

# LenguajeVHDL

## Lenguaje VHDL de descripción hardware

<b>1. Introducción.....</b>	<b>2</b>
<b>2. Descripción del lenguaje.....</b>	<b>2</b>
2.1. Objetos.....	4
2.2. Atributos.....	5
2.3. Tipos de datos.....	5
2.3.1. Conversión de tipos .....	6
2.3.2. Operadores lógicos, relacionales y aritméticos .....	7
2.4. Entidad.....	7
2.5. Arquitectura.....	9
2.6. Sentencias concurrentes.....	9
2.6.1. La sentencia process.....	9
2.6.2. Asignación a señal concurrente .....	10
2.6.3. Asignación concurrente condicional (WHEN-ELSE) .....	11
2.6.4. Asignación concurrente con selección (WITH-SELECT-WHEN-OTHERS).....	11
2.7. Sentencias secuenciales.....	11
2.7.1. Sentencias condicionales (IF-ELSIF-ELSE y CASE-WHEN-OTHERS) .....	11
2.7.2. Sentencias iterativas (LOOP, NEXT, EXIT).....	13
2.7.3. La sentencia WAIT.....	14
2.7.4. Llamada a subprogramas [No entra] .....	14
2.8. Subprogramas (PROCEDURE y FUNCTION) [No entra] .....	15
<b>3. Fichero de estímulos.....</b>	<b>16</b>
<b>4. Descripción de circuitos combinacionales .....</b>	<b>19</b>
4.1. Conexión de varias salidas a una línea bus.....	19
4.2. Multiplexor .....	20
4.3. Convertidor BCD a 7 segmentos .....	21
4.4. Sumador.....	23
<b>5. Descripción de circuitos secuenciales.....</b>	<b>24</b>
5.1. Biestable tipo D.....	24
5.2. Registro de 8 bits con reset asíncrono .....	26
5.3. Registro de 8 bits con reset asíncrono, señal de carga y salida en alta impedancia .....	27
5.4. Registro de desplazamiento de 8 bits con reset asíncrono .....	27
5.5. Contador de N bits con reset asíncrono y señal de habilitación .....	28
<b>6. Diseño de máquinas de estado .....</b>	<b>30</b>
6.1. Ejercicio del control de la barrera del tren.....	31

## 1. Introducción

Un lenguaje de descripción hardware (HDL, Hardware Description Language) permite el diseño y simulación de circuitos electrónicos digitales complejos con un nivel de abstracción muy superior a las técnicas tradicionales, como son los mapas de Karnaugh o las ecuaciones Booleanas. Los lenguajes HDL han supuesto para la electrónica un avance similar al que supuso la aparición de lenguajes de alto nivel, como el C, frente a la programación en ensamblador. En este tema se va a estudiar el lenguaje de descripción hardware más popular llamado **VHDL** (**VHSIC HDL**; donde VHSIC: Very High Speed Integrated Circuits).

Las ventajas fundamentales que aporta un lenguaje como VHDL frente a las técnicas tradicionales son las siguientes:

- *Potencia y flexibilidad.*  
VHDL permite realizar descripciones esquemáticas o de comportamiento de los circuitos. Es un lenguaje de simulación y de síntesis. La síntesis consiste en la traducción del código VHDL a puertas lógicas.
- *Independencia de la tecnología.*  
VHDL permite la descripción funcional previa de un circuito sin especificar un dispositivo concreto para su implementación final.
- *Portabilidad.*  
VHDL es un estándar que permite que el código sea portable y reutilizable entre entornos de trabajo de distintos fabricantes.
- *Reducción del ciclo de diseño.*  
Como consecuencia de lo anterior, el diseño con VHDL permite un flujo de diseño rápido puesto que el código puede simularse y depurarse antes de realizar la síntesis (traducción a puertas lógicas). De esta forma se acelera el diseño, los plazos de entrega y la colocación de un nuevo producto en el mercado.

En el mercado existe gran variedad de herramientas para la simulación y síntesis de circuitos digitales con el lenguaje VHDL. La gran competencia entre ellas ha ocasionado que las prestaciones que ofrecen sean muy similares.

## 2. Descripción del lenguaje

Una descripción VHDL de un circuito de baja complejidad requiere, al menos de estos 3 elementos:

- *Librerías (library).* La potencialidad de un lenguaje depende en gran medida de sus librerías. En la cabecera de cada diseño se deben incluir aquellas librerías que se necesiten.
- *Entidad (entity).* Describe las entradas y salidas del diseño, como si se tratase de una caja negra.
- *Arquitectura (architecture).* Describe el contenido de ese diseño.

