VHDL 1: Estructura de un fichero VHDL

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

Acknowledgement to Fernando Muñoz,
Universidad de Sevilla

Contexto docente

BT01: Introducción a VHDL y su aplicación en FPGAs

- Tema 1: Estructura de un fichero VHDL
- Tema 2: Describiendo la funcionalidad
- Tema 3: Diseño de circuitos síncronos
- Tema 4: Simulación con testbenches

Conocimientos previos requeridos:

- Electrónica digital básica (puertas lógicas y biestables)

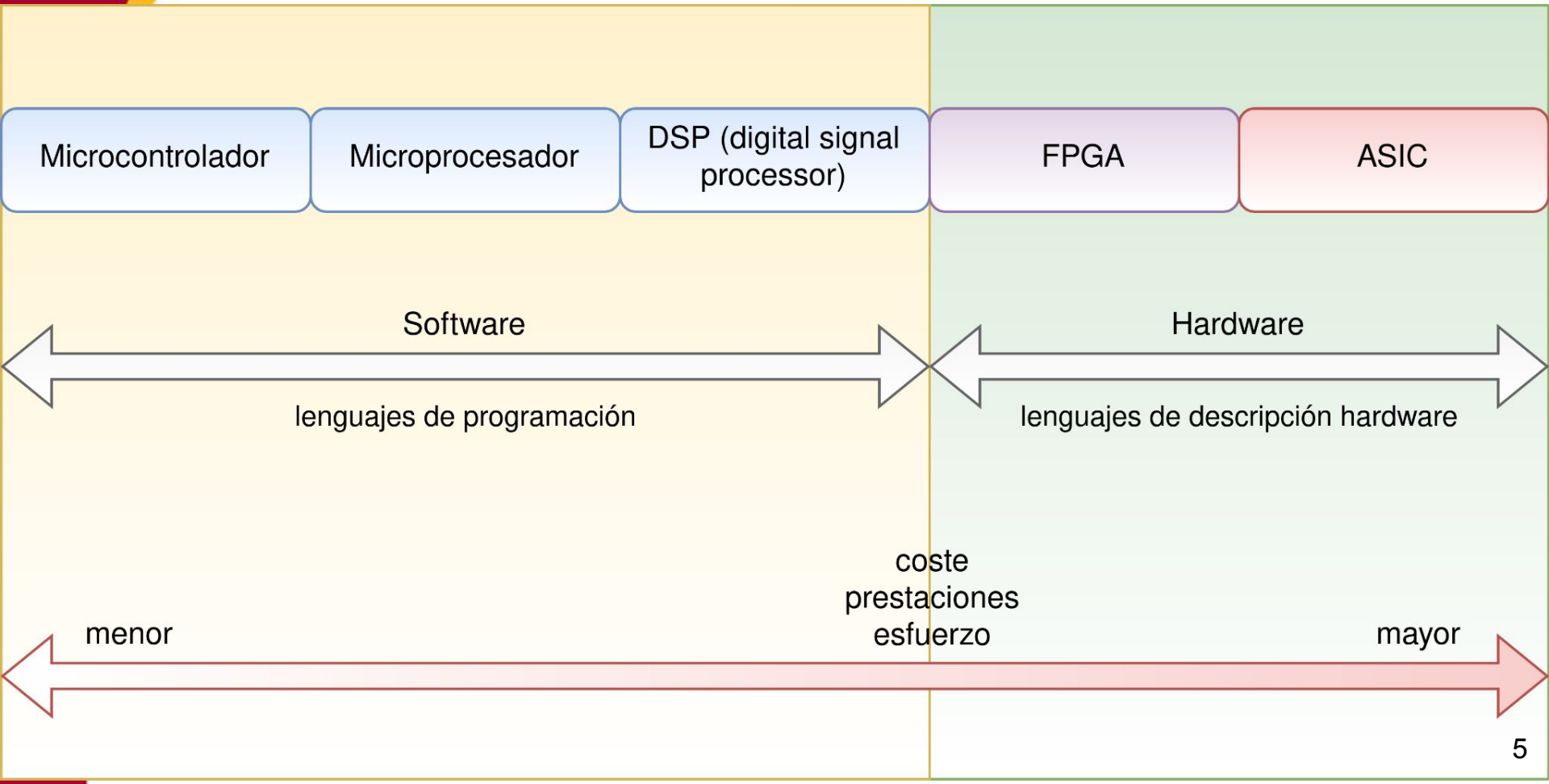
Objetivos de aprendizaje

- Revisar las posibilidades existentes para diseño de sistemas embebidos
- Encajar los dispositivos FPGA en dichas posibilidades
- Conocer qué es una FPGA
- Motivar el aprendizaje de diseño digital usando FPGAs
- Conocer la estructura de un fichero VHDL
- Conocer los tipos de datos más comunes en VHDL y cómo se manejan

Contenido

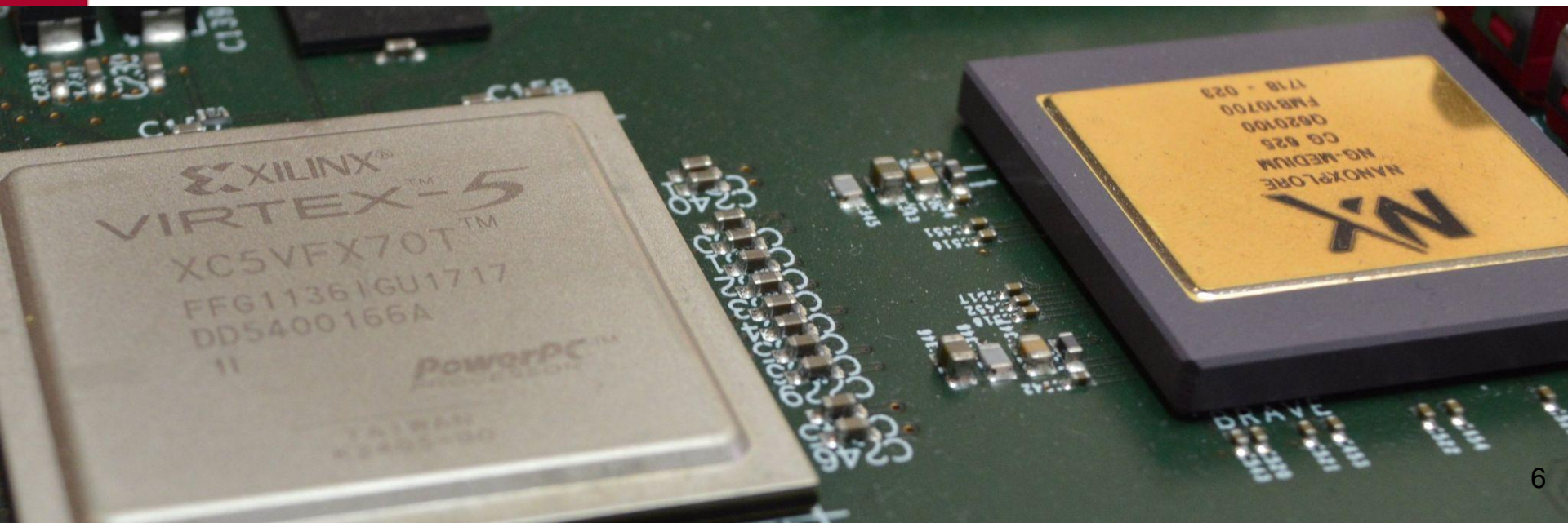
- Soluciones para sistemas embebidos
- ¿Qué es una FPGA y por qué aprender a usarlas?
- Niveles de abstracción
- Flujo de diseño
- Programa vs diseño
- ¿Qué es VHDL?
- Ejemplo básico
- Sección `library` y tipos de datos
- Sección `entity`
- Sección `architecture`
- Sección `configuration`

Múltiples opciones para sistemas 'embedded'



¿Qué es una FPGA?

