

# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

VHDL

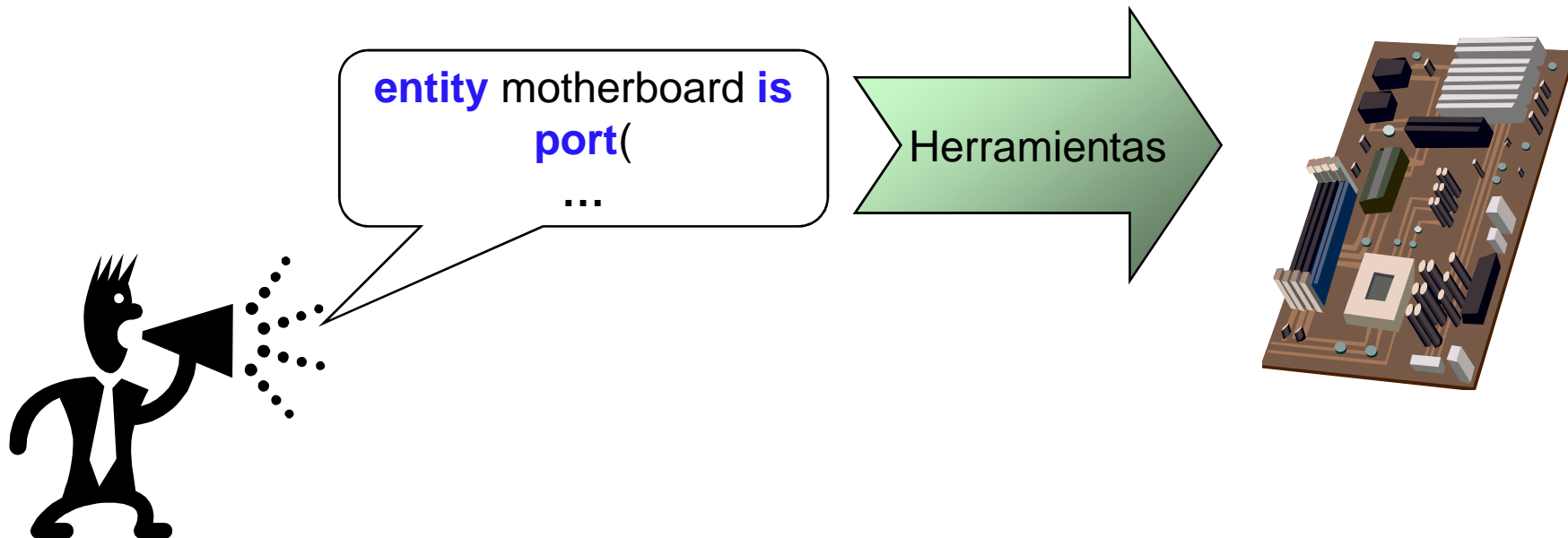
Ing. Pablo Cayuela  
[pablo.cayuela@gmail.com](mailto:pablo.cayuela@gmail.com)  
Ing. Federico Paredes  
[fedesor@gmail.com](mailto:fedesor@gmail.com)

# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

¿Qué es VHDL?

VHDL es el acrónimo de “Very High Speed Integrated Circuit Hardware Description Language”.

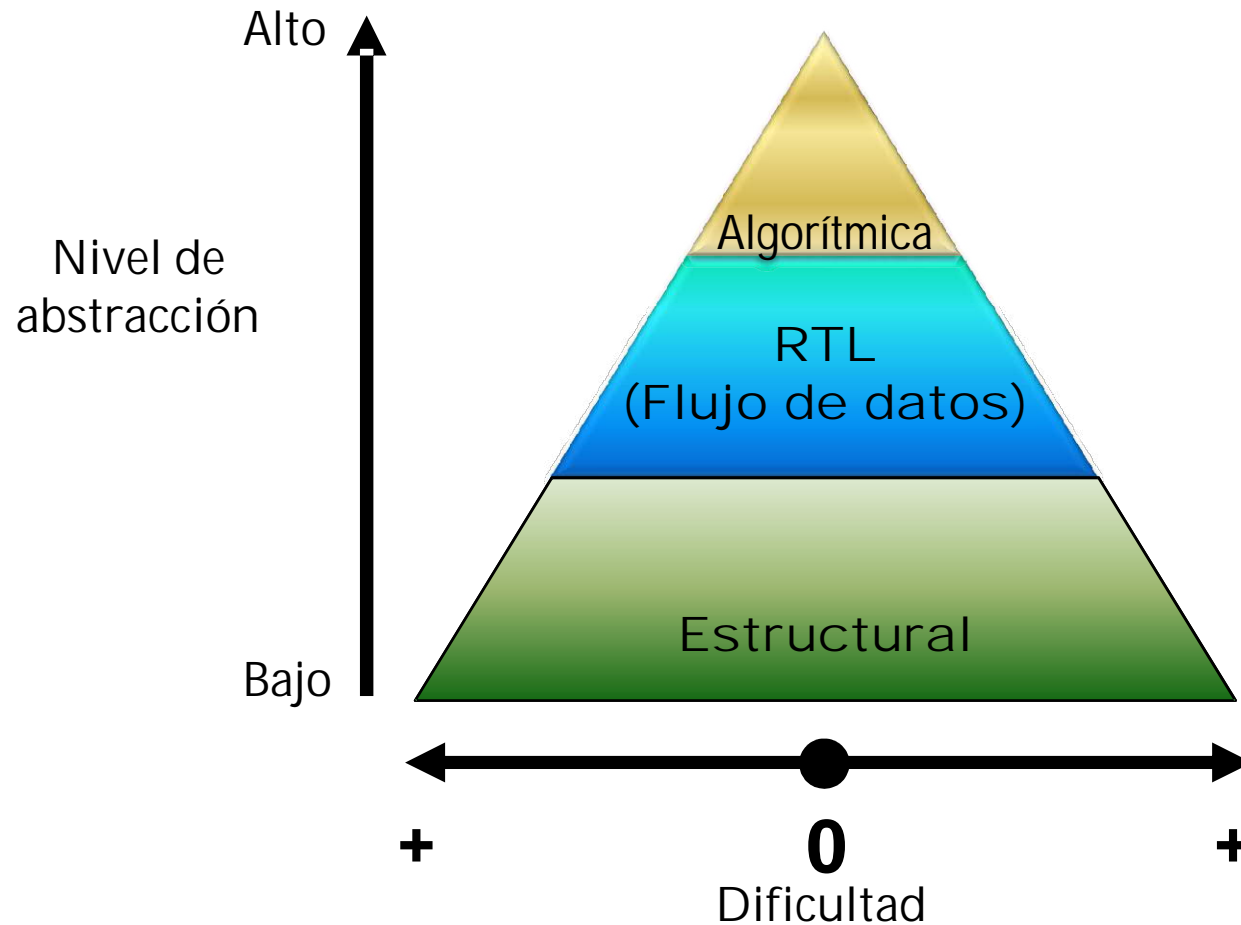
Es un lenguaje desarrollado a principios de los '80 por el departamento de defensa de los EE.UU. para la documentación, modelado y simulación de circuitos digitales. Con el tiempo, también adquirió importancia en la síntesis de estos circuitos, siendo hoy una de sus principales aplicaciones.



# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

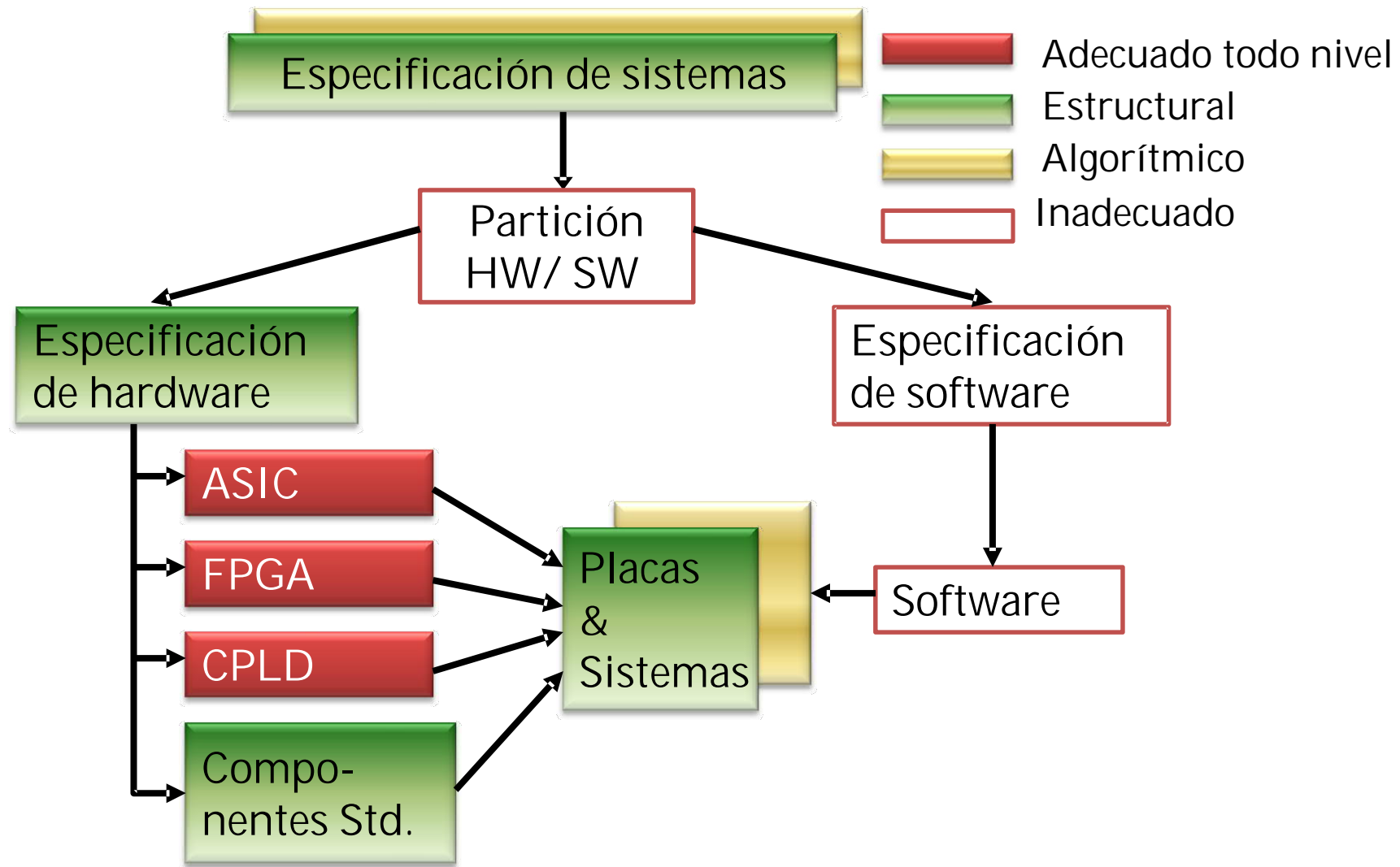
Sumergiéndose en VHDL

Las distintas descripciones poseen niveles de abstracción y dificultad diferentes



# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

Áreas de aplicación de VHDL



# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

---

Algunas características importantes de VHDL

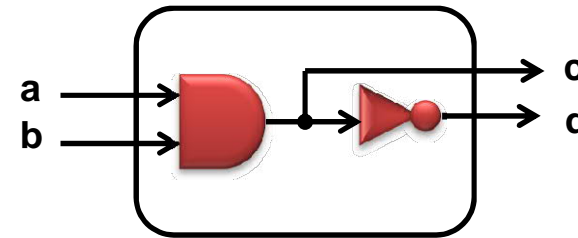
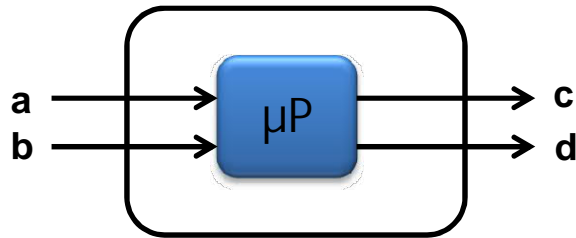
- VHDL es un lenguaje concurrente, sus sentencias se ejecutan en paralelo.
- Es un lenguaje extenso con más de 100 palabras reservadas.
- Posibilita la integración de diferentes herramientas de CAD.
- Posee una sintaxis amplia y flexible, permitiendo distintos tipos de descripción.
- Permite dividir un diseño en unidades más pequeñas, incrementando la modularidad de éste.
- Se encuentra estandarizado por el IEEE.
- Permite diseñar, modelar y simular un sistema desde altos niveles de abstracción hasta niveles muy cercanos al hardware final.

# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

Secuencia

versus

Concurrencia



```
void function f1(bool a, bool b)
{
    -- a=1 b=0 c=1
    c = a & b; -- c=0
    d = !c;    -- d=1
}
```

```
architecture interna of f1 is
begin
    c <= a and b; -- c=0
    d <= not c;   -- d=0
end;
```

```
void function f2(bool a, bool b)
{
    -- a=1 b=0 c=1
    d = !c;    -- d=0
    c = a & b; -- c=0
}
```

```
architecture interna of f2 is
begin
    d <= not c; -- d=0
    c <= a and b; -- c=0
end;
```

# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

---

## Importancia de las herramientas

### Flujos de diseño

- Especificación
- Simulación
- Implementación

### Herramientas

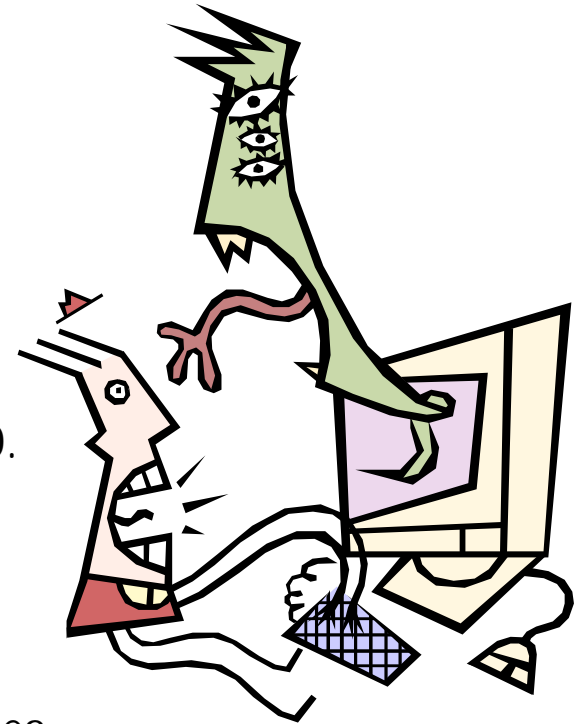
- Compilador
- Simulador
- Sintetizador

# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

---

Adelantos para ir perdiendo el miedo

- Todas las sentencias terminan en punto y coma ;
- Los comentarios comienzan con dos guiones --
- Todo identificador debe comenzar con una letra y contener solo letras, números y guiones bajos \_
- Los números se consideran por omisión en base 10.
- Pueden especificarse otras bases con el carácter numeral #. Ej.: 2#10110# (binario).
- Mayúsculas y minúsculas se consideran equivalentes.
- Las cadenas se definen mediante comillas dobles, los caracteres con comillas simples. Ej.: "cadena", 'a'.