El primer ejemplo es muy simple y muestra como se diseña una puerta AND de 2 entradas en VHDL. Se aprecia claramente las 3 partes o elementos necesarios para el diseño de un circuito con este lenguaje: librerías, entidad y arquitectura.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
```

```
ENTITY and2IS
PORT (
    a,b: IN std_logic;  --Pines de entrada
    s: OUT std_logic    --Pin de salida. Sin ';' antes de ');'
);
END and2;
```

```
ARCHITECTURE and2_arqOF and2IS
BEGIN
    s <= a AND b;
END and2_arq;
```

Las dos primeras líneas son dos sentencias utilizadas habitualmente en el código VHDL para incluir el paquete estándar `std_logic_1164`. Esta librería define un tipo de dato llamado `std_logic`, utilizado habitualmente en la síntesis y simulación de circuitos digitales. A continuación se declara la entidad, que incluye las entradas y salidas, y, finalmente, la arquitectura, que es la descripción de una puerta AND, utilizando el operador AND de VHDL.

Las palabras reservadas del lenguaje VHDL se escribirán en mayúsculas en este documento para distinguirlas del resto. En los entornos de desarrollo de VHDL no es necesario hacerlo ya que disponen de editores que las muestran con otro color por lo que no es necesario utilizar las mayúsculas para destacarlas.

Los identificadores utilizados por el usuario no pueden ser iguales a estas palabras reservadas y pueden estar formados por letras, dígitos o el carácter "\_". El primer carácter siempre debe ser una letra, y un identificador no puede acabar con el carácter "\_", así como tampoco se pueden usar dos seguidos "\_\_". Por ejemplo, son identificadores legales:

```
tx_clk; Three_State_Enable; sel7D; HIT_1029
```

Por contra no son válidos los identificadores:

```
_tx_clk; 8B10B; large#num; register; clk_; link__bar
```

Como es habitual en cualquier lenguaje de programación, están prohibidas las vocales acentuadas así como la letra "ñ".

La estructura genérica de un código VHDL es la indicada a continuación. El doble carácter '--' indica que el texto que viene a continuación es un comentario hasta el final de la línea.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
```

```
ENTITY eeeIS
PORT (
    -- declaración de los puertos de entrada y salida
);
END eee;
```

```
ARCHITECTURE aaaOF eeeIS
    -- declaraciones de tipos de datos y objetos
BEGIN
    -- líneas de código
END aaa;
```

## 2.1. Objetos

Los objetos que se manejan en un código VHDL son constantes, señales y variables. Todos ellos deben ser declarados en su lugar correspondiente dentro de la arquitectura. La declaración genérica de un objeto es la siguiente:

- *Constantes* (**CONSTANT**). Una constante define un valor que no varía dentro de la descripción VHDL. Se utilizan para mejorar la legibilidad del código y facilitar su modificación. La sintaxis VHDL para declarar una constante es la siguiente:

```
CONSTANT nombre [, ...] : TIPO [ := expresión ];
```

Algunos ejemplos de declaración de constantes son los siguientes:

```
CONSTANT rango      : INTEGER      := 8;
CONSTANT PI         : REAL         := 3.1415927;
CONSTANT cierto     : BOOLEAN     := true;
CONSTANT ciclo      : TIME        := 20ns;
```

- *Señales* (**SIGNAL**). Las señales pueden representar conexiones reales del circuito. Los puertos de entrada o salida de la declaración de la entidad son señales que se pueden utilizar en el bloque de la arquitectura sin necesidad de volverlos a declarar. La declaración debe realizarse en la arquitectura justo antes del BEGIN. Nunca dentro de un proceso. La sintaxis genérica es la siguiente:

```
SIGNAL nombre [, ...] : TIPO [ := expresión ];
```

A continuación se muestran varios ejemplos de declaración de señales:

```
SIGNAL contador      : std_logic_vector(3 DOWNTO 0) := "1001";
SIGNAL valor         : integer range 0 to 9;
SIGNAL resultado     : bit;
```

La asignación de una señal no es inmediata. Se realiza al finalizar el proceso activo en el que se encuentra. La sintaxis de asignación de una señal utiliza los caracteres <=. Ejemplo :

```
contador <= "0110";
```