- FPGA = Field Programmable Gate Array
- Es un circuito integrado cuya arquitectura interna puede ser configurada por el cliente (diseñador)
- Normalmente se configura utilizando un Lenguaje de Descripción Hardware (HDL), y herramientas asociadas para la síntesis e implementación del diseño

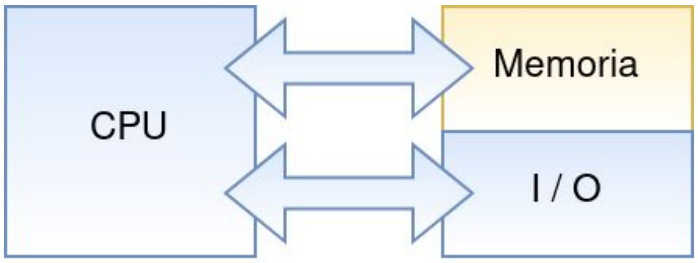


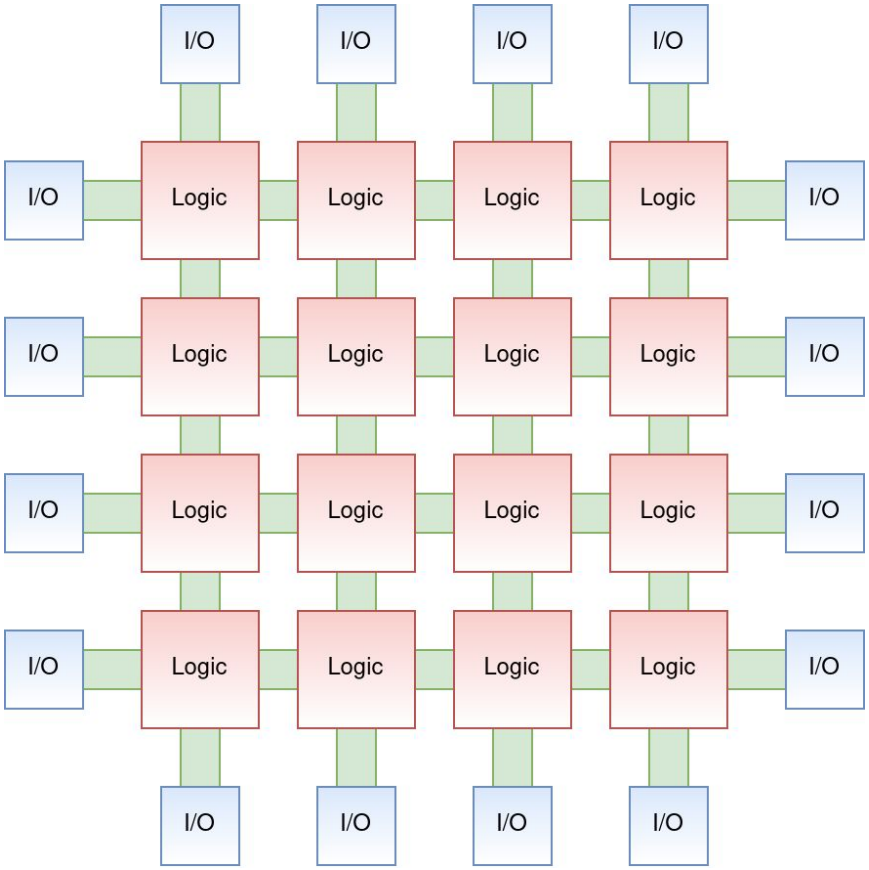
Programabilidad
FPGA vs microprocesador

"Pero… un microprocesador también es programable, no?"

La **separación entre CPU y memoria** es lo que hace que el microprocesador sea programable: la CPU siempre es la misma, y el ingeniero puede cambiar el **programa** almacenado en la memoria

Microprocesador vs FPGA

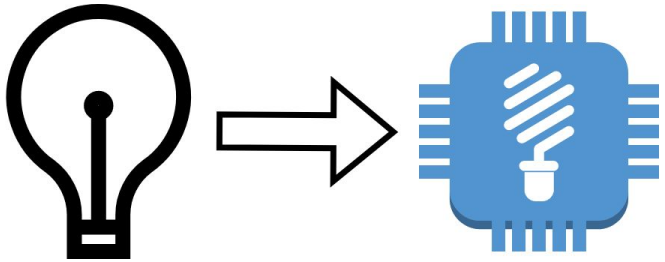




¿Por qué aprender a usar FPGAs?

Prototipado rápido de circuitos digitales a medida

- No es necesario fabricar un ASIC
  - Menor esfuerzo
  - Reducción de costes para tiradas pequeñas
- Reducción del 'Time to Market'
- I+D+i
- Confidencialidad



¿Por qué aprender a usar FPGAs?

Rendimiento / prestaciones

- Procesamiento en paralelo

Aplicaciones en múltiples industrias:

- Astronomía
- Física de partículas
- Aeronáutica
- Espacio
- Banca / inversiones

Aplicaciones

- Astronomía
  - BEE2 (Berkeley Emulation Engine 2)
    - Usado en el Allen Telescope Array
    - PetaOp/second
- Física de partículas
  - Detector del CMS del LHC en CERN
- Aeronáutica
  - Sistemas de RADAR y comunicaciones
- Espacio
  - Solar Orbiter, Misiones OSIRIS, Misión Rosetta y muchas otras…
- Banca / inversiones
  - High Frequency Trading

Oportunidad laboral

FPGA jobs

Sort by: **relevance** - date 3,008 jobs

FPGA Engineer
Quantum Dimension
Huntington Beach, CA
We are seeking a talented FPGA design engineer to join our design & development team at our corporate office in Huntington Beach, CA. This FPGA position…
Posted 30+ days ago · More...

ASIC-FPGA Design Engineer
Intellibee
Phoenix, AZ
Description: Analyze, design, simulate, and implement algorithms in hardware descriptor languages, HDL (VHDL, Verilog, System Verilog), based on customer…
Posted 30+ days ago · More...

PWB Designer - Staff
Qualcomm
San Diego, CA
Company: Qualcomm Technologies, Inc. Job Area: Engineering Services Group, Engineering Services Group > PWB Design General Summary: The successful…
Posted 6 days ago · More...

Senior FPGA Design Engineer

<- indeed.com

(FPGA, United States)

linkedin.com ->

(FPGA, Spain)

302 VHDL Jobs in Spain (21 new)

FPGA ENGINEER
onhunters
Madrid, Community of Madrid, Spain
Be an early applicant
2 weeks ago

Power Electronics Engineer
Applus+ IDIADA
la Bisbal del Penedès, Catalonia, Spain
Be an early applicant
6 days ago

FPGA's Design Engineer
Airbus
Madrid, Community of Madrid, Spain
Be an early applicant
2 hours ago

FPGA Engineer
Gradiant
Vigo, Galicia, Spain
3 weeks ago

Electronic Hardware Designer
SET Europa
San Sebastián, Basque Country, Spain
Be an early applicant
2 months ago

Fpga Engineer Luxquanta, Spain
LuxQuanta
Madrid, Community of Madrid, Spain
Be an early applicant
2 weeks ago