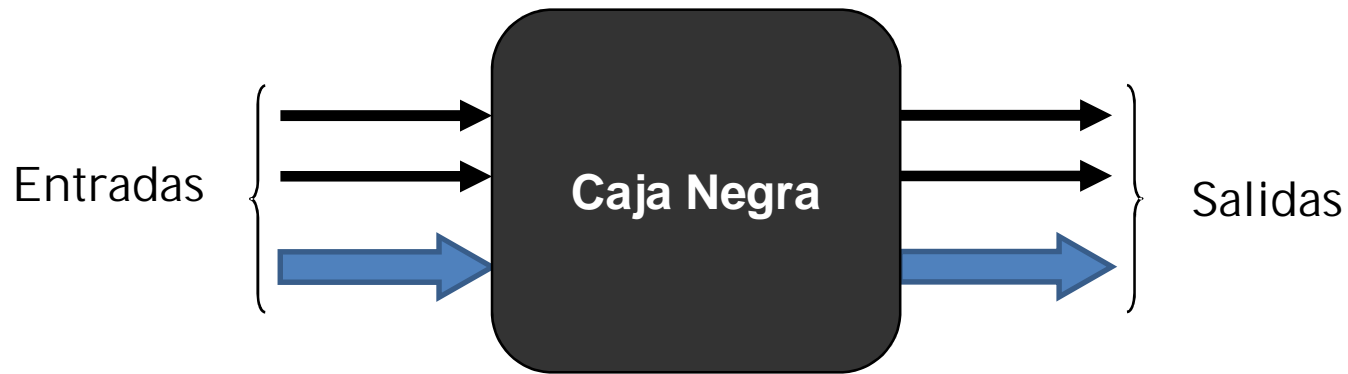




# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

---

Diseño de hardware: El concepto de caja negra



Todo diseño de hardware comienza como un bloque definido por sus entradas, sus salidas y la relación entre estas. En VHDL esa caja negra se denomina **ENTITY** (entidad).

# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

---

## La Entidad

Todo diseño en VHDL comienza con una entidad:

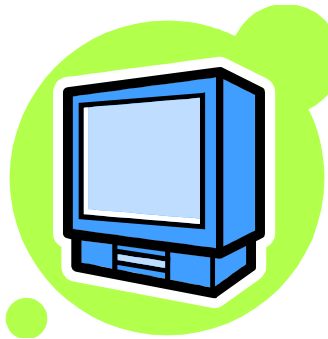


```
entity celular is  
...  
end entity celular;
```



```
entity procesador is  
...  
end entity procesador;
```

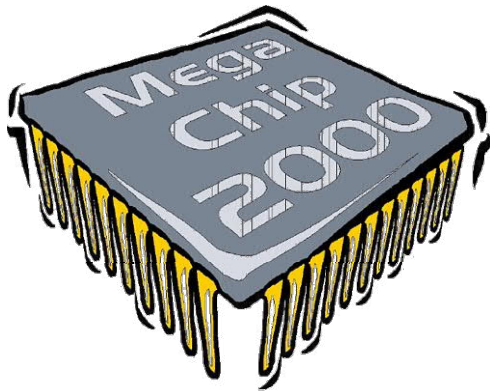
```
entity televisor is  
...  
end entity televisor;
```



# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

## La Entidad

En la sección de entidad se definen las entradas y salidas (puertos) del hardware. Además se declaran los parámetros y propiedades (genéricos) de este.



```
entity megachip is  
  generic(  
    fmax = 100 MHz  
  );  
  port(  
    entradas = 8 bit  
    salidas = 8 bit  
  );  
end entity megachip;
```

La palabra “**generic**” se utiliza para definir parámetros.

Dentro del bloque “**port**” declaramos los puertos de nuestra entidad.



Los puertos y genéricos se muestran sólo de manera figurada y no siguen las reglas del VHDL.

# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

---

## La Entidad

La entidad especifica cómo se conecta el bloque al mundo exterior y permite establecer los parámetros de conexión.

Resumiendo:

**ENTIDAD = INTERFAZ**

¡Sin entidad, no existe hardware!

# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

---

## La Arquitectura

Definida la entidad, el paso que sigue es definir el contenido de esta. Esto se realiza en un bloque llamado ARCHITECTURE (Arquitectura).

Descripción de  
arquitectura

**architecture** contenido **of** mi\_entidad **is**

[ sección declarativa ]

**begin**

[ sección ejecutiva: sentencias ]

**end architecture** contenido;



Una arquitectura siempre se asocia con una entidad determinada.

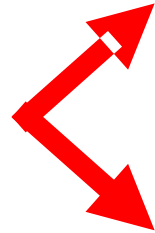
# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

---

## La Arquitectura

Una entidad puede tener muchas arquitecturas.

```
entity teléfono is  
...  
end entity teléfono;
```



```
architecture móvil of teléfono is  
...  
end architecture móvil;
```



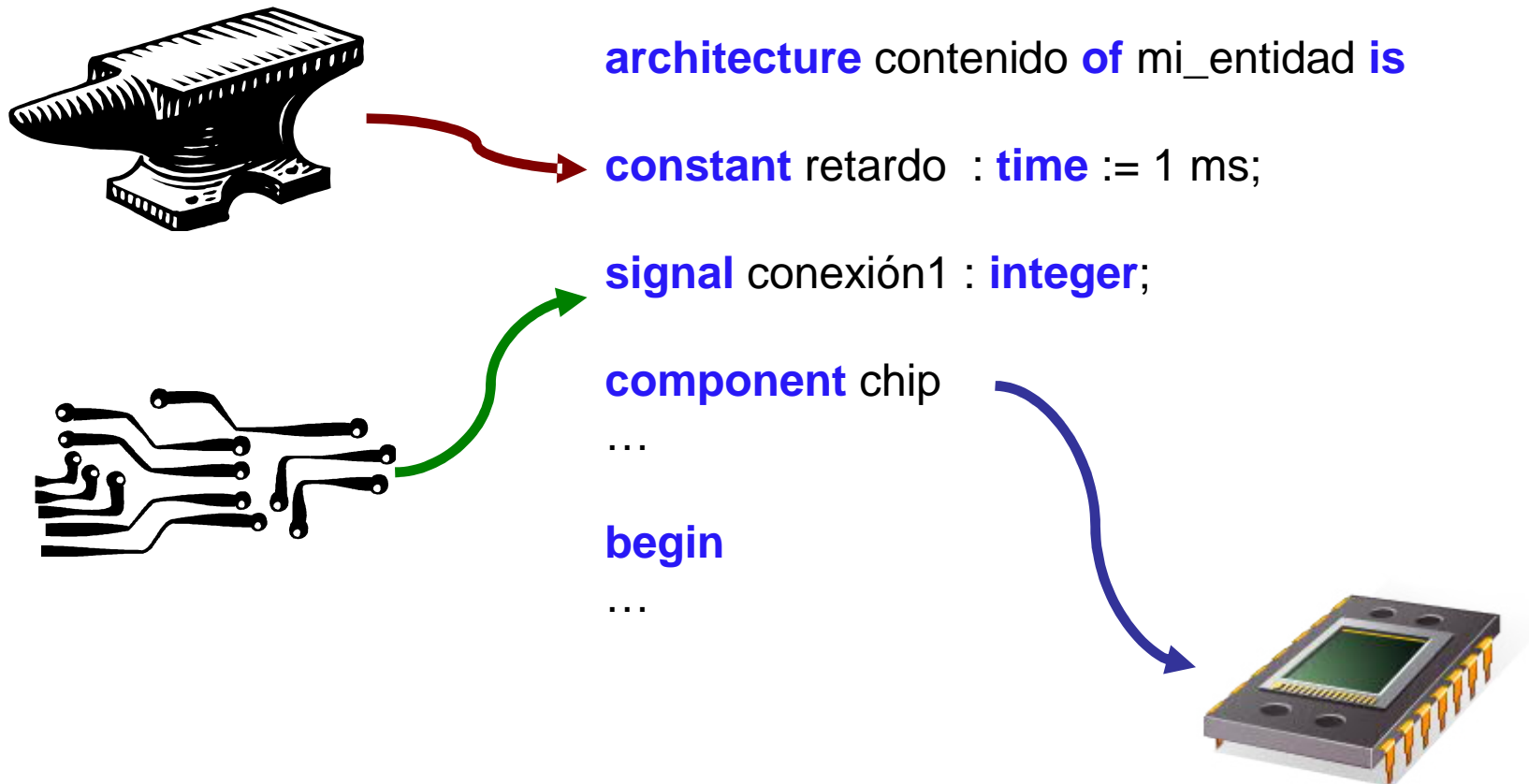
```
architecture fijo of teléfono is  
...  
end architecture fijo;
```



# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

## La Arquitectura

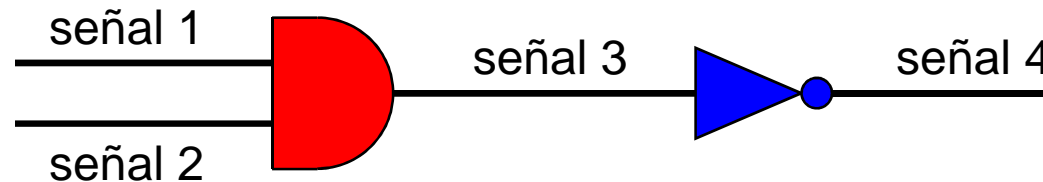
La porción declarativa de la arquitectura permite declarar señales, constantes y componentes entre otras cosas.



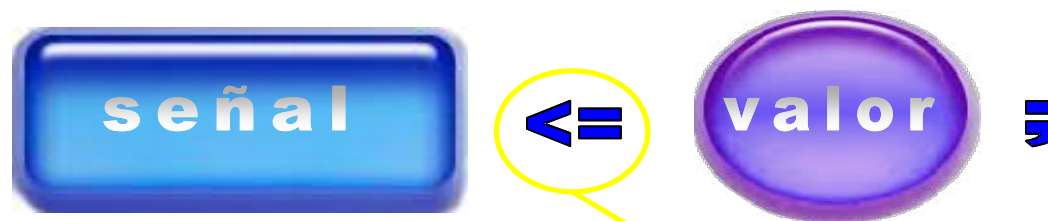
# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

## Señales

Una señal cumple en VHDL la misma función que una conexión en un circuito esquemático.



Además, se pueden considerar como el equivalente en VHDL de las variables en otros lenguajes de programación. Y al igual que éstas, pueden ser sujeto de asignaciones.



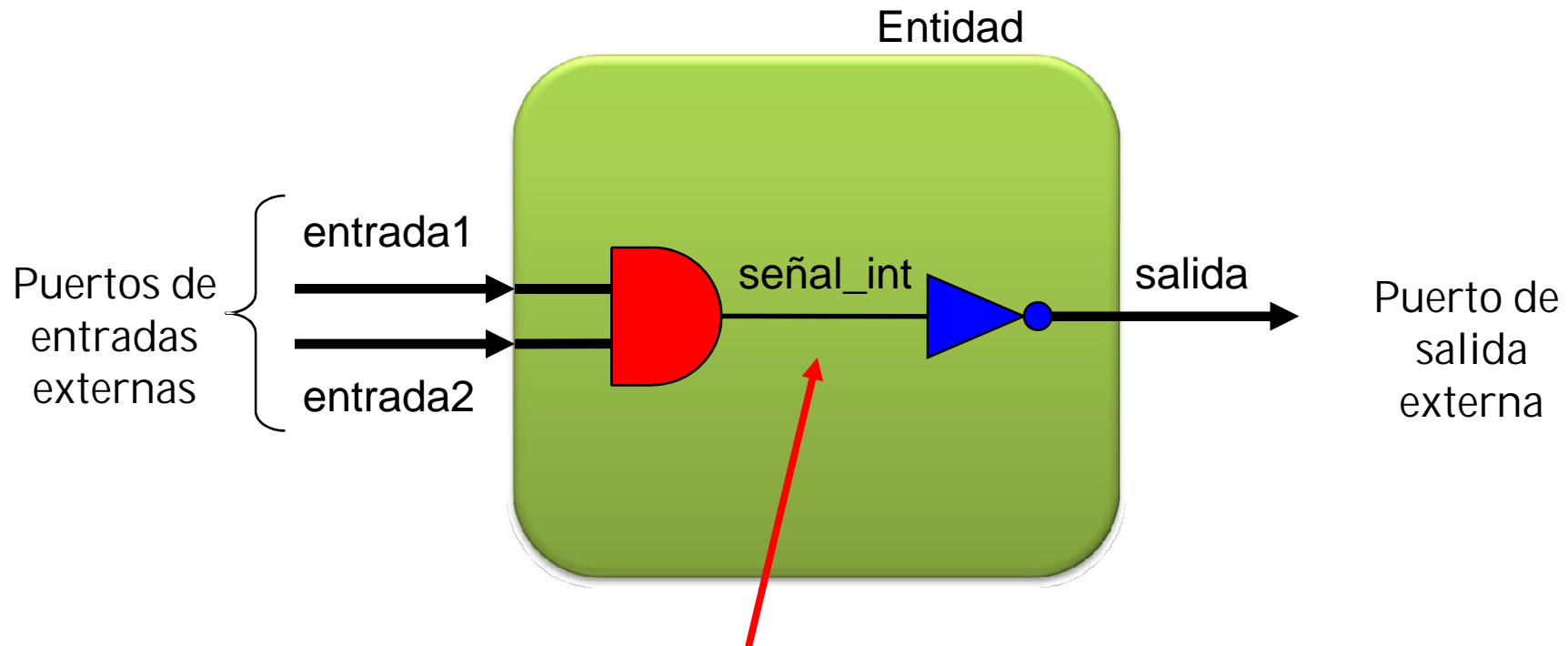
Operador de asignación para señales



# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

## Señales

Las señales se pueden clasificar en señales internas y señales externas (puertos).

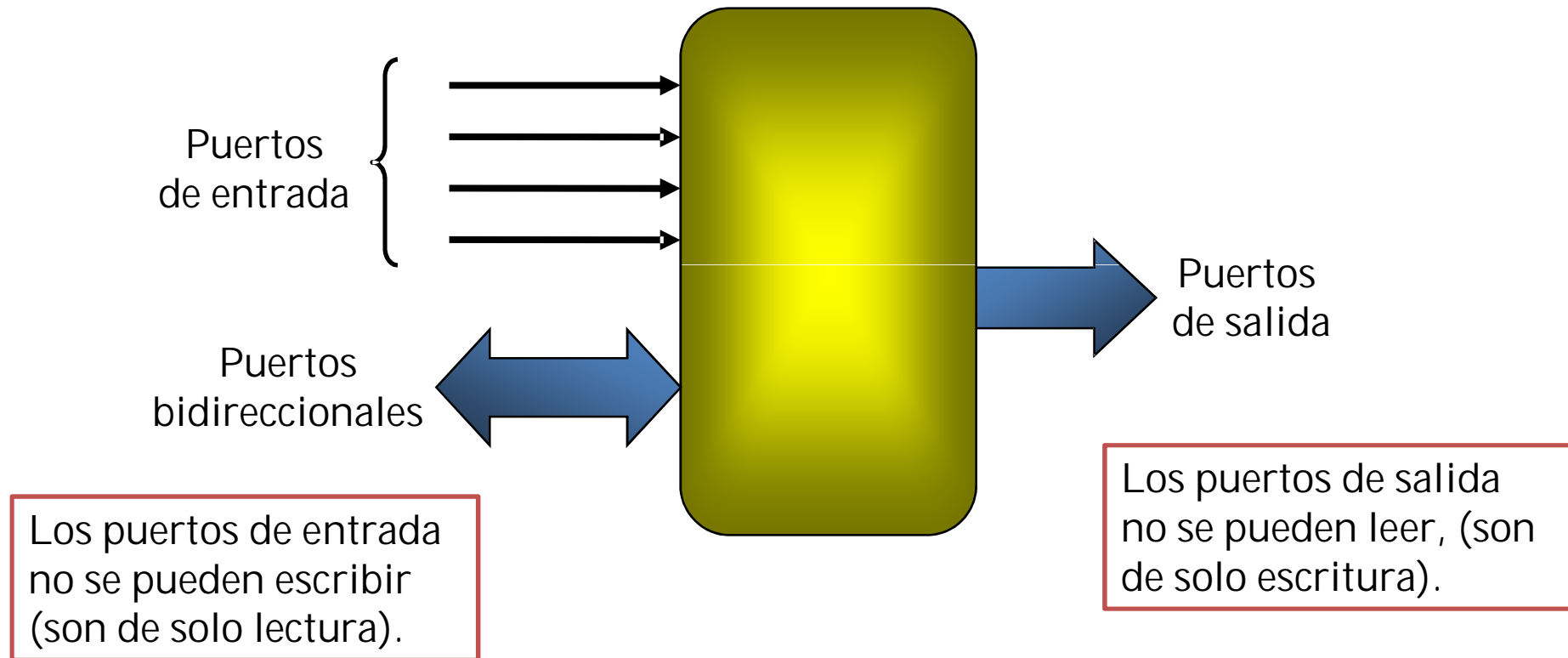


Las señales internas se pueden leer y escribir

# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

## Señales

Las señales externas pueden ser de entrada, de salida o bidireccionales.

