



Tema 1: Diseño y modelado hardware con VHDL

Tecnología y Organización de Computadores

Grado en Ingeniería Informática

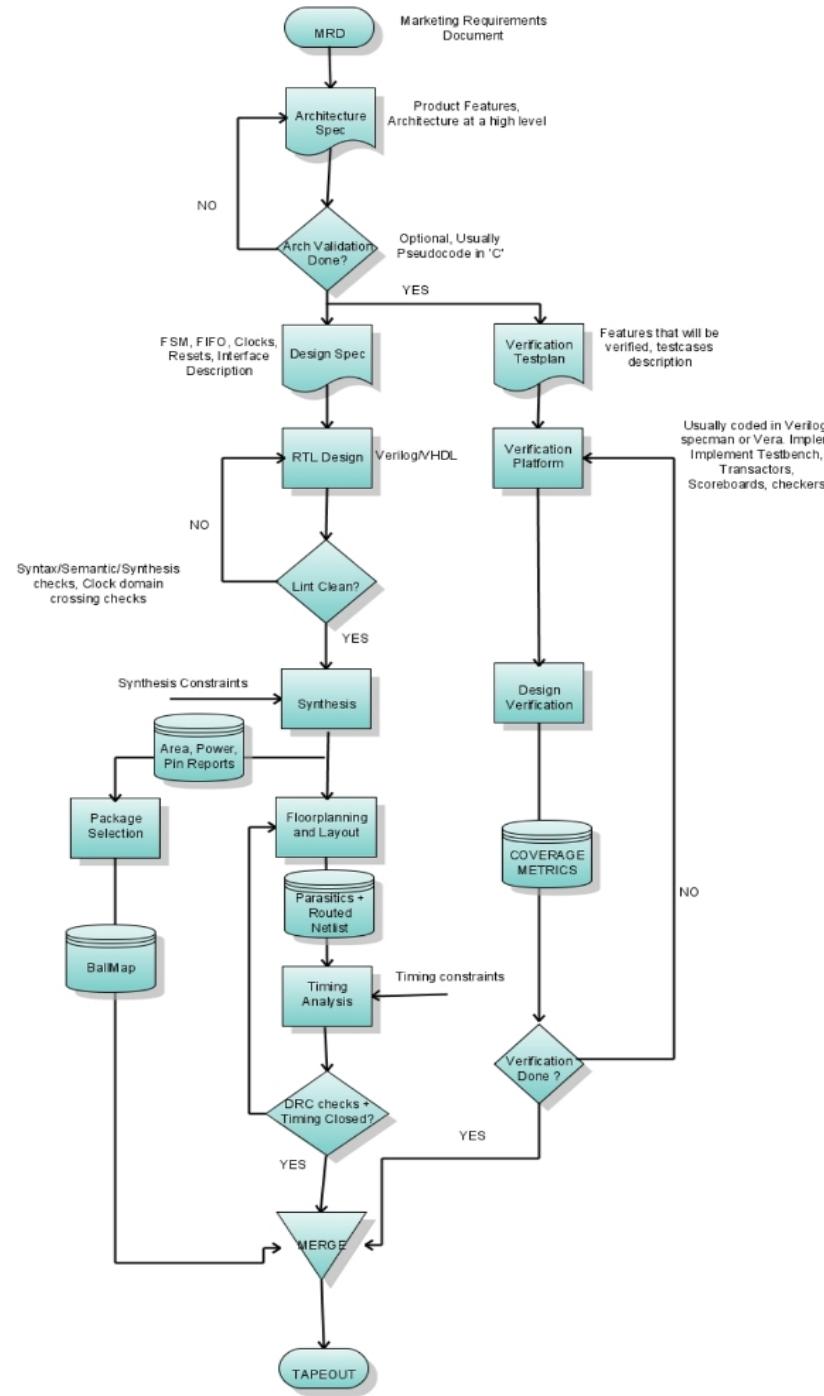
Grado en Ingeniería de Computadores

Índice



1. Flujo de diseño
2. Lenguajes de descripción hardware
3. Simulación con VHDL
4. Estructura de un modelo VHDL
5. Elementos básicos de VHDL
6. Procesos
7. Temporización
8. Operadores básicos
9. Aritmética en VHDL
10. Diseño estructural
11. Lógica secuencial: FSM
12. Paquetes
13. Testbench

Sección 1.1 Flujo de diseño





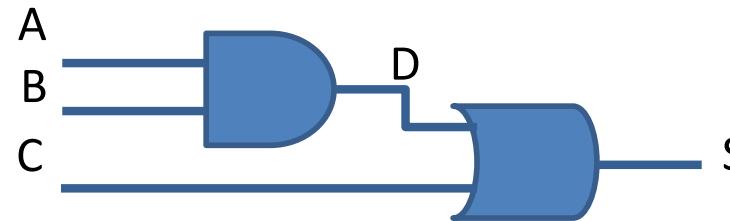
Lenguajes de Descripción Hardware

- ¿Qué es HDL?
 - Lenguaje específicamente creado para el diseño de circuitos:
 - » Nivel de puerta (gate level).
 - » Nivel de comportamiento (behavioral level).
 - La estructura del lenguaje sugiere el diseño hardware.
- ¿Por qué usar HDL?
 - Verificación temprana: descubrir problemas en el diseño antes de su implementación física.
 - Mejorar la productividad del proceso de diseño.
 - Interoperabilidad de software y metodologías.



¿Por qué son necesarios?

- HDL tiene que ser capaz de simular el comportamiento real del HW sin que el programador necesite imponer restricciones.



A and B
D or C



¿Por qué son necesarios?

$t = 5\text{ns}$	$t = 10\text{ns}$
$A = 0$	$A = 1$
$B = 1$	$B = 1$
$C = 0$	$C = 0$

Descripción 1

$D = A \text{ and } B;$
 $S = D \text{ or } C;$

Descripción 2

$S = D \text{ or } C;$
 $D = A \text{ and } B;$

¿Se obtiene el mismo resultado?



Más usados

- VHDL
 - VHSIC (Gobierno de EE.UU. 1980).
 - IEEE VHDL'87.
 - www.vhdl.org
- Verilog
 - Desarrollado por CADENCE.
 - IEEE 1364.
 - www.eda.org
- SystemVerilog
 - Descripción hardware y verificación.
 - Extensión IEEE 1364
 - www.systemverilog.org

VHDL



- Características
 - Modelar sistemas altamente concurrentes
 - Distintos tipos de descripción:
 - Estructural
 - Descomposición e interconexión de sub-circuitos
 - Comportamiento
 - Permite la especificación de la funcionalidad de un circuito utilizando formas familiares de lenguajes de programación
 - Permite la simulación del circuito antes de su fabricación
 - Verificar y comparar alternativas de diseño sin necesidad de prototipos hardware



Fases simulación

- VHDL realiza la simulación siguiendo la técnica de **simulación por eventos discretos** (*Discrete Event Time Model*).
 - Permite avanzar el tiempo a intervalos variables, en función de la planificación de ocurrencia de eventos
- La simulación consta de tres fases:
 - Fase 0 (fase de inicialización): a las señales se les asignan valores iniciales y se pone el tiempo interno de simulación a cero ($t = 0$). La asignación se hace rellenando una lista de eventos para $t = 0$.
 - Fase 1: todas las transiciones planificadas para ese tiempo son ejecutadas
 - Fase 2: Las señales que se han modificado como consecuencia de las transiciones planificadas en el instante t se escriben en la lista de eventos planificándose para el instante $t + \delta$. Donde δ es un incremento infinitesimal.