De abstracto a específico

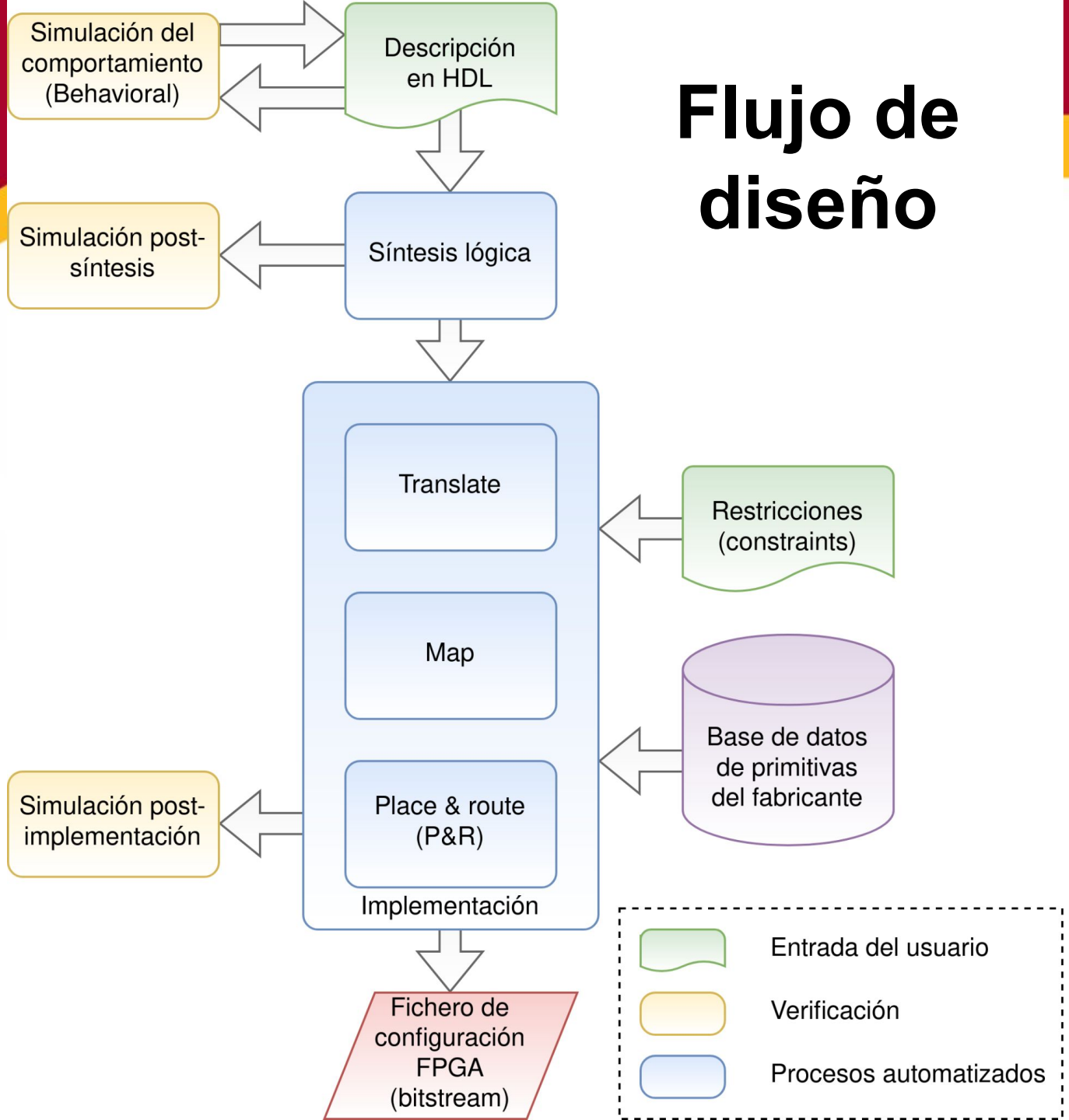
Nivel	Descripción
System Level	Visión general (diagrama de bloques)
High Level	Implementación funcional en lenguajes de programación de alto nivel (C, C++, SystemC, Matlab, Python, …)
Behavioral Level	Descripción 'cycle-accurate' del circuito en lenguaje de descripción hardware (Verilog, VHDL, …)
Register-Transfer Level (RTL)	Lista de registros y funciones lógicas. Típicamente una netlist, pero también puede ser un código en HDL (Verilog, VHDL, …)
Logic Gate Level	Netlist de biestables y puertas lógicas básicas (and, or, not, …)
Physical Gate Level	Netlist de puertas o primitivas disponibles en la arquitectura objetivo (ya sea FPGA o ASIC)
Switch Level	Netlist de transistores individuales

Fuente: Claire Xenia Wolf, Yosys Presentation

Uso típico en diseño HW

Nivel	Uso típico
System Level	Planteamiento a nivel general del sistema y división en bloques funcionales
High Level	Creación de un modelo de referencia (golden model) para verificación. En algunos casos, síntesis de alto nivel utilizando herramientas tipo Vitis HLS
Behavioral Level	Diseño del hardware en sí utilizando VHDL o Verilog. Simulación funcional (behavioral simulation)
Register-Transfer Level (RTL)	Diseño del hardware en sí utilizando VHDL o Verilog. Simulación funcional (behavioral simulation)
Logic Gate Level	Diseño mapeado a la tecnología FPGA concreta. Simulación post-síntesis
Physical Gate Level	Diseño rutado. Simulación post-implementación
Switch Level	Configuración del dispositivo FPGA

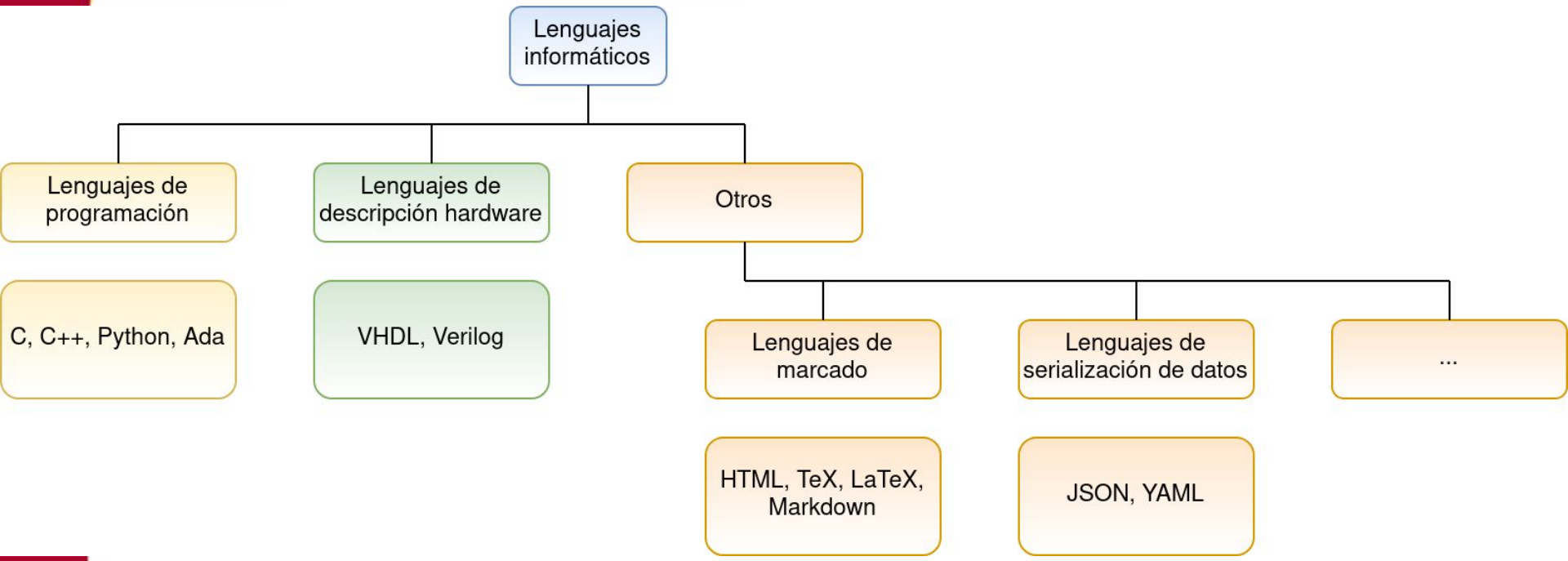
Flujo de diseño



Síntesis e Implementación

- **Síntesis**: convierte de Behavioral a RTL
- **Translate**: parte de todo el RTL y las ‘constraints’ (pines, timing, etc) y los une en un único conjunto
- **Map**: describe el diseño utilizando únicamente las primitivas que tenemos disponibles en la FPGA (Look-up-Tables, Flip-flops, Block RAMs, etc)
- **Place and Route**: decide qué recursos exactos va a utilizar dentro del silicio y configura las conexiones entre estos

Tipos de lenguajes informáticos



Programación vs Descripción Hardware

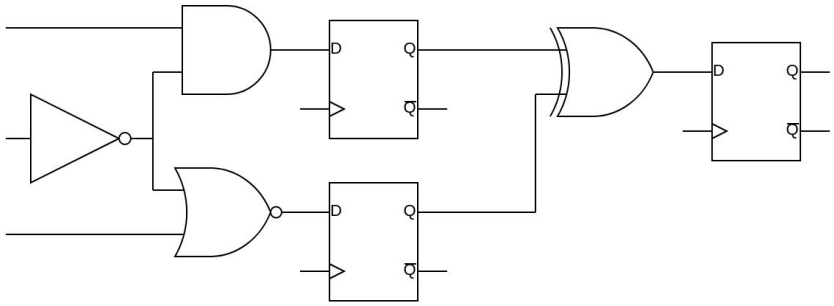
Lenguaje de programación

- Describe una secuencia de instrucciones
- Secuencialidad
  - El orden es importante!

