

# Tema 3: VHDL avanzado

Hipólito Guzmán Miranda  
Departamento de Ingeniería Electrónica  
Universidad de Sevilla  
[hguzman@us.es](mailto:hguzman@us.es)

# Contexto docente

BT02: Lenguajes de descripción hardware

- Tema 3: VHDL avanzado
- Tema 4: VHDL para procesamiento de señal

Conocimientos previos requeridos:

- VHDL básico (SEC GITT / Complementos de Electrónica MUIT)
  - Diseño con dos procesos

## Objetivos de aprendizaje

- Ampliar el vocabulario de sentencias y palabras clave en VHDL
- Familiarizarse con el potencial de VHDL para elevar el nivel de abstracción en diseño, sin perder de vista el comportamiento en síntesis
- Adquirir capacidades para reducir la duplicidad de código e incrementar su reutilizabilidad

## Contenido

- Motivación
- Records
- Functions y Procedures
- Sentencias For y Generate
- Packages y Libraries
- std\_ulogic vs std\_logic
- Versiones del estándar
- Conclusiones
- Bibliografía

## Contenido

- Motivación
- Records
- Functions y Procedures
- Sentencias For y Generate
- Packages y Libraries
- std\_ulogic vs std\_logic
- Versiones del estándar
- Conclusiones
- Bibliografía

## El VHDL que conocéis (síntesis)

comb: **process** (<lista\_de\_sensibilidad>)

**if** ... **elsif** ... **else** ... **end if**;

**case** ... **when =>** ... **end case**;

sinc: **process** (<lista\_de\_sensibilidad>)

**if** (rst = '1') **then** ...

**elsif** (rising\_edge(clk)) **then** ...

**end if**;

Instancias de componentes

## El VHDL que conocéis (simulación)

clk\_process: **process**

- invertir clk, **wait for** clk\_period/2

stim\_process: **process**

- Secuencia de estímulos generada manualmente (muy tedioso en tests complejos)

## ‘Cargo cult programming’

En C, código descuidado normalmente produce malos resultados, y es más difícil de depurar y modificar

En VHDL, código descuidado puede producir hardware que funcione, pero también será difícil de depurar y modificar -> **código que nadie quiere tocar**



## **VHDL es un lenguaje de ALTO nivel**

- Describid a un mayor nivel de abstracción
- Dejad que el sintetizador infiera el circuito
- El hardware sintetizado funciona igual de bien (o mejor), pero el código es más sencillo de leer y mantener

Pero vayamos poco a poco...

## Unas palabras de advertencia

Todo lo que se explica aquí cuesta recursos hardware (en implementación)

Las operaciones no se realizan secuencialmente sino concurrentemente

Paradigma de diseño es el mismo:  
operaciones se convierten en lógica -> pero  
tendréis más recursos para estructurar  
vuestro código

## Contenido

- Motivación
- Records
- Functions y Procedures
- Sentencias For y Generate
- Packages y Libraries
- std\_ulogic vs std\_logic
- Versiones del estándar
- Conclusiones
- Bibliografía

## El tipo de dato record

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

- Esto también aplica cuando describimos hardware
- Los records son un tipo de dato que está compuesto de otros datos
- Son el equivalente VHDL a los ‘structs’ de C
- Agrupad señales o puertos del mismo contexto en records

## Ejemplo: signals

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

## Ejemplo: puertos

```
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**

- El record entero tiene que tener una única dirección (IN o OUT)
- Añadir una nueva señal al puerto sólo implica cambiar la definición del record!

```
    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;
```



## Contenido

- Motivación
- Records
- Functions y Procedures
- Sentencias For y Generate
- Packages y Libraries
- std\_ulogic vs std\_logic
- Versiones del estándar
- Conclusiones
- Bibliografía

## Leer señales, devolver valor

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

Cada llamada a la función generará un inversor al sintetizar!

## Ejemplos

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

Cada llamada a la función generará un sumador al sintetizar

## 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;
```

## ¿Por qué usarlas?

Ya que producen el mismo hardware, merece la pena usarlas para:

- Encapsular operaciones que reutilizas
- Multiplexar o invertir generics, constants o señales en un **generic map** o **port map**

Insisto: no son subrutinas, son **hardware!**

## Múltiples entradas, múltiples salidas

Aparentemente similares a los `functions` pero:

- Tienen parámetros IN y OUT
- Pueden leer de los IN y modificar los OUT

## 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;
```

(No es sintetizable ya que contiene un wait)

## Diferencias:

- Las funciones no modifican nada, simplemente devuelven un valor
- Los procedimientos cambian el valor de señales

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

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



## Contenido

- Motivación
- Records
- Functions y Procedures
- Sentencias For y Generate
- Packages y Libraries
- std\_ulogic vs std\_logic
- Versiones del estándar
- Conclusiones
- Bibliografía

## Bucle for

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

- El sintetizador **expande el bucle durante la síntesis**
- El rango del bucle debe ser estático (para que pueda sintetizarse)
- Cada paso por el bucle no es una 'iteración', sino una repetición del hardware

## Ejemplo

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

## Es equivalente a:

```
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 loop;
end process;
```

## Instanciación condicional de componentes

- Instancia un componente o no, según se cumpla o no una condición
- Esta condición debe ser estática de forma que se sepa si se cumple **durante la síntesis**
- El sintetizador es el que instancia o no el componente

## if condition generate

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

## Instanciación múltiple de componentes

- Usando `for parameter in range`
- Al igual que antes, el sintetizador **expande el bucle durante la síntesis**
- El rango del bucle debe ser estático (para que pueda sintetizarse)
- Cada paso por el bucle no es una 'iteración', sino una **instancia** del componente

```
for parameter in range  
generate
```

```
regdesp:
```

```
for i in 0 to 3 generate
```

```
myreg : reg port map (
```

```
clk => clk,
```

```
rst => rst,
```

```
din => data(i),
```

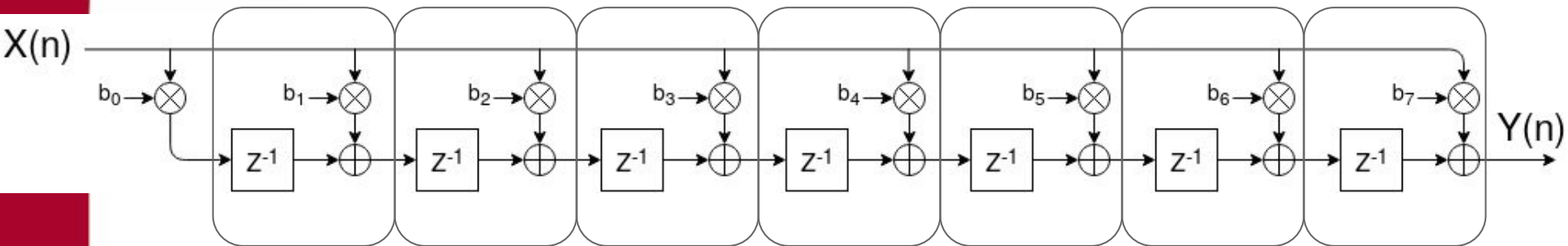
```
dout => data(i+1));
```

```
end generate regdesp;
```



## Filtro FIR

Como conjunto de etapas



```

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;

```

## Contenido

- Motivación
- Records
- Functions y Procedures
- Sentencias For y Generate
- Packages y Libraries
- `std_ulogic` vs `std_logic`
- Versiones del estándar
- Conclusiones
- Bibliografía

## Encapsular todo lo anterior

En un package VHDL podemos definir:

- Tipos de datos
- Constantes
- Funciones
- Procedimientos
- Componentes

## Encapsular todo lo anterior

En lugar de redeclarar todo lo que necesitamos en cada vhd de cada entidad, simplemente añadimos a la sección `library`:

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

```
library IEEE;
```

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

```
package mypackage is
```

```
-- declaración de tipos de datos
```

```
-- declaración de constantes
```

```
-- declaración de componentes
```

```
-- declaración de funciones y procedimientos
```

```
end mypackage;
```

```
package body mypackage is
```

```
-- definición de funciones y procedimientos
```

```
end mypackage;
```

## Conjuntos de packages

Se incluyen por completitud, pero no os hará falta crearlas para la asignatura

Por ejemplo, `std_logic_1164` es un package del library IEEE:

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

Vuestros packages custom pertenecen al library `work` por defecto

## Conjuntos de packages

Para utilizar libraries de terceros, deben compilarse/sintetizarse aparte, y en la sección library:

```
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;
```



## Contenido

- Motivación
- Records
- Functions y Procedures
- Sentencias For y Generate
- Packages y Libraries
- `std_ulogic` vs `std_logic`
- Versiones del estándar
- Conclusiones
- Bibliografía

## std\_ulogic: no resuelto std\_logic: resuelto

```
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;
```

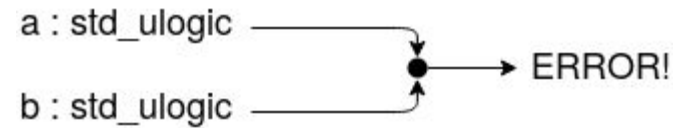
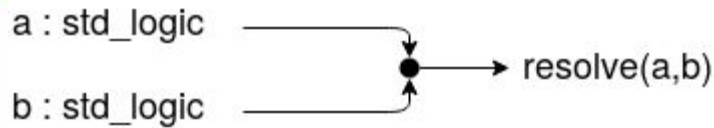
## Función de resolución

```
-- resolution function
```

```
CONSTANT resolution_table : stdlogic_table := (
```

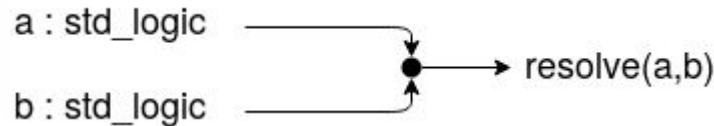
```
-- -----
-- | U   X   0   1   Z   W   L   H   -   | |
-- -----
( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U |
( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X |
( 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X', 'X' ), -- | 0 |
( 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X', 'X' ), -- | 1 |
( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X', 'X' ), -- | Z |
( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X', 'X' ), -- | W |
( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X', 'X' ), -- | L |
( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X', 'X' ), -- | H |
( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ) -- | - |
);
```

## ¿Qué significa esto?



- No siempre queremos esto! (de hecho, casi nunca lo queremos)
- El sintetizador te avisa de los multi-source
- Los simuladores no! (aparecen 'X' en los waveform)
- Nos sirve de chequeo en tiempo de síntesis / compilación

## ¿Cuándo lo queremos?



### Síntesis:

- Puertos bidireccionales

### Simulación:

- Modelado de pull-ups, pull-down, resistencias (fuera de nuestro diseño digital)
- Buses compartidos

## Contenido

- Motivación
- Records
- Functions y Procedures
- Sentencias For y Generate
- Packages y Libraries
- `std_ulogic` vs `std_logic`
- Versiones del estándar
- Conclusiones
- Bibliografía

- VHDL'87
  - Primera versión
- VHDL'93
  - Versión con mayor soporte por las herramientas de síntesis y simulación propietarias. Introduce variables compartidas
- VHDL 2002
  - Añade tipos protegidos para variables compartidas. Se añade VHPI (VHDL Procedural Interface) en 2007
- VHDL 2008 (más información [aquí](#))
  - Integración de PSL (Property Specification Language). Generics en tipos, paquetes y subprogramas. Soportado en síntesis por Synopsys, y en simulación por QuestaSim y GHDL. Múltiples mejoras de usabilidad (`process (all) ;`)
- VHDL 2019
  - Pendiente de aprobación

## Contenido

- Motivación
- Records
- Functions y Procedures
- Sentencias For y Generate
- Packages y Libraries
- std\_ulogic vs std\_logic
- Versiones del estándar
- Conclusiones
- Bibliografía



## Conclusiones y recomendaciones

- VHDL da opciones para estructurar el código de manera que sea más mantenible
- No es obligatorio usarlo todo
- Aunque esté encapsulado, sigue siendo HW!
- Se recuerda que Xilinx ISE genera plantillas para todo lo mencionado anteriormente (en Project -> New source y en Edit -> Language Templates)

## Contenido

- Motivación
- Records
- Functions y Procedures
- Sentencias For y Generate
- Packages y Libraries
- `std_ulogic` vs `std_logic`
- Versiones del estándar
- Conclusiones
- Bibliografía

## Bibliografía

- 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

## Resultados de aprendizaje

- Saber utilizar el tipo record para reestructurar los datos procesados por un diseño
- Diferencias entre function y procedure
- Uso de for y generate para generar instancias de hardware
- Saber que se puede mover código común a un package para evitar duplicidad de código
- ¿Cuándo se debe utilizar std\_ulogic y cuándo std\_logic?