Finanziato dall'Unione europea NextGenerationEU
Ministero dell'Università e della Ricerca
Italiadomani PIANO NAZIONALE DI RIPRESA E RESILIENZA
Fondazione ICSC

Advanced course to FPGA programming

Piero Vicini, Ottorino Frezza, Francesca Lo Cicero, Francesco Simula,
INFN Rome1

INFN Istituto Nazionale di Fisica Nucleare

Course Overview

Introduction to FPGA

- Architecture
- Advantages and limitations
- Design Flow
- Simulation tool

Tools & Programming languages

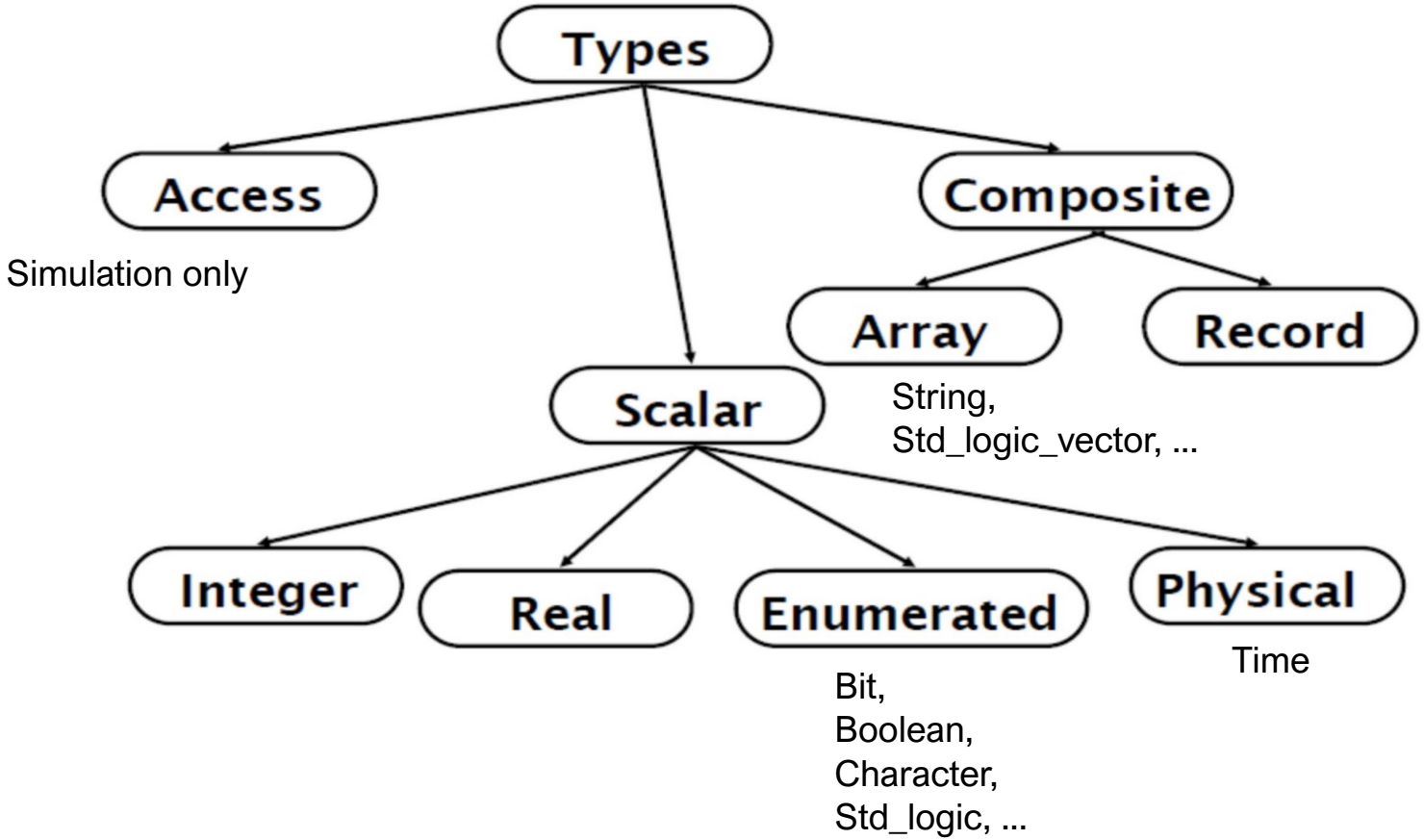
- Advanced VHDL
- Vivado tool overview
  - Design Flow
    - Project creation
    - Ip core integration
    - Simulation
    - Synthesis
    - Implementation
    - Tcl scripting
  - Timing analysis
  - Custom IP Design & Integration
  - FPGA Test & Debug