- *Variables* (**VARIABLE**). A diferencia de las señales no tienen relación con el resultado final que genere la síntesis del circuito. Además su asignación es inmediata. Son de gran utilidad en instrucciones iterativas para recorrer un bucle. Se recomienda utilizarlas solo dentro de un proceso.

```
VARIABLE nombre [, ...] : TIPO [ := expresión ];
```

A continuación se muestran algunos ejemplos de declaración de variables:

```
VARIABLE i,j,k       : INTEGER := 0; -- La asignación a 0 es opcional.
VARIABLE resultado   : BOOLEAN;
```

Ejemplos de VHDL para asignar una variable:

```
i := 3;
j := i+1;
```

## 2.2. Atributos

Los elementos de VHDL -como las variables y señales- pueden tener información adicional llamada *atributos*. El atributo se representa añadiéndole un apóstrofe y el nombre del atributo al elemento deseado. Los atributos más comunes son los que se detallan a continuación.

Suponga la siguiente declaración en una entidad: `a : IN std_logic_vector(5 downto 2);`

a'LEFT	Límite izquierdo del rango de a. Devuelve 5.
a'RIGHT	Límite derecho del rango de a. Devuelve 2.
a'LOW	Límite inferior del rango de a. Devuelve 2.
a'HIGH	Límite superior del rango de a. Devuelve 5.
a'LENGTH	Número de elementos de a. Devuelve 4 (elementos 2,3,4,5)
a'RANGE	Devuelve "5 DOWNT0 2". Útil para una sentencia iterativa.

Se pueden usar los siguiente atributos para una determinada señal:

s'EVENT	Se ha producido un cambio en la señal s.
s'STABLE(t)	Indica si estuvo estable durante el último periodo t.

## 2.3. Tipos de datos

Los tipos de datos que se van a utilizar en este texto son los dos siguientes:

- *Escalares*. Utilizan la palabran reservada *RANGE*. Algunos ejemplos:
 

<b>TYPE</b> byte	<b>IS RANGE</b> 0 <b>TO</b> 255;	-- tipo entero
<b>TYPE</b> indice	<b>IS RANGE</b> 10 <b>DOWNT0</b> 0;	-- tipo entero
<b>TYPE</b> INTEGER	<b>IS RANGE</b> -2147483647 <b>TO</b> 2147483647;	-- Predefinido
<b>TYPE</b> tension	<b>IS RANGE</b> 0.0 <b>TO</b> 3.3;	-- tipo real
<b>TYPE</b> REAL	<b>IS RANGE</b> -1E38 <b>TO</b> 1E38;	-- Predefinido
- *Enumerado*. La enumeración define una lista que un objeto de ese tipo soporta. Se utiliza habitualmente para definir las máquinas de estado. Por ejemplo, se podría declarar una señal (signal) tipo 'estado' de la siguiente manera:
 

```
TYPEestado IS (inicio,estado1,estado2,estado3,estado4,final);
SIGNAL estado_actual, estado: estado;
```
- *std\_logic, std\_logic\_vector, y std\_ulogic\_vector*. Constituyen los tipos lógicos estándar para diseños digitales. Se dispone de ellos incluyendo el paquete std\_logic\_1164. Los valores definidos en este estándar aparecen en la tabla 4.1.

