Advanced VHDL

Hipólito Guzmán Miranda
Departamento de Ingeniería Electrónica
Universidad de Sevilla
hguzman@us.es

Teaching context

B02: Advanced Programmable Logic Systems

- Tema 1: FPGA architecture
- Tema 2: Advanced digital design methodologies
- Tema 3: Advanced VHDL
- Tema 4: Verification capabilities for digital circuits

Required prior knowledge:

- Basic VHDL
  - Design with two processes

Learning objectives

- Expand the vocabulary of sentences and keywords in VHDL
- Become familiar with VHDL's potential to raise the level of abstraction in design, without losing sight of the behavior in synthesis
- Acquire skills to reduce code duplication and increase code reusability

Contents

Contents

Motivation

The VHDL you know (synthesis)

```
comb: process (<sensitivity_list>)
    if … elsif … else … end if;
    case … when => … end case;

sinc: process (rst, clk)
    if (rst = '1') then  …
    elsif (rising_edge(clk) then …
    end if;
```

Instances of components and entities

Motivation

The VHDL you know (simulation)

clk_process: **process**

- invert clk, **wait for** clk_period/2

stim_process: **process**

- Manually-generated stimuli sequence (very tedious to write for complex tests)

Motivation

‘Cargo cult programming’

In C, sloppy code usually produces poor results and is harder to debug and modify.

In VHDL, sloppy code can produce working hardware, but it will also be difficult to debug and modify -> **code that nobody wants to touch.**

Motivation

VHDL is a HIGH level language

- Describe at a higher abstraction level
- Let the synthesizer infer the circuit
- Synthesized hardware works just as well (or better), but the code is easier to read and maintain

But let's take it one step at a time...

Motivation

A few words of warning

Everything explained here requires hardware resources (in implementation)

Operations are not performed sequentially but concurrently

The design paradigm remains the same: operations become logic -> but you'll have more resources to structure your code

Contents

Records

The `record` datatype

“Bad programmers worry about the code. Good programmers worry about data structures and their relationships.” - Linus Torvalds

- This also applies when describing hardware
- `record(s)` are a type that is composed of other data
- They are the VHDL equivalent to C’s `struct`
- Group signals or ports of the same context into `record(s)`

Example: signals

```
type transceiver_data is
    record
      data : std_logic_vector (15 downto 0);
      valid : std_logic;
    end record;

signal datain, dataout : transceiver_data;
```

Example: ports

```
entity transceiver is
  Port (
    clk        : in std_logic;
    rst        : in std_logic;
    data_in    : in transceiver_data;
    data_out_I : out transceiver_data;
    data_out_Q : out transceiver_data
  );
end transceiver;
```

Ejemplo: puertos

```
entity transceiver is
```

- The full record has a single direction (IN o OUT)
- Adding a new signal to the port only requires changing the record definition!

```
        data_out_Q : out transceiver_data
    );
end transceiver;
```

Asignación y uso

Acceder a **`record.dato`** :

```
if (data_in.valid = '1') then
    data_out.data <= data_in.data;
    data_out.valid <= '1';
end if;
```

Contents

Functions

Read signals, return value

```
function invert (data: std_logic) return std_logic is
  begin
    return not data;
  end function invert;
```

On synthesis, each call to the function will generate an inverter!

Ejemplos

```
function sum (a: integer; b: integer)
return integer is
  begin
    return a+b;
  end function sum;
```

On synthesis, each call to the function will generate an adder

Functions

Ejemplos

```
function sel (cond: boolean; if_true,
if_false: integer) return integer is
  begin
    if cond = true then
      return (if_true);
    else
      return (if_false);
    end if;
  end function sel;
```

Functions

Why use them?

Since they produce the same hardware, it's worthwhile to use them for:

- Encapsulating operations for reuse
- Multiplexing or inverting generics, constants, or signals in a **`generic map`** or **`port map`**

I must insist: they are not subroutines, they are **hardware**!

Procedures

Multiple inputs, multiple outputs

Apparently similar to `functions` but:

- Have IN and OUT parameters
- Can read from the IN parameters and modify the OUT parameters

Procedures

Ejemplo:

```
procedure vect_write
(constant data: in std_logic_vector(31 downto 0);
signal vector_ctrl : out fifo_ctrl) is
  begin
    vector_ctrl.datai <= data;
    vector_ctrl.wr_en <= '1';
    wait for 10 ns;
    vector_ctrl.wr_en <= '0'; --after 10 ns;
  end procedure;
```

(Not synthesizable since it contains a wait statement)

Functions and Procedures

Differences:

- Functions do not modify anything, they simply return a value

```
data <= a_function (other_data);
```

- Procedures change the values of signals

```
my_procedure (signals_in, signals_out);
```

Contents

For loop

For loop

```
for i in 0 to 7 loop
```

- The synthesizer **expands the loop during synthesis**
- The range of the loop must be static (so that it can be synthesized)
- Each pass through the loop is not an ‘iteration’, but a repetition of the hardware