Ejemplo I

```
d <= a and b;  
s <= d or c;
```

Valores iniciales

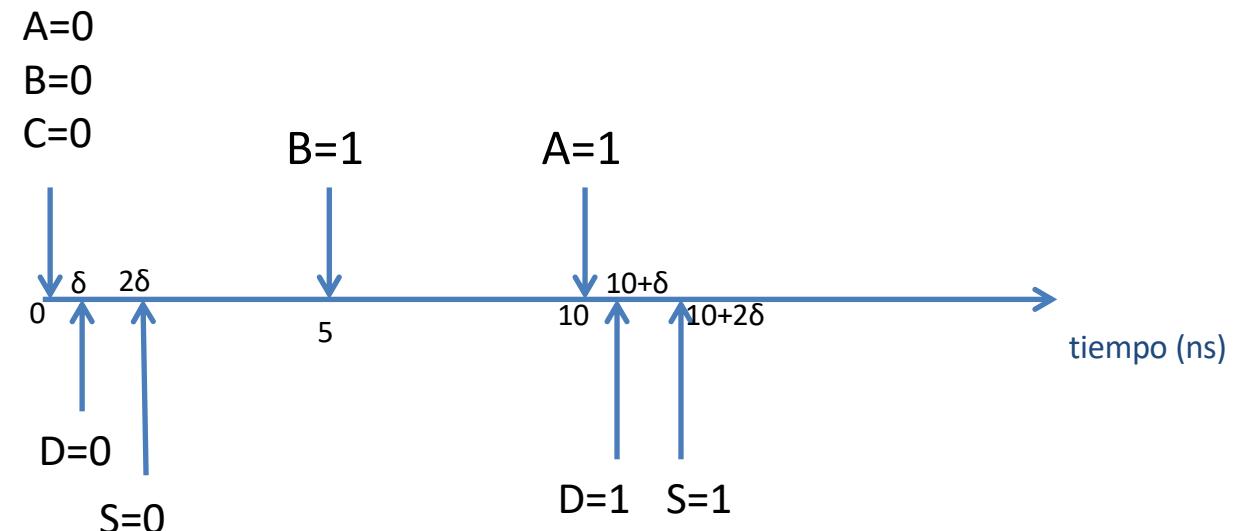
A=U

B=U

C=U

D=U

S=U





Ejemplo II

```
s <= d or c;  
d <= a and b;
```

Valores iniciales

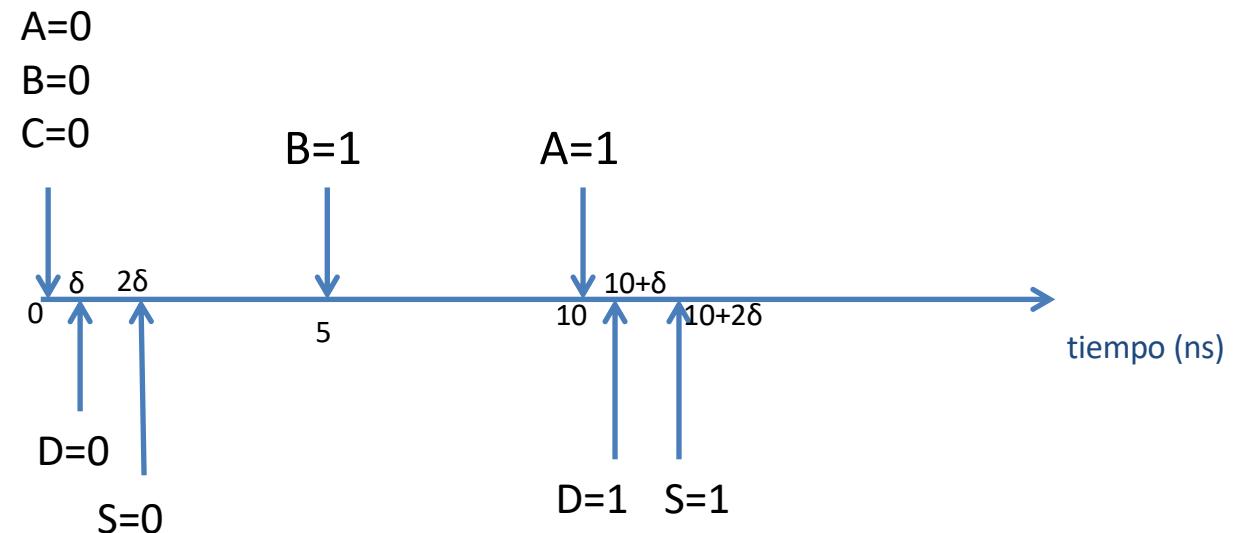
A=U

B=U

C=U

D=U

S=U





Ejemplo III

$C \leq B;$

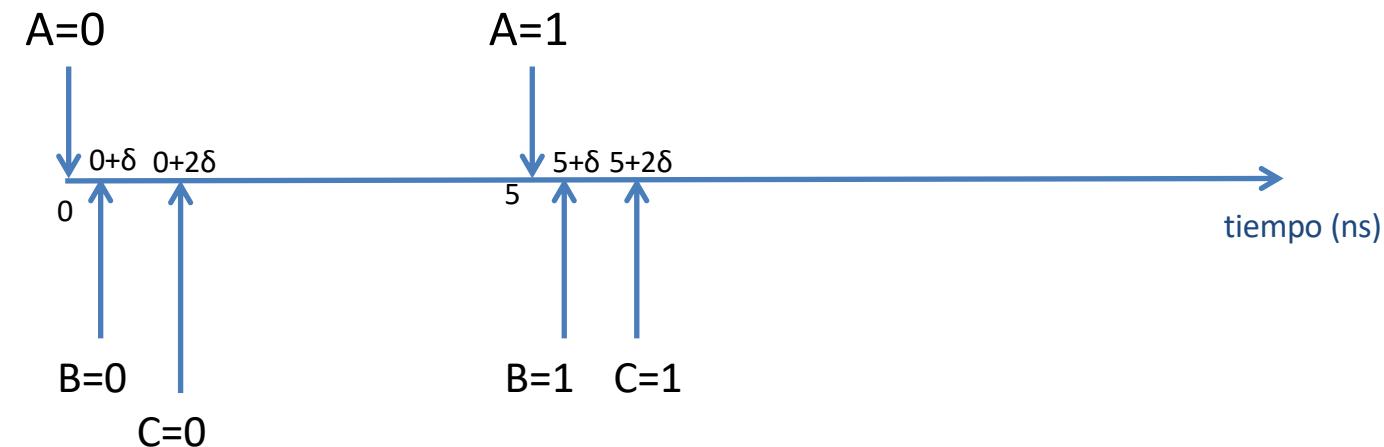
$B \leq A;$

Valores iniciales

$A=U$

$B=U$

$C=U$



$t(ns)$	0	$0 + \delta$	$0 + 2\delta$	No hay más cambios	5	$5 + \delta$	$5 + 2\delta$	No hay más cambios
A	0	0	0		1	1	1	
B	U	0	0		0	1	1	
C	U	U	0		0	0	1	



Ejemplo IV

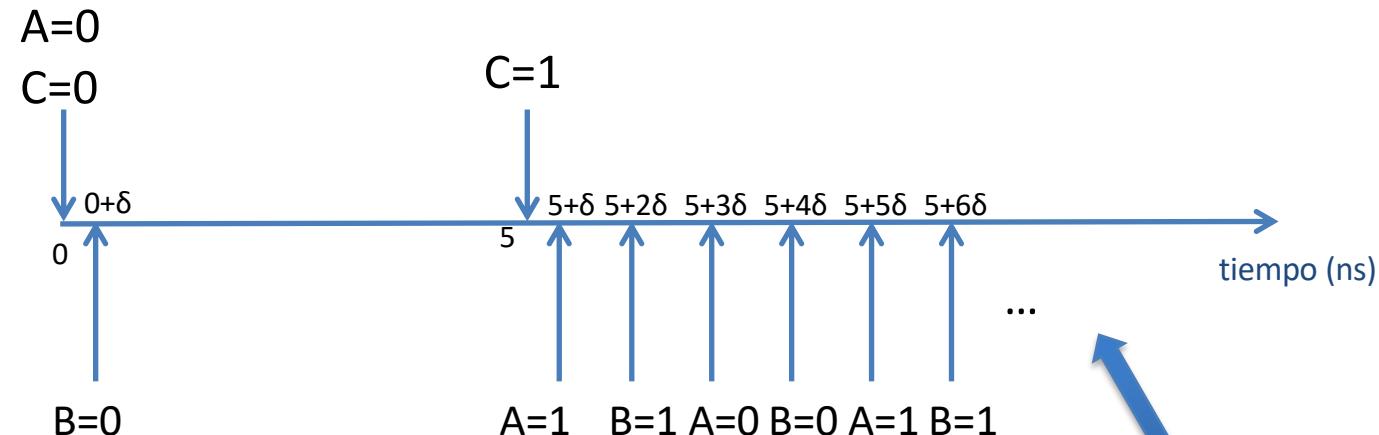
```
A <= C xor B;  
B <= A;
```

Valores iniciales

A=U

B=U

C=U

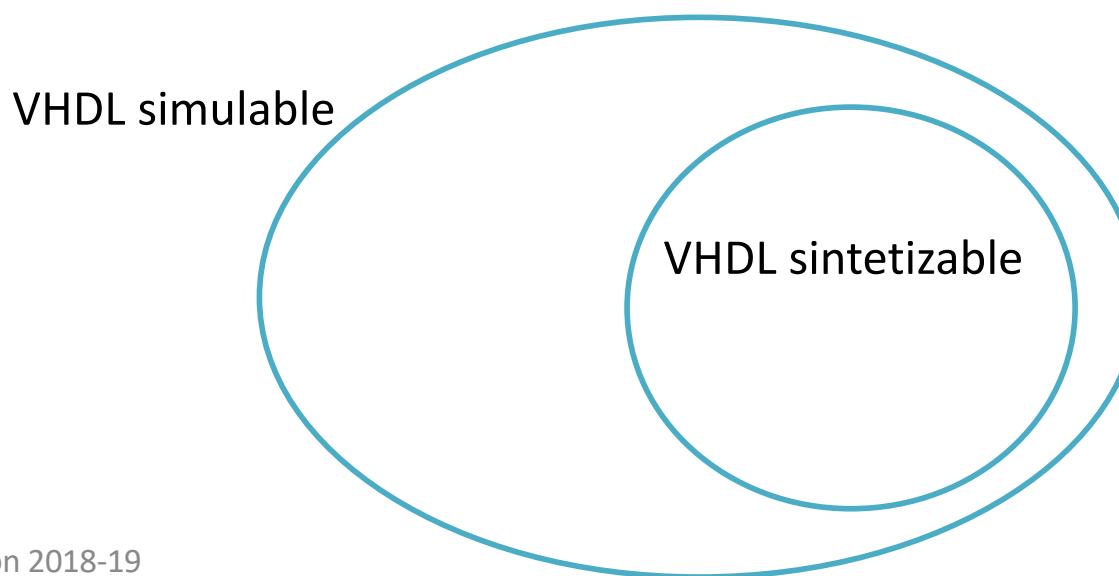


Oscila indefinidamente



Simulable vs sintetizable

- **Cualquier código VHDL correcto es simulable pero ...**
cualquier código NO es sintetizable
 - Si queremos que nuestra descripción del circuito sea sintetizable debemos usar construcciones y estilos de descripción que permitan a las herramientas inferir la lógica del circuito





Descripción de un sistema digital

- Un modelo VHDL está formado por dos partes
 1. Definición de la entidad (**entity**): descripción de los interfaces: entradas y salidas
 1. Nombre y tipos de los puertos de entrada y salida
 2. Contiene la información para conectar el sistema con otros circuitos
 3. Tiene la información necesaria para conectar el circuito a otros circuitos
 2. Descripción de la arquitectura (**architecture**): cómo se calculan las salidas a partir de las entradas
 - Define la funcionalidad del circuito
 - Contiene señales, funciones, procesos, ...

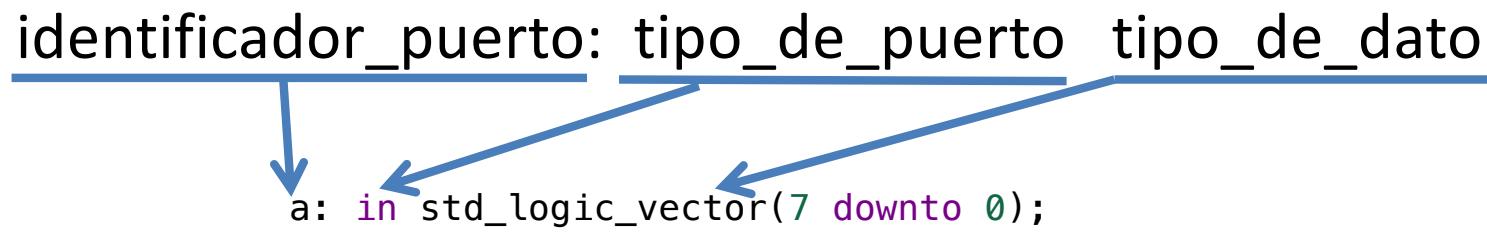


Entidad. Ejemplos

Ejemplo 1

```
1 entity ex1 is
2     port (a: in std_logic_vector(7 downto 0);
3             b: out std_logic_vector(7 downto 0);
4             c: out std_logic_vector(7 downto 0));
5         );
6 end ex1;
```

Definición de los puertos



Ejemplo 2

```
20 entity ex2 is
21     generic(width : natural := 8);
22     port (a: in std_logic_vector(width-1 downto 0);
23             b: out std_logic_vector(width-1 downto 0));
24         );
25 end ex2;
```

Mediante un genérico se pueden parametrizar los anchos de los puertos



Arquitectura (I)

```
nombre_arquitectura
16 architecture rtl of ex1 is
17   -- Definiciones de:
18   -- * tipos
19   -- * señales
20   -- * componentes
21
22 begin
23   -- sentencias concurrentes
24   -- procesos
25   -- componentes
26 end architecture rtl;
```

nombre_entidad

Parte declarativa arquitectura

Parte procedural arquitectura



Arquitectura (II)

- El código VHDL propiamente dicho se escribe dentro de ***architecture***.
- Cada ***architecture*** va asociada a una ***entity*** y se indica en la primera sentencia.
- A continuación, y antes de ***begin*** se definen:
 - Señales
 - Tipos
 - Componentes: otros circuitos ya definidos y compilados de los cuales conocemos su interfaz en VHDL.
- Desde ***begin*** hasta ***end*** escribiremos todas las sentencias propias de VHDL
 - No todas pueden utilizarse en cualquier parte del código.



Entidad y arquitectura. Ejemplo

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

entity ex2 is
    generic(width : natural := 8);
    port (a : in std_logic_vector(width-1 downto 0);
          b : out std_logic_vector(width-1 downto 0)
        );
end ex2;

architecture rtl of ex2 is
begin
    b <= not(a);
end architecture rtl;
```



Tipos de objetos

- Un objeto es un elemento (*item*) con nombre en un modelo VHDL. El tipo de objeto define la naturaleza y el comportamiento del elemento.
- Antes de poder usarse, los objetos deben ser declarados
- Tipos de objetos:
 - Señales
 - Variables
 - Constantes
 - Archivos



Señales

- En el diseño, representan elementos de memoria o conexiones y en el VHDL sirven para comunicar las sentencias concurrentes del lenguaje
- Pueden usarse en cualquier linea de código dentro de la arquitectura. Los puertos de una entidad son señales
- El operador de asignacion <=
- Están sujetas al modelo de temporizacion basado en deltas
- Se pueden usar señales en el lado de la derecha de asignaciones de señal o de variables
- Para declararlas se usa la palabra reservada **signal**
 - No es recomendable inicializar sus valores

```
signal a1 : std_logic_vector(7 downto 0);
```

↑ ↑
identificador_señal tipo_de_dato



Señales. Ejemplo

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

entity ex3 is
    port (op1 : in std_logic_vector(7 downto 0);
          op2 : in std_logic_vector(7 downto 0);
          op3 : in std_logic_vector(7 downto 0);
          op4 : in std_logic_vector(7 downto 0);
          sal : out std_logic_vector(7 downto 0)
        );
end ex3;

architecture rtl of ex3 is
    signal z : std_logic_vector(7 downto 0);
    signal add : std_logic_vector(7 downto 0);
begin
    z <= op1 or op2;
    add <= z and op3;
    sal <= add xor op4;
end architecture rtl;
```

Puertos son señales

Declaración de señales

Asignación de señales



Atributos de las señales

- **S'DELAYED(t)** is the signal value of S at time now - t .
- **S'STABLE** is true if no event is occurring on signal S.
- **S'STABLE(t)** is true if no even has occurred on signal S for t units of time.
- **S'QUIET** is true if signal S is quiet. (no event this simulation cycle)
- **S'QUIET(t)** is true if signal S has been quiet for t units of time.
- **S'TRANSACTION** is a std_logic signal, the inverse of previous value each cycle S is active.
- **S'EVENT** is true if signal S has had an event this simulation cycle.
- **S'ACTIVE** is true if signal S is active during current simulation cycle.
- **S'LAST_EVENT** is the time since the last event on signal S.
- **S'LAST_ACTIVE** is the time since signal S was last active.
- **S'LAST_VALUE** is the previous value of signal S.