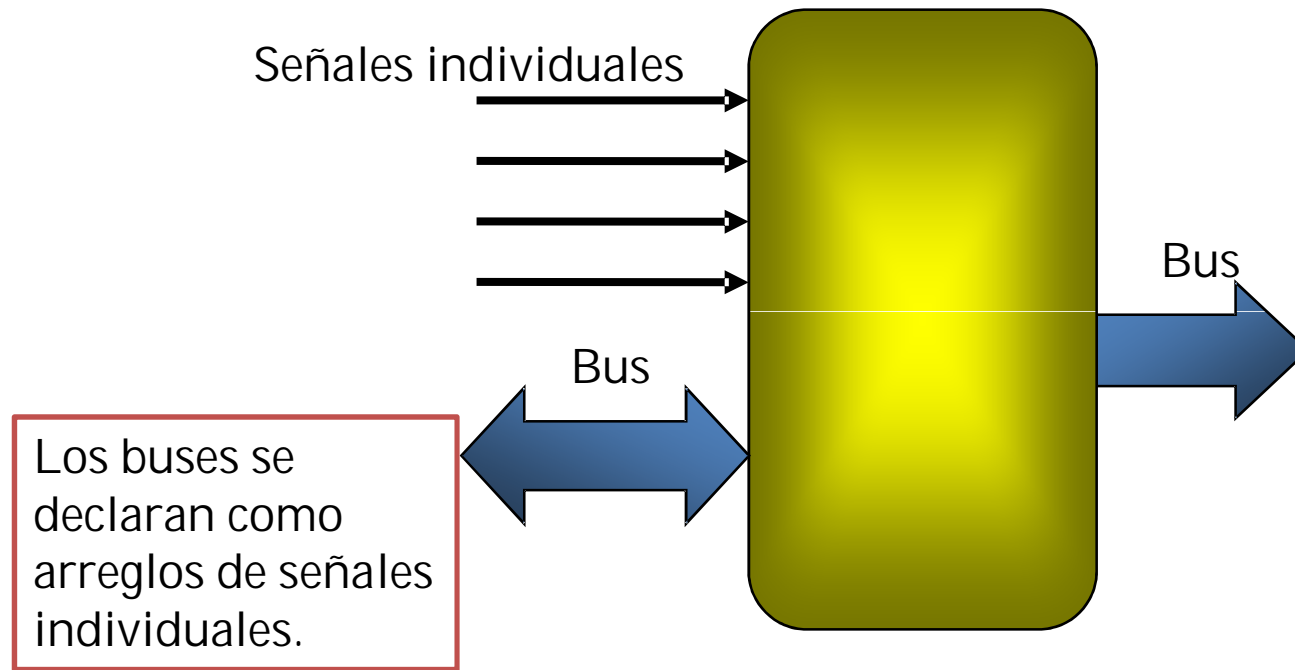


- Las señales de entrada se especifican como **in**
- Las señales de salida se especifican como **out**
- Las bidireccionales se especifican como **inout**

# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

## Señales

También se pueden clasificar en señales individuales y buses.

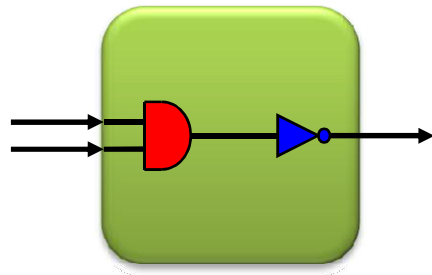


Por supuesto, varias señales individuales se pueden unir (concatenar) para formar un bus y éste también se puede descomponer en señales únicas.

# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

## Señales

Las señales externas se declaran en la entidad, entre los paréntesis de la sentencia **port** :



```
entity compuerta is
  port(
    entrada1 : in  bit;
    entrada2 : in  bit;
    salida   : out bit
  );
end entity compuerta;
```

Las señales internas se definen en la sección declarativa de la arquitectura:



```
architecture estructura of compuerta is
  signal señal_int : bit;
begin
```

```
  señal_int <= entrada1 and entrada2;
  salida    <= not señal_int;
```

```
end architecture compuerta;
```



# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

## Tipos de datos

Todas las señales deben poseer un tipo de dato definido.

Ejemplos de datos predefinidos en VHDL:

- bit



- integer

-1 0 1 2 3 ...

- time

10 ns  
5 ms



- file



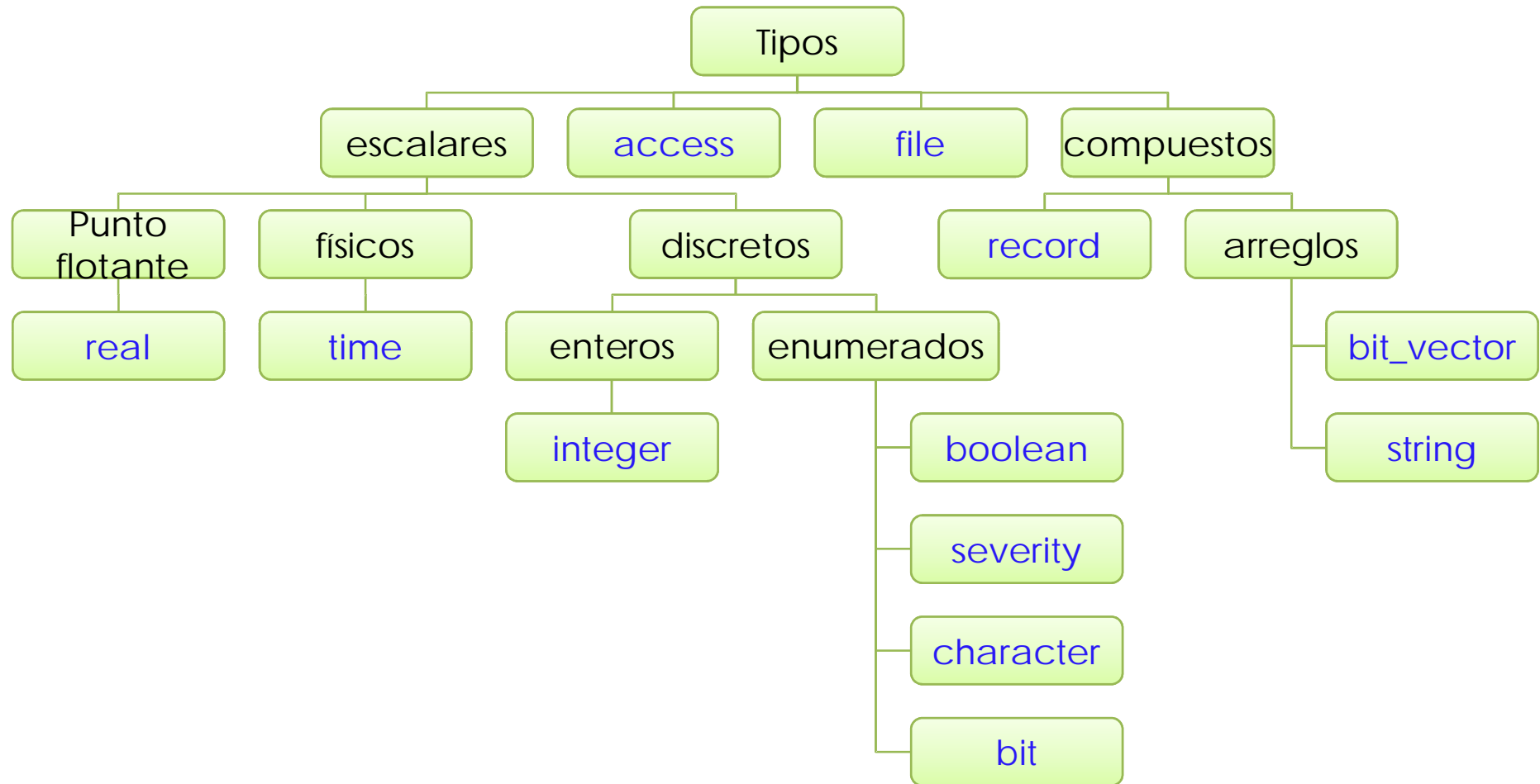
*Archivo.vhd*

- real

$\pi = 3,1415926535897$

# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

Clasificación de los tipos de datos predefinidos

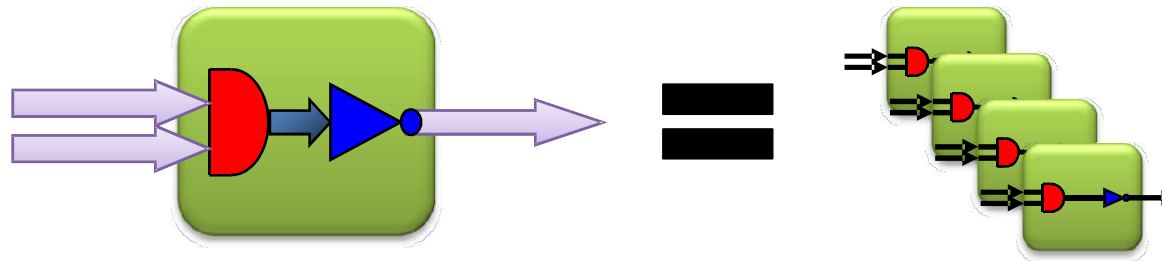


# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

## Tipos de datos

En la declaración de nuestra compuerta, utilizamos el tipo de datos **bit**. Este tipo de datos tiene dos valores posibles: '0' y '1', y se utiliza para señales individuales. Si quisiéramos aumentar la cantidad de bits de nuestra compuerta, podríamos utilizar el tipo **bit\_vector** en la declaración de señales:

```
entity compuerta is
  port(
    entrada1 : in    bit_vector (3 downto 0);
    entrada2 : in    bit_vector (3 downto 0);
    salida   : out   bit_vector (3 downto 0)
  );
end entity compuerta;
```

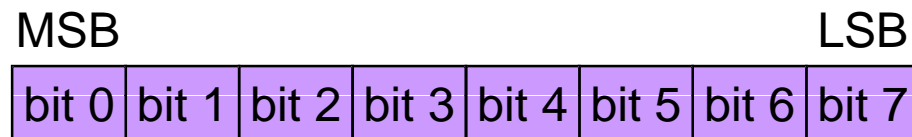


# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

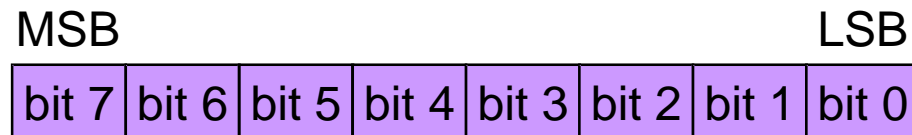
## Tipos de datos

En toda declaración de vectores se utilizan las palabras reservadas **downto** y **to** para determinar el tamaño del vector y la ubicación del bit más significativo.

**signal** mi\_vector : **bit\_vector** (0 **to** 7)



**signal** mi\_vector : **bit\_vector** (7 **downto** 0)



Para evitar confusiones  
utilice sólo un tipo de  
declaración.

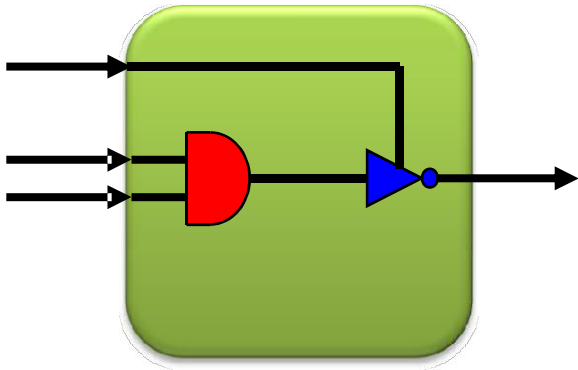


# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

## Tipos de datos

¿Son suficientes los tipos **bit** y **bit\_vector** para los sistemas digitales?

Supongamos que deseamos agregar a nuestro circuito la capacidad de poner su salida en alta impedancia.



Pero estos tipos de datos solo pueden representar los valores '0' y '1'.



¿Y Z?

# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

## Tipos de datos

Resulta que los tipos de datos predefinidos no alcanzan para describir un sistema digital en su totalidad. Por eso se agrega un tipo de datos llamado **std\_logic**.

Valores que pueden tomar las señales **std\_logic**:

<b>std_logic</b>	U	Sin inicializar
	X	Desconocido fuerte
	0	Nivel lógico bajo (cero fuerte)
	1	Nivel lógico alto (uno fuerte)
	Z	Alta impedancia
	W	Desconocido débil
	L	Cero débil (pull-down)
	H	Uno débil (pull-up)
	-	No importa

**std\_logic** está  
definido en la librería  
**ieee.std\_logic\_1164**



# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

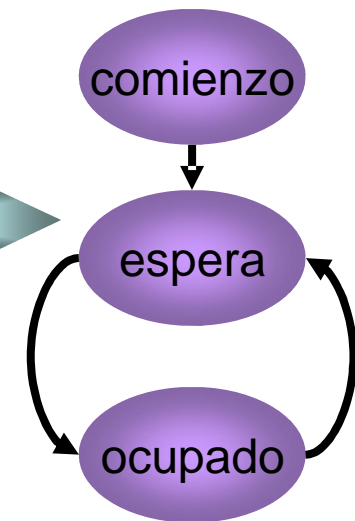
## Tipos de datos

También es posible definir nuevos tipos. Para esto se utiliza la sentencia **type**. Se pueden definir subconjuntos de tipos existentes, tipos enumerados o arreglos de varias dimensiones. Las definiciones se realizan en la parte declarativa de la arquitectura.

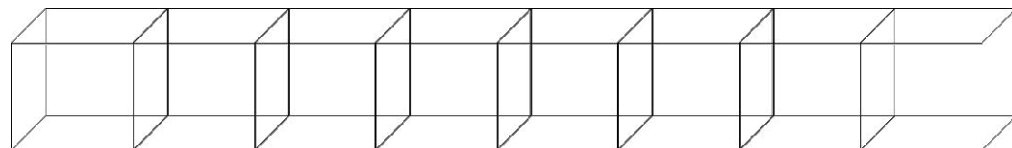
```
type estados is (comienzo, espera, ocupado);  
signal estado_actual : estados;
```

También se pueden definir arreglos de una o más dimensiones.

Los tipos enumerados suelen utilizarse para crear máquinas de estado.



```
type vector is array (7 downto 0) of integer;  
signal mi_vector : vector;
```



# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

## Operadores

Al igual que en otros lenguajes, existe una importante cantidad de operadores. A continuación se detallan algunos.