FPGA interconnection

- Quick intro
- AURORA Xilinx
  - Test and Debug
  - Fine tuning
  - Timing and Resources analysis
  - Optimization

VHDL Data type

- **Pre-defined type**



- **User-defined type**

- **Operation between different data types are not allowed!**

VHDL User-defined type

- **Create a new enumerated type**

```
type <type_name> is (<list of values>);
```

  - The list of values is a comma separated list of all the values of new type can have.
  - Once the type has been created, an instance of it can be declared in the code, and any of the defined values can be assigned to it.

```
type fsm_t is (idle, starting, running, stopping)
-- Declare a signal which uses our custom type
signal fsm : fsm_t;
-- Example of assigning a value to our signal
fsm <= stopping;
```

- **Create a subtype**
  - Restrict the range of valid values of existing type.

```
subtype <type_name> is <type> range <valid_values>;
```

  - Operations are allowed between subtype and the type from which it was derived
  - Assignment that are out of the subtype range result in error

```
Subtype integer_8_bit is integer range 0 to 255;
Signal in1 : integer_8_bit;
```

- **Create an array**

```
type <type_name> is array (<range>) of <type>;
```

- The <range> can be defined using the **downto** or **to** VHDL keywords.
- To create an unconstrained array type use
  - natural <range> (allows for 0)
  - positive <range>

```
type byte_array_type is array (natural <range>) of std_logic_vector(7 downto 0);
signal example : byte_array_type(7 downto 0);
```

VHDL User-defined type

- **Create a record**

Record is used to collect one or more elements of different types in a single construct.

- Elements can be any vhdl data type and can be accessed through field name.

```
type <record_name> is record
   <element_name> : <type>;
   <element_name> : <type>;
end record <record_name>;
```

- After we have declared a record type, we can use it in the exact same manner as any type.
- We can assign data to individual elements in the record or to the entire array.

```
signal <signal_name> : <record_name>;
<signal_name> <= (
    <element_name> => <value>,
    <element_name> => <value>);
<signal_name>.<element_name> <= <value>;
```

```
type uart_record_t is record
  rx : std_logic;
  cts : std_logic;
  tx : std_logic;
end record uart_record_t;
```

```
signal uart_example : uart_record_t;
-- Assigning data to all elements in
-- a record
uart_example <=
( rx => '0', cts => '0', tx => '0');
-- Set the tx signal to 1b
uart_example.tx <= '1';
```

VHDL User-defined Package

The primary purpose of a package is to encapsulate elements that can be shared (globally) among two or more entities.

The **package declaration** contains the declaration of the items visible to any design unit that uses the package .
Es: Type, subtype declaration

The **package body** defines the items declared package declaration.

```
USE LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
PACKAGE math IS
         TYPE st16 IS ARRAY(0 TO 15) OF std_logic;
END math;
PACKAGE BODY math IS
End math
```

To use a package, you need to include the library containing it and add **use statement** before the entity declaration (std and work libraries are visible by default).

```
library library_name ;
use library_name.package_name.all ;
```

Generic

Generics provide an ability to pass information into an entity during instantiation.

- Create flexible and easily reused code

```
entity entity_name is
[generic (
generic_name : generic_type := default_value;
…..
generic_name : generic_type := default_value
);]
port (
port_name : mode port_type;
port_name : mode port_type
);
end entity_name;
```

Note that the last generic definition does not end with a semicolon (;)

```
library IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
entity constant_mult is
generic ( multiplier: integer:=5);
port (
  port_in   : in integer;
  port_out  : out integer);
end constant_mult ;

architecture behavioural of constant_mult is
begin
  port_out <= port_in* multiplier ;
end behavioural;
```

Generate statement

Generate statements are used to accomplish one of two goals:

- Replicating Logic in VHDL

```
Generate_name: for variable in low_value to high_value
generate
   Sequence of statement
end generate Generate_name;
```

- Turning on/off blocks of logic in VHDL for debugging purposes, or for switching out different components without having to edit lots of code.

```
Generate_name: if condition generate
   Sequence of statement
end generate Generate_name;
```