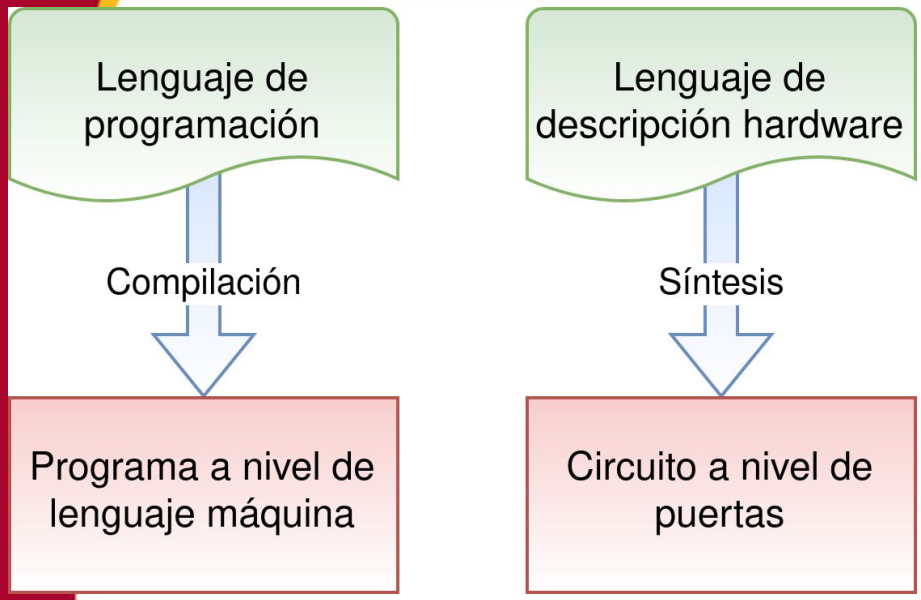
```
load 3, R0
load 4, R1
mul R0, 3
add R1, 3
mul R0, R1
jnz R2 ...
```

Lenguaje de descripción hardware

- Describe un circuito
- Concurrencia
  - Todo ocurre a la vez!



HDL: Programa vs diseño



- La sintaxis es muy similar
  - VHDL se parece a Ada
  - Verilog se parece a C
- En VHDL no se programa, se **describe**
- Hay que pensar siempre que la descripción se corresponde con **circuitos** funcionando en paralelo

¿Qué es VHDL?

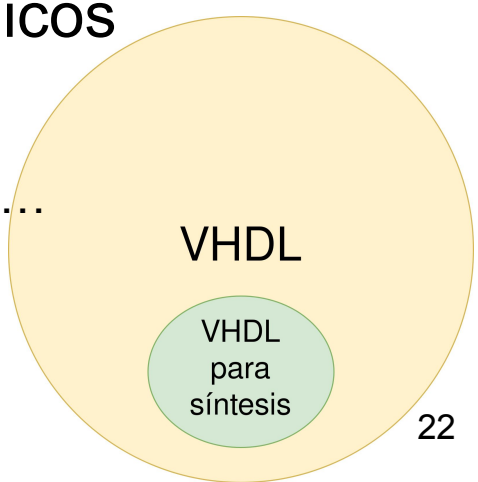
- VHDL es un **Lenguaje de Descripción Hardware**
- VHDL son las siglas de “VHSIC Hardware Description Language”
  - VHSIC son, a su vez, las siglas de “Very High Speed Integrated Circuits”
- Su sintaxis está basada en la de Ada
- Es un estándar de IEEE
  - Tiene distintas versiones: 1987, **1993**, 2000, 2002, 2007, **2008**, 2019
  - En general hay bastante compatibilidad hacia atrás entre versiones, sobre todo en código sintetizable

Generalidades de VHDL

- Es insensible a mayúsculas / minúsculas
  - En inglés: ‘case-insensitive’
  - **`MYSIGNAL`** es el mismo objeto que **`MySignal`** o **`mYsIgNaL`**
- Es insensible a espacios en blanco
  - En inglés: ‘whitespace insensitive’
  - Puedes cortar las líneas por donde quieras mientras no rompas un keyword en dos ni fusiones dos keywords
- Sólo tiene comentarios de línea, precedidos por dos guiones
  - `-- Esto es un comentario`
- Es de tipado duro
  - En inglés: ‘strongly typed’
  - No puedes realizar una asignación si lo que hay a ambos lados no es EXACTAMENTE del mismo tipo de dato
  - `a <= b;  -- a y b deben ser del mismo tipo`

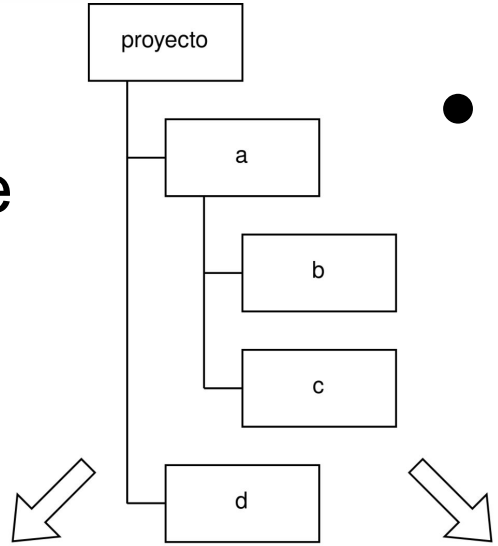
Generalidades de VHDL

- El VHDL completo es complejo, pero:
  - El subconjunto necesario para síntesis es muy pequeño
  - Sólo se utilizan en síntesis:
    - Asignaciones
      - `estado_siguiente <= reposo;  -- para señales`
      - `my_var := my_value;  -- para variables`
    - Comparaciones y operadores lógicos
      - **=** (igual), **/=** (distinto), **>** (mayor que), **<=** (menor o igual que), etc…
      - **and**, **xor**, **or**, **nand**, **nor**, **xnor**, **not**, etc …
    - Sentencias **if**
    - Sentencias **case**
    - Sentencias **process**



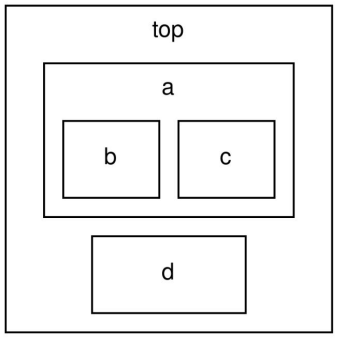
Unidades mínimas de funcionamiento

- En software, la unidad mínima de funcionalidad es la función



- En VHDL, la unidad mínima de funcionalidad es la entidad

```
main()
{
    function_a();
    function_d();
}

function_a()
{
    function_b();
    function_c();
}
```



Estructura de un fichero VHDL

```
LIBRARY nombre_librería;
USE librería.paquete.all;
```

Inclusión de librerías y paquetes (código que reutilizamos)

```
ENTITY nombre_entity IS
  GENERIC(.........);
  PORT(..............);
END nombre_entity;
```

Definición de la entidad (descripción de 'caja negra')

```
ARCHITECTURE nombre_architecture OF nombre_entity IS
  --Declaración de elementos que vayamos a necesitar
BEGIN
  --Descripción de la funcionalidad
END nombre_architecture;
```

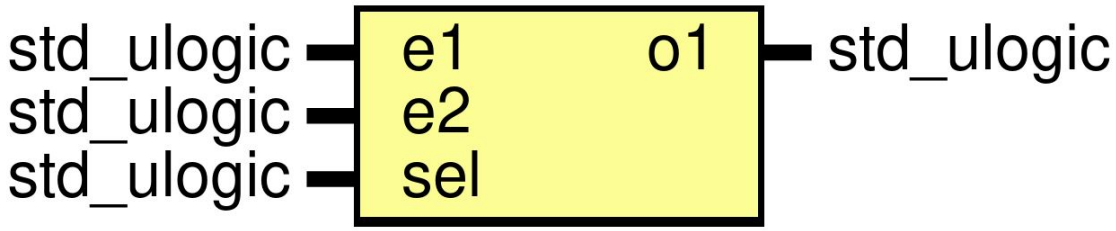
Descripción de la arquitectura ('cómo es por dentro')

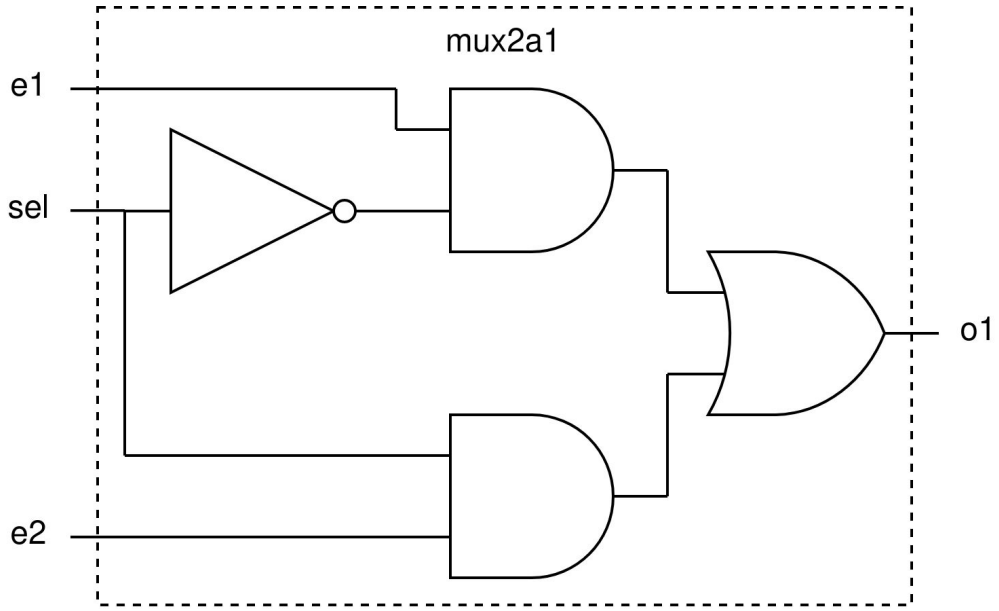
```
CONFIGURATION nombre_configuracion OF nombre entidad IS
FOR nombre_arquitectura
--Cuerpo de la configuración
END nombre_configuracion;
```