### Operadores aritméticos



Suma

$a \leq b + c;$



Resta

$a \leq b - c;$



Multiplicación

$a \leq b * c;$



División

$a \leq b / c;$



**a, b y c son  
tipos numéricos  
(la suma y resta  
funciona también  
con std\_logic y  
std\_logic\_vector)**

**rem**

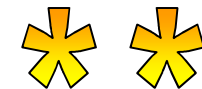
Resto

$a = b \text{ rem } c;$

**mod**

Módulo

$a = b \text{ mod } c;$



Potenciación

$a = b ** c;$

**abs()**

Valor absoluto

$a = \text{abs}(c);$

# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

## Operadores

### Operadores lógicos y binarios a nivel de bits

**and**

$x \leq y \text{ and } z;$

**or**

$x \leq y \text{ or } z;$

**not**

$x \leq \text{not } y;$

Operador de  
concatenación

**&**

$x \leq y \& z;$



**x, y, z son  
tipos boolean,  
bit,  
std\_logic,  
std\_logic\_vector**

**nand**

$x \leq y \text{ nand } z;$

**nor**

$x \leq y \text{ nor } z;$

**xor**

$x \leq y \text{ xor } z;$

# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

---

Operadores

Operadores relacionales



Igualdad



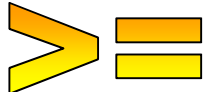
Desigualdad



Mayor



Menor



Mayor o igual



Menor o igual



**El resultado es un valor  
booleano (true / false), y  
se utilizan en las  
sentencias  
condicionales**

# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

---

## Librerías

Las librerías contienen declaraciones de tipos, definición de operadores, componentes, entidades, etc. Permiten incluir elementos predefinidos facilitando la programación.

Una librería importante es la librería IEEE.  
Ésta se encuentra subdividida en paquetes.  
Algunos paquetes importantes:

- **std\_logic\_1164**
- **std\_logic\_signed**
- **std\_logic\_arith**

Para emplear todos los paquetes de una librería:

```
library nombre_librería;  
use nombre_librería.nombre_paquete.all
```



# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

---

Ejemplo de código completo

```
library IEEE;                                     -- Declaración de librerías
use IEEE.STD_LOGIC_1164.ALL;

entity compuerta is
  Port (
    entrada1 : in  STD_LOGIC_VECTOR (3 downto 0);
    entrada2 : in  STD_LOGIC_VECTOR (3 downto 0);
    triestado : in  STD_LOGIC;
    salida1   : out STD_LOGIC_VECTOR (3 downto 0));
end compuerta;

architecture Behavioral of test is
  signal señal_int : STD_LOGIC_VECTOR (3 downto 0);
begin
  señal_int <= entrada1 and entrada2;
  salida1   <= not señal_int when triestado='0' else (others=>'Z');
end Behavioral;
```





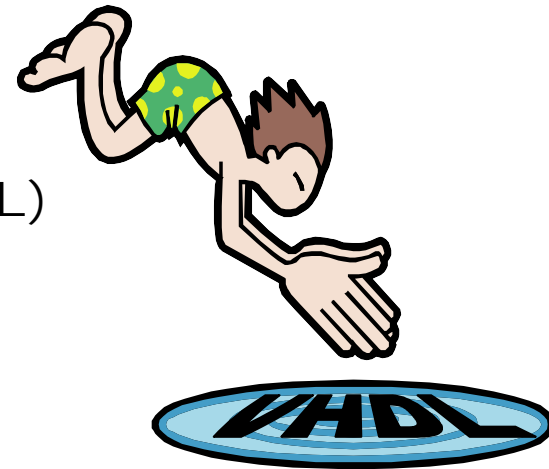
# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

---

Sumergiéndose en VHDL

En VHDL existen tres tipos principales de descripciones de hardware:

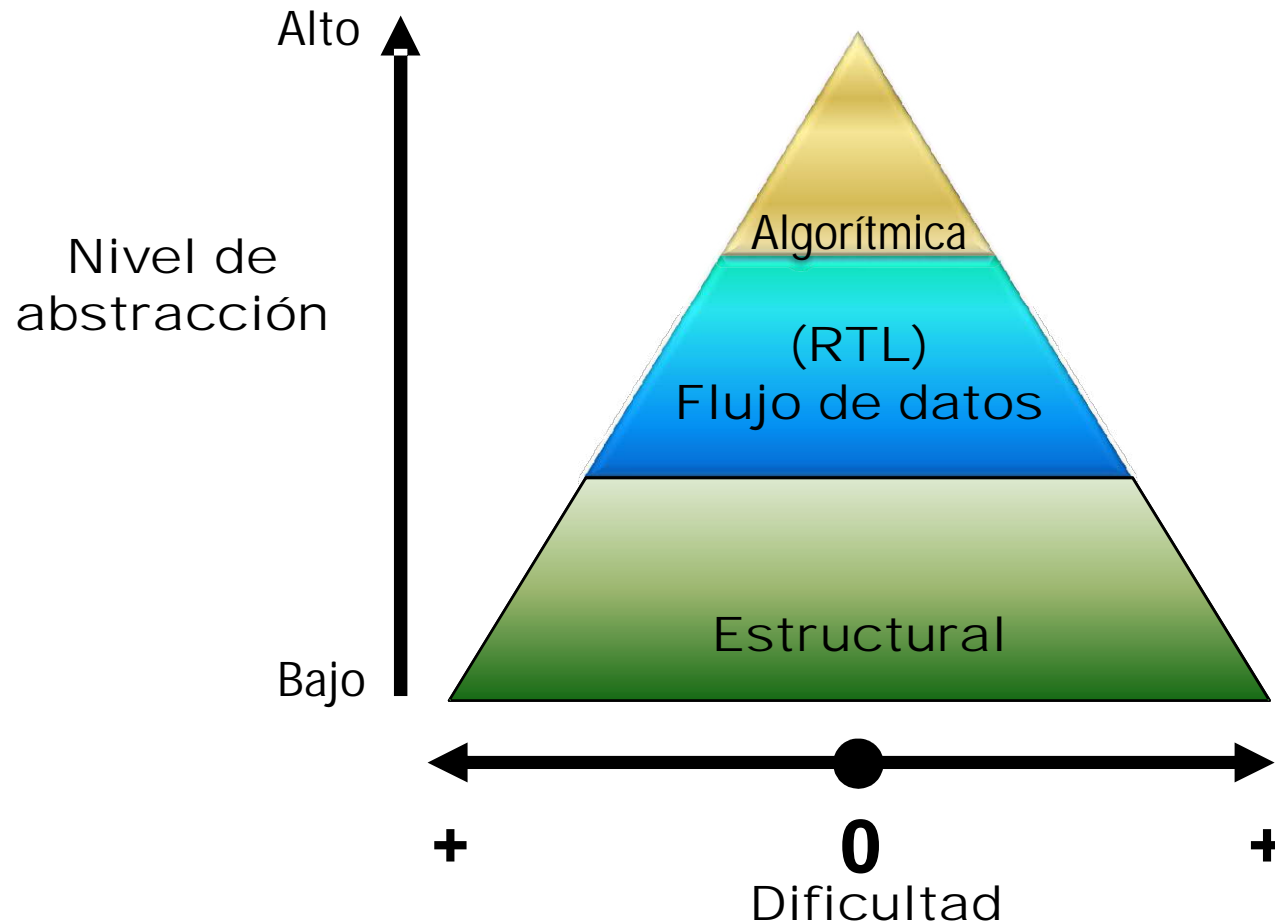
- Descripción Algorítmica
- Descripción de Flujo de Datos (RTL)
- Descripción Estructural



# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

Sumergiéndose en VHDL

Las distintas descripciones poseen niveles de abstracción y dificultad diferentes



# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

---

Sumergiéndose en VHDL

Tipos de descripciones de hardware

- Descripción Flujo de datos



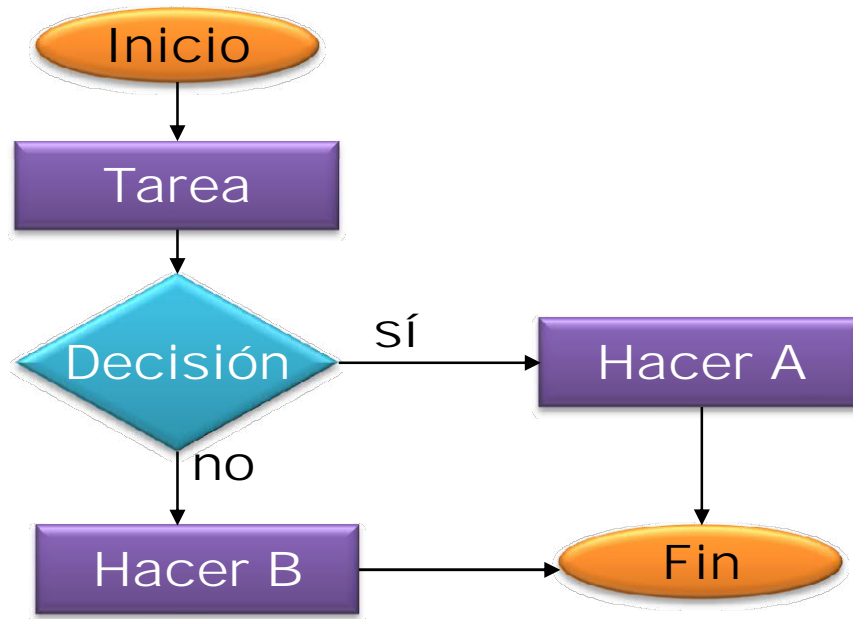
Divide todo diseño en lógica combinacional y registros. La descripción se realiza mediante sentencias concurrentes, con la excepción de los registros, que se describen de manera secuencial.

# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

Sumergiéndose en VHDL

Tipos de descripciones de hardware

- Descripción Algorítmica



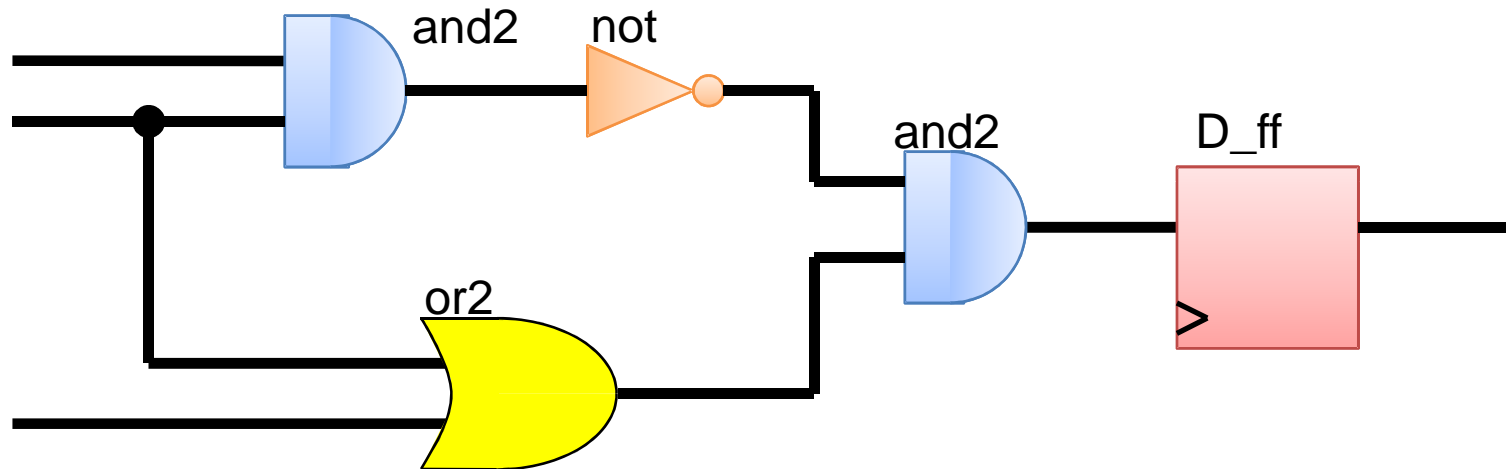
Define un diseño mediante algoritmos secuenciales, similares a los utilizados en otros lenguajes de programación. Por ende, utiliza sentencias secuenciales.

# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

Sumergiéndose en VHDL

Tipos de descripciones de hardware

- Descripción Estructural



Una descripción estructural utiliza elementos previamente definidos para representar un diseño. Este tipo de descripción consiste en la instanciación de las partes que la componen y su conexionado.

# **INTRODUCCIÓN A LA LÓGICA PROGRAMABLE**

---

## **Descripción Flujo de Datos**

# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

## Descripción Flujo de Datos

Sentencias utilizadas por la descripción flujo de datos

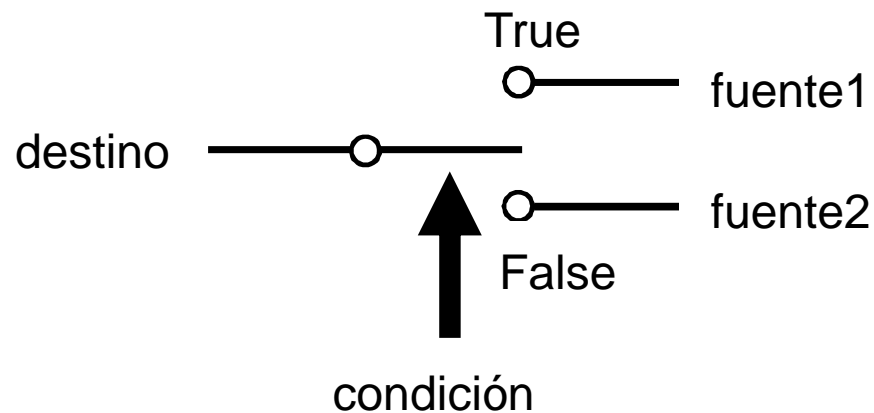
sentencia **when ... else**

destino  $\leftarrow$  *fuente1* **when** condición **else** *fuente2*

*fuente1* y *fuente2* pueden ser expresiones, señales o valores constantes  
condición es una expresión con resultado booleano.



Funciona como un interruptor inversor



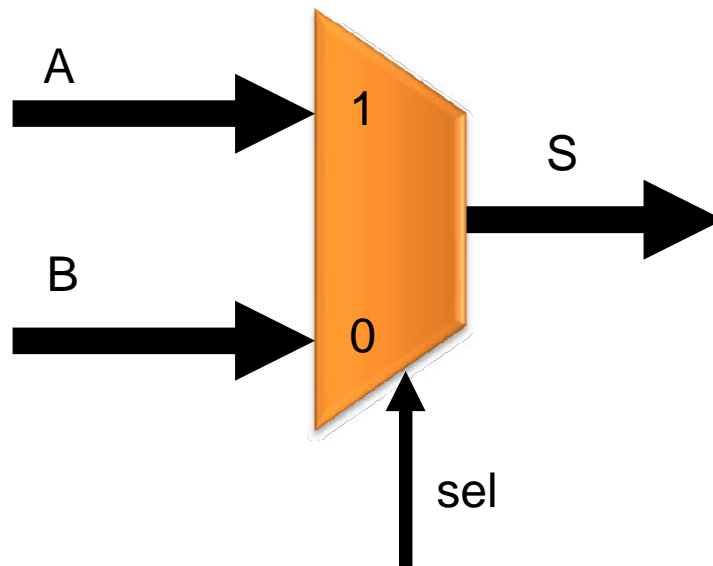
# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

## Descripción Flujo de Datos

Sentencias utilizadas por la descripción flujo de datos

sentencia **when ... else**

Es ideal para describir multiplexores



$S \leq A$  **when** sel='1' **else** B;

Se pueden anidar varias sentencias por ejemplo, para formar un mux más grande:

$S \leq A$  **when** sel="00" **else**  
B **when** sel="01" **else**  
C **when** sel="10" **else**  
D;