Variables

- No representan conexiones o estados de memoria
- Se pueden utilizar como índices en instrucciones de bucle o modelar componentes.
- Sólo pueden estar en los procesos.
- El operador de asignación :=
- No pueden compartirse entre procesos*
- No están sujetas al modelo de temporización basado en deltas
- Se pueden usar señales en el lado de la derecha de asignaciones de señal o de variables
- Se declaran usando la palabra reservada **variable**
 - Es recomendable inicializarlas

```
variable a2 : std_logic_vector(7 downto 0) := (other => '0');
```

(*) Existen las variables compartidas (shared) pero no se van a utilizar en este curso.



Constantes

- Mecanismo para proporcionar un nombre y un tipo para un valor.
 - Sólo pueden estar en los procesos.
 - El operador de asignación :=
 - No están sujetas al modelo de temporización basado en deltas
 - Se pueden usar señales en el lado de la derecha de asignaciones de señal o de variables
 - Se declaran usando la palabra reservada **constant**
 - Es recomendable inicializarlas
- ```
constant ancho_byte : natural := 4;
constant ancho_bit : natural := ancho_byte * 8;
```



# Tipos de datos escalares

- Tipo: conjunto de valores con sus posibles operaciones
- Tipos escalares
  - **integer**: sintetizable. Su tamaño depende de la herramienta de síntesis.
  - **natural**: sintetizable. Su tamaño depende de la herramienta de síntesis.
  - **real**: no son sintetizables.
  - Tipos físicos
    - **time**: no es sintetizable. Ej.: `constant periodo : time := 23 ns;`
  - Tipos enumerados: conjunto de posibles valores
    - **bit**: {'0', '1'}
    - **boolean** : {true, false}
    - **character** : {ascii}
    - **std\_logic** : {'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-'}

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



# Tipos de datos escalares

- Definición nuevos tipos y subtipos

- Tipos

identificador      definición\_tipo

```
type state is (st0, st1, st2);
type int_4b is range 0 to 15;
type month is range 1 to 12;

variable fsm : state := st0;
variable init_val : int_4b := 7;
```

Definición del rango: n\_min to n\_max  
n\_max downto n\_min

- Subtipos. Ejemplo

definición\_tipo

```
subtype short_int is integer range 0 to 255;
```



# Tipos de datos compuestos

- **std\_logic\_vector**
  - Requiere usar la librería del IEEE

```
library IEEE;
use IEEE.std_logic_1164.all;
```
- **Arrays. Ejemplos**
  - Por ejemplo, array de bits

| Rango del array                                          | tipo |
|----------------------------------------------------------|------|
| <pre>type mem_word is array (0 to 63) of bit;</pre>      |      |
| <pre>type example1 is array (127 downto 0) of bit;</pre> |      |



# Tipos de datos compuestos

- Arrays. Ejemplos

- Rango del array usando un tipo enumerado

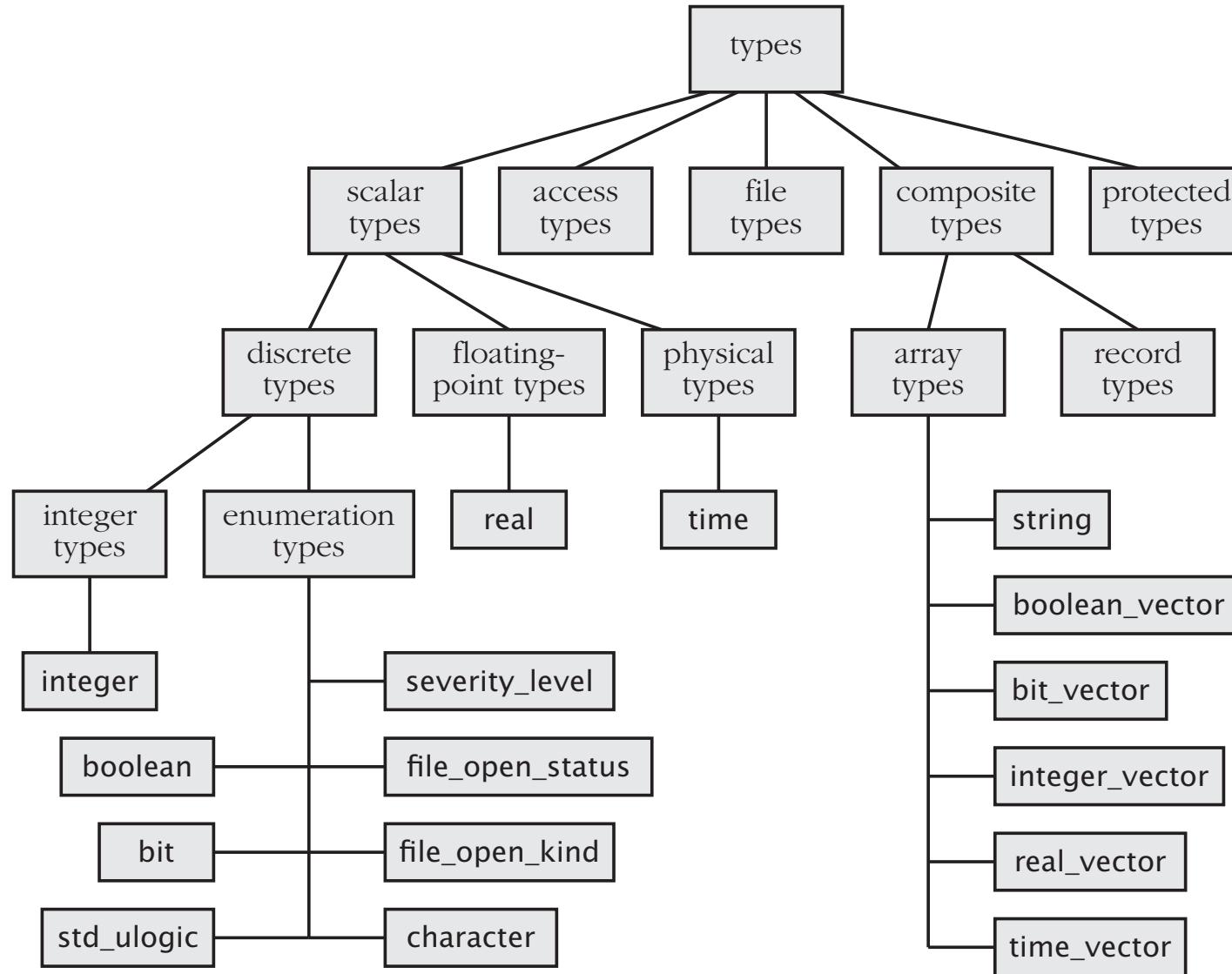
```
type state is (st0, st1, st2);
type state_array is array (st0 to st2) of natural;
```

- Rango usando tipo enumerado y tipo base usando subtipos

```
subtype short_int is integer range 0 to 127;
type state is (st0, st1, st2);
type short_array is array(short_int) of state;
```



# Tipos de datos



Fuente: Ashenden, P. J. *The Designer's Guide to Vhdl*, 3rd ed.; The Morgan Kaufmann Series in Systems on Silicon; Morgan Kaufmann: 2008.



# Sentencias concurrentes (I)

- Siempre fuera de los procesos.
- Asignación de valor a una señal.
- Aparecen en cualquier punto del programa (después del begin de la arquitectura).
- Sintetizadas como lógica combinacional pura.
- Siempre tienen un *else final* o un *when others*.
- Sintaxis

```
identificar_señal <= valor_1 when condicion_1 else
 valor_2 when condicion_2 else
 ...
 valor_n when condicion_n else ← OBLIGATORIO
 valor_por_defecto: ←
```



# Sentencias concurrentes (II)

## Ejemplos

```
entity ex4 is
 port(entrada : in std_logic_vector(4 downto 0);
 salida : out std_logic_vector(4 downto 0)
);
end entity ex4;
```

```
architecture rtl of ex4 is
begin
 salida <= "00" when entrada = "0001" else
 "01" when entrada = "0010" else
 "10" when entrada = "0100" else
 "11";
end architecture rtl;
```

Circuito combinacional



```
entity mux2a1 is
 port(op1 : in std_logic_vector(4 downto 0);
 op2 : in std_logic_vector(4 downto 0);
 sel : in std_logic;
 mux : out std_logic_vector(4 downto 0)
);
end entity mux2a1;
```

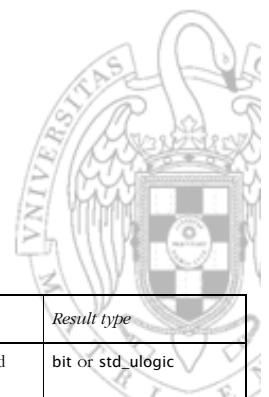
```
architecture rtl of mux2a1 is
begin
 mux <= op1 when sel = '1' else
 op2;
end architecture rtl;
```



# Operadores básicos (I)

| <i>Operator</i>                                                                   | <i>Operation</i>                                                                                                                                                                    | <i>Left operand type</i>  | <i>Right operand type</i>                                                      | <i>Result type</i>      |
|-----------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------|--------------------------------------------------------------------------------|-------------------------|
| <b>**</b>                                                                         | exponentiation                                                                                                                                                                      | integer or floating-point | integer                                                                        | same as left operand    |
| <b>abs</b>                                                                        | absolute value                                                                                                                                                                      |                           | numeric                                                                        | same as operand         |
| <b>not</b>                                                                        | logical negation                                                                                                                                                                    |                           | boolean, bit, std_ulogic,<br>1-D array of boolean or<br>bit, std_ulogic_vector | same as operand         |
| <b>and</b><br><b>or</b><br><b>nand</b><br><b>nor</b><br><b>xor</b><br><b>xnor</b> | logical and reduction<br>logical or reduction<br>negated logical and<br>reduction<br>negated logical or<br>reduction<br>exclusive or reduction<br>negated exclusive or<br>reduction |                           | 1-D array of boolean or<br>bit, std_ulogic_vector                              | element type of operand |

Fuente: Ashenden, P. J. *The Designer's Guide to Vhdl*, 3rd ed.; The Morgan Kaufmann Series in Systems on Silicon; Morgan Kaufmann: 2008.