Selección de las configuraciones (opcional, avanzado)

Ejemplo básico

```vhdl
-- Multiplexor de dos entradas
library ieee;
use ieee.std_logic_1164.all;

entity mux2a1 is
  port (
    e1 : in  std_ulogic;
    e2 : in  std_ulogic;
    sel: in  std_ulogic;
    o1 : out std_ulogic
  );
end mux2a1;

architecture mux2a1_arch of mux2a1 is

begin

  process(e1,e2, sel)
  begin
    if (sel='0') then
      o1 <= e1;
    else
      o1 <= e2;
    end if;
  end process;

end mux2a1_arch;
```





La sección library

Library

Inclusión de librerías y paquetes con:

Tipos de datos, funciones, componentes, ...

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

Sección library

Ejemplo:

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

Sintaxis:

```
library lib_name;
use lib_name.package_name.all;
```

Sección library

Paquetes a utilizar:

```
ieee.std_logic_1164.all;
```

Siempre lo incluiremos

```
ieee.numeric_std.all;
```

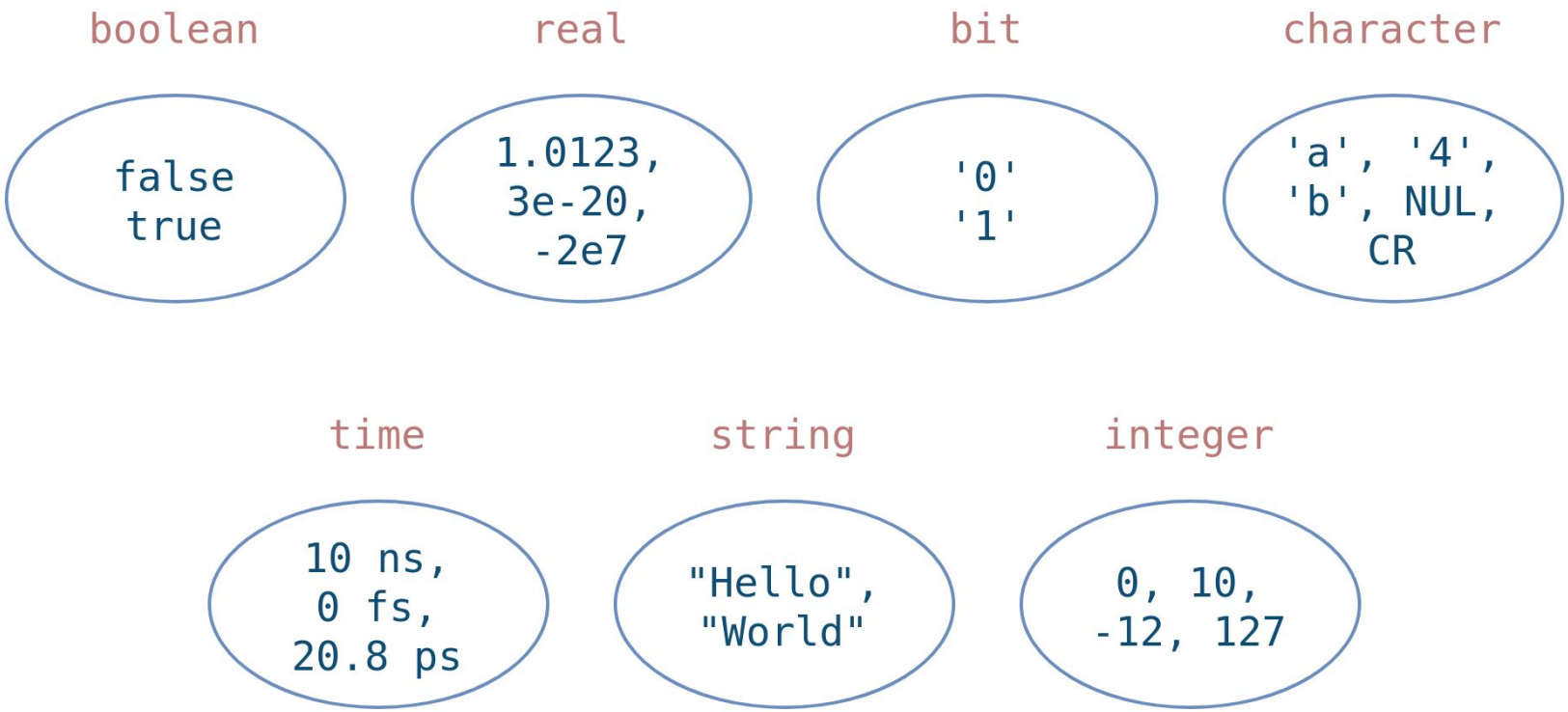
Lo incluiremos si necesitamos realizar operaciones aritméticas con vectores

Hablemos de tipos de datos

- Toda señal, variable, constante, expresión, etc., tiene un tipo de dato
- VHDL es de tipado duro (strongly typed)
  - En una asignación, lo que aparece a ambos lados debe ser del mismo tipo de dato
  - `a <= b; -- a y b deben ser EXACTAMENTE del mismo tipo`
  - Si a y b son –por ejemplo– vectores de 4 bits pero de distinto tipo (por ejemplo uno tiene signo y el otro no), no puede hacerse: la herramienta nos da un error
- Los tipos de datos son **muy** importantes
  - En cualquier lenguaje, pero VHDL *nos obliga* a tenerlos siempre en cuenta

Tipos de datos estándar

- Definidos en el propio lenguaje VHDL (estándar de IEEE)
  - Un objeto con un tipo de dato específico sólo puede adoptar unos valores determinados, dependiendo del tipo de dato concreto
  - El usuario puede definir sus propios tipos



Subtipos: tipo + restricción

- Un subtipo es compatible con su tipo base
  - `my_integer <= my_positive;  -- Legal, porque son del mismo tipo`
- Un subtipo puede utilizarse como operando en las operaciones definidas para su tipo base
- Algunos subtipos definidos en el estándar:

```
subtype NATURAL is INTEGER range 0 to INTEGER'HIGH;
subtype POSITIVE is INTEGER range 1 to INTEGER'HIGH;
type STRING is array (POSITIVE range <>) of CHARACTER;
```

El problema del tipo `bit`

- A pesar de que está definido en el estándar, es insuficiente para modelar el comportamiento de circuitos de mínima complejidad
- Sólo puede tomar los valores ‘0’ y ‘1’
- ¿Cómo modelamos un registro que no se ha inicializado? ¿Un valor lógico desconocido? ¿Alta impedancia?

std_ulogic al rescate

- El tipo **std_ulogic** está definido en el paquete **ieee.std_logic_1164**
- Un **std_ulogic** puede tomar 9 valores distintos:

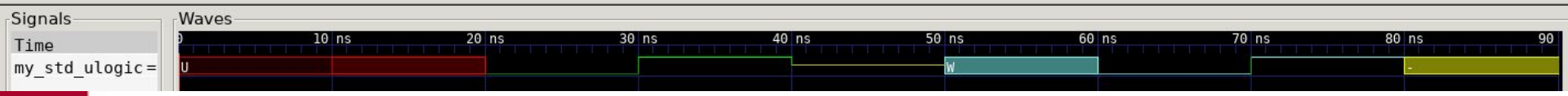
```
TYPE std_ulogic IS ( 'U',  -- Uninitialized
                     'X',  -- Forcing  Unknown
                     '0',  -- Forcing  0
                     '1',  -- Forcing  1
                     'Z',  -- High Impedance
                     'W',  -- Weak     Unknown
                     'L',  -- Weak     0
                     'H',  -- Weak     1
                     '-'   -- Don't care
                   );
```

Sección `library` y tipos de datos

`std_ulogic` explicado

	Fuerte		Débil		Especiales	
Bajo	'0'	Cero lógico fuerte	'L'	Cero lógico débil (Low = pulldown)	'Z'	Alta impedancia (High impedance)
Alto	'1'	Uno lógico fuerte	'H'	Uno lógico débil (High = pullup)	'U'	Sin inicializar (Uninitialized)
Desconocido	'X'	Valor fuerte desconocido (Unknown)	'W'	Valor débil desconocido (Weak)	'-'	No importa (Don't care)

¿Por qué tantos?, ¿para qué usamos cada uno? ¿cuándo tiene sentido utilizarlos?