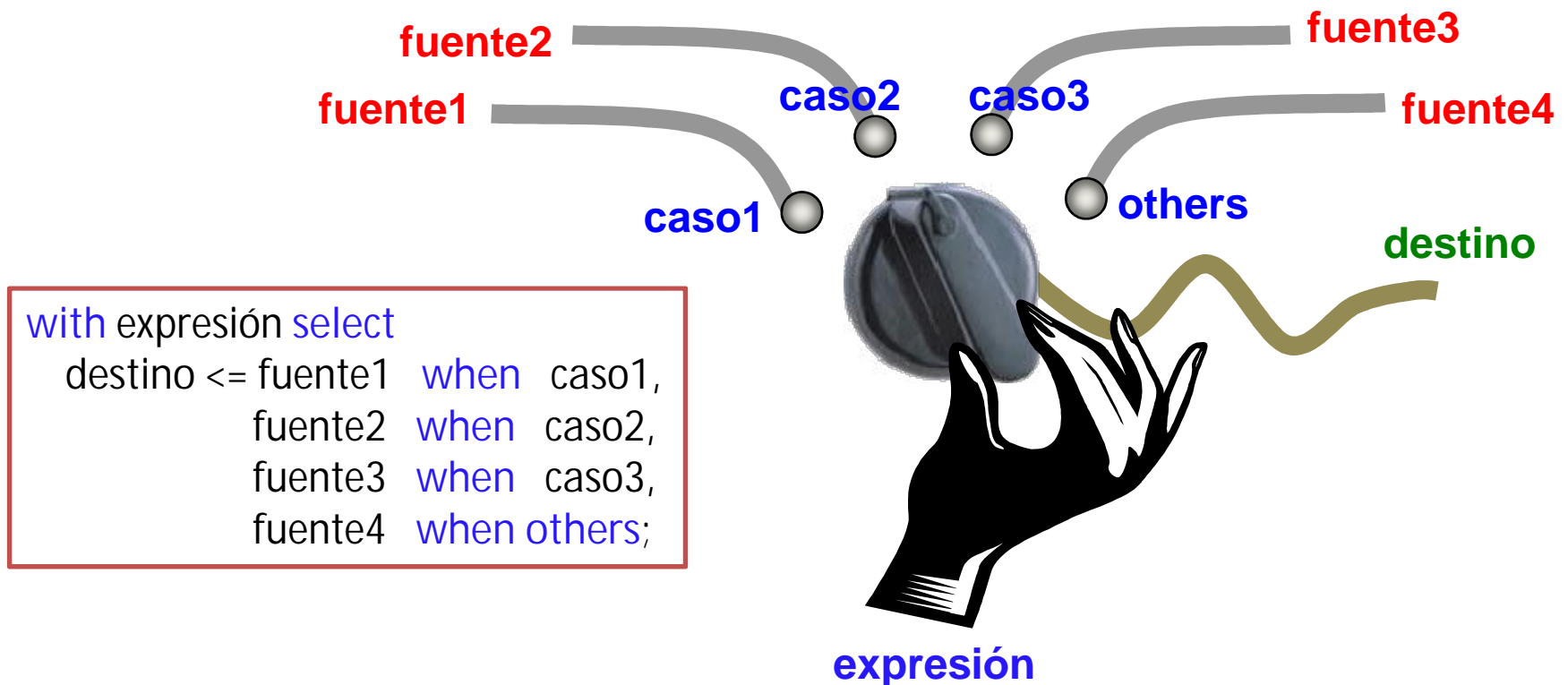


# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

Descripción Flujo de Datos

Sentencias utilizadas por la descripción flujo de datos

sentencia **with ...select ... then**



# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

-- Ejemplo de descripción flujo de datos

**library** IEEE;

**use** IEEE.STD\_LOGIC\_1164.**ALL**;

**entity** multi\_compuerta **is**

**port** ( sel              : **in** std\_logic\_vector(1 **downto** 0);

        T\_en             : **in** std\_logic;

        A                : **in** std\_logic;

        B                : **in** std\_logic;

        S                : **out** std\_logic);

**end entity** multi\_compuerta;

**architecture** flujo\_de\_datos **of** multi\_compuerta **is**

**signal** salida\_interna : std\_logic;

**begin**

**with** sel **select**

    salida\_interna <= A and B **when** "00",

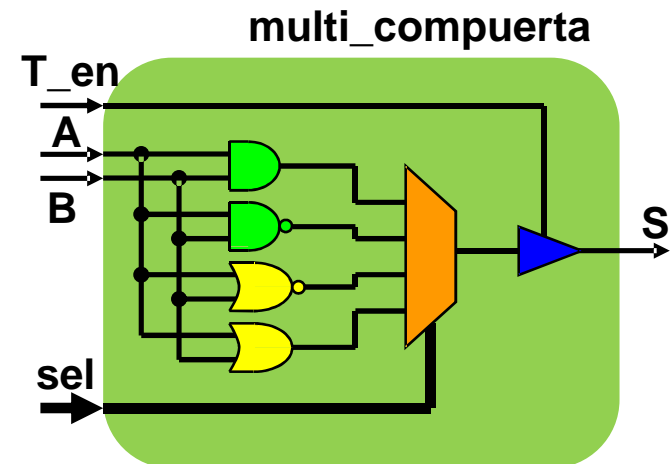
                  A nand B **when** "01",

                  A nor B **when** "10",

                  A or B **when others**;

  S <= salida\_interna **when** T\_en='0' **else** 'Z';

**end architecture** flujo\_de\_datos;



# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

---

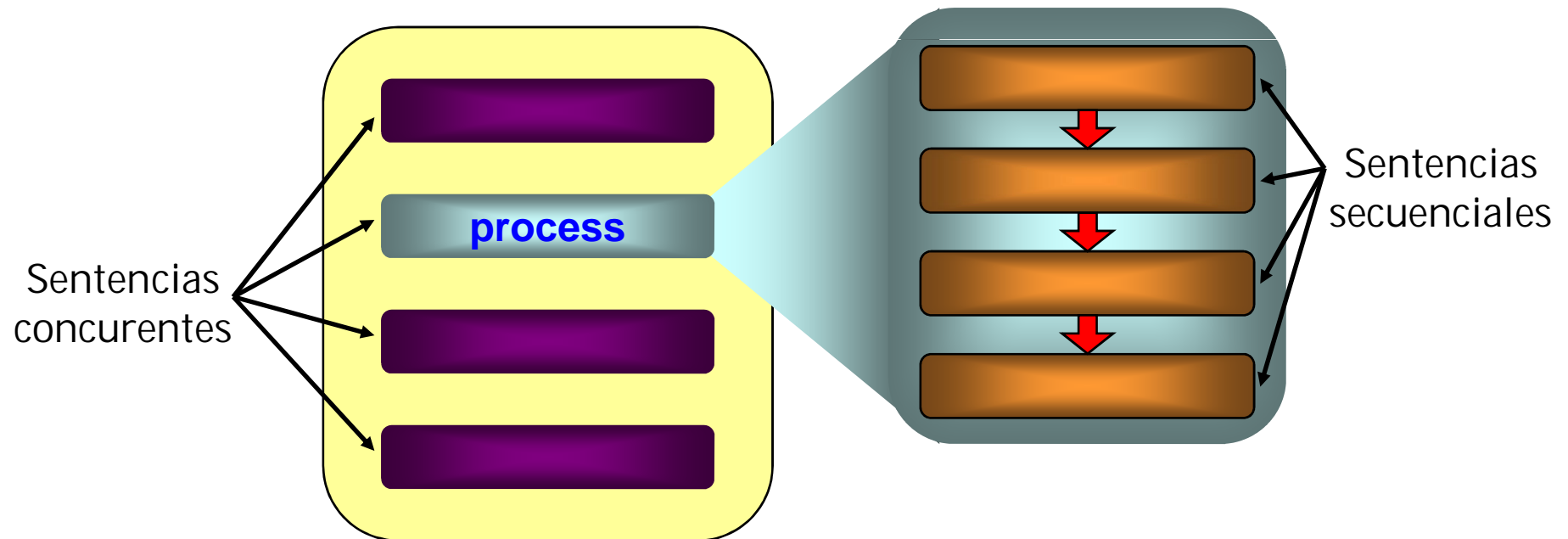
## **Descripción Algorítmica**

# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

## Descripción Algorítmica

La base de las descripciones algorítmicas: el bloque **process**

El bloque **process**, o proceso, es una estructura que permite, en un lenguaje predominantemente concurrente como VHDL, la declaración de sentencias secuenciales.



# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

## Descripción Algorítmica

El bloque **process**

Estructura de un proceso:

```
[ id_proc: ]  
process( lista_de_sensibilidad)  
  [ declaraciones ]  
begin  
  sentencias_serie;  
  ...  
end process;
```

El identificador de proceso (id\_proc) es opcional. Las declaraciones permiten la creación de variables y también son opcionales.



# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

## Descripción Algorítmica

### El bloque **process**

#### Variables

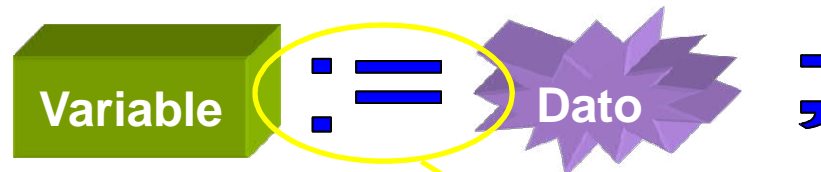
Los procesos permiten la declaración de elementos denominados variables. Éstas cumplen la misma función que en otros lenguajes.

La palabra reservada **variable** indica una declaración.

**variable** mi\_variable tipo\_de\_dato := valor\_inicial;

**integer**  
**bit**  
**std\_logic**  
etc.

Asignación de variables:



Este es el operador utilizado para las asignaciones de variables.

# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

---

## Descripción Algorítmica

El bloque **process**

### Variables

Una variable cumple la función de almacenar datos y permitir operaciones sobre los mismos en un proceso.



Pero... ¿no pueden las señales hacer eso?

La respuesta es: sí, pero de manera diferente.

Una señal es un elemento concurrente, y como tal, es concurrente con todo el proceso, por lo tanto las asignaciones realizadas a una señal toman efecto al finalizar el mismo.

# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

## Descripción Algorítmica

El bloque **process**

Señales

VS.

Variables

sin\_variable:  
**process** (a,b,c)  
**begin**  
Nunca se realiza  $\rightarrow$  a <= 2;  
b <= a + 4;  
a <= c + a;  
**end process** sin\_variable;

Inicialmente  
a = 1  
b = 2  
c = 1  $\rightarrow$  3

después del proceso  
a = 4  
b = 5  
c = 3



con\_variable:  
**process** (a,b,c)  
variable x : integer;  
**begin**  
x := 2;  $\leftarrow$  Asignación instantánea  
b <= x + 4;  
a <= c + x;  
**end process** con\_variable;

después del proceso  
a = 5  
b = 6  
c = 3



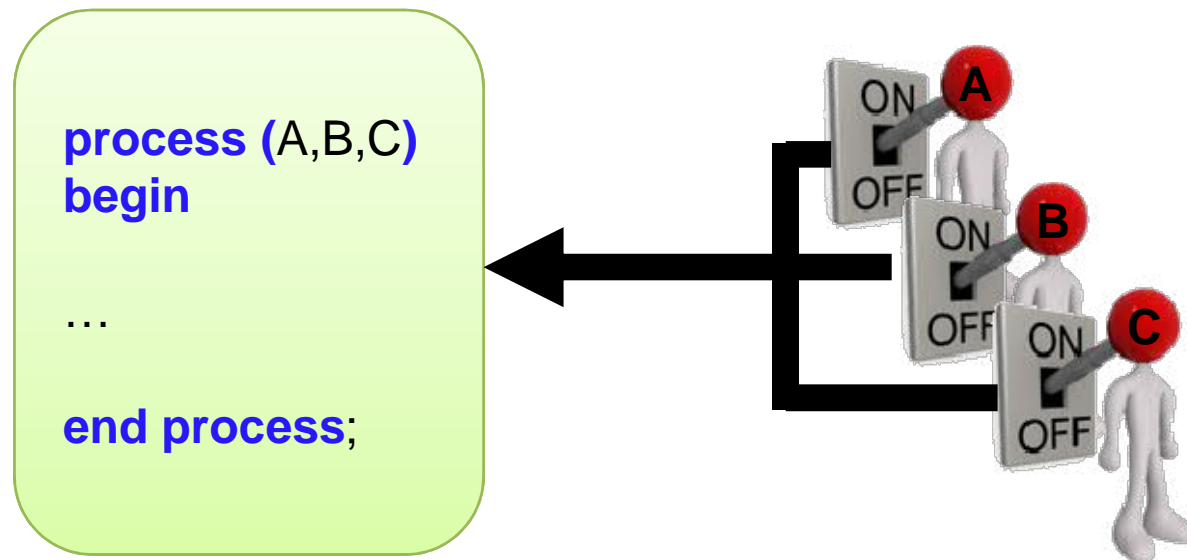


# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

## Descripción Algorítmica

El bloque **process**

La lista de sensibilidad define a través de las señales listadas en ella, el momento de evaluación del proceso. Cuando alguna de estas señales cambie de valor, éste se ejecutará.



# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

---

## Descripción Algorítmica

El bloque **process**

La sentencia **wait**.

Esta sentencia constituye una alternativa a la lista de sensibilidad. Su función es la de detener la ejecución de un proceso hasta que se cumpla alguna condición.



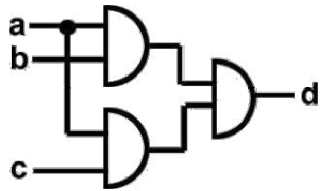
# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

## Descripción Algorítmica

El bloque **process**

La sentencia **wait**.

Existen diferentes condiciones posibles.



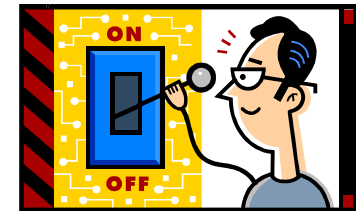
Espera una condición lógica  
**wait until** d='1' ;

Esperar el cambio de alguna señal  
**wait on** a, b, clock ;



Esperar un tiempo  
**wait** 10 ms;

Esperar indefinidamente (matar el proceso)  
**wait**;



# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

## Descripción Algorítmica

El bloque **process**

Sentencias secuenciales

Los procesos utilizan sentencias diferentes a las concurrentes.  
A continuación explicaremos algunas de ellas.