# Operadores básicos (II)

| <i>Operator</i> | <i>Operation</i>         | <i>Left operand type</i>                                                          | <i>Right operand type</i>                                                                   | <i>Result type</i>                                                             |
|-----------------|--------------------------|-----------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------|
| *               | multiplication           | integer or floating-point<br>physical<br><b>integer or real</b>                   | same as left operand<br><b>integer or real</b><br>physical                                  | same as operands<br>same as left operand<br>same as right operand              |
| /               | division                 | integer or floating-point<br>physical<br>physical                                 | same as left operand<br>integer or real<br>same as left operand                             | same as operands<br>same as left operand<br>universal integer                  |
| <b>mod</b>      | modulo                   | integer or physical                                                               | same as left operand                                                                        | same as operands                                                               |
| <b>rem</b>      | remainder                |                                                                                   |                                                                                             |                                                                                |
| +               | identity                 |                                                                                   | numeric                                                                                     | same as operand                                                                |
| -               | negation                 |                                                                                   |                                                                                             |                                                                                |
| +               | addition                 | numeric                                                                           | same as left operand                                                                        | same as operands                                                               |
| -               | subtraction              |                                                                                   |                                                                                             |                                                                                |
| &               | concatenation            | 1-D array<br>1-D array<br>element type of right operand<br>element type of result | same as left operand<br>element type of left operand<br>1-D array<br>element type of result | same as operands<br>same as left operand<br>same as right operand<br>1-D array |
| <b>sll</b>      | shift-left logical       | 1-D array of <b>boolean</b> or <b>bit, std_ulogic_vector</b>                      | integer                                                                                     | same as left operand                                                           |
| <b>srl</b>      | shift-right logical      |                                                                                   |                                                                                             |                                                                                |
| <b>rol</b>      | rotate left              |                                                                                   |                                                                                             |                                                                                |
| <b>ror</b>      | rotate right             |                                                                                   |                                                                                             |                                                                                |
| <b>sla</b>      | shift-left arithmetic    | 1-D array of <b>boolean</b> or <b>bit</b>                                         | integer                                                                                     | same as left operand                                                           |
| <b>sra</b>      | shift-right arithmetic   |                                                                                   |                                                                                             |                                                                                |
| =               | equality                 | any except file or protected type                                                 | same as left operand                                                                        | <b>boolean</b>                                                                 |
| /=              | inequality               |                                                                                   |                                                                                             |                                                                                |
| <               | less than                | scalar or 1-D array of any discrete type                                          | same as left operand                                                                        | <b>boolean</b>                                                                 |
| <=              | less than or equal to    |                                                                                   |                                                                                             |                                                                                |
| >               | greater than             |                                                                                   |                                                                                             |                                                                                |
| >=              | greater than or equal to |                                                                                   |                                                                                             |                                                                                |

| <i>Operator</i> | <i>Operation</i>                  | <i>Left operand type</i>                                                 | <i>Right operand type</i> | <i>Result type</i>       |
|-----------------|-----------------------------------|--------------------------------------------------------------------------|---------------------------|--------------------------|
| ?=              | matching equality                 | bit, std_ulogic or 1-D array of bit or std_ulogic                        | same as left operand      | <b>bit or std_ulogic</b> |
| ?/=             | matching inequality               |                                                                          |                           |                          |
| ?<              | matching less than                | bit or std_ulogic                                                        | same as left operand      | <b>bit or std_ulogic</b> |
| ?<=             | matching less than or equal to    |                                                                          |                           |                          |
| ?>              | matching greater than             |                                                                          |                           |                          |
| ?>=             | matching greater than or equal to |                                                                          |                           |                          |
| <b>and</b>      | logical and                       | boolean, bit, std_ulogic, 1-D array of boolean or bit, std_ulogic_vector | same as left operand      | same as operands         |
| <b>or</b>       | logical or                        |                                                                          |                           |                          |
| <b>nand</b>     | negated logical and               |                                                                          |                           |                          |
| <b>nor</b>      | negated logical or                |                                                                          |                           |                          |
| <b>xor</b>      | exclusive or                      |                                                                          |                           |                          |
| <b>xnor</b>     | negated exclusive or              |                                                                          |                           |                          |
| ??              | condition conversion              |                                                                          | <b>bit or std_ulogic</b>  | <b>boolean</b>           |

Fuente: Ashenden, P. J. *The Designer's Guide to Vhdl*, 3rd ed.; The Morgan Kaufmann Series in Systems on Silicon; Morgan Kaufmann: 2008.



# Procesos

- Sentencias concurrentes “vitaminadas”



```
<etiqueta> : process (lista de sensibilidad)
begin
 -- sentencias secuenciales
 -- sentencias condicionales
 -- bucles
end process <etiqueta>;
```

Sólo se ejecutan las sentencias que se encuentran dentro del process si alguna de las señales de la lista de sensibilidad cambia de valor

- Sólo se ejecutan las sentencias que se encuentran dentro del proceso si alguna de las señales de la lista de sensibilidad ha cambiado de valor.
- El código dentro del proceso se ejecuta **secuencialmente**
- El proceso es concurrente con:
  - El resto de procesos
  - Las sentencias concurrentes de la architecture



# Procesos

- Diferencias en la temporización de las sentencias dentro y fuera de un proceso

```
architecture rtl of ex is
 signal b : std_logic;
begin

 p_wire : process (a)
 begin
 b <= a;
 c <= b;
 end process p_wire;

end architecture rtl;
```

Sentencias secuenciales

```
architecture rtl of ex is
 signal b : std_logic;
begin
 b <= a;
 c <= b;
end architecture rtl;
```

Sentencias concurrentes



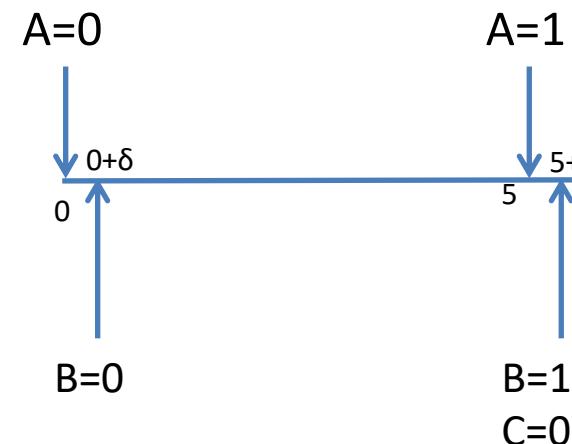
# Simulación de un proceso (I)

Valores iniciales

A=U

B=U

C=U



```
architecture rtl of ex is
 signal b : std_logic;
begin
 p_wire : process (a)
 begin
 b <= a;
 c <= b;
 end process p_wire;
end architecture rtl;
```





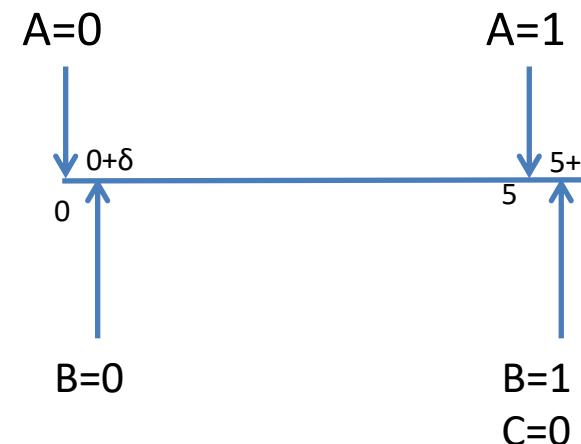
# Simulación de un proceso (II)

Valores iniciales

A=U

B=U

C=U



```
architecture rtl of ex is
 signal b : std_logic;
begin
```

```
 p_wire : process (a)
 begin
 c <= b;
 b <= a;
 end process p_wire;
```

```
end architecture rtl;
```



# Simulación de un proceso (III)

- Las variables se definen en los procesos

Valores iniciales

A=U

B=U

C=U

A=0

0+ $\delta$

B=0

C=0

A=1

5+ $\delta$

B=1

C=1

```
architecture rtl of ex is
 signal b : std_logic;
begin
 p_wire : process (a)
 variable d, e : std_logic_vector;
 begin
 d := a;
 e := d;
 c <= d;
 b <= e;
 end process p_wire;
end architecture rtl;
```



# Sentencias secuenciales

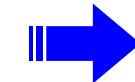
- VHDL permite utilizar otro tipo de sentencias condicionales más parecidas a los lenguajes de programación usados
- Se utilizan siempre dentro de una estructura process. ¿Por qué?
- Como veremos mas adelante sentencias condicionales incompletamente descritas dan lugar a HW secuencial



# Sentencias condicionales: if

## Plantilla

```
if <cond_l> then
 sequential statements
elsif <cond-n> then
 sequential statements
else
 sequential statements
end if;
```



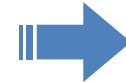
```
architecture rtl of ex is
begin
p_if_example : process (a, b)
begin
 if a >= b then
 c <= a or b;
 elsif a < b then
 c <= b;
 else
 c <= "0000";
 end if;
end process p_if_example;
end architecture rtl;
```



# Sentencias condicionales: case (I)

## Plantilla

```
case < expression > is
 when choice_l =>
 seq. statements;
 when choice_n =>
 seq. statements;
 when others =>
 seq. statements;
end case;
```



```
architecture rtl of ex is
begin
 p_case_example : process (din)
 begin
 case din is
 when "111" => r <= '0';
 when "100" => r <= '1';
 when "110" => r <= '1';
 when others => r <= '0';
 end case;
 end process p_case_example;
end architecture rtl;
```



# Sentencias condicionales. Ejemplo (I)

```
architecture rtl of ex is
begin
 p_case_example : process (din, f)
 begin
 case din is
 when "111" =>
 r <= '0';
 when "100" =>
 if f = '1' then
 r <= '1';
 else
 r <= '0';
 end if;
 when "110" =>
 if f = '0' then
 r <= '0';
 else
 r <= '1';
 end if;
 when others =>
 r <= '0';
 end case;
 end process p_case_example;
end architecture rtl;
```

## Sección 1.6 Procesos

```
entity mux8a1 is
 port(op0 : in std_logic;
 op1 : in std_logic;
 op2 : in std_logic;
 op3 : in std_logic;
 op4 : in std_logic;
 op5 : in std_logic;
 op6 : in std_logic;
 op7 : in std_logic;
 sel : in std_logic_vector(2 downto 0);
 mux : out std_logic
);
end entity mux8a1;
```

```
architecture uso_if of mux8a1 is
begin
 p_if : process (op0, sel) is
 begin
 if sel = "000" then
 mux <= op0;
 elsif sel = "001" then
 mux <= op1;
 elsif sel = "010" then
 mux <= op2;
 elsif sel = "011" then
 mux <= op3;
 elsif sel = "100" then
 mux <= op4;
 elsif sel = "101" then
 mux <= op5;
 elsif sel = "110" then
 mux <= op6;
 elsif sel = "111" then
 mux <= op7;
 end if;
 end process p_if;
end architecture uso_if;
```

```
architecture uso_case of mux2a1 is
begin
 p_case : process (op0, sel) is
 begin
 case sel is
 when "000" => mux <= op0;
 when "001" => mux <= op1;
 when "010" => mux <= op2;
 when "011" => mux <= op3;
 when "100" => mux <= op4;
 when "101" => mux <= op5;
 when "110" => mux <= op6;
 when "111" => mux <= op7;
 end case;
 end process p_case;
end architecture uso_case;
```



```
entity mux8a1_vector is
 port(op : in std_logic_vector(7 downto 0);
 sel : in std_logic_vector(2 downto 0);
 mux : out std_logic
);
end entity mux8a1_vector;
```

```
architecture uso_case of mux2a1_vector is
begin
 p_case : process (op0, sel) is
 begin
 case sel is
 when "000" => mux <= op0;
 when "001" => mux <= op1;
 when "010" => mux <= op2;
 when "011" => mux <= op3;
 when "100" => mux <= op4;
 when "101" => mux <= op5;
 when "110" => mux <= op6;
 when "111" => mux <= op7;
 end case;
 end process p_case;
end architecture uso_case;
```



# Ejemplo (II): Multiplexores





# Wait

- Sintaxis
  - [label:] **wait** [sensitive clause] [condition clause];
- Se usa dentro procesos, procedimientos y funciones
  - Si el proceso tiene lista de sensibilidad entonces **NO** puede contener wait
- Suspender la ejecución del proceso, procedimiento o función
- No siempre son sintetizables. Depende de qué se describa con ellos
  - Evitarlos en el diseño del circuito
- Tres tipos:
  - **wait for** <timeout clause, time delay>
  - **wait until** <condition>
  - **wait on** <sensitive clause, event>
- Ejemplos:
  - **wait for** 10 ns;
  - **wait until** clk='1';
  - **wait on** in1;



# Bucles

- **Bucle loop**

```
[label:] loop
 sequential statements
 exit statement to get out
end loop [label];
```

- **Bucle for**

```
[label:] for var in range loop
 sequence-of-statements
end loop [label];
```