For loop

Example

```
reorder_data: process (data_in)
begin
  for i in 0 to 7 loop
    data_out(i) <= data_in(7-i);
  end loop;
end process;
```

For loop

Is equivalent to:

```
reorder_data: process (data_in)
begin
   data_out(0) <= data_in(7);
   data_out(1) <= data_in(6);
   data_out(2) <= data_in(5);
   data_out(3) <= data_in(4);
   data_out(4) <= data_in(3);
   data_out(5) <= data_in(2);
   data_out(6) <= data_in(1);
   data_out(7) <= data_in(0);
end process;
```

Conditional instantiation of components

- Instances a component/entity or not, depending on whether a condition is met
- This condition must be static so that it is known **at synthesis time**
- The synthesizer instances (or not) the component

Generate sentence

`if condition generate`

```
second_instance: if GENERATE_TWO=true
generate
  inst2: cont port map (
    clk => clk,
    rst => rst,
    count => count2 );
end generate second_instance;
```

Multiple component instantiation

- Using `for parameter in range`
- The same as before, the synthesizer **expands the loop during synthesis**
- The loop range must be static (so that it can be synthesized)
- Each pass of the loop is not an 'iteration', is an **instance** of the hardware (component/entity)

Generate sentence

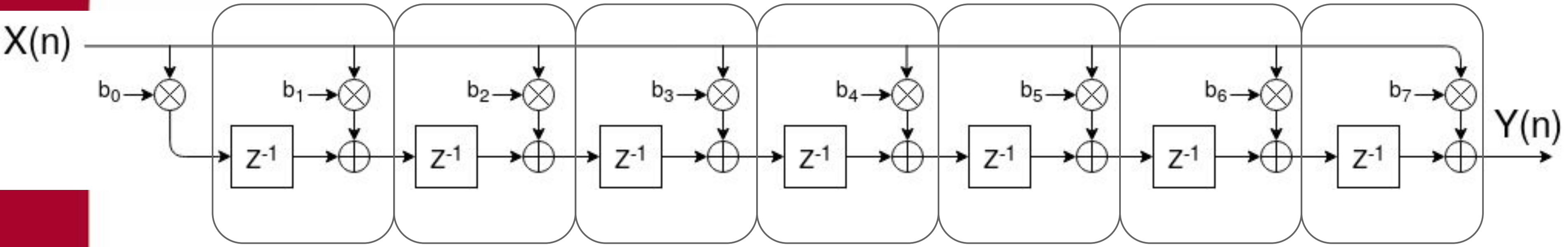
for parameter in range generate

```
regdesp:
  for i in 0 to 3 generate
    myreg : reg port map (
      clk => clk,
      rst => rst,
      din => data(i),
      dout => data(i+1));
   end generate regdesp;
```

Generate sentence

FIR filter

As a set of taps (stages)



```vhdl
channel_filter: for i in 0 to 23 generate
  taps: tap generic map(
      INPUT_WIDTH    => 9,
      OUTPUT_WIDTH   => 10,
      TRUNC_BITS     => 8,
      COEF           => coefs(sel(i<12, i, 23-i)),
      SAT_MULT_BITS => 2)
    port map(
      clk => clk,
      rst => rst,
      valid => cfilterin_valid,
      input => cfilterin,
      prev => d_aux(i),
      output => d_aux(i+1)
    );
  end generate;
```

Contents

Packages

Encapsulate all we have previously seen

In a VHDL **package** we can define:

- Data types
- Constants
- Functions
- Procedures
- Components

Packages

Encapsulate all we have previously seen

Instead of redeclaring everything we need in each `.vhd` file of each entity, we simply add the following to the `library` section:

```
use work.mypackage.all;
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

package mypackage is
    -- declaration of data types
    -- declaration of constants
    -- declaration of components
    -- declaration of functions and procedures
end mypackage;

package body mypackage is
    -- definition of functions and procedures




end mypackage;
```

Sets of packages

Included here for completeness, but you won’t need to create them for this subject

For example, `std_logic_1164` is a package of the `IEEE` library:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
```

You packages will belong to library **`work`** by default

Sets of packages

To use third-party libraries, they must be compiled/synthesized separately, and in the library section:

```
library uvvm_util;
use uvvm_util.types_pkg.all;
use uvvm_util.string_methods_pkg.all;
use uvvm_util.adaptations_pkg.all;
use uvvm_util.methods_pkg.all;

use <library>.<package>.all;
```

Contents

std_ulogic vs std_logic

std_ulogic: unresolved
std_logic: resolved

```
TYPE std_ulogic 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
                   );