Sentencia condicional **if ... then ... else**

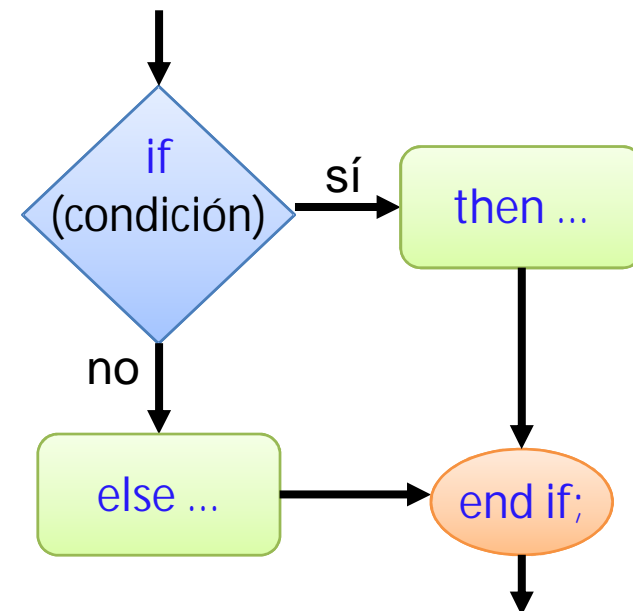
**if** (condición) **then**

...  
[ **elsif** ]

...  
[ **else** ]

...  
**end if;**

condición es una  
expresión con  
resultado booleano



# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

Descripción Algorítmica

El bloque **process**

Sentencias secuenciales

Sentencia de selección: **case**

**case** expresión **is**

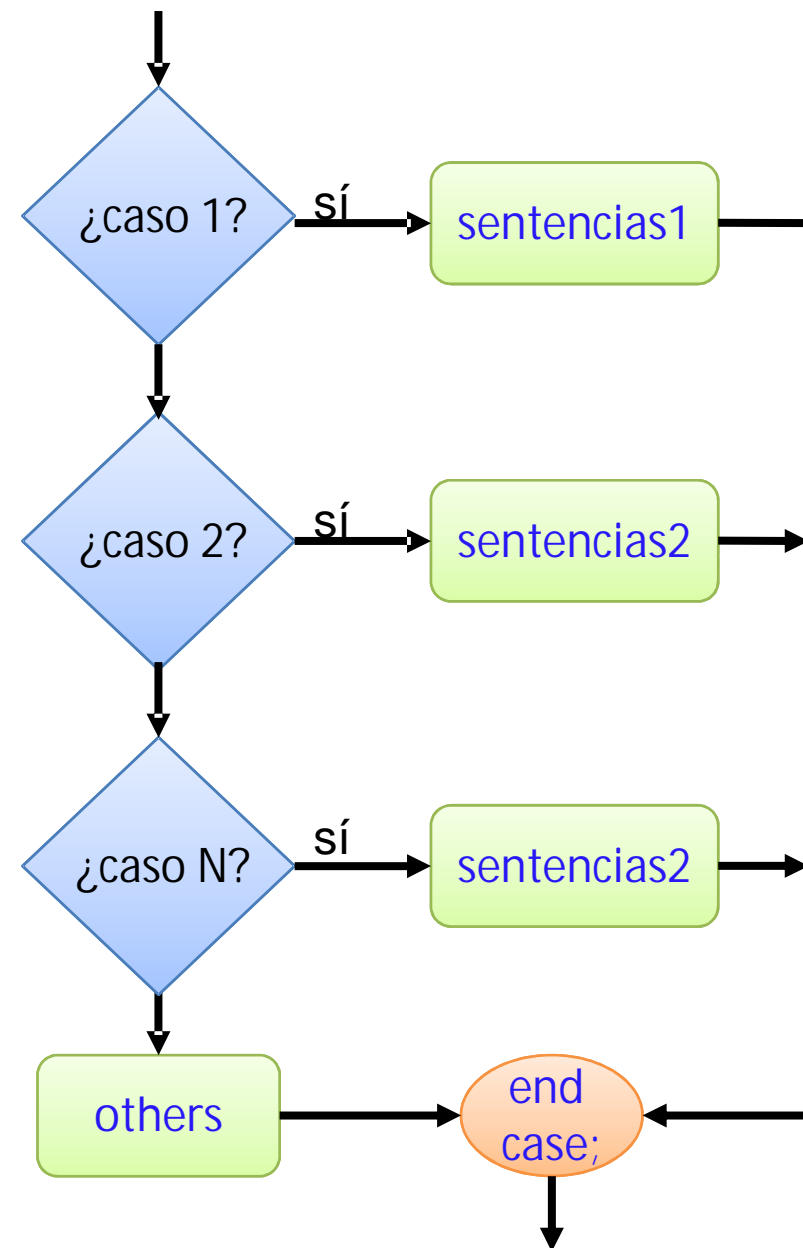
**when** caso1 => sentencias1;

**when** caso2 => sentencias2;

...

**when others** => sentencias;

**end case;**



# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

Descripción Algorítmica

El bloque **process**

Sentencias secuenciales

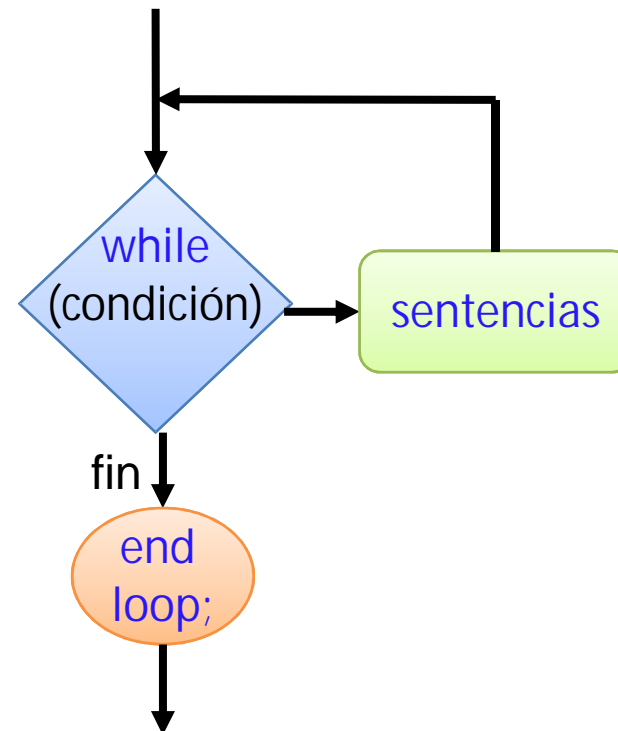
Bucles **for** y **while**



**while** condición  
**loop**  
...  
**end loop;**

**for** cuenta **in** rango  
**loop**  
...  
**end loop;**

Puede ser ascendente: 0 **to** N  
o descendente: N **downto** 0



# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

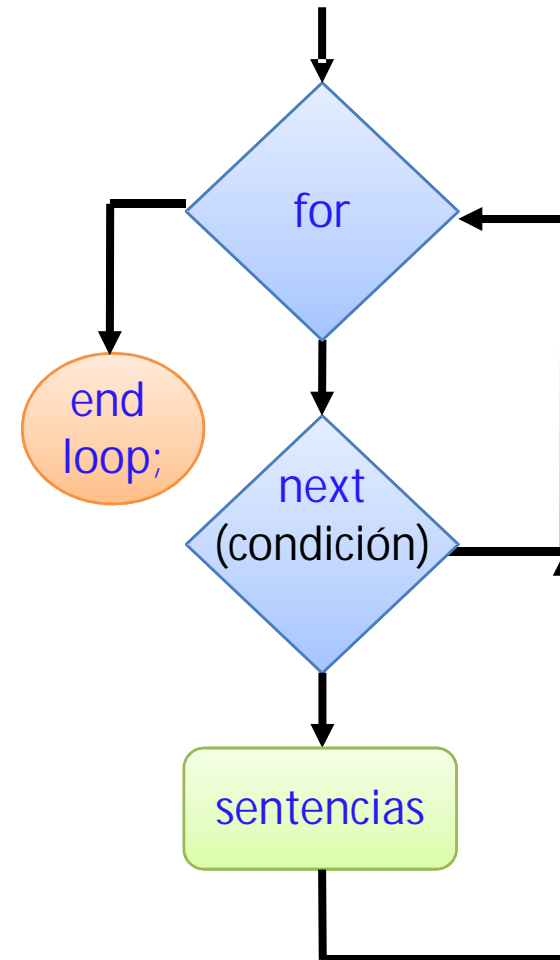
## Descripción Algorítmica

El bloque **process**

Sentencia **next**

Permite detener la ejecución de la iteración actual y pasar a la siguiente:

**next** id\_lazo: **when** condición;



# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

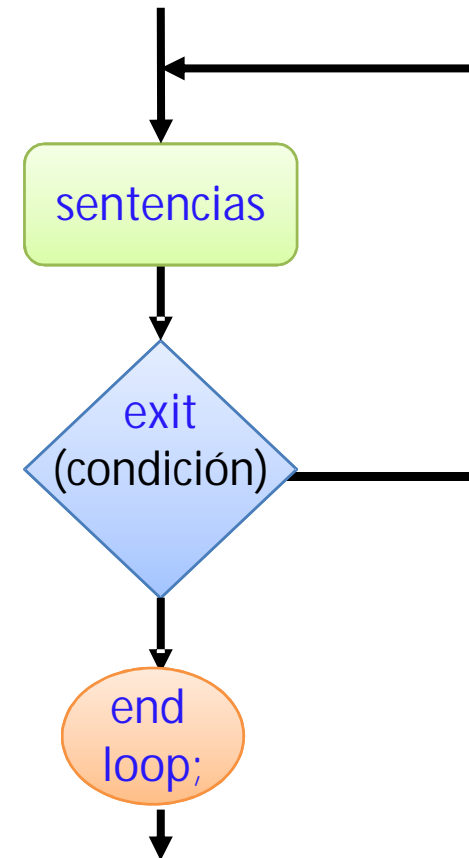
## Descripción Algorítmica

El bloque **process**

Sentencia **exit**

Permite salir del lazo de iteración:

**exit** id\_bucle **when** condición;

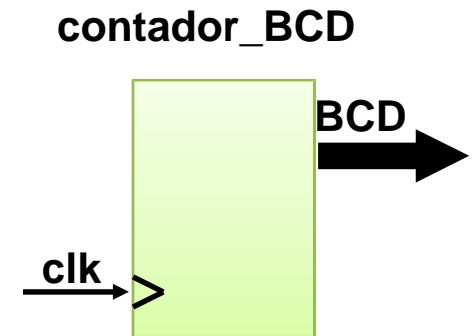




# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

-- Ejemplo de descripción algorítmica

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity contador_BCD is
    port ( clk          : in  std_logic;
          BCD          : out std_logic_vector (3 downto 0));
end entity contador_BCD;
architecture algorítmica of contador_BCD is
    signal cuenta : std_logic_vector (3 downto 0):=(others=>'0');
begin
    process(clk)
    begin
        if (clk'event and clk='1') then
            if cuenta="1001" then
                cuenta <= (others=>'0');
            else
                cuenta <= cuenta + 1;
            end if;
        end if;
    end process;
end architecture algorítmica;
```



# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

---

## **Descripción Estructural**

# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

## Descripción Estructural

Un diseño estructural está compuesto por muchas entidades interconectadas. Para instanciar entidades se utiliza la siguiente estructura:

El nombre por omisión de la librería de usuario es work.

Luego de la librería se coloca el nombre de la entidad que deseamos replicar y la arquitectura a utilizar en esta.

id\_inst:

```
entity work.nombre_entidad(nombre_arquitectura)
generic map ( param_entidad => param_instancia
               ... );
port map ( puertos_entidad => conexión_instancia
            ... );
```



# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

## Descripción Estructural

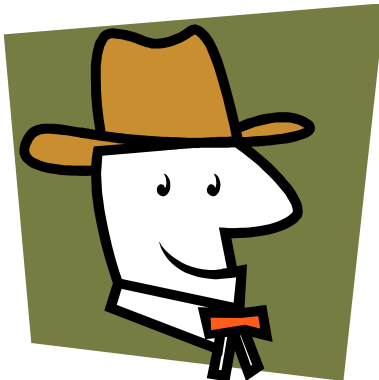
El identificador de instancia es obligatorio y se usa para diferenciar una instancia de otra.

id\_inst:

```
entity work.nombre_entidad(nombre_arquitectura)  
generic map ( param_entidad => param_instancia  
              ... );
```

```
port map ( puertos_entidad => conexión_instancia  
          ... );
```

Por último, se definen las conexiones de la instancia con el resto del diseño.

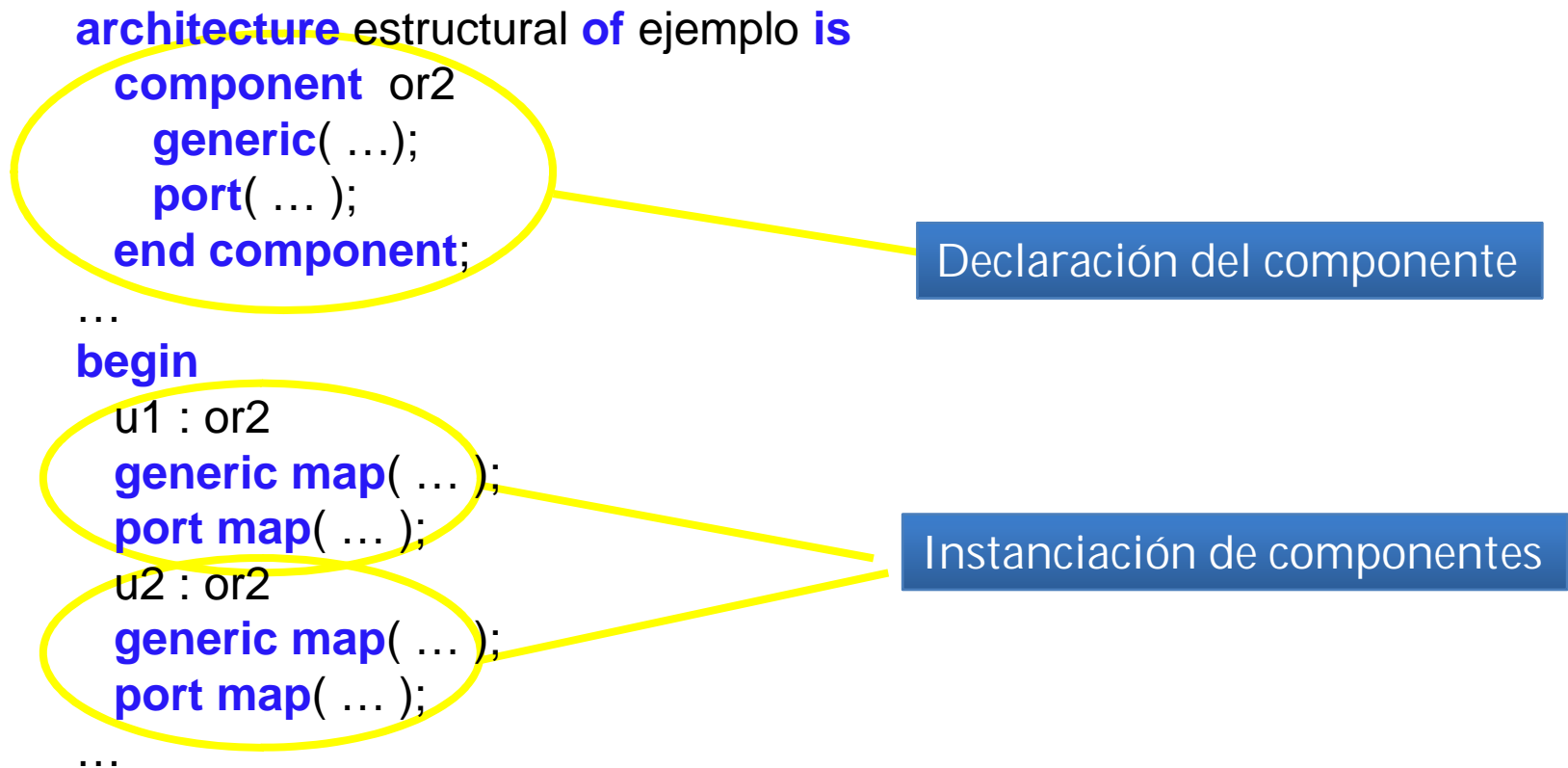


Este tipo de instanciación se denomina instanciación de entidad directa.

# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

## Descripción Estructural

Otra manera es utilizando la sentencia **component**, para declarar un componente y luego instanciarlo.

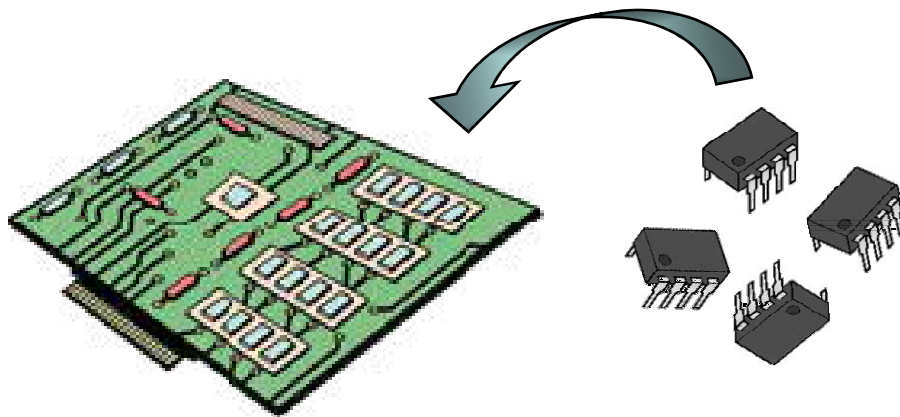


# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

## Descripción Estructural

La referencia de un componente hacia una entidad se hace mediante el bloque **configuration**.

**configuration** estructural **of** ejemplo **is**  
**for** estructural  
  **for** u1 : or2 **use entity** TTL\_or2(rapida);  
  **for** u2 : or2 **use entity** TTL\_or2(normal)  
**end component**;



Muchas de las herramientas CAD realizan esta referencia de manera automática.



# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

## Descripción Estructural

Consideraciones sobre el mapeo de puertos y genéricos.

**generic map** ( parametroA => valor,  
                  parametroB => parametroC);

**port map**( puerto\_entradaA => señal,  
            puerto\_salidaB => vector(5),  
            puerto\_salidaC => **open**);



Los puertos listados dentro del **port map** deben corresponderse exactamente a aquellos en la entidad siendo instanciada.



Se debe respetar la dirección y el tipo de datos de los puertos de la entidad al realizar las conexiones.



La dirección de la flecha dentro del **port map** es contraria a la dirección en las asignaciones.



Se pueden especificar puertos sin conectar con la palabra reservada **open**.

# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

## Descripción Estructural

Replicación de estructuras: las sentencias **for ... generate** e **if ... generate**  
Estas sentencias permiten el control de la instanciación de múltiples elementos de manera sencilla y efectiva.

Permite repetir  
elementos una  
cantidad N de  
veces.