- **Bucle while**

```
[label:] while cond loop
 sequence-of-statements
end loop [label];
```



# Señales en lista de sensibilidad

- Si el proceso describe lógica combinacional entonces la lista de sensibilidad debe contener todas las señales de las que depende la lógica.

```
architecture rtl of ex is
begin
 p_if_example : process (a, b)
 begin
 if a >= b then
 c <= a or b;
 elsif a < b then
 c <= b;
 else
 c <= "0000";
 end if;
 end process p_if_example;
end architecture rtl;
```



| $t = 5 \text{ ns}$ | $t = 10 \text{ ns}$ | $t = 15 \text{ ns}$ |
|--------------------|---------------------|---------------------|
| $a \leq "0010"$    | $a \leq "0010"$     | $a \leq "0010"$     |
| $b \leq "0010"$    | $b \leq "0110"$     | $b \leq "0001"$     |
| c                  | c                   | c                   |



# Señales en lista de sensibilidad

- Si el proceso describe lógica combinacional entonces la lista de sensibilidad debe contener todas las señales de las que depende la lógica.
- Ejemplo proceso con errores

```
architecture rtl of ex is
begin
 p_if_example : process (a, b)
 begin
 if a >= b then
 c <= a or b;
 elsif a < b then
 c <= b;
 end if;
 end process p_if_example;
end architecture rtl;
```

| $t = 5 \text{ ns}$ | $t = 10 \text{ ns}$ | $t = 15 \text{ ns}$ |
|--------------------|---------------------|---------------------|
| $a \leq "0010"$    | $a \leq "0010"$     | $a \leq "0010"$     |
| $b \leq "0010"$    | $b \leq "0110"$     | $b \leq "0001"$     |
| c                  | c                   | c                   |





# Señales en lista de sensibilidad

- Si el proceso describe lógica combinacional entonces la lista de sensibilidad debe contener todas las señales de las que depende la lógica.
- Ejemplo proceso con errores

```
architecture rtl of ex is
begin
 p_if_example : process (a)
 begin
 if a >= b then
 c <= a or b;
 elsif a < b then
 c <= b;
 else
 c <= "0000";
 end if;
 end process p_if_example;
end architecture rtl;
```

| $t = 5 \text{ ns}$ | $t = 10 \text{ ns}$ | $t = 15 \text{ ns}$ |
|--------------------|---------------------|---------------------|
| $a \leq "0010"$    | $a \leq "0010"$     | $a \leq "0010"$     |
| $b \leq "0010"$    | $b \leq "0110"$     | $b \leq "0001"$     |
| c                  | c                   | c                   |



# Detección de transiciones

- La detección de transiciones en señales es fundamental para implementar lógica secuencial
- Detección de transiciones de subida o bajada
  - En la lista de sensibilidad de los procesos
  - Usando el atributo event de la señal. P.e.: `clk'event`
- Detección de transiciones de subida:
  - Bien usando la construcción: `clk'event and clk = '1'`
  - Recomendado: Usando la función predefinida `rising_edge`  
`rising_edge(clk)`



# Detección de transiciones

- Detección de transiciones de bajada:
  - Bien usando la construcción: `clk'event and clk = '0'`
  - Recomendado: Usando la función predefinida `falling_edge(clk)`



# Ejemplos (I)

Flip-flop: biestable D disparado por flanco

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

entity ff is
port (d : in std_logic;
 clk : in std_logic;
 q : out std_logic);
end entity ff;

architecture rtl of ff is
begin
 p_ff : process (clk, d) is
 begin
 if rising_edge(clk) then
 q <= d;
 end if;
 end process p_ff;
end architecture rtl;
```

Flip-flop con reset sincrono

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

entity ff is
port (d : in std_logic;
 clk : in std_logic;
 rst : in std_logic;
 q : out std_logic);
end entity ff;

architecture rtl of ff is
begin
 p_ff : process (clk) is
 begin
 if rising_edge(clk) then
 if rst = '1' then
 q <= d;
 end if;
 end if;
 end process p_ff;
end architecture rtl;
```



# Ejemplos (II)

## Flip-flop con reset asincrono

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

entity ff is
port (d : in std_logic;
 clk : in std_logic;
 rst : in std_logic;
 q : out std_logic);
end entity ff;

architecture rtl of ff is
begin
 p_ff : process (clk) is
 begin
 if rst = '1' then
 if rising_edge(clk) then
 q <= d;
 end if;
 end if;
 end process p_ff;
end architecture rtl;
```

## Registro 8 bits con reset asincrono

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

entity reg is
port (d : in std_logic_vector(7 downto 0);
 clk : in std_logic;
 rst : in std_logic;
 q : out std_logic_vector(7 downto 0));
end entity reg;

architecture rtl of ff is
begin
 p_ff : process (clk) is
 begin
 if rst = '1' then
 if rising_edge(clk) then
 q <= d;
 end if;
 end if;
 end process p_ff;
end architecture rtl;
```



# Operadores disponibles en VHDL-93 (sin incluir ningún paquete)

| Operador               | Descripción    | Tipo de operando a          | Tipo de operando b  | Tipo del resultado          |
|------------------------|----------------|-----------------------------|---------------------|-----------------------------|
| <code>a ** b</code>    | Exponenciación | integer                     | integer             | integer                     |
| <code>abs a</code>     | Valor absoluto | integer                     |                     | integer                     |
| <code>not a</code>     | Negación       | boolean, bit,<br>bit_vector |                     | boolean, bit,<br>bit_vector |
| <code>a * b</code>     | Multiplicación | integer                     | integer             | integer                     |
| <code>a / b</code>     | División       |                             |                     |                             |
| <code>a mod b</code>   | Módulo         |                             |                     |                             |
| <code>a rem b</code>   | Resto          |                             |                     |                             |
| <code>+ a</code>       | Identidad      | integer                     | integer             | integer                     |
| <code>- a</code>       | Negación       |                             |                     |                             |
| <code>a + b</code>     | Suma           | integer                     | integer             | integer                     |
| <code>a - b</code>     | Resta          |                             |                     |                             |
| <code>a &amp; b</code> | Concatenación  | Array 1-D, elemento         | Array 1-D, elemento | Array 1-D                   |
| <code>a = b</code>     | Igual          | Cualquiera                  | Mismo que a         | boolean                     |
| <code>a /= b</code>    | No igual       |                             |                     |                             |

Integer: -(2<sup>31</sup>-1) to +(2<sup>31</sup>-1)    Natural: 0 to +(2<sup>31</sup>-1)



# Operadores disponibles en VHDL-93 (sin incluir ningún paquete) (cont.)

| Operador         | Descripción       | Tipo de operando a          | Tipo de operando b | Tipo del resultado |
|------------------|-------------------|-----------------------------|--------------------|--------------------|
| <b>a sll b</b>   | Despl. Lóg. Izq.  | bit_vector                  | integer            | bit_vector         |
| <b>a srl b</b>   | Despl. Lóg. Der.  |                             |                    |                    |
| <b>a sla b</b>   | Despl. Arit. Izq. |                             |                    |                    |
| <b>a sra b</b>   | Despl. Arit. Der. |                             |                    |                    |
| <b>a rol b</b>   | Rotación Izq.     |                             |                    |                    |
| <b>a ror b</b>   | Rotación Der.     |                             |                    |                    |
| <b>a &lt; b</b>  | Menor que         | Escalar o array 1-D         | Mismo que a        | boolean            |
| <b>a &lt;= b</b> | Menor o igual que |                             |                    |                    |
| <b>a &gt; b</b>  | Mayor que         |                             |                    |                    |
| <b>a &gt;= b</b> | Mayor o igual que |                             |                    |                    |
| <b>a and b</b>   | and               | boolean, bit,<br>bit_vector | Mismo que a        | Mismo que a        |
| <b>a or b</b>    | or                |                             |                    |                    |
| <b>a xor b</b>   | xor               |                             |                    |                    |
| <b>a nand b</b>  | nand              |                             |                    |                    |
| <b>a nor b</b>   | nor               |                             |                    |                    |
| <b>a xnor b</b>  | xnor              |                             |                    |                    |



# Operadores y funciones del paquete IEEE std\_logic\_1164

| Operador                                                                                                  | Tipo de operando a            | Tipo de operando b | Tipo del resultado |
|-----------------------------------------------------------------------------------------------------------|-------------------------------|--------------------|--------------------|
| <b>not</b> a                                                                                              | std_logic_vector<br>std_logic | -                  | Mismo que op. a    |
| a <b>and</b> b<br>a <b>or</b> b<br>a <b>xor</b> b<br>a <b>nand</b> b<br>a <b>nor</b> b<br>a <b>xnor</b> b | std_logic_vector<br>std_logic | Mismo que op. a    | Mismo que op. a    |

| Función              | Tipo de operando a | Tipo del resultado |
|----------------------|--------------------|--------------------|
| to_bit(a)            | std_logic          | bit                |
| to_stdlogic(a)       | bit                | std_logic          |
| to_bitvector(a)      | std_logic_vector   | bit_vector         |
| to_stdlogicvector(a) | bit_vector         | std_logic_vector   |



# Aritmética en VHDL



- El tipo `std_logic_vector` **NO** sirve para realizar operaciones aritméticas
- ¿Por qué? Es necesario especificar qué interpretación (con o sin signo) se le dará a cada dato
  - Binario puro
  - Complemento a 2
- El paquete `ieee.numeric_std.all` es la librería recomendada para la realización de operaciones aritméticas. La compatibilidad está garantizada sea cual sea la herramienta que utilicen
- `numeric_std` no permite la operación “`a+b`” sobre señales de tipo `std_logic_vector`.
- El paquete `ieee.numeric_std.all` describe dos nuevos tipos de datos
  - `signed`: números en complemento a 2

```
signal d : signed(127 downto 0);
```
  - `unsigned`: números sin signo

```
signal a, b : unsigned(8 downto 0);
```



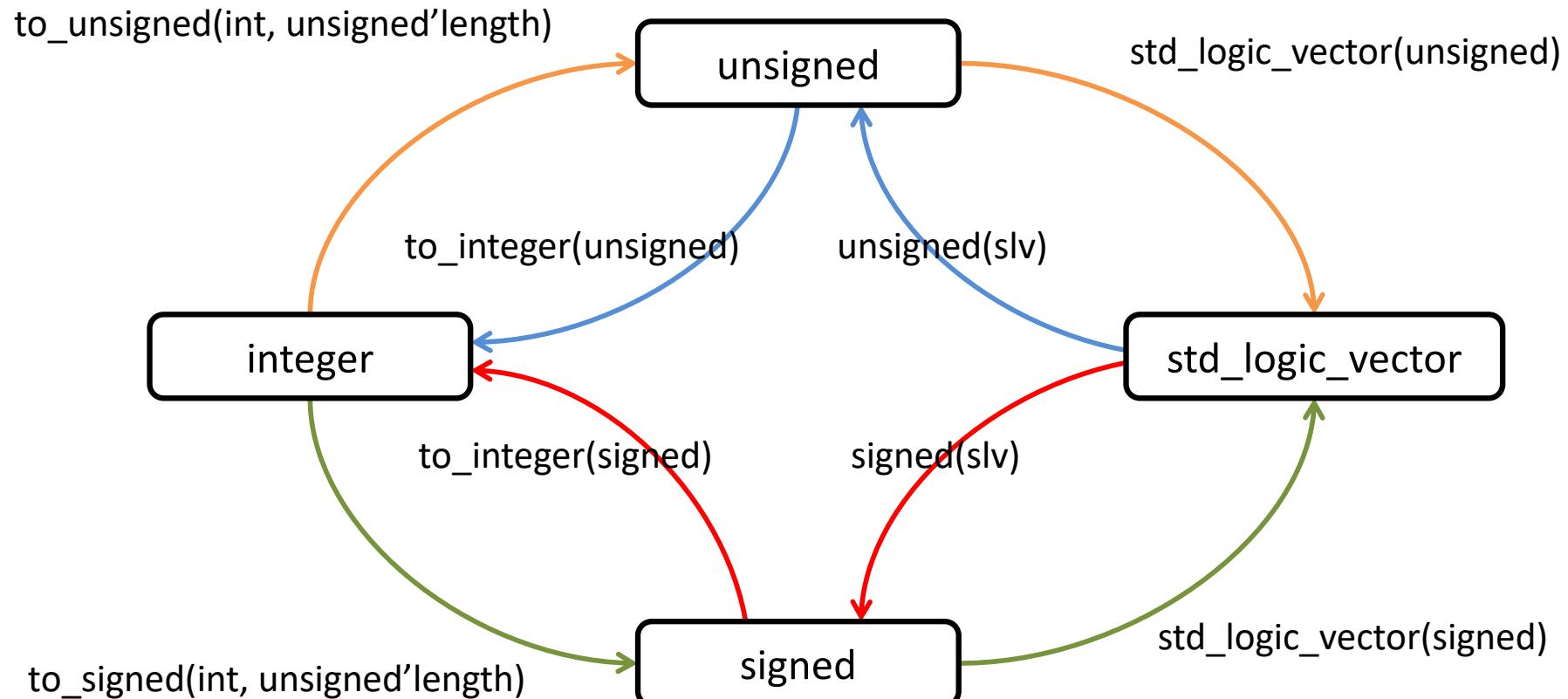
# Operadores del paquete IEEE *numeric\_std*