‘U’ ‘X’ ‘0’ ‘1’ ‘Z’ ‘W’ ‘L’ ‘H’ ‘-’

Sección `library` y tipos de datos

`std_ulogic` explicado

- ‘0’, ‘1’: Uso normal
- ‘Z’: Cuando necesitamos poner algo en alta impedancia (buses compartidos, normalmente sólo se puede poner en los pines de la FPGA, que es el único sitio donde suele haber puertas triestado)
- ‘U’: Nos avisa en simulación de que no hemos inicializado algo correctamente (ej: resets mal implementados)
- ‘X’: Nos avisa en simulación de cortocircuitos, o de operaciones realizadas sobre valores ‘U’
- ‘L’, ‘H’: Modelado de pulldowns/pullups en simulación
- ‘W’: Nos avisa en simulación de cortocircuitos entre ‘L’ y ‘H’
- ‘-’: Puede usarse como ‘comodín’ al comparar vectores (`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)

La sección library típica

- Todos nuestros diseños comenzarán por:

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

- Y opcionalmente también escribiremos:

```
use ieee.numeric_std.all;
```

El paquete ieee.std_logic_1164

- Paquete básico de la librería IEEE
- Define los tipos de datos:
  - std_ulogic
  - std_ulogic_vector(MSB downto LSB)
  - std_logic
  - std_logic_vector(MSB downto LSB)
- Define operaciones binarias entre estos tipos de datos (and, nand, or, nor, xor, xnor, not)

¿Y qué es std_logic?

```
SUBTYPE std_logic IS resolved std_ulogic;
[...]
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' )  -- | - |
    );
```

¿Debería usar std_logic o std_ulogic?

- Si bien los fabricantes suelen 'ayudar' generando código e interfaces en los que todo es `std_logic` y `std_logic_vector`, siempre que podamos es preferible utilizar `std_ulogic` y `std_ulogic_vector` en su lugar, porque así nos avisa de los cortocircuitos también en simulación
- (Más información en el tema de VHDL Avanzado)

El paquete ieee.numeric_std

- Introduce buses con significado numérico: signed y unsigned
- Ejemplos:
  - `my_slv : std_ulogic_vector(3 downto 0) := “1001”;`
    - Representa un grupo de bits sin significado numérico
  - `my_signed : signed(3 downto 0) := “1001”;`
    - Representa un entero con signo en complemento a dos (“1001” = -7)
  - `my_unsigned : unsigned(3 downto 0) := “1001”;`
    - Representa un entero sin signo (“1001” = 9)

Operadores en VHDL y std_logic_1164

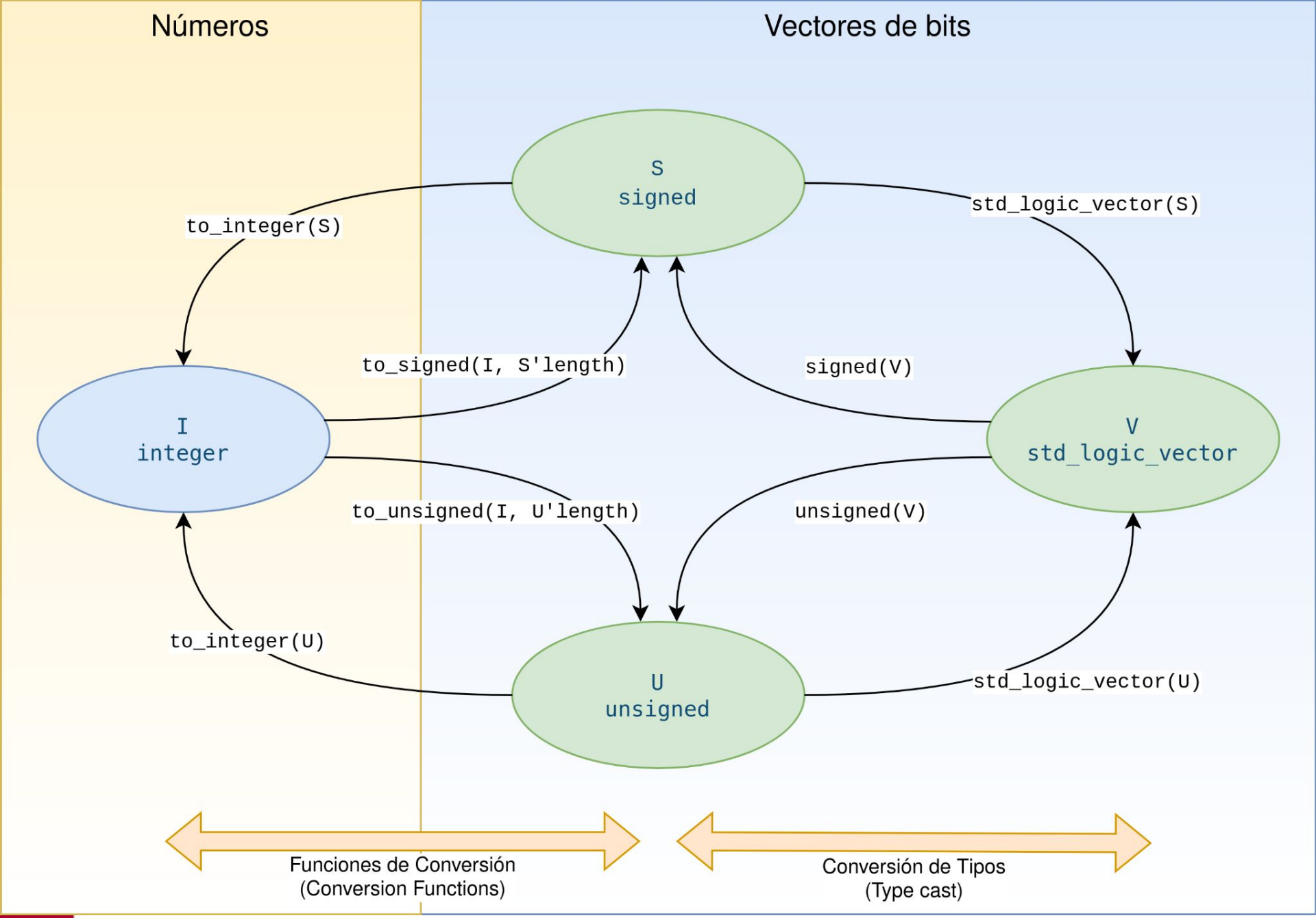
Operador	Descripción	Tipo de dato de operandos	Tipo de dato de resultados
a + b	Suma	integer	integer
a - b	Resta		
a * b	Multiplicación		
a / b	División		
a ** b	Exponenciación		
a & b	Concatenación	1-d array	1-d array
a = b	Igual	cualquiera	cualquiera
a /= b	Distinto		
a < b	Menor que	integer	integer
a <= b	Menor o igual que		
a > b	Mayor que		
a >= b	Mayor o igual que		
not a	Negación	boolean, bit, std_[u]logic, std_[u]logic_vector	mismo tipo que los operandos
a **and** b	And		
a **or** b	Or		
a **xor** b	Xor		

IEEE numeric_std: sobrecarga de operadores

Operador	Descripción	Tipo de dato de operandos	Tipo de dato de resultados
a + b	Operadores aritméticos	unsigned, natural, signed, integer	unsigned, signed
a - b			
a * b			
a / b			
a = b	Comparaciones	unsigned, natural, signed, integer	boolean
a /= b			
a < b			
a <= b			
a > b			
a >= b			

Conversiones entre vectores de bits y tipos numéricos

	Tipo de dato origen	Tipo de dato destino	Función de conversión
Conversión entre vectores de bits	unsigned, signed	std_logic_vector	std_logic_vector(a)
	signed, std_logic_vector	unsigned	unsigned(a)
	unsigned, std_logic_vector	signed	signed(b)
	unsigned, signed	integer	to_integer(a)
Conversión entre vectores de bits y enteros	natural	unsigned	to_unsigned(a,length)
	integer	signed	to_signed(a,length)



Traducido del [original de Shannon Hilbert]

Ejemplo de asignaciones y operaciones aritméticas

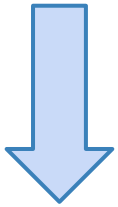
```
signal my_slv: std_logic_vector(3 downto 0);
signal my_uns: unsigned(3 downto 0);

my_unsigned <= my_slv; -- ERROR: type mismatch
my_slv <= my_unsigned; -- ERROR: type mismatch
my_unsigned <= 3;      -- ERROR: type mismatch
my_slv <= 3;           -- ERROR: type mismatch
```