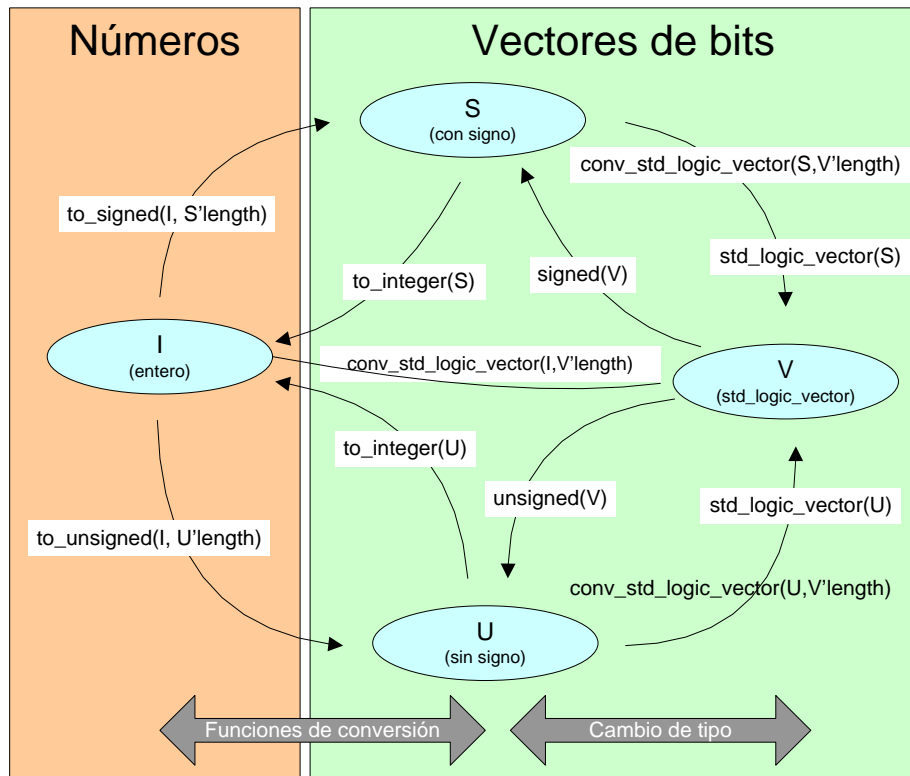
Sintaxis	Valor
'0'	fuerza un cero lógico
'1'	fuerza un uno lógico
'Z'	alta impedancia
'L'	cero lógico débil (pull-down).
'H'	uno lógico débil (pull-up).
'W'	Indefinido débil
'-'	no importa

Cuadro 4.1: Valores lógicos definidos en el estándar std\_logic\_1164

- *unsigned y signed*. El tipo unsigned es un vector de bits sin signo similar al tipo std\_ulogic\_vector pero no son compatibles. Se disponen de ellos incluyendo el paquete [ieee.numeric\\_std.ALL](#).

### 2.3.1. Conversión de tipos

En la siguiente figura se muestra cómo pasar de un tipo a otro.



Conviene resaltar varias cosas:

- Las funciones que permiten la conversión de tipos están en alguna librería. Conviene diseñar código estándar VHDL. Éste lo garantiza la librería `ieee.numeric_std.ALL`. Cualquier función que pertenezca a las librerías `ieee.std_logic_arith` o `ieee.std_logic_signed/unsigned` no es código estándar y no conviene usarlo.
- Se puede pasar directamente de integer a std\_logic\_vector pero no es código estándar y no se recomienda. Por el contrario, no se puede pasar directamente del tipo "std\_logic\_vector" al tipo "integer". Hay que pasar previamente por el tipo signed o unsigned. Por ejemplo:
- El dibujo es igualmente válido si en lugar de "std\_logic\_vector" se tiene "std\_u`logic_vector`".

Algunos ejemplos de conversión de tipo son los siguientes:

```
SIGNAL    a : std_logic_vector(3 downto 0);
VARIABLE  b : std_ulogic_vector(3 downto 0);
SIGNAL    c : unsigned(3 downto 0);
SIGNAL    i : integer;

b := std_ulogic_vector(c); -- Las variables se asignan con :=
c <= unsigned(a);
a <= std_logic_vector(to_unsigned(i, a'length)); -- recomendado
a <= conv_std_logic_vector(b, a'length);         -- No recomendado

d <= to_integer(c);
d <= to_integer(unsigned(a));
```

### 2.3.2. Operadores lógicos, relacionales y aritméticos

Los operadores que se pueden aplicar a los datos de un diseño son los siguientes:

Lógicos	Relacionales	Aritméticos	VHDL-93
<b>OR</b>	= (igual que)	+, -, *, /	<b>SLA</b> – despl. aritm. izq.
<b>AND</b>	/= (distinto que)	<b>Abs</b> (valor absoluto)	<b>SRA</b> – despl. aritm. der.
<b>NOR</b>	> (mayor que)	<b>Mod</b> (módulo)	<b>SLL</b> – despl. lógicoizq.
<b>NAND</b>	< (menor que)	<b>Rem</b> (Resto)	<b>SRL</b> – despl. lógico der.
<b>XOR</b>	>= (mayor o igual que)	** (potencia)	<b>ROL</b> – rotaciónizquierda
	<= (menor o igual que)	& (concatenación)	<b>ROR</b> – rotaciónderecha

### 2.4. Entidad

Una declaración de una entidad describe el módulo cuya descripción VHDL se va a implementar (como si se tratase de un símbolo de una captura esquemática). Cada entrada o salida en la declaración de una entidad se denomina puerto. El puerto se define indicando su nombre, modo y tipo. El nombre es un identificador válido, según las reglas expuestas. El modo define la dirección en la que se transfiere el dato a través de ese puerto. La siguiente tabla 4.2 muestra los diferentes tipos de puertos y sus modos de funcionamiento.

Tipo	Modo de Operación
<b>IN</b>	el dato sólo entra a la entidad a través del puerto
<b>OUT</b>	el dato sólo sale de la entidad a través del puerto
<b>INOUT</b>	el dato es bidireccional, es decir, entra o sale de la entidad a través del puerto
<b>BUFFER</b> (No usar nunca)	el dato sale de la entidad a través del puerto, pero es realimentado internamente, pudiendo actuar de entrada a elementos de la arquitectura. Una señal de salida es tipo buffer cuando aparece en el lado derecho de una asignación o cuando actúa como condición. Es decir, cuando influye en el valor de otra señal interna o de otro puerto de salida

Cuadro 4.2: Tipos de puertos definidos en VHDL

En la figura 4.1 se muestra un ejemplo de los modos de funcionamiento de los puertos detallados anteriormente.

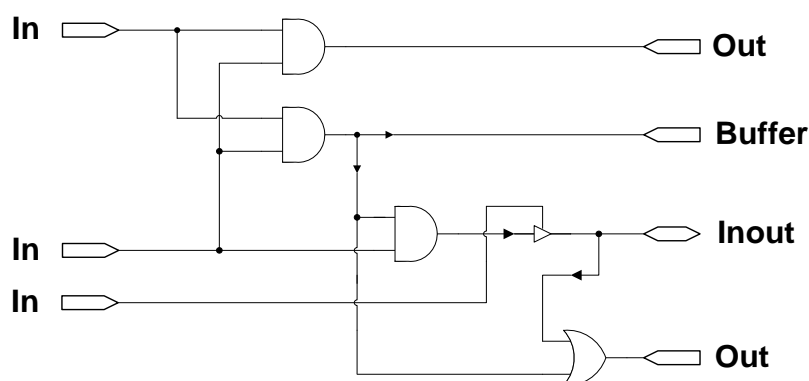


Figura 4.1: Modos de un puerto

Se evitará utilizar el tipo *buffer*. Esto es posible utilizando una salida tipo *Out* más una señal interna de la arquitectura que se asigna a la salida tipo *Out*. Se verá más adelante tanto en el circuito del registro de desplazamiento como en el circuito del contador

Por ejemplo, la declaración de una entidad correspondiente a un sumador de 4 bits (figura 4.2) y su esquema serían de la forma:

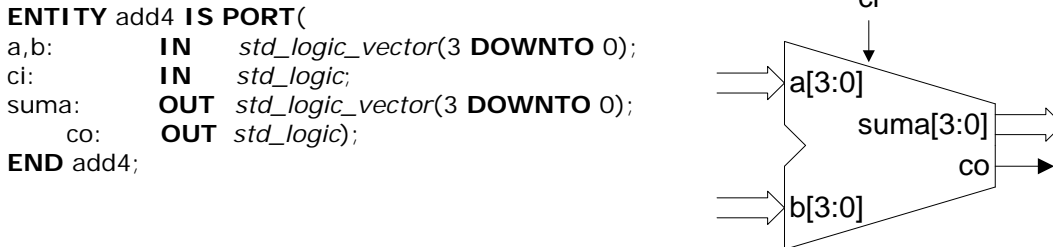


Figura 4.2: Sumador de 4 bits

La sintaxis de la entidad de forma general es la siguiente (todo lo que está entre dos corchetes significa que es opcional):

```

[etiqueta:] ENTITY nombre_entidad IS
  [ GENERIC(lista_generica); ]
  [ PORT(lista_puertos); ]
  [ BEGIN                                -- muy raro su uso
  [ sentencias ]                          -- muy raro su uso
END ENTITY nombre_entidad;

```

Las entidades para los estímulos de una simulación (no describen un circuito) están vacías.

La instrucción **GENERIC** permite declarar propiedades o constantes. Se les puede asignar un valor por defecto que puede ser cambiado cuando se cree el componente.

Si se desea realizar un diseño de una puerta **AND** de tamaño variable, la entidad sería la siguiente:

```

ENTITY andx IS

  GENERIC ( N : integer := 2 );      -- 2 es el valor por defecto

  PORT(
    entradas: IN std_logic_vector(N-1 DOWNTO 0);
    salida:  OUT std_logic-- No lleva ';' porque viene un ')'
  );

END ENTITY andx;

```

Si se quieren crear tres puertas lógicas **AND** de 2, 3 y 4 entradas respectivamente habrá que escribir en la arquitectura correspondiente (normalmente del fichero de estímulos) 3 líneas para declarar cada uno de los objetos que se van a crear. También habrá que declarar previamente las entradas y salidas que tendrá cada una de las puertas:

```

ARCHITECTURE test_arq OF test IS-- fichero de estímulos
SIGNAL Entradas2: std_logic_vector(1 DOWNTO 0);  -- Dos  entradas
SIGNAL Entradas3: std_logic_vector(2 DOWNTO 0);  -- Tres  entradas
SIGNAL Entradas4: std_logic_vector(3 DOWNTO 0);  -- Cuatro entradas
SIGNAL salidas1, salida2, salida3: std_logic;

```



**BEGIN**

```
-- Se crean los tres componentes de diferentes tamaños
U1: andx generic map (N => 2) PORT MAP(entradas2, salida1); -- AND2
U2: andx generic map (N => 3) PORT MAP(entradas3, salida2); -- AND3
U3: andx generic map (N => 4) PORT MAP(entradas4, salida3); -- AND4

--otras sentencias

ENDtest_arq;
```

## 2.5. Arquitectura

La arquitectura describe la funcionalidad de la caja negra declarada anteriormente. Básicamente, existen tres estilos fundamentales para describir esta funcionalidad dentro del cuerpo de la arquitectura:

- Descripción de comportamiento ('behavioral'). Alto nivel de abstracción.
- Descripción de flujo de datos ('dataflow'). Mediano/Bajo nivel de abstracción.
- Descripción estructural ('structural'). Nulo nivel de abstracción.

El tipo de descripción de la arquitectura depende del tipo de elementos y estructuras que se utilicen dentro de ella. Por ejemplo la descripción de mayor nivel de abstracción ('behavioral') solo utiliza procesos. En este texto se utilizarán, dentro de la arquitectura, los diferentes elementos de sintaxis que se proponen en los siguientes apartados sin prestar atención al tipo de descripción que resulte para la arquitectura.

## 2.6. Sentencias concurrentes

La característica común a todas ellas es que indican conexiones o normas que han de cumplirse. Se ejecutan en paralelo y el efecto es como si se estuvieran ejecutando indefinidamente.

Se declaran en las arquitecturas de los modelos y no estarán contenidas en ningún proceso. Las tres sentencias concurrentes que se estudiarán en este apartado son:

- La sentencia **PROCESS** (proceso).
- Asignación a señal concurrente.
- Asignación concurrente condicional (**WHEN-ELSE**).
- Asignación concurrente con selección (**WITH-SELECT-WHEN-OTHERS**).

### 2.6.1. La sentencia process

Los procesos son bloques que se van a evaluar en paralelo con otras sentencias concurrentes: otros procesos o asignaciones. Sin embargo, el contenido de un proceso se evalúa secuencialmente. Este hecho obliga a que las sentencias que constituyen el cuerpo de un proceso sean sentencias secuenciales. La síntesis de la descripción VHDL generará un circuito digital que se comporte según ese código que se ejecuta de forma secuencial (ver ejemplo de la sentencia **if**). Dentro de una arquitectura pueden existir varios procesos, que se ejecutan de manera concurrente entre sí.

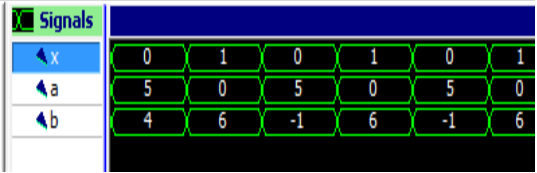
La sintaxis genérica de un proceso se muestra a continuación. La etiqueta y la declaración de variables son opcionales.

```
[etiqueta:] PROCESS [(lista sensible)]
    --declaraciones de variables (no admite señales)
BEGIN
    --sentencias secuenciales
END PROCESS [etiqueta];
```

Normalmente, el proceso se encuentra en modo suspendido hasta que se produce un cambio en una de las señales que figura en la lista de variables sensibles ('sensitivity list'). Cuando se produce un evento en una de estas señales, se ejecutan las líneas de código del proceso, realizándose la asignación de las señales al final de la ejecución del proceso. Una vez finalizado el proceso, entra de nuevo en modo suspendido hasta que vuelva a producirse un nuevo evento en una de las señales de la lista sensible.

En el caso de que una señal se asigne más de una vez dentro del proceso, la última asignación es la que prevalece. Supongamos el siguiente ejemplo en el que las señales son de tipo entero siendo la señal **x** de entrada y **a, b** de salida.

```
PROCESS (x)
VARIABLE m : INTEGER;
BEGIN
a <= 5;           -- podría ser a=5 al final
m := a;          -- m=valor anterior de a
b <= m-1;        -- podría ser b=m-1
IF (x=1) THEN
a <= 0;          -- como x=1 -> a=0
b <= m+1;        -- como x=1 -> b=m+1
END IF;
END PROCESS;
```



Signal	0	1	0	1	0	1
x	0	1	0	1	0	1
a	5	0	5	0	5	0
b	4	6	-1	6	-1	6

Si un proceso contiene la sentencia *WAIT*, utilizada para la simulación, entonces no puede tener lista sensible. El siguiente proceso se utiliza en el fichero de estímulos para generar la señal de reloj:

```
PROCESS -- No lleva lista sensible
BEGIN -- Se repite indefinidamente. La ejecución del proceso no depende del
clk <= '0'; -- cambio ninguna señal
WAITFOR 1us;

clk <= '1'; -- Generará un reloj cuyo periodo es 2us
WAITFOR 1us;
END PROCESS;
```

En los circuitos síncronos, todos los elementos secuenciales cambian su valor con el flanco de reloj. En consecuencia, la descripción en VHDL de cualquier elemento de memoria síncrono deberá utilizar un proceso con el reloj en su lista sensible. Todas las asignaciones que se realizan dentro de un proceso se hacen efectivas al final del mismo a efectos de interpretación y simulación.

### 2.6.2. Asignación a señal concurrente

Se encuentran fuera de un proceso y son evaluadas en paralelo con el resto de procesos y asignaciones concurrentes que están presentes en la arquitectura. La sintaxis genérica es la siguiente:

[etiqueta:] nombre\_señal <= expresión\_o\_dato;

En realidad este tipo de sentencia es equivalente a un proceso cuya lista sensible está compuesta por las variables que aparecen en el término de la derecha de la asignación. A continuación se muestra el proceso equivalente de la sentencia  $y <= a$  (a la izquierda), y el proceso equivalente a la sentencia  $y <= (a \text{ AND } (\text{NOT } b)) \text{ OR } (a \text{ AND } c)$  (a la derecha).

<pre>PROCESS (a) BEGIN y &lt;= a END PROCESS;</pre>	<pre>PROCESS (a,b,c) BEGIN y &lt;= (a AND (NOT b)) OR (a AND c) END PROCESS;</pre>
---	--

### 2.6.3. Asignación concurrente condicional (WHEN-ELSE)

La asignación condicional concurrente que estudiaremos es de la forma WHEN-ELSE. Su sintaxis, de forma genérica, es:

```
nombre_señal <= valor_1 WHEN condicion1 ELSE
                valor_2 WHEN condicion2 ELSE
                . . .
                valor_n WHEN condicionn ELSE
                valor_x;
```

Según las condiciones, a la señal se le asigna un valor. La prioridad va según el orden de aparición. Es decir, una vez que una de las condiciones se cumple, es indiferente si alguna otra condición más abajo se verifica, se asigna el valor de la primera condición que se cumple. Es importante poner siempre un ELSE al final para evitar que la síntesis introduzca latches en la implementación en puertas, normalmente indeseados en una lógica puramente combinacional.

En la tabla 3.3 se detallan las sentencias concurrentes, así como algunos ejemplos.

Sentencias concurrentes	Ejemplos
Asignación a señal	<code>x &lt;= (a AND (NOT sel1)) OR (b AND sel1);</code> <code>g &lt;= NOT (y AND sel2);</code>
Asignación condicional	<code>y &lt;= d WHEN (sel='1') ELSE c;</code> <code>h &lt;= '0' WHEN (x='1' AND sel2='0') ELSE '1';</code> <code>y &lt;= a WHEN (sel="00") ELSE</code> <code>     b WHEN (sel="01") ELSE</code> <code>     c WHEN (sel="10") ELSE</code> <code>     d ;</code>

Cuadro 3.3: Ejemplos de sentencias que se interpretan concurrentemente en VHDL

### 2.6.4. Asignación concurrente con selección (WITH-SELECT-WHEN-OTHERS)

La asignación condicional concurrente que estudiaremos es de la forma WHEN-ELSE. Su sintaxis, de forma genérica, es:

```
WITH expresion SELECT
nombre_señal <= opcion_1 WHEN valor1,
                opcion_2 WHEN valor2,
                . . .
                opcion_x WHEN OTHERS;
```

En el apartado 4.2 se muestra un ejemplo de uso de la sentencia WITH para la descripción de un convertidor de código de BCD a 7 segmentos.

## 2.7. Sentencias secuenciales

Las sentencias secuenciales se ubican solo en el cuerpo de un proceso o de un subprograma. Permiten describir la funcionalidad de un componente.

Aunque los circuitos digitales trabajan en paralelo, pueden ser modelados por una serie de expresiones secuenciales, de forma similar a como se haría con un lenguaje de propósito general.

### 2.7.1. Sentencias condicionales (IF-ELSIF-ELSE y CASE-WHEN-OTHERS)

#### La sentencia IF-ELSIF-ELSE

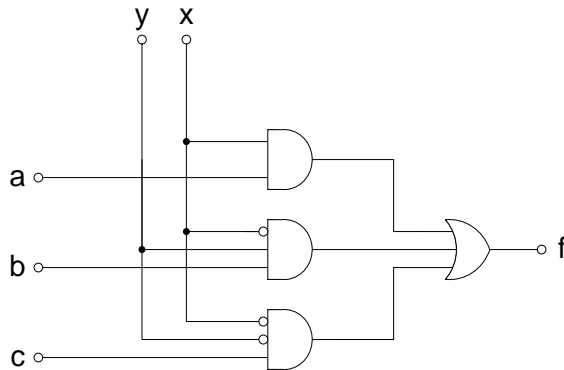
Su sintaxis genérica es:

```

IF condicion1 THEN
  – sentencias secuenciales
ELSIF condicion2 THEN
  – sentencias secuenciales
ELSE
  – sentencias secuenciales
END IF;

```

A continuación se muestra un ejemplo y el resultado de su síntesis en puertas. Se aprecia que el circuito, resultado de la síntesis, cumple con el orden secuencial de la sentencia *if*.



```

PROCESS (a,b,c,x,y)
BEGIN
  IF x='1'THEN
    f <= a;
  ELSIF y='1'THEN
    f <= b;
  ELSE
    f <= c;
  END IF;
END PROCESS;

```

La interpretación secuencial del código anterior supone que la salida tomará el valor de la entrada 'a' siempre que 'x' valga 1. La salida tomará el valor de la entrada 'b' no sólo cuando 'y' valga 1, sino cuando además se verifique que 'x' valga cero. La secuencialidad del código supone que no se comprueba la condición del *ELSIF* a menos que no se cumpla la condición del *IF*. Finalmente, la salida será igual a 'c' cuando no se cumpla ninguna de las dos condiciones previas. La síntesis refleja la interpretación indicada.

El último *ELSE* es vital para garantizar que el circuito sea puramente combinacional. Si no estuviera y se diera ese caso ([x,y]='00'), el sistema intentaría que la salida *f* mantuviera su valor anterior, es decir, introduciría un biestable para almacenar el valor de *f* por lo que el circuito dejaría de ser combinacional.

### La sentencia **CASE-WHEN-OTHERS**

La estructura condicional CASE-WHEN especifica una serie de asignaciones condicionales en función de una señal de selección. Su sintaxis, de forma genérica, es:

```

CASE valor_seleccion IS
  WHEN valor_1 =>
    -- sentencias_secuenciales;
  WHEN valor_2 TO valor_7 => -- consecutivos
    -- sentencias_secuenciales;
  WHEN valor_8 | valor_9 =>
    -- sentencias_secuenciales;
  ...
  WHEN valor_n =>
    -- sentencias_secuenciales;

  WHEN OTHERS =>
    -- sentencias_secuenciales;
END CASE;

```

Con la última condición, **WHEN OTHERS**, se cubren las condiciones no contempladas en los valores anteriores. En los valores expresados a continuación de la palabra reservada **WHEN** deben de cubrirse todos los posibles valores de la señal de selección; por tanto, usar **WHEN OTHERS** es opcional