| Operador                                                             | Descripción                | Tipo de operando a                                         | Tipo de operando b                                         | Tipo del resultado                       |
|----------------------------------------------------------------------|----------------------------|------------------------------------------------------------|------------------------------------------------------------|------------------------------------------|
| <b>abs</b> a<br>- a                                                  | Valor absoluto<br>Negación | signed                                                     | -                                                          | signed                                   |
| a * b<br>a / b<br>a <b>mod</b> b<br>a <b>rem</b> b<br>a + b<br>a - b | Operaciones aritméticas    | unsigned<br>unsigned, natural<br>signed<br>signed, integer | unsigned, natural<br>unsigned<br>signed, integer<br>signed | unsigned<br>unsigned<br>signed<br>signed |
| a = b<br>a /= b<br>a < b<br>a <= b<br>a > b<br>a >= b                | Operaciones relacionales   | unsigned<br>unsigned, natural<br>signed<br>signed, integer | unsigned, natural<br>unsigned<br>signed, integer<br>signed | boolean<br>boolean<br>boolean<br>boolean |



# Funciones de conversión de datos / casting

- Para la conversión de señales de un tipo a otro se deben usar las siguientes funciones y operaciones de casting:





# Funciones de conversión de datos / casting

| Tipo de dato inicial          | Tipo de dato final | Función de conversión /casting |
|-------------------------------|--------------------|--------------------------------|
| unsigned, signed              | std_logic_vector   | std_logic_vector(a)            |
| signed,<br>std_logic_vector   | unsigned           | unsigned(a)                    |
| unsigned,<br>std_logic_vector | signed             | signed(a)                      |
| unsigned, signed              | integer            | to_integer(a)                  |
| natural                       | unsigned           | to_unsigned(a, size)           |
| integer                       | signed             | to_signed(a, size)             |

Chu, Pong P. "RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability"



# Ejemplos

Sumador sin signo de 8 bits

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

entity adder is
port(a : in std_logic_vector(7 downto 0);
 b : in std_logic_vector(7 downto 0);
 sum : out std_logic_vector(7 downto 0);
 co : out std_logic);
end adder;

architecture rtl of adder is
 signal a_i, b_i, sum_i : unsigned(8 downto 0); ←
begin
 a_i <= unsigned("0" & a); ←
 b_i <= unsigned("0" & b); ←
 sum_i <= a_i + b_i + ci_i; ←

 sum <= std_logic_vector(sum_i(7 downto 0)); ←
 co <= sum_i(8);
end rtl;
```



# Método

1. Declarar el componente que se desea utilizar en la parte declarativa de la arquitectura
2. Se instancia el componente en la parte procedural de la arquitectura

```
architecture arch_name of entity_name is
 ...
 component component_name is
 generic (<declaracion_genéricos>);
 port (<declaracion_puertos>);
 end component component_name;

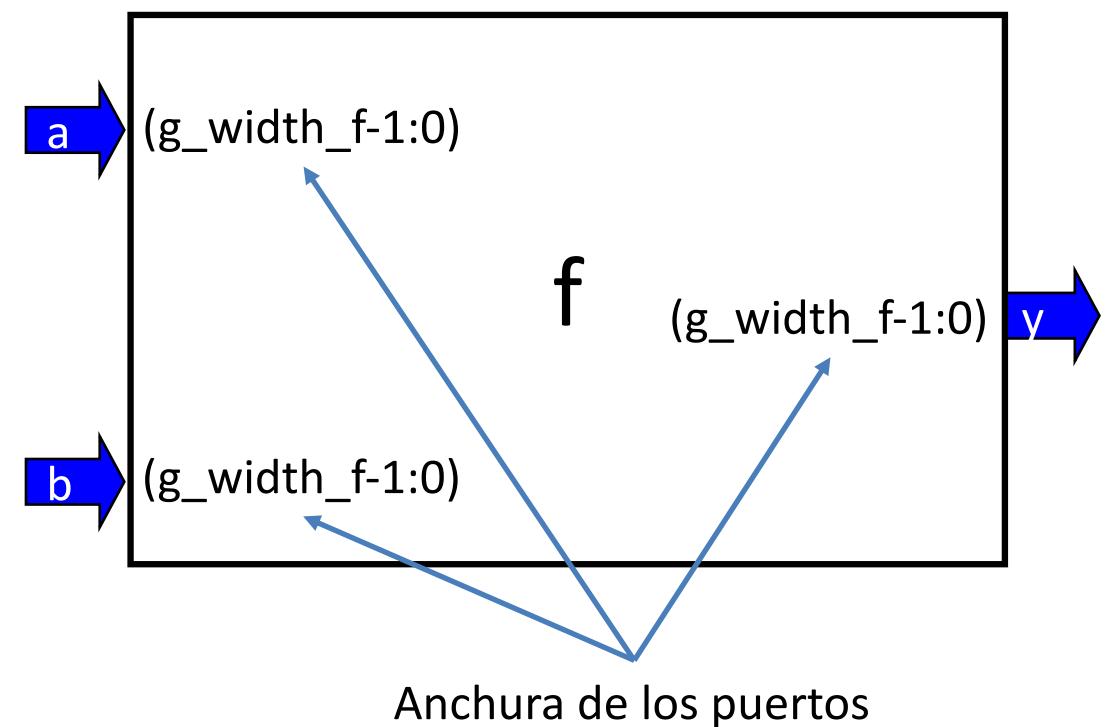
begin
 <etiqueta> : component_name
 generic map (nombre_gen1 => valor_gen1,
 nombre_gen2 => valor_gen2,
 ...
)
 port map (nombre Puerto1 => señal1,
 nombre Puerto2 => señal2,
 ...);
end arch_name;
```



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

entity f is
 generic (
 g_width_f : natural := 8);
 port (
 a : in std_logic_vector(g_width_f-1 downto 0);
 b : in std_logic_vector(g_width_f-1 downto 0);
 y : out std_logic_vector(g_width_f-1 downto 0));
end entity f;
```

# Ejemplo



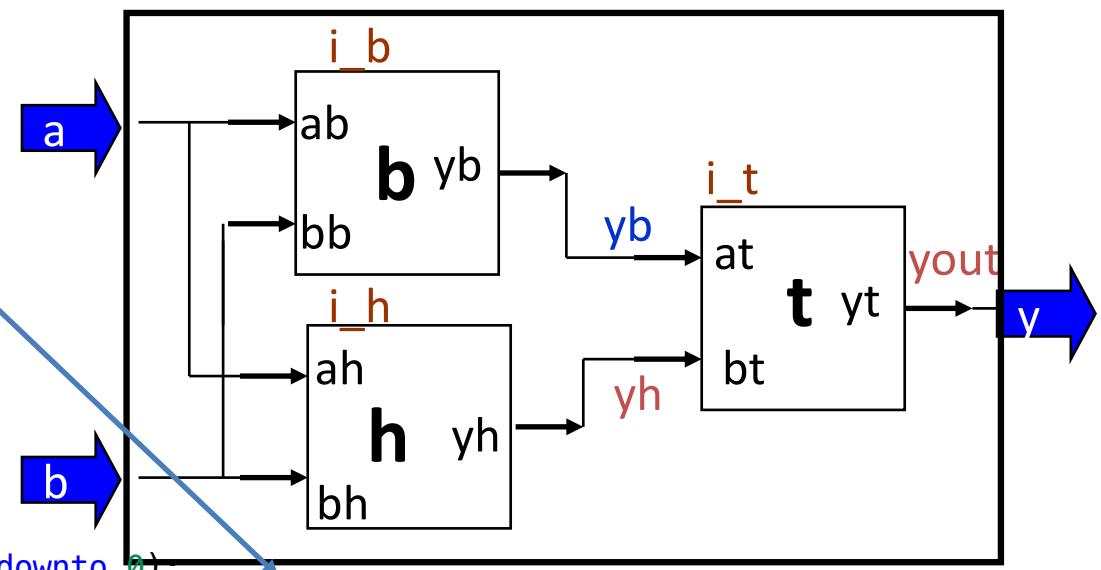


# Ejemplo

Definidos en archivos .vhdl  
diferentes

```
entity h is
 generic (
 g_width_h : natural := 4);
 port (
 ah : in std_logic_vector(g_width_h-1 downto 0),
 bh : in std_logic_vector(g_width_h-1 downto 0);
 yh : out std_logic_vector(g_width_h-1 downto 0));
end entity h;
```

```
entity b is
 generic (
 g_width_b : natural := 16);
 port (
 ab : in std_logic_vector(g_width_b-1 downto 0);
 bb : in std_logic_vector(g_width_b-1 downto 0);
 yb : out std_logic_vector(g_width_b-1 downto 0));
end entity b;
```



```
entity t is
 generic (
 g_width_t : natural := 11);
 port (
 at : in std_logic_vector(g_width_t-1 downto 0);
 bt : in std_logic_vector(g_width_t-1 downto 0);
 yt : out std_logic_vector(g_width_t-1 downto 0));
end entity t;
```

# Fiemnlo



```

architecture struct of f is
 component b is
 generic (
 g_width_b : natural);
 port (
 ab : in std_logic_vector(g_width_b-1 downto 0);
 bb : in std_logic_vector(g_width_b-1 downto 0);
 yb : out std_logic_vector(g_width_b-1 downto 0));
 end component b;

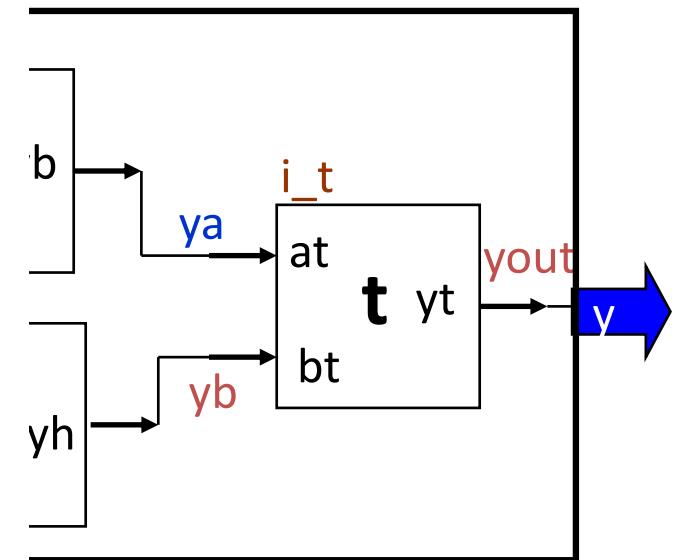
 component h is
 generic (
 g_width_h : natural);
 port (
 ah : in std_logic_vector(g_width_h-1 downto 0);
 bh : in std_logic_vector(g_width_h-1 downto 0);
 yh : out std_logic_vector(g_width_h-1 downto 0));
 end component h;

 component t is
 generic (
 g_width_t : natural);
 port (
 at : in std_logic_vector(g_width_t-1 downto 0);
 bt : in std_logic_vector(g_width_t-1 downto 0);
 yt : out std_logic_vector(g_width_t-1 downto 0));
 end component t;

 signal ya, yb : std_logic_vector(g_width_f-1 downto 0);
 signal yout : std_logic_vector(g_width_f-1 downto 0);

begin