Ejemplo de asignaciones y operaciones aritméticas

```
signal my_slv: std_logic_vector(3 downto 0);
signal my_uns: unsigned(3 downto 0);

my_unsigned <= unsigned(my_slv);                    -- OK!
my_slv <= std_logic_vector(my_unsigned);            -- OK!
my_unsigned <= to_unsigned(3);                      -- OK!
my_slv <= std_logic_vector(to_unsigned(3,4));       -- OK!
```

Ejemplo de asignaciones y operaciones aritméticas

```
signal slv1, slv2, slv3: std_logic_vector(3 downto 0);
signal uns1, uns2, uns3: unsigned(3 downto 0);

uns1 <= uns2 + uns3; -- OK! Mismo tipo de datos
uns1 <= uns2 + 3;    -- OK! Sobrecarga de operadores

slv1 <= slv2 + slv3; -- ERROR: Para sumar hace falta un
slv1 <= slv2 + 1;    -- ERROR: significado numérico
```



Usamos funciones de conversión

```
slv1 <= std_logic_vector(unsigned(slv2) + unsigned(slv3)); -- OK!
slv1 <= std_logic_vector(unsigned(slv2) + 3);              -- OK!
```

La sección entity

Entity

- Descripción de 'caja negra': entradas, salidas y parámetros (generics)
- Contiene las subsecciones **generic** y **port**

```
entity contador is
  generic (Nbit : integer := 8);
  port (clk    : in  std_ulogic;
        rst    : in  std_ulogic;
        enable : in  std_ulogic;
        Q      : out std_ulogic_vector ((Nbit - 1) downto 0)
       );
end contador;
```

Sintaxis de la sección entity

Sintaxis:

Direction debe ser in, out o bidir

```
entity entity_name is
  Generic (gen_name : data_type := default_value;
           <another generic>;
           <last port doesn't have separating ;>
          );
  Port ( port_name  : direction data_type;
         <another port>;
         <last port doesn't have separating ;>
       );
end entity_name;
```

Declaración de generics

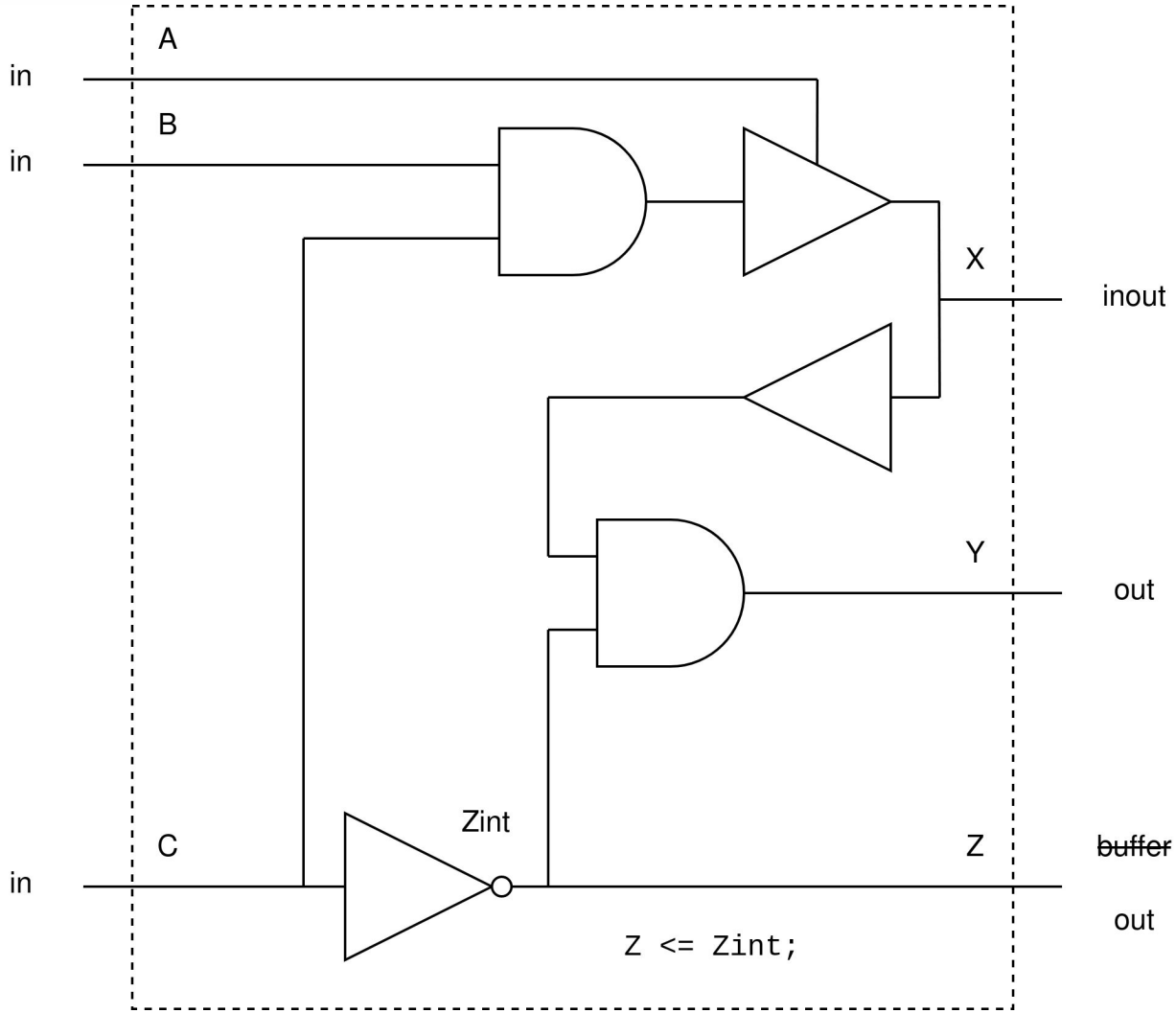
- Un generic es un parámetro
- Sirve para hacer diseños parametrizables
  - Por ejemplo, podemos tener un contador de N bits que instanciaremos en 4, 8, 16 y 32 bits en lugar de tener que describir cuatro contadores diferentes, cada uno con un tamaño
- Un generic tiene:
  - Tipo de dato
  - Valor por defecto (opcional)
    - Es el valor que tomará si no se le asigna ninguno al instanciarlo

Declaración de ports

- Un port es un puerto de la entidad
- Sirve para declarar los pines de entrada y salida que nos permite conectar unas entidades con otras
- Un port tiene:
  - Nombre
  - Dirección
    - in / out / inout / ~~buffer~~
  - Tipo de dato

Sección `entity`

Ejemplo



La sección architecture

```
architecture mi_entidad_arch of mi_entidad is

    -- Definición de tipos de datos
    -- Declaración de señales (signal)
    -- Declaración de componentes (component)

begin

    -- Sentencias concurrentes
    -- Procesos (process)
    -- Instancias de componentes y entidades

end mi_entidad_arch;
```

Declaración de tipos de datos

- VHDL nos permite declarar nuevos tipos de datos
- `type nombre_datatype is <definición del tipo de dato>;`
- El uso más típico son los tipos enumerados
  - Muy útiles para hacer máquinas de estados
- Pero también se pueden definir arrays, records (análogos a structs en C), y otros

Declaración de tipos de datos

Ejemplos:

```
type dia_mes_t is integer range 0 to 31;  -- Entero con restricción de valores
type estado_t is (fetch, decode, execute, writeback);  -- Tipo enumerado
type ram_type is array (0 to DEPTH-1) of std_logic_vector (WIDTH-1 downto 0);  -- Array
type std_logic_vector is array ( natural range <> ) of std_logic; -- En el paquete std_logic_1164
type UNSIGNED is array ( NATURAL range <> ) of STD_LOGIC;  -- En el paquete numeric_std
type SIGNED is array ( NATURAL range <> ) of STD_LOGIC;  -- En el paquete numeric_std
```

Declaración de señales

- `signal my_signal_name: datatype := initial_value;`
- Valor inicial no siempre soportado en síntesis
  - Y si no sabemos bien qué estamos haciendo, realmente puede no ser buena idea usarlo para síntesis: nosotros lo que tenemos que definir son los valores de reset de los biestables y asegurarnos de que activamos la señal de reset
  - En síntesis suele ser implementado usando los valores de INIT de los biestables (no confundir con los valores de reset)
- El valor inicial sí está soportado en simulación
  - Esto es útil en los testbenches
- Importante! `signal` es diferente de `variable`!
  - Las variables las veremos más adelante

Declaración de señales

Ejemplos:

```
signal my_std_ulogic: std_ulogic;
signal my_std_logic : std_logic;
signal my_slv       : std_logic_vector(7 downto 0);   -- 8 bits
signal my_unsigned  : unsigned(7 downto 0);           -- 8 bits
signal my_signed    : signed(7 downto 0);             -- 8 bits
signal my_integer   : integer range 0 to 255;         -- 8 bits
```

Declaración de componentes

- Si queremos reutilizar en un fichero VHDL una entidad que ya tengamos escrita en otro fichero, una forma es instanciarla como componente
- Para ello, es obligatorio declararla como componente en la arquitectura, antes del begin

Declaración de componentes

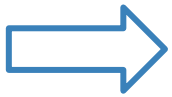
```
component nombre
    generic (
        -- declaración de generics
    );
    port (
        -- declaración de puertos
    );
end component;
```

- Las secciones **generic** y **port** son literalmente copia de lo que tengamos escrito en el **entity**

Declaración de componentes

Ejemplo:

```
component mux2a1
    port (
        e1 : in  std_logic;
        e2 : in  std_logic;
        sel: in  std_logic;
        o1 : out std_logic
    );
end component;
```



Copia del entity!