```
id_for:  
for indice in 0 to N generate  
  
    [declaraciones ]  
  
begin  
  
    [sentencias_a_replicar]  
  
end generate;
```





# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

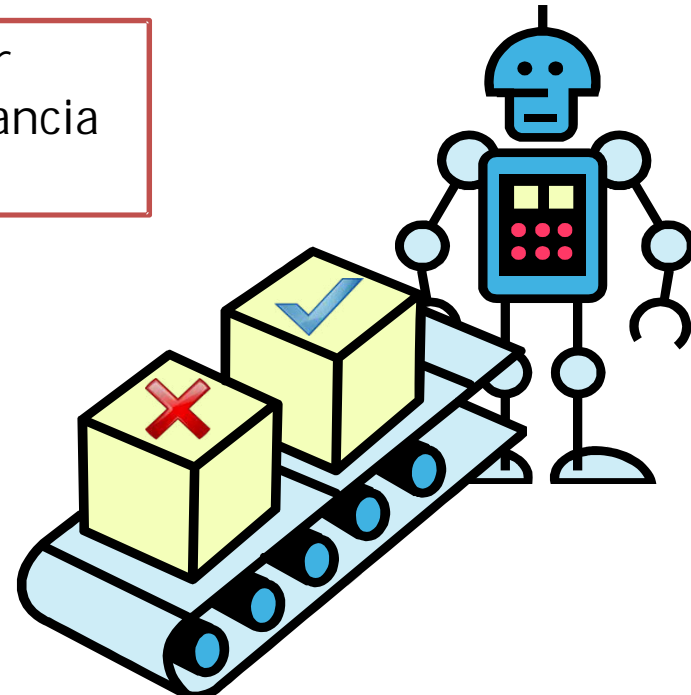
## Descripción Estructural

Replicación de estructuras: las sentencias **for** ... **generate** e **if** ... **generate**

```
id_gen:  
if condición generate  
  
begin  
  
[sentencias]  
  
end generate;
```



Permite decidir  
cuando se instancia  
un elemento.



# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

-- Ejemplo de descripción estructural

**library** IEEE;

**use** IEEE.STD\_LOGIC\_1164.**ALL**;

**entity** registro\_desplazamiento **is**

**generic** ( Nbits : **integer** :=8);

**port** ( clk : **in** **std\_logic**;

          shift\_in : **in** **std\_logic**;

          shift\_out : **out** **std\_logic**);

**end entity** registro\_desplazamiento;

**architecture** estructural **of** registro\_desplazamiento **is**

**component** dff

**port**( D : **in** **std\_logic**;

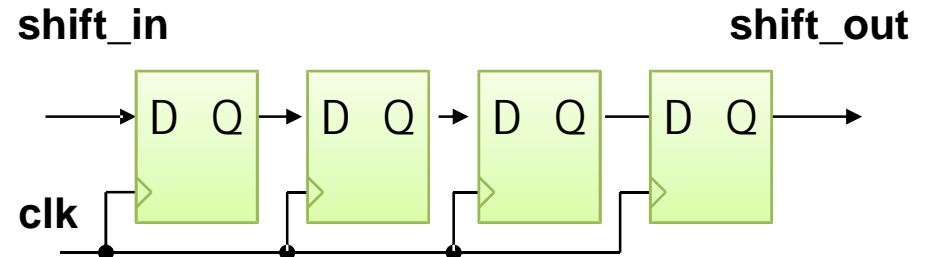
        Q : **out** **std\_logic**;

        clk : **in** **std\_logic**);

**end component**;

  signal interna : **std\_logic\_vector** (Nbits-2 **downto** 0);

**begin**



# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

reg\_desp:

**for** indice\_bit **in** 0 **to** Nbits-1 **generate**

primero: **if** (indice\_bit=0) **generate**

pff: dff

**port map**( D => shift\_in,  
Q => interna(0),  
clk=> clk);

**end generate**;

medio: **if** (indice\_bit/=0 **and** indice\_bit /= Nbits-1) **generate**

mff : dff

**port map**( D => interna(indice\_bit-1),  
Q => interna(indice\_bit),  
clk=> clk);

**end generate**;

ultimo: **if** (indice\_bit = Nbits-1) **generate**

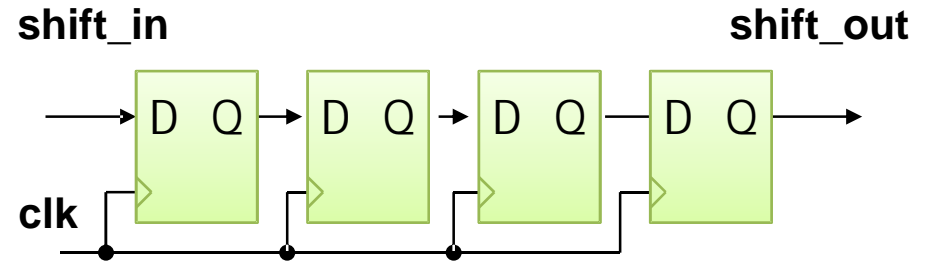
uff: dff

**port map**( D => interna(indice\_bit-1),  
Q => shift\_out,  
clk=> clk);

**end generate**;

**end generate**;

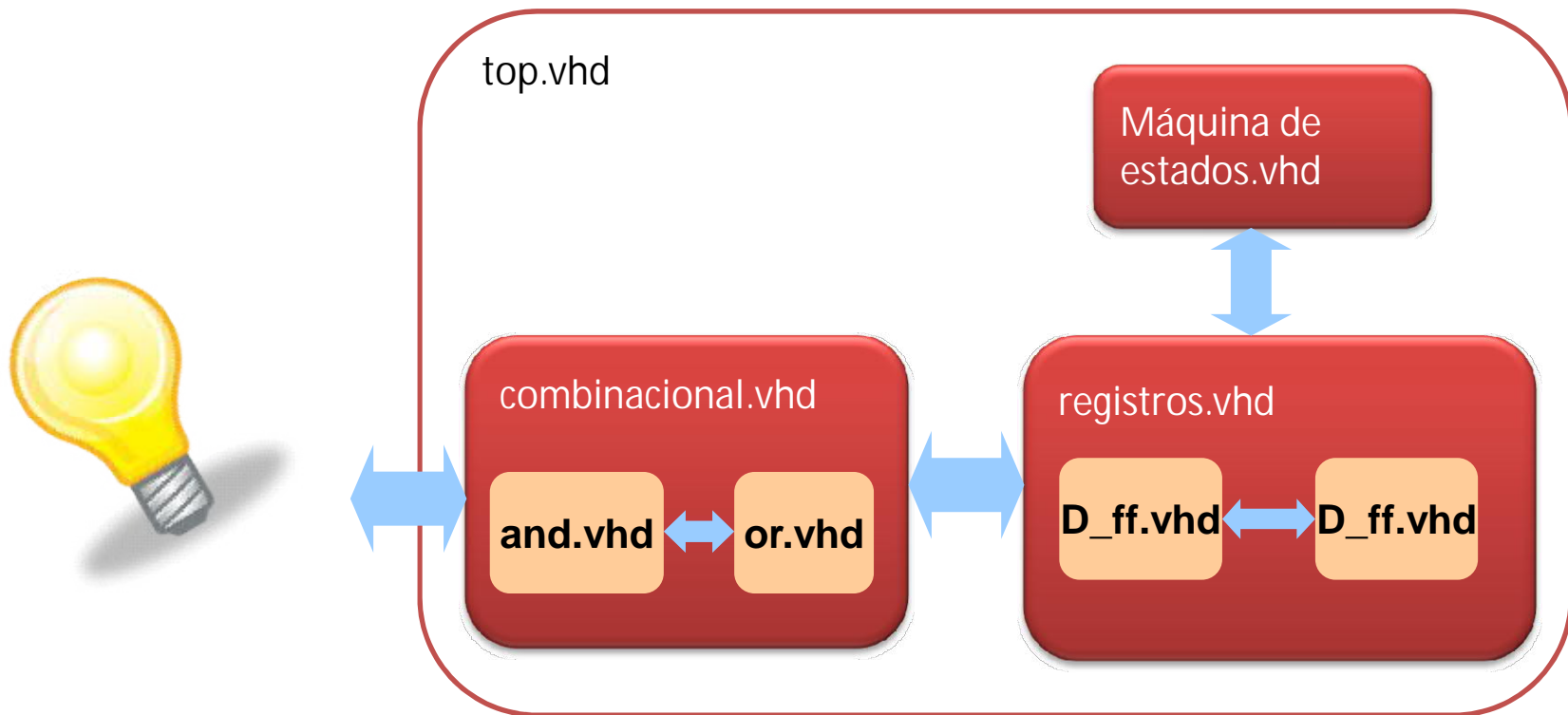
**end architecture** estructural;



# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

Descripción Estructural: Diseño jerárquico

Mediante este estilo de diseño se puede dividir un sistema en partes de fácil resolución, para luego integrarlas y formar el sistema completo.



# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

---

## **VHDL para síntesis**

# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

---

VHDL para síntesis

No siempre un código VHDL es sintetizable.

Sólo un subconjunto reducido del lenguaje se utiliza para sintetizar circuitos.



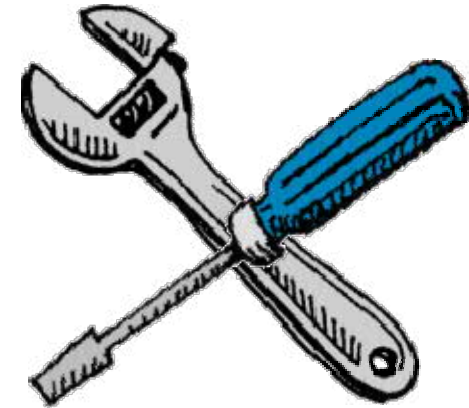
# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

---

VHDL para síntesis

Algunos consejos para realizar descripciones sintetizables:

- Evítense las sentencias **wait**.
- Evítense excesivos **if** anidados.
- Utilícese **case** en vez de varios **if**.
- Utilícese un reloj por proceso.
- No se olviden señales en las listas sensibles.
- Al momento de optar entre descripciones diferentes, utilizar la más sencilla de interpretar.

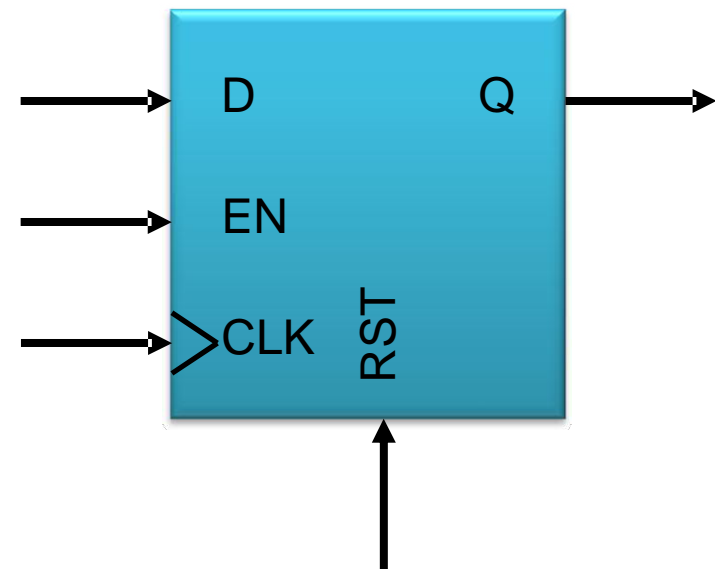


# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

VHDL para síntesis

Descripción de flip flops tipo D con reset  
síncrono y clock enable:

```
process (CLK)
begin
  if (CLK'event and CLK='1') then
    if RST='1' then
      Q <= '0';
    elsif (EN='1') then
      Q <= D;
    end if;
  end if;
end process;
```



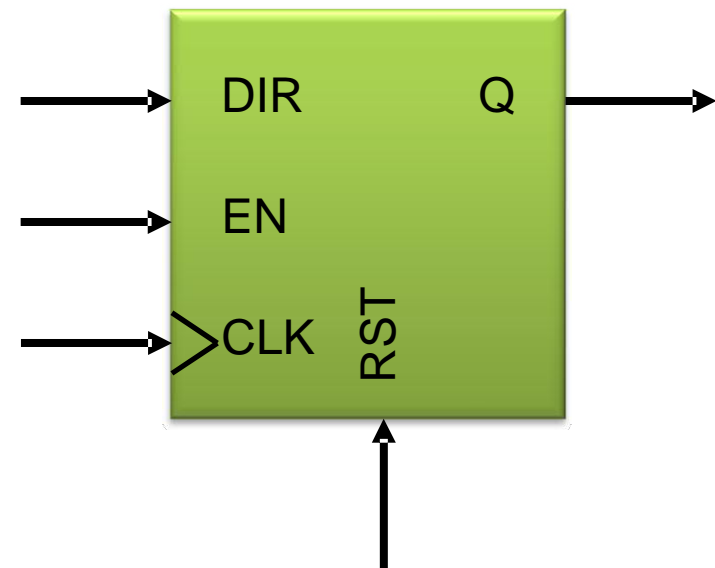


# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

VHDL para síntesis

Descripción de contadores up-down:

```
process (CLK)
  variable Q_int : std_logic_vector (3 downto 0);
begin
  if (CLK'event and CLK='1') then
    if RST='1' then
      Q_int := (others => '0');
    elsif (EN='1') then
      if (DIR='1') then
        Q_int := Q_int + 1;
      else
        Q_int := Q_int - 1;
      end if;
    end if;
  end if;
  Q <= Q_int;
end process;
```

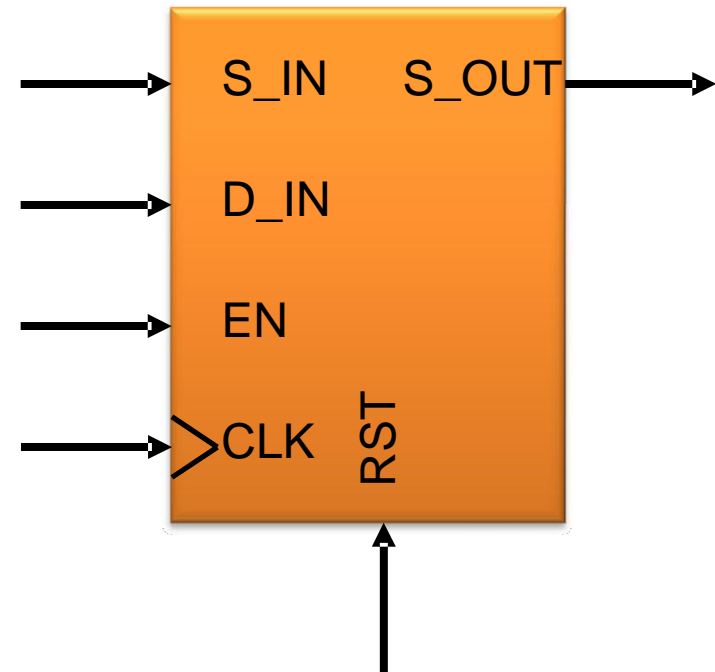


# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

VHDL para síntesis

Descripción de registros de desplazamiento con load y clock enable:

```
process (CLK)
begin
  if (CLK'event and CLK='1') then
    if RST='1' then
      D <= (others => '0');
    elsif (EN='1') then
      if load='1' then
        D <= D_IN;
      else
        D <= D(6 downto 0) & S_IN;
      end if;
    end if;
  end if;
end process;
S_OUT <= D(7);
```

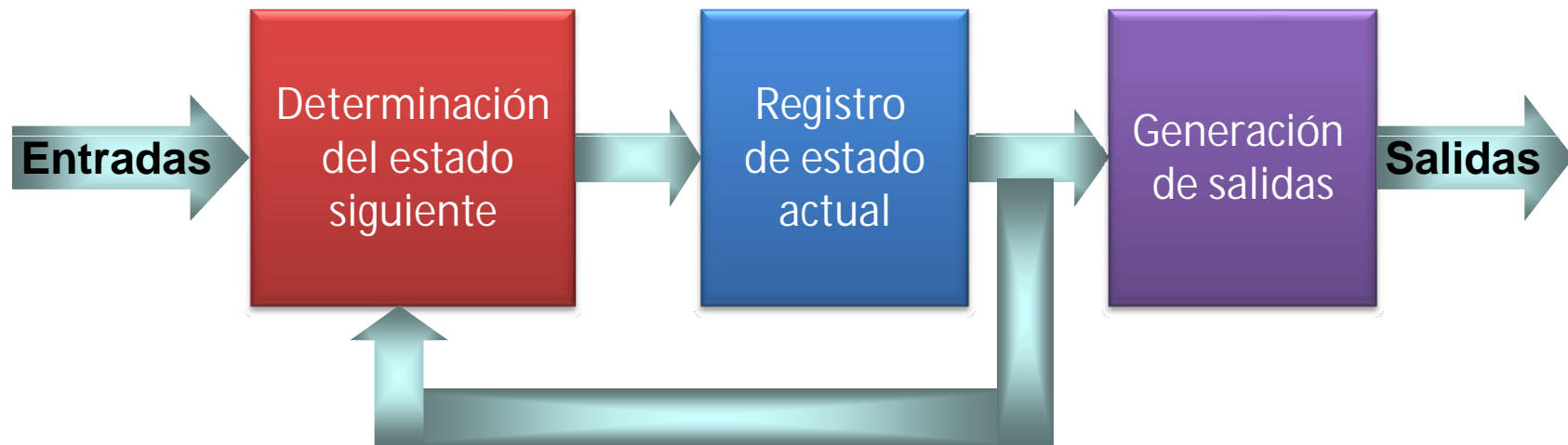


# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

---

VHDL para síntesis

El esquema de una máquina de estados de Moore es el siguiente:

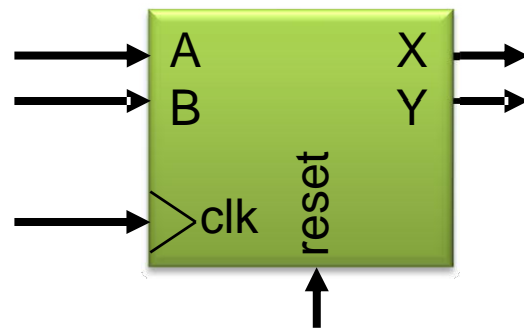


# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

VHDL para síntesis

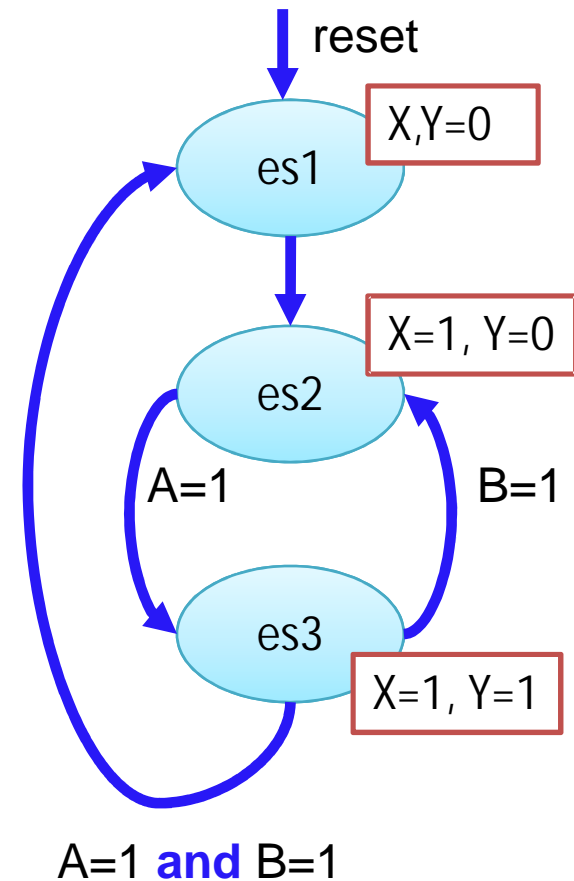
Descripción de máquinas de estado de Moore:

Supongamos la siguiente máquina de estados:



La entidad es:

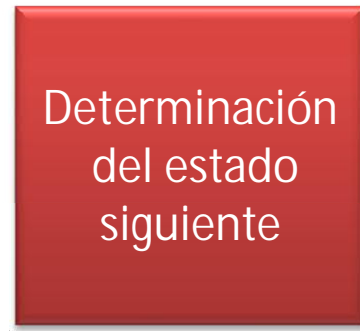
```
entity moore is
  port ( A,B      : in  std_logic;
         clk, reset : in  std_logic;
         X,Y      : out std_logic);
end entity moore;
```



# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

VHDL para síntesis

Descripción de máquinas de estado de Moore:



```
architecture behavioral of moore is
  type estados is (es1, es2, es3);
  signal actual, siguiente : estados;
begin
  ...
```

det\_siguiente\_estado:

```
process (A,B, actual)
begin
  case actual is
    when es1 =>
      siguiente<= es2;
    when es2 =>
      if (A='1') then
        siguiente <= es3;
      else
        siguiente <= es2;
      end if;
    when es3 =>
      if (A='1' and B='1') then
        siguiente <= es1;
      elsif (B='1') then
        siguiente <= es2;
      else
        siguiente <= es3;
      end if;
    end case;
  end process;
```

# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

---

VHDL para síntesis

Descripción de máquinas de estado de Moore:

```
reg_estado_actual:  
process (clk)  
begin  
if (clk='1' and clk'event) then  
    if reset='1' then  
        actual<=es1;  
    else  
        actual<=siguiente;  
    end if;  
end if;  
end process;
```



Registro  
de estado  
actual


# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

---

VHDL para síntesis

Descripción de máquinas de estado de Moore:

```
process (actual)
begin
  case actual is
    when es1 =>
      X<='0';
      Y<='0';
    when es2 =>
      X<='1';
      Y<='0';
    when es3 =>
      X<='1';
      Y<='1';
  end case;
end process;
```



Generación  
de salidas

# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

---

## **VHDL para simulación**



# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

---

## VHDL para simulación

El lenguaje VHDL fue diseñado inicialmente como medio para el modelado y la simulación de sistemas digitales, por lo cual no impone limitaciones en cuanto a la simulación.

**VHDL  
para simulación**

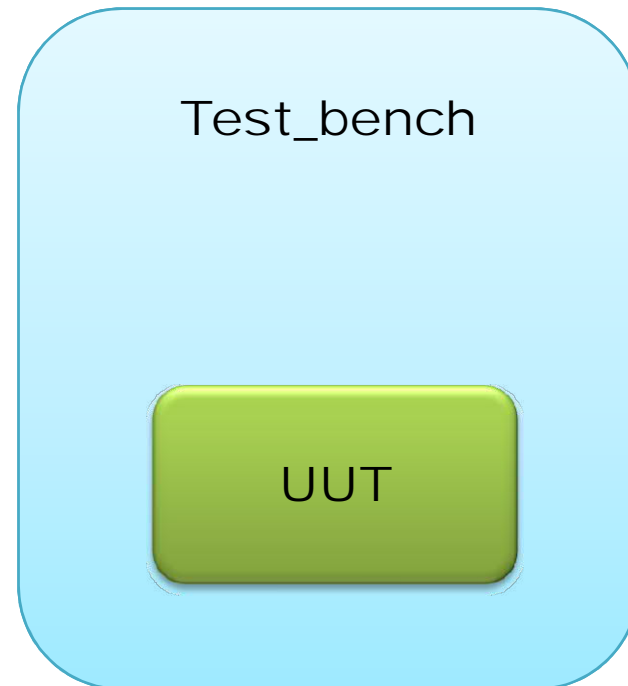


# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

VHDL para simulación

El test bench (banco de pruebas)

Un test bench se encarga de generar los estímulos necesarios para comprobar un circuito y corroborar los resultados de la simulación.



Un testbench es una entidad sin puertos que se utiliza para probar un diseño.

UUT: Unit Under Test

# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

---

VHDL para simulación

Pasos para crear un testbench

1. Crear una entidad sin puertos.
2. Instanciar la unidad bajo prueba en la arquitectura de dicha entidad.
3. Declarar una señal por cada puerto de la señal bajo prueba con el mismo nombre (e inicializar las entradas a algún valor).
4. Conectar las señales homónimas.
5. Crear los estímulos (generalmente en un proceso).
6. Escribir las sentencias que comprobarán los resultados.

# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

VHDL para simulación

La sentencia **assert**

Esta sentencia se utiliza para evaluar los resultados de la unidad bajo prueba y comprobar si se corresponden con los esperados. Su sintaxis es:

**assert** condición **report** “mensaje” **severity** nivel



Mensaje a mostrar si NO se cumple la condición.

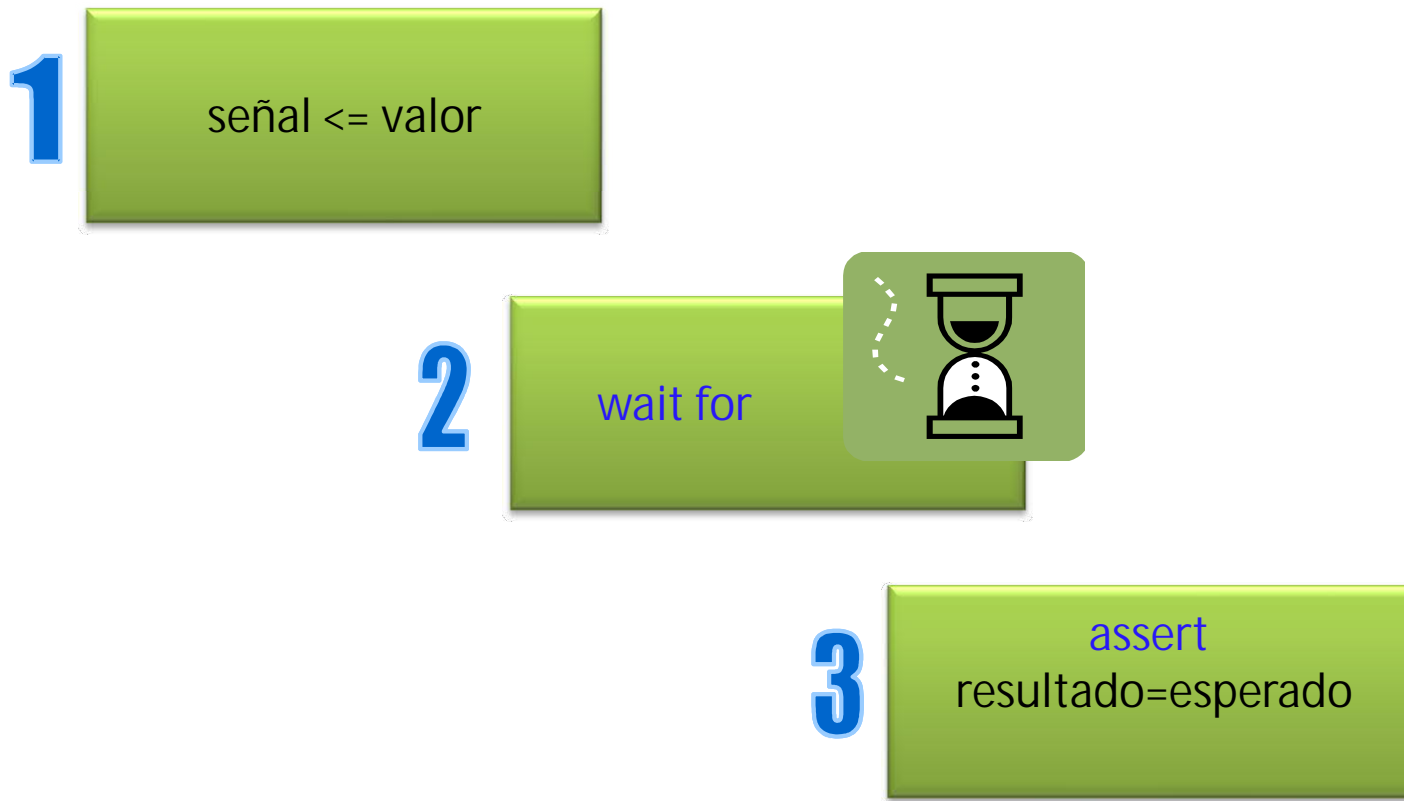
- NOTE
- WARNING
- ERROR
- FAILURE

# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

---

VHDL para simulación

Esquema básico para la generación de estímulos y comprobación de resultados:

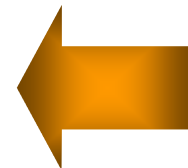
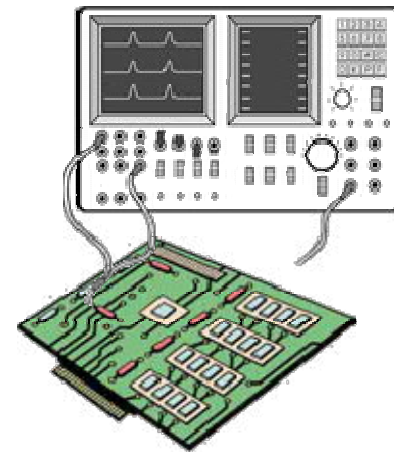


# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

VHDL para simulación

Ejemplo de testbench:

```
entity testbench is
end entity testbench
architecture behavior of testbench is
  component dff
    port( D : in  std_logic;
          Q : out std_logic;
          clk : in  std_logic);
  end component;
  signal D,Q,clk std_logic;
begin
  uut : dff
  port map( D =>D,
            Q =>Q,
            clk =>clk);
```



Instanciación de la  
unidad bajo prueba.

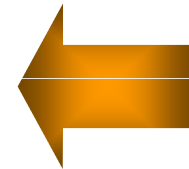
# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

---

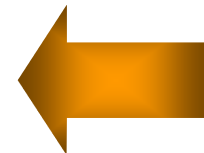
VHDL para simulación

tb:

```
process()  
begin  
    D<='0';  
    wait for 1 us;  
    assert Q='0' report "Falla" severity error;  
    D<='1';  
    wait for 1 us;  
    assert Q='1' report "Falla" severity error;  
    wait;  
end process;  
clk_gen:  
process()  
    clk <='0';  
    wait for 0.5 us;  
    clk <='1';  
    wait for 0.5 us;  
end process;  
end architecture behavior;
```



Proceso de prueba.



Proceso para la  
generación de reloj.

# INTRODUCCIÓN A LA LÓGICA PROGRAMABLE

---

**¡Muchas Gracias!**

**¿Preguntas?**