```





# Ejemplo

```

begin
 i_b : b
 generic map (
 g_width_b => g_width_f)
 port map (
 ab => a,
 bb => b,
 yb => ya);

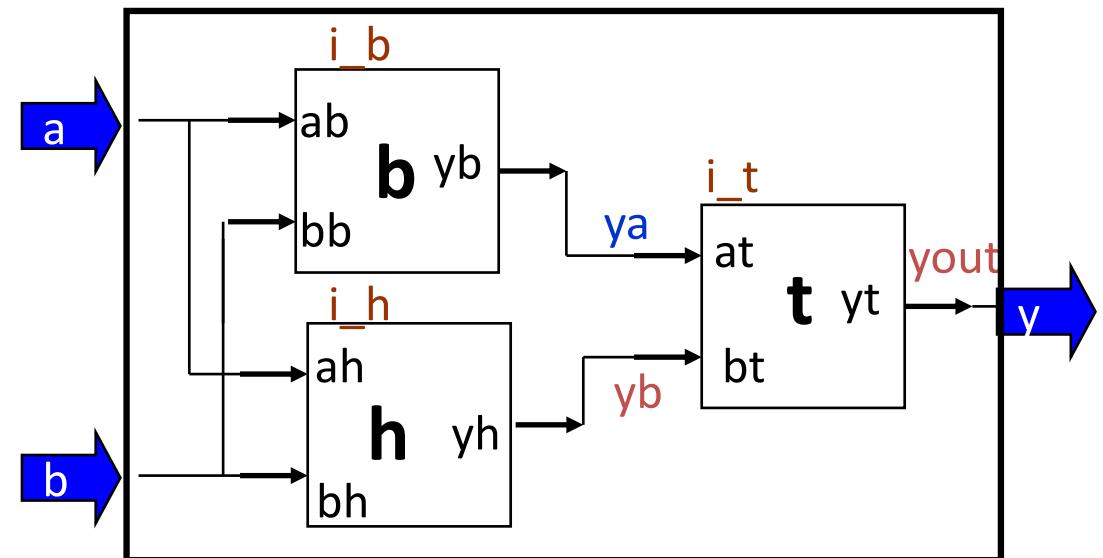
 i_h : h
 generic map (
 g_width_h => g_width_f)
 port map (
 ah => a,
 bh => b,
 yh => yb);

 i_t : t
 generic map (
 g_width_t => g_width_f)
 port map (
 at => ya,
 bt => yb,
 yt => yout);

 salida : y <= yout;

end architecture struct;

```

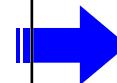




# Generate

- **GENERATE:** Las secuencias de generación de componentes permiten crear una o más copias de un conjunto de interconexiones, lo cual facilita el diseño de circuitos mediante descripciones estructurales.

```
for indice in rango generate
 -- instrucciones concurrentes
end generate;
```

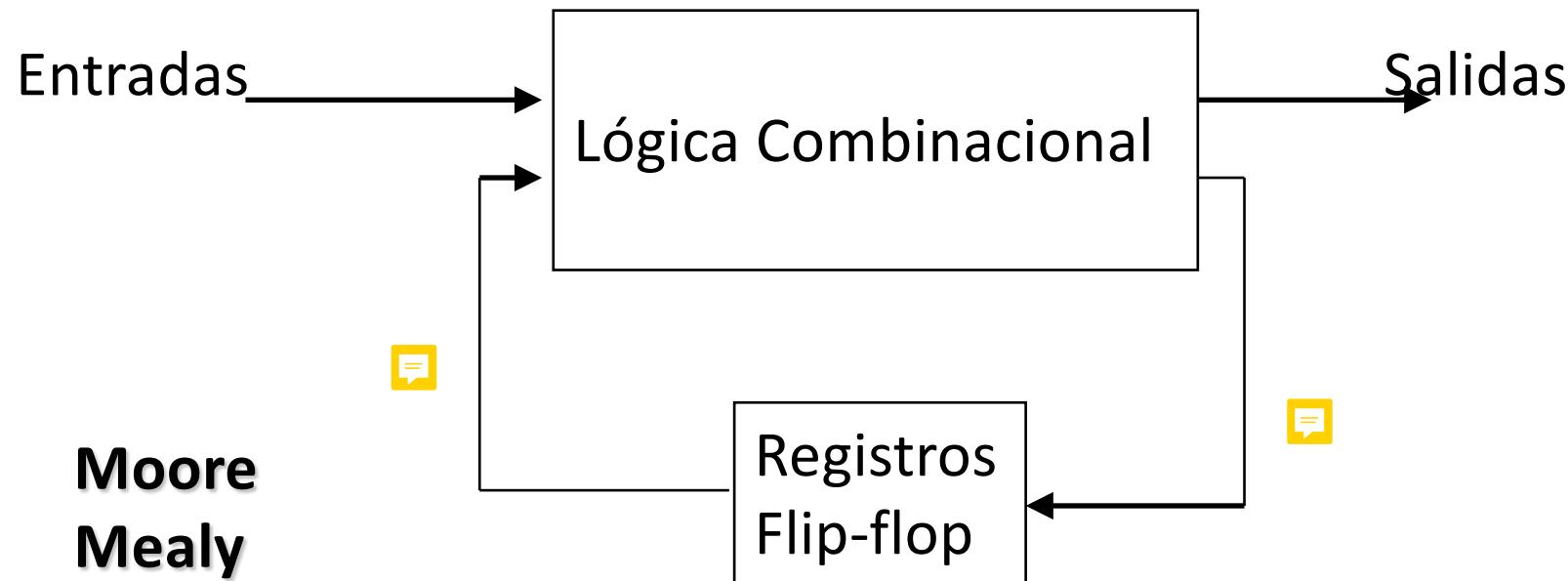


```
...
component comp
 port(x: in std_logic;
 y: out std_logic);
end component
...
signal a, b: std_logic_vector(0 to 7);
...
begin
...
gen: for i in 0 to 7 generate
 u: comp port map (
 x =>a(i),
 y => b(i));
end generate gen;
...
```



# Máquina de estados finita (FSM)

- Todo máquina de estados finitos está compuesta por:
  - Un circuito combinacional que calcula la salida del circuito y el siguiente estado
  - Elementos de memoria para almacenar el estado actual





# Estructura modelo VHDL

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

entity fsm is
 port (
 entrada : in std_logic;
 clk : in std_logic;
 rst : in std_logic;
 salida : out std_logic);
end entity fsm;
```

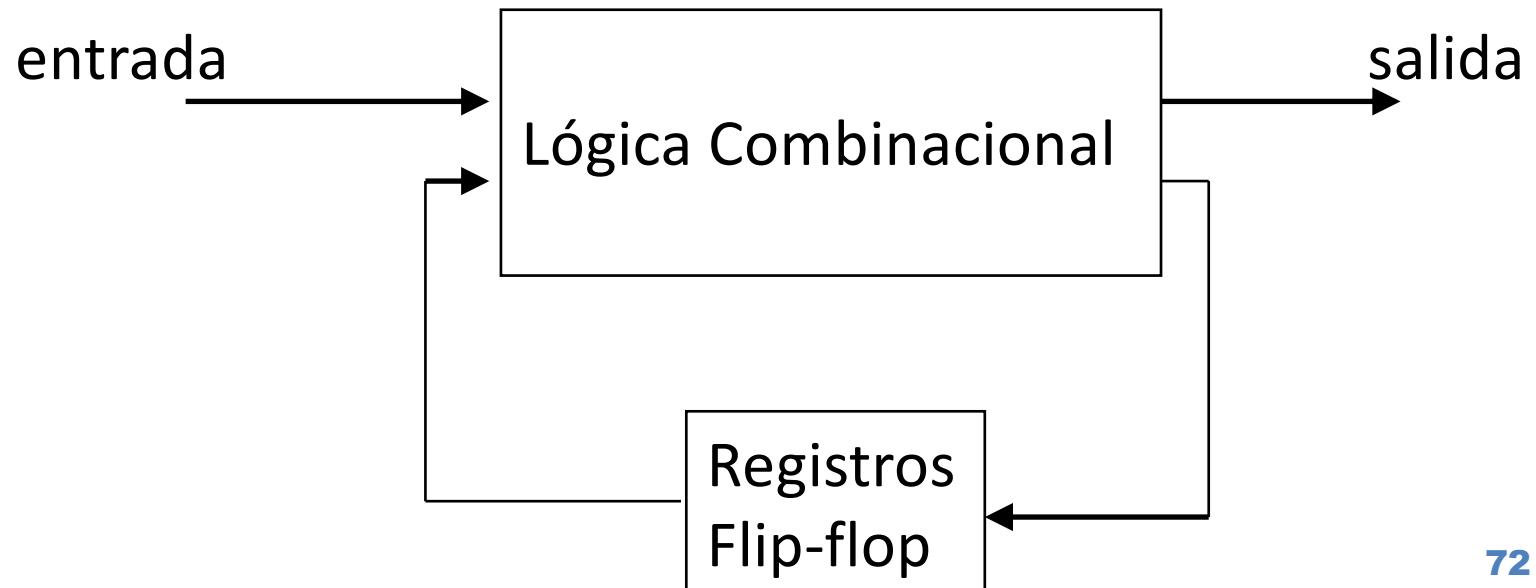




# Estructura modelo VHDL

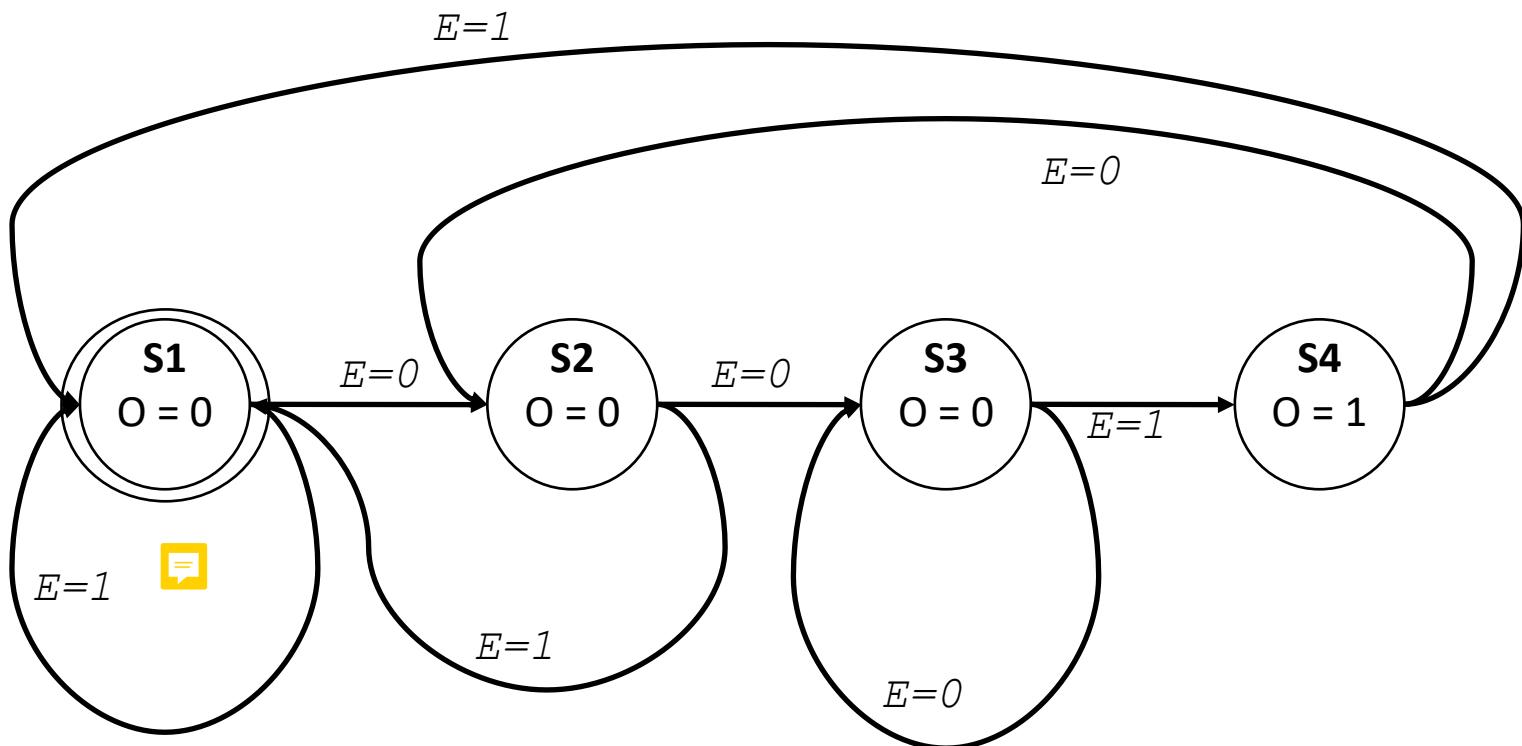
```
architecture rtl of FSM is
 signal estado_actual, estado_siguiente : <type>;
 signal internas : <type>;
begin
 p_comb : process (sensitivity list)
 begin
 vhdl describiendo la lógica combinacional
 end process p_comb;