SUBTYPE std_logic IS resolved std_ulogic;
```

std_ulogic vs std_logic

std_ulogic explained

- ‘0’, ‘1’: Normal use
- ‘Z’: When we need to put something at high impedance (such as shared buses; it can usually only be put on the FPGA pins, which is the only place where tri-state gates are usually found)
- ‘U’: It warns us in simulation that we haven't initialized something correctly (e.g., poorly implemented resets)
- ‘X’: It warns us in simulation about short circuits, or about some operations performed over ‘U’ values
- ‘L’, ‘H’: Modeling of pulldowns/pullups in simulation
- ‘W’: Warns us in simulation about short circuits between ‘L’ and ‘H’
- ‘-’: Can be used as a wildcard when comparing vectors

  (`if vect = “11-0-1--” then`)

	Fuerte		Débil	
Bajo	'0'	Cero lógico fuerte	'L'	Cero lógico débil (Low = pulldown)
Alto	'1'	Uno lógico fuerte	'H'	Uno lógico débil (High = pullup)
Desconocido	'X'	Valor fuerte desconocido (Unknown)	'W'	Valor débil desconocido (Weak)

Especiales	
'Z'	Alta impedancia (High impedance)
'U'	Sin inicializar (Uninitialized)
'-'	No importa (Don't care)

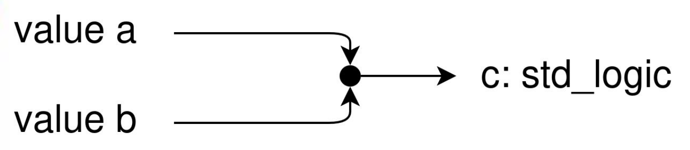
std_ulogic vs std_logic

Resolution function

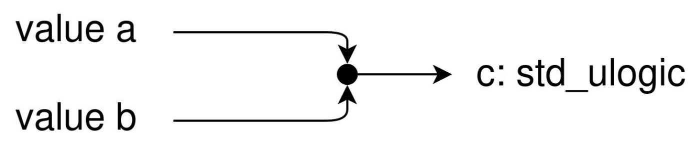
```
-------------------------------------------------------------------
-- resolution function
-------------------------------------------------------------------
CONSTANT resolution_table : stdlogic_table := (
--      ---------------------------------------------------------
--      |  U    X    0    1    Z    W    L    H    -        |   |
--      ---------------------------------------------------------
        ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U |
        ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X |
        ( 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X' ), -- | 0 |
        ( 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X' ), -- | 1 |
        ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), -- | Z |
        ( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), -- | W |
        ( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), -- | L |
        ( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), -- | H |
        ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' )  -- | - |
        );
```

std_ulogic vs std_logic

What does this mean?



OK! c takes the value resolve(a,b)



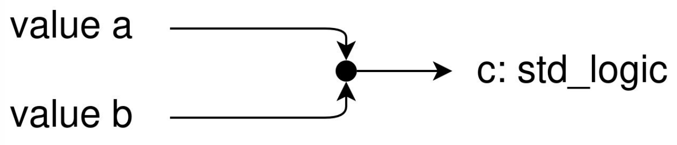
ERROR!

- We don't always want this! (In fact, we almost never do!)
- Synthesizers warn you about multi-sources
- Simulators don't! (you get 'X's in the waveforms)

- Very useful as a check during synthesis / compilation time

std_ulogic vs std_logic

When do we want `std_logic`?



OK! c takes the value resolve(a,b)

Synthesis:

- Bidirectional ports (sometimes)

Simulation:

- Modeling of pull-ups, pull-down, resistors (outside of our digital design)
- Shared buses

Contents

Standard versions

- VHDL'87
  - First version
- VHDL'93
  - Version with greater support from proprietary synthesis and simulation tools. Introduces shared variables
- VHDL 2002
  - Adds protected types for shared variables. VHPI (VHDL Procedural Interface) was added in 2007
- VHDL 2008 (more information here)
  - Integration of PSL (Property Specification Language). Generics in types, packages and subprograms. Supported in synthesis by Synopsys, and in simulación by QuestaSim and GHDL. Multiple usability improvements (`process(all);`)
- VHDL 2019
  - Latest version (let's give the tools time...)

Contents

Conclusions and recomendations

- VHDL offers options for structuring code to make it more maintainable
- Using all of these options is not mandatory
- Even though it's encapsulated, it's still hardware!
- Remember that vendor tools typically generate templates for everything mentioned above

Contents

Bibliography

- Brian Mealy, Fabrizio Tappero, *Free Range VHDL*. Free Range Factory, 2018
- *The VHDL Golden Reference Guide.* Doulos, 1995
- Ricardo Jasinski, *Effective Coding with VHDL: principles and best practice.* The MIT Press, 2016

Learning outcomes

- Understanding how to use the record type to restructure data processed by a design
- Understanding the differences between functions and procedures
- Using for loops and generate to create hardware instances
- Understanding that common code can be moved to a package to avoid code duplication
- When to use `std_ulogic` and when to use `std_logic`?