_logic := '0';

begin

    first_instance: entity work.LogicalAND
    generic map(
        PARAMETER => true
    )
    port map(
        a_i => a,
        b_i => b,
        o_o => tmp
    );

end Behavioral;
```

- Generics can also be set when instantiating an entity.
- The values of the generics are only applied to one instance and not globally.

Project Structure

- When instantiating entities, the structure of your project will change.



- A tree of entities will appear.

- In this case, the LogicalAND entity doesn't exist so an interrogation mark is displayed instead of the entity itself.

VHDL to FPGA

RTL and Technology Schematics

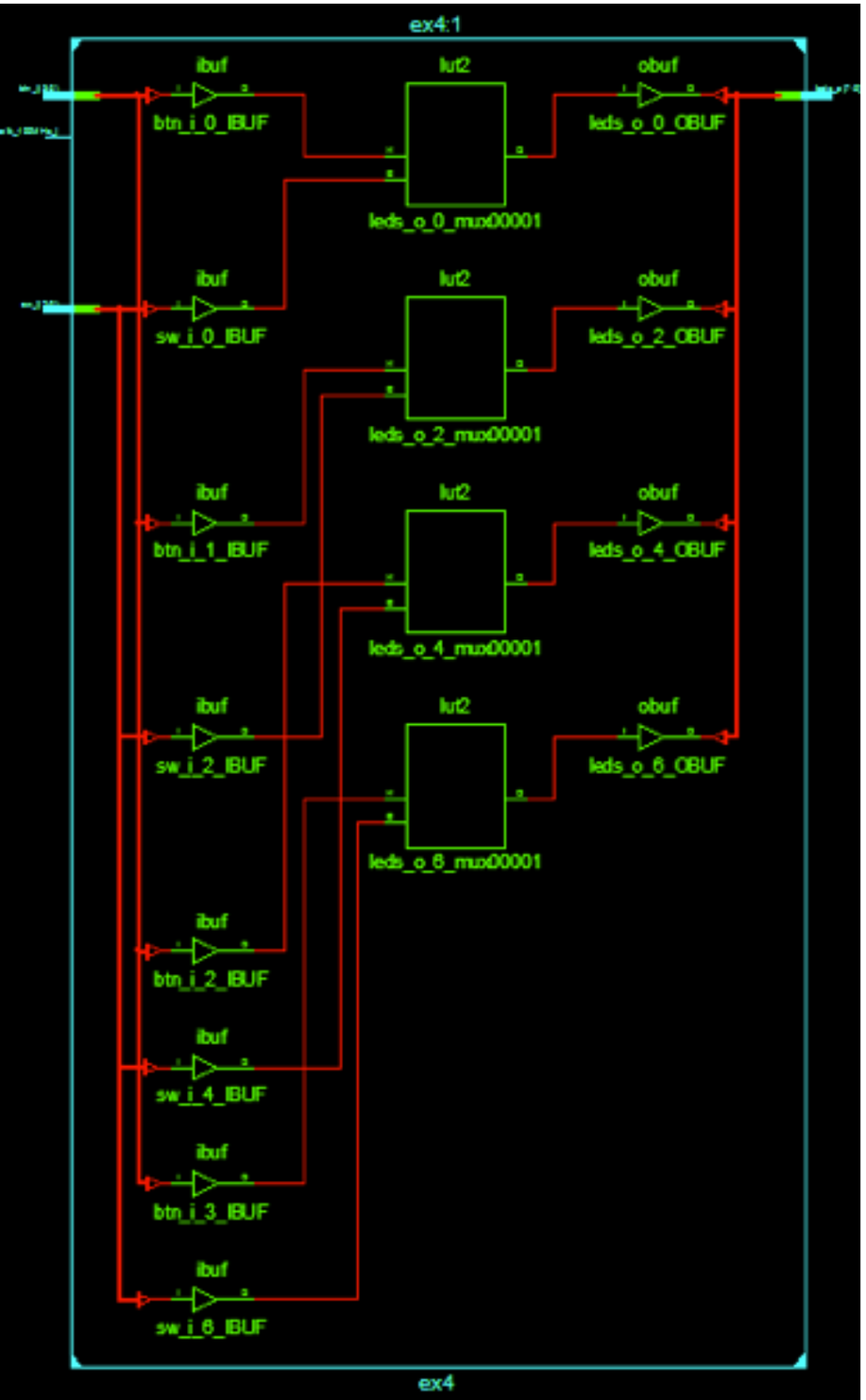
- If you want to visualise your design at the hardware component level, run the “Synthesize - XST” tool and then select the “View RTL Schematic” or “View Technology Schematic” options.

- The RTL schematic is your design translated in building blocks (logic gates, multiplexers, …).

- The technology schematic is your design adapted for the FPGA you are working with. It will only use ressources that are available.

LEDs example

Technology



- This is the result we get if we use the design of the LEDs with the generate conditions.

RTL