 p_reg : process (clk, rst)
 begin
 vhdl describiendo los registros
 end process p_reg;
end rtl;
```





# Ejemplo





# Ejemplo

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

entity fsm is
 port(e : in std_logic;
 rst : in std_logic;
 clk : in std_logic;
 o : out std_logic);
end fsm;

architecture rtl of fsm is
 type estado_t is (s1, s2, s3, s4); ←
 signal estado_actual, estado_siguiente : estado_t;
begin

 p_reg : process(clk)
 begin
 if rising_edge(clk) then
 if rst = '1' then ←
 estado_actual <= s1;
 else
 estado_actual <= estado_siguiente;
 end if;
 end if;
 end process p_reg;
```



# Ejemplo. Cont'd (Opción 1)

```
p_comb : process(estado_actual, e)
begin
 case estado_actual is
 when s1 =>
 o <= '0';
 if (e = '0') then
 estado_siguiente <= s2;
 else
 estado_siguiente <= s1;
 end if;
 when s2 =>
 o <= '0';
 if (e = '0') then
 estado_siguiente <= s3;
 else
 estado_siguiente <= s1;
 end if;
 when s3 =>
 o <= '0';
 if (e = '0') then
 estado_siguiente <= s3;
 else
 estado_siguiente <= s4;
 end if;
 when s4 =>
 o <= '1';
 if (e = '0') then
 estado_siguiente <= s2;
 else
 estado_siguiente <= s1;
 end if;
 end case;
end process p_comb;
end architecture rtl;
```



# Ejemplo. Cont'd (Opción 2)

```
p_sig_estado : process(estado_actual, e)
begin
 case estado_actual is
 when s1 =>
 if (e = '0') then
 estado_siguiente <= s2;
 else
 estado_siguiente <= s1;
 end if;
 when s2 =>
 if (e = '0') then
 estado_siguiente <= s3;
 else
 estado_siguiente <= s1;
 end if;
 when s3 =>
 if (e = '0') then
 estado_siguiente <= s3;
 else
 estado_siguiente <= s4;
 end if;
 when s4 =>
 if (e = '0') then
 estado_siguiente <= s2;
 else
 estado_siguiente <= s1;
 end if;
 end case;
end process p_sig_estado;
```

```
p_sal : process(estado_actual, e)
begin
 case estado_actual is
 when s1 => o <= '0';
 when s2 => o <= '0';
 when s3 => o <= '0';
 when s4 => o <= '1';
 end case;
end process p_sal;

end architecture rtl;
```



# Ejemplo. Contador mod 8

```

entity counter is
 port (
 d : in std_logic;
 clk : in std_logic;
 rst : in std_logic;
 q : out std_logic(2 downto 0));
end entity counter; Opción 1

architecture rtl of counter is
 signal count : unsigned(2 downto 0);
begin

 p_counter: process (clk) is
 begin
 if rising_edge(clk) then
 if rst = '1' then
 count <= (others => '0');
 else
 count <= count + 1;
 end if;
 end if;
 end process p_counter;

 q <= std_logic_vector(count);

end architecture rtl;

```

## Opción 2

```

architecture rtl of counter is
 subtype nat_3b is natural range 0 to 7;
 signal count : nat_3b;
begin

 p_counter : process (clk) is
 begin
 if rising_edge(clk) then
 if rst = '1' then
 count <= 0;
 else
 count <= count + 1;
 end if;
 end if;
 end process p_counter;

 q <= std_logic_vector(to_unsigned(count, 3));

end architecture rtl;

```



# ¿Qué son?

- Son la implementación del concepto de librería de otros lenguajes
- Agrupaciones de definiciones de tipos, funciones y procedimientos personalizados.
  - Pueden ser incluidos en uno o varios diseños
  - Normalmente, se definen en ficheros .vhd dedicados.
- Definición
  - package nombre is  
    tipos personalizados  
    constantes  
    funciones y procedimientos (declaraciones)  
end nombre;
  
  - package body nombre is  
    funciones y procedimientos (cuerpo)  
end nombre;



# Ejemplo

fsm\_pack.vhd

```
package fsm_pack is
 type state_t is (S0, S1, S2, S3, S4, S5);
end package fsm_pack;
```

fsm.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use work.fsm_pack.all;

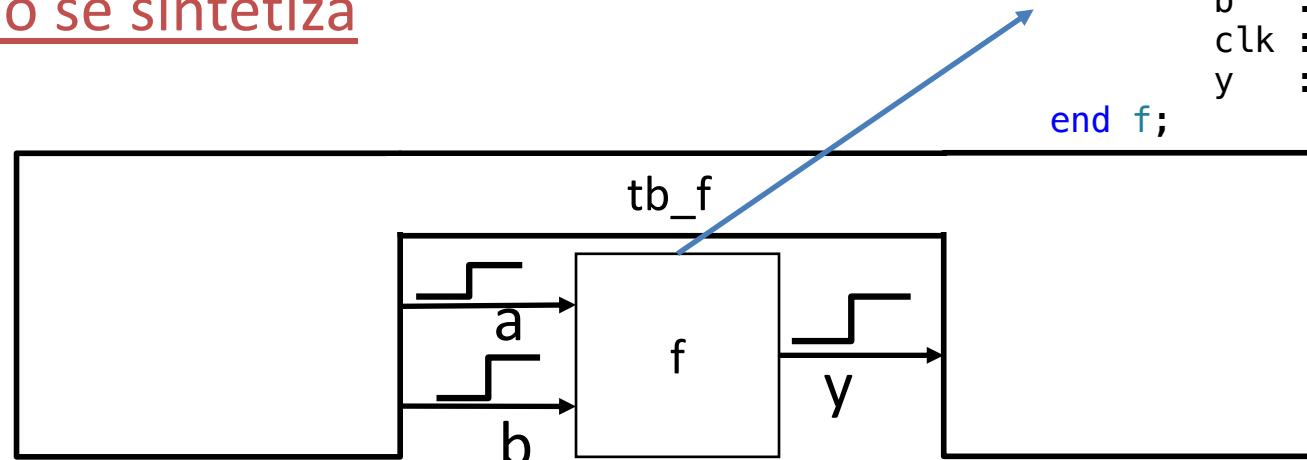
entity fsm is
 generic (g_width : natural := 32);
 port (a : in std_logic_vector(g_width-1 downto 0);
 b : in std_logic_vector(g_width-1 downto 0);
 c : in std_logic_vector(g_width-1 downto 0);
 d : out state_t);
end fsm;
```



# ¿Qué es?

- Modelo VHDL del entorno donde operará el diseño.
- Se utiliza durante la simulación para verificar el correcto funcionamiento del circuito.
  - Introduce estímulos por las entradas y comprueba que las salidas obtenidas son las esperadas
  - No se sintetiza

```
entity f is
port (a : in std_logic;
 b : in std_logic;
 clk : in std_logic;
 y : out std_logic);
end f;
```





# Estructura básica

```
entity tb_f
end tb_f;

architecture beh of tb_f is

component f is
 port (a : in std_logic;
 b : in std_logic;
 clk : in std_logic;
 y : out std_logic);
end component f;

signal a, b, y : std_logic;
signal clk : std_logic;

begin
 dut : f port map(
 a => a,
 b => b,
 clk => clk,
 y => y);

 p_tb : process
 ...
end architecture;
```

1. Se crea una entidad de simulación sin puertos de entrada ni de salida

2. Se declara el componente del diseño a verificar

3. Se definen tantas señales como puertos del diseño a verificar

4. Se instancia el componente y se asignan las señales internas del testbench a los puertos del componente



# Estructura básica

```
p_tb : process
begin
 a <= '0';
 b <= '0';
 wait for 50 ns;
 wait until rising_edge(clk);
 a <= '1';
 wait until rising_edge(clk);
 b <= '1';
 wait until rising_edge(clk);
 a <= '0';
 wait; -- espera para siempre
end process p_tb;

p_clk : process
 constant clk_period : time := 30ns;
begin
 clk <= '0';
 wait for clk_period/2;
 clk <= '1';
 wait for clk_period/2;
end process p_clk;

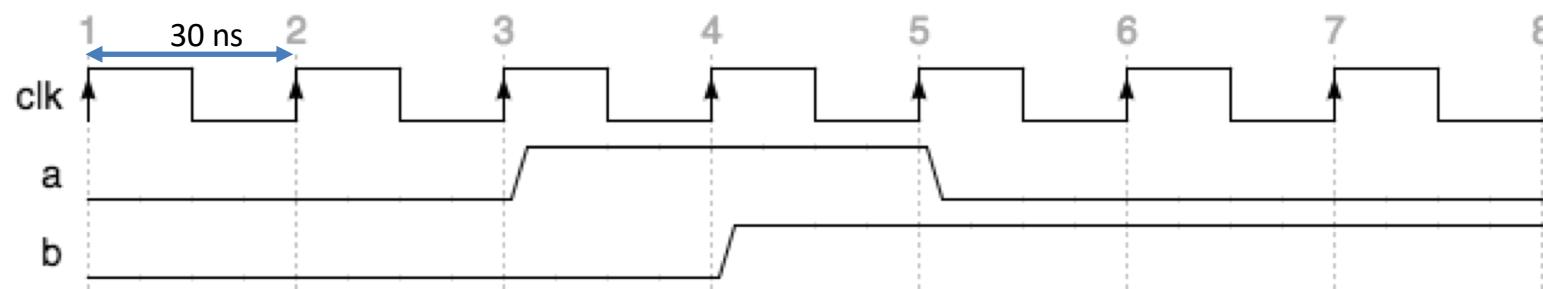
end architecture beh;
```

5. Se define el proceso de simulación.  
NO tiene lista de sensibilidad
6. Se definen los valores iniciales ( $t=0$ )  
de las entradas
7. Se definen los cambios en los valo-  
res de las entradas



# Estructura básica

```
p_tb : process
begin
 a <= '0';
 b <= '0';
 wait for 50 ns;
 wait until rising_edge(clk);
 a <= '1';
 wait until rising_edge(clk);
 b <= '1';
 wait until rising_edge(clk);
 a <= '0';
 wait; -- espera para siempre
end process p_tb;
```





# Reloj

- ¿Cómo implementar la entrada periódica de reloj, clk?

```
p_clk : process
 constant clk_period : time := 20ns;
begin
 clk <= '0';
 wait for clk_period/2;
 clk <= '1';
 wait for clk_period/2;
end process p_clk;
```

Puede sustituirse por un genérico

- Como este *process* no acaba con un *wait (sin for)*, se repetirá indefinidamente