Generate statement to replicate logic

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

entity VectorScaler is
   port (
        input_vec : in  integer_vector(0 to 1);
        output_vec: out integer_vector(0 to 1)
    );
end entity VectorScaler;

architecture rtl of VectorScaler is
Constant SCALE_FAC: integer := 15;
Begin

  output_vec(0) <= input_vec(0) * SCALE_FAC;
  output_vec(1) <= input_vec(1) * SCALE_FAC;

end architecture rtl;
```

=

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

entity VectorScaler is
generic ( N: integer:=2);
   port (
        input_vec   : in  integer_vector(0 to N-1);
        output_vec : out integer_vector(0 to N-1)
    );
end entity VectorScaler;

architecture rtl of VectorScaler is
Constant SCALE_FAC: integer := 15;
begin
  generate_loop: for i in 0 to N-1 loop
    output_vec(i) <= input_vec(i) * SCALE_FAC;
  end generate_loop;
end architecture rtl;
```

Generate statement to turn on/off logic block

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

entity VectorScaler is
generic ( N: integer:=2; SCALE_FAC: integer := 15 );
   port (
        input_vec  : in  integer_vector(0 to 1);
        output_vec : out integer_vector(0 to 1)
    );
end entity VectorScaler;

architecture rtl of VectorScaler is
Begin
generate_check_0: if SCALE_FAC> 0 generate
   generate_loop: for i in 0 to N-1 loop
      output_vec(i) <= input_vec(i) * SCALE_FAC;
   end generate_loop;
end generate_check_0;
generate_check_1: if SCALE_FAC= 0 generate
       output_vec <= input_vec;
end generate_check_1;
end architecture rtl;
```

- The same signal can be driven by multiple generate statements.
    - The designer needs to ensure that generate blocks are mutually exclusive, such that no two can be active at the same time. Otherwise, there will be a problem with the same signal being driven by two sources.

VHDL subprograms

- VHDL allows code reuse and abstraction through *subprograms*

  - Subprograms improve readability, maintainability, and modularity of designs.
  - They can be declared:
    - inside an architecture (local use)
    - inside a package (reusable across multiple files).

    - **Functions** → compute and return a value.
      - Great for combinational logic and calculations.

    - **Procedures** → perform actions on signals/variables.
      - Useful for signal manipulation or testbench tasks

VHDL subprograms: function

Functions

- Return a single value.
- No wait statements allowed.
- Parameters: input.

```
function <function_name> (<parameters>) return <output_type>
[local declaration part]
begin
 <Sequence of statements>
 return <value>
end function;
```

Optional local declaration part can contain:

- variable declarations
- constant declarations
- local type or subtype definitions

```
function add_bits(a, b : std_logic_vector) return std_logic_vector is
begin
  return a + b;
end function;
```

VHDL subprograms: procedure

Procedures

- Do not return a value directly.
- Parameters can be in, out, or inout.
- May contain wait statements.

```
procedure <procedure_name> (<parameters>) is
[local declaration part]
begin
 Sequence of statements
end procedure;
```

Optional local declaration part can contain:

- variable declarations
- constant declarations
- local type or subtype definitions

```
procedure swap(signal a, b : inout std_logic) is
  variable tmp : std_logic;
begin
  tmp := a;
  a   := b;
  b   := tmp;
end procedure;
```

VHDL for-loop statements

for - loop statement is used to repeat a block for a **fixed** number of iterations

```
[loop_label :] For <parameter> in <range> loop
 Sequence of statements
end loop [loop_label];
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity shift_left_1 is
Port ( a : in STD_LOGIC_VECTOR(3 downto 0);
       y : out STD_LOGIC_VECTOR(3 downto 0) );
end shift_left_1;
architecture Behavioral of shift_left_1 is
begin
 process(a)
 begin
  y(0) <= '0';
  for i in 1 to 3 loop y(i) <= a(i - 1);
  end loop;
 end process;
end Behavioral;
```

- <parameter> field is a variable within the loop and it is not necessary to declare it separately. However, we can't change the value of this parameter within the loop code.
- <range> field to specify how many times the for loop code is repeated.
  - incrementing range from A to B (with B>A)

    ```
    In A to B
    ```

  - decrementing range from A to B (with A>B)

    ```
    In A downto B
    ```

- **In synthesis for-loop statement is unrolled by the synthesizer, resulting in parallel hardware implementation**

VHDL while-loop statements

While ... Loop statement repeats a block while a condition is true

- **exit**: exits the loop early
- **next**: skips to the next iteration