Declaración de componentes

- ¿Cuándo la usamos?
  - Si la entidad **está descrita en VHDL**:
    - No suele merecer la pena declararla e instanciarla como componente, ya que:
    - Se puede **instanciar directamente como entidad**
      - Escribimos menos y sobre todo evitamos duplicidad de código
  - Si la entidad **no está descrita en VHDL**:
    - Por ejemplo porque esté en otro lenguaje (Verilog), porque sea una primitiva de la FPGA o un IP core de terceros en formato no-VHDL
    - Es **obligatorio** declararla e instanciarla como componente

Después del begin

- Aquí es donde describimos todo el funcionamiento del circuito (señales conmutando, cálculos, toma de decisiones, instanciación de entidades y componentes, etc)
- Lo veremos en los siguientes temas

```
begin
    -- Sentencias concurrentes
    -- Procesos (process)
    -- Instancias de componentes y entidades
end mi_entidad_arch;
```

La sección configuration

- Es un mecanismo para asignar entidades a instancias de componentes
- En ausencia de esta sección, las instancias de componentes usan una entidad que exista con el mismo nombre y mismos puertos que el componente
- En la gran mayoría de ficheros VHDL esta sección no se utiliza
- Las herramientas de síntesis normalmente no soportan esta sección
- En simulación sí ve algo más de uso

Sección configuration

Sintaxis

```
configuration nombre_configuracion of nombre_entidad is
    -- declaraciones ...
    for nombre_arquitectura
        for nombre_instancia : nombre_componente
            use entity work.nombre_entity;
        end for;
        for nombre_instancia2 : nombre_componente2
            use configuration work.nombre_configuracion;
        end for;
        -- puede haber otros for para otras instancias ...
    end for;
end nombre_configuracion;
```

- Se pueden definir varias configuraciones diferentes con distintos nombres, de forma que en el nivel jerárquico superior se decida cuál de ellas se utiliza
- Puede haber una jerarquía de configuraciones (`use configuration` …)

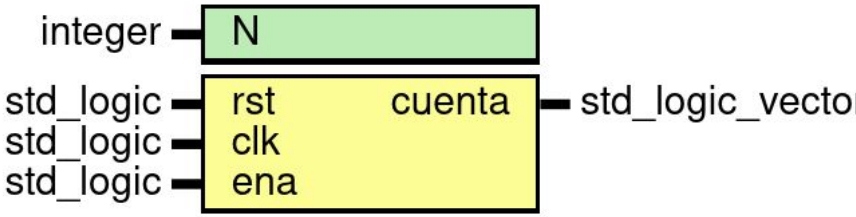
Tarea

Describir la sección entity de los siguientes bloques:

Entity: contador

- **File**: contador.vhd

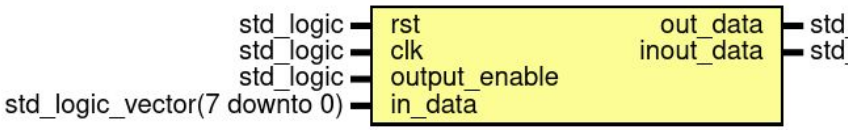
Diagram



Entity: buffer_bus_bidireccional

- **File**: buffer_bus_bidireccional.vhd

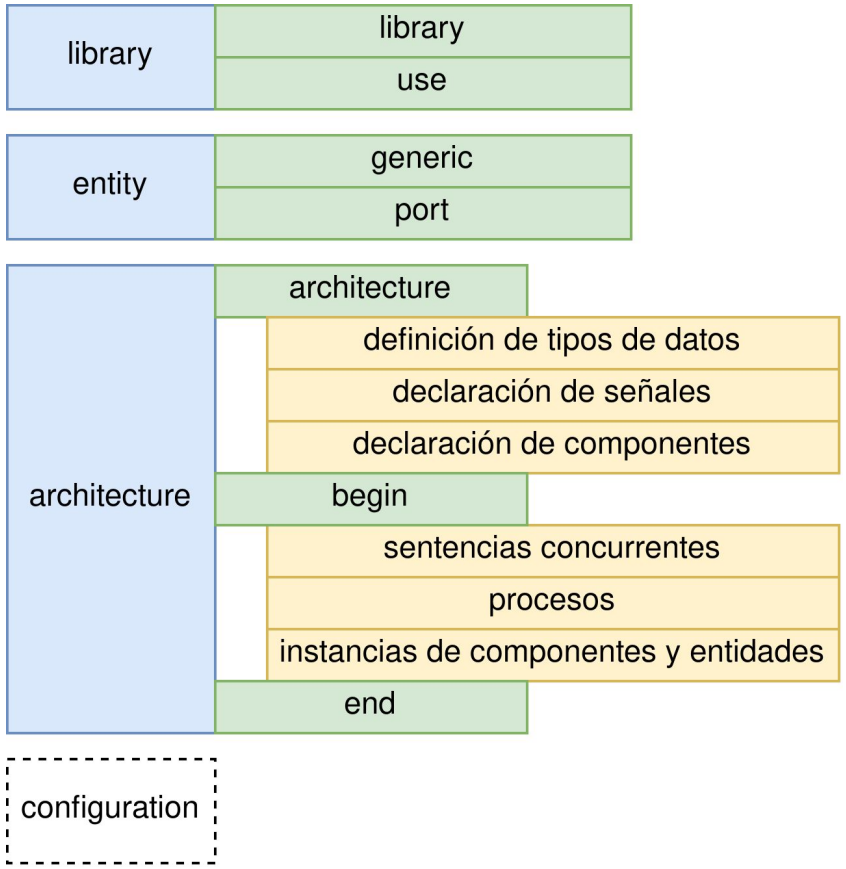
Diagram



(`inout_data` debe ser de dirección `inout`)

Resumen

- Library
  - Inclusión de librerías y paquetes
- Entity
  - Descripción de caja negra
- Architecture
  - Descripción de la funcionalidad
- Configuration
  - Normalmente no se utiliza



Conclusiones

- Las **FPGAs** son **dispositivos programables** con **múltiples aplicaciones** en ingeniería
- VHDL es un **lenguaje de descripción hardware** que puede utilizarse para trabajar con FPGAs
- El lenguaje es **complejo**, pero el subconjunto necesario para síntesis es **pequeño**
- La unidad mínima de funcionalidad en VHDL es la **entidad**
- Un fichero VHDL contiene normalmente **3 secciones**: `library`, `entity` y `architecture`
- De estas tres secciones, todo salvo lo que va tras el begin de la arquitectura es **estático** y relativamente sencillo de entender

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

- Conocer, de manera descriptiva, qué es una FPGA, en qué se diferencia de un microprocesador y qué posibles aplicaciones puede tener
- Comprender las diferencias existentes entre un lenguaje de programación y un lenguaje de descripción hardware
- Conocer, de manera descriptiva, en qué consisten los procesos de síntesis e implementación

Resultados de aprendizaje

- Conocer las secciones de un fichero VHDL y para qué se utiliza cada una
- Ser capaz de localizar en un fichero VHDL las distintas secciones, comprendiendo las secciones `library` y `entity`
- Saber que VHDL es un lenguaje de tipado duro y comprender qué implica esto
- Saber determinar los tipos de datos que intervienen en una asignación que contenga una expresión sencilla en VHDL, conociendo los tipos de datos de los objetos que intervienen en ella, y ser capaz de determinar si dicha sentencia es VHDL válido o no