```
While <condition> loop
 Sequence of statements
end loop;
```

- When writing **while true loop**, an infinite loop is created unless an internal exit statement is included to break the loop.

A while-loop in VHDL can be synthesized only if the synthesis tool can prove that it terminates.

- The maximum number of iterations can be determined at compile time.
- The loop is bounded by a signal or constant with a fixed known size (e.g., array length, integer range).
- The exit condition is guaranteed to be met.

- A While-Loop is suitable in simulation when the exact number of iteration cannot be determined in advance.
  - Typically used in testbenches to generate stimulus.

Hands-on

Hands-on info

- **TOOL**:
  - GHDL
  - GTKwave
  - Vivado

- **REPOSITORY**
  - Clone the repository
    Git clone https://gitlab.com/francescalocicero/advanced_fpga_programming
    or, if you have already cloned the course repository, Pull the latest changes
    git pull (from inside the folder)

- **DIRECTORY STRUCTURE OF THE REPOSITORY**
  - The repository is organized into multiple example folders.
  - Each example folder contains
    - Design Sources (.vhd and .xci files in **/src** subfolder)
    - Simulation Sources (testbench .vhd and waveform .wcfg in **/sim** subfolder)
    - Solution Sources (exercise's solution in **/solution** subfolder)
    - Optional additional files (e.g Makefile, .tcl scripts in /tcl)

Sensor project

Scenario:

- N sensors send a temperature value in signed format (STD_LOGI_VECTOR_LEN bit).
- Each temperature value is scaled by a constant calibration factor (e.g., scaling factor).
- A conditional check is applied: if the scaled temperature exceeds a certain threshold (defined as a std_logic_vector with length STD_LOGIC_VECTOR_LEN), an alarm is triggered.

Sensor project

Scenario:

- **N** sensors send temperature values in signed format.
- Each temperature value is scaled by a constant **calibration factor** (e.g., a scaling factor).
- A conditional check is applied: if the scaled temperature of a sensor exceeds a certain **threshold** (defined as signed std_logic_vector with length **STD_LOGIC_VECTOR_LEN**), an alarm is triggered for that specific sensor.

Let's define the constants!

N	Number of sensors	2
CALIBRATION_FACTOR	Constant used to scale the temperature value	5
STD_LOGIC_VECTOR_LEN	Length of std_logic_vector used to represent the threshold	10
TEMP_THRESHOLD	Temperature threshold	`"0001101000"`

- Use GHDL to compile and run the VHDL sources (provided in /src and /sim directories) - a Makefile is provided to simplify the compilation and simulation process.
- Use GTKWave to view and analyze the simulation waveforms produced by GHDL

Sensor project
Data_processor.vhd

```
entity DataProcessor is
    port (
        sensor_temp_0 : in  signed(9 downto 0);
        sensor_temp_1 : in  signed(9 downto 0);
        alarm_signal  : out std_logic_vector(1 downto 0)
    );
end entity DataProcessor;

architecture Behavioral of DataProcessor is

    signal scaled_temp_0, scaled_temp_1 : signed(9 downto 0);
    constant THRESHOLD : signed(9 downto 0) := "0001101000";

begin

    -- Scale temperatures by 5
    scaled_temp_0 <= to_signed(to_integer(sensor_temp_0) * 5, 10);
    scaled_temp_1 <= to_signed(to_integer(sensor_temp_1) * 5, 10);

    -- Alarm if scaled temperature > binary threshold directly written
    alarm_signal(0) <= '1' when scaled_temp_0 > signed(THRESHOLD) else '0';
    alarm_signal(1) <= '1' when scaled_temp_1 > signed(THRESHOLD) else '0';

end architecture Behavioral;
```

Sensor project
Data_processor.vhd

```
entity DataProcessor is
    port (
        sensor_temp_0 : in  signed(9 downto 0);
        sensor_temp_1 : in  signed(9 downto 0);
        alarm_signal  : out std_logic_vector(1 downto 0)
    );
end entity DataProcessor;

architecture Behavioral of DataProcessor is

    signal scaled_temp_0, scaled_temp_1 : signed(9 downto 0);
    constant THRESHOLD : signed(9 downto 0) := "0001101000";

begin

    -- Scale temperatures by 5
    scaled_temp_0 <= to_signed(to_integer(sensor_temp_0) * 5, 10);
    scaled_temp_1 <= to_signed(to_integer(sensor_temp_1) * 5, 10);

    -- Alarm if scaled temperature > binary threshold directly written
    alarm_signal(0) <= '1' when scaled_temp_0 > signed(THRESHOLD) else '0';
    alarm_signal(1) <= '1' when scaled_temp_1 > signed(THRESHOLD) else '0';

end architecture Behavioral;
```

There are constants and similar signals ...

Improvements:
- move the constants into a package
- define arrays to group related data

Sensor project
Data_processor.vhd

```
entity DataProcessor is
    port (
        sensor_
        sensor_
        alarm_s
    );
end entity Data

architecture Be

    signal scal
    constant TH

begin

    -- Scale temperatures by 5
    scaled_temp_0 <= to_signed(to_integer(sensor_temp_0) * 5, 10);
    scaled_temp_1 <= to_signed(to_integer(sensor_temp_1) * 5, 10);

    -- Alarm if scaled temperature > binary threshold directly written
    alarm_signal(0) <= '1' when scaled_temp_0 > signed(THRESHOLD) else '0';
    alarm_signal(1) <= '1' when scaled_temp_1 > signed(THRESHOLD) else '0';

end architecture Behavioral;
```

- Combine sensor_temp_0 and sensor_temp_1 into a two-elements input array
- Combine scaled_temp_0 and scaled_temp_1 signals into a two-elements array
- Combine temp_signed_0 and temp_signed_1 signals into two elements array
- Define all necessary array types into a package
- Add all project constants into the same package
- **Make sure to include the package in the file Data_processor.vhd**
- Update the testbench entity and architecture to use arrays.
- Run the simulation and verify the results.

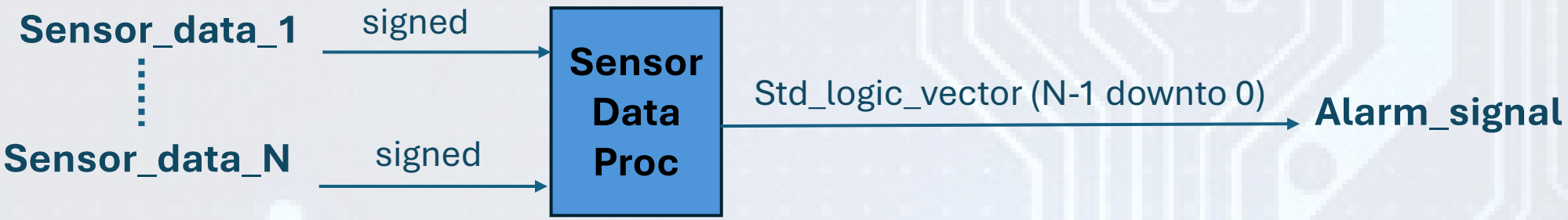
Sensor project
Data_processor.vhd

```
entity DataProcessor is
    Port ( sensor_temp  : in signed_array ( 1 downto 0);
           alarm_signal : out std_logic_vector ( 1 downto 0)
);
end DataProcessor;

architecture Behavior
    -- Scaled tempera
    signal scaled_tem

begin
    -- Scale each tem
    scaled_temp(0) <=
STD_LOGIC_VECTOR_LEN);
    scaled_temp(1) <= to_signed(to_integer(sensor_temp(1)) * CALIBRATION_FACTOR,
STD_LOGIC_VECTOR_LEN);

    -- Check against threshold
    alarm_signal(0) <= '1' when scaled_temp(0) > TEMP_TH
    alarm_signal(1) <= '1' when scaled_temp(1) > TEMP_THRESHOLD else '0';

end architecture Behavioral;
```

The code is still dependent on the number of ports, meaning that if the number of sensors changes, the entity and the internal logic must be manually updated.
We need something more….

- Use generic and generate statement!

Sensor data processor



Further improvement:

- Implement functions in the package:
  - function scale_temperature (temp: signed; scale_factor: integer) return signed;
    - Returns the temperature scaled by the calibration factor.
  - function exceeds_threshold (scaled_temp: integer) return boolean;
    - Returns true if the scaled temperature exceeds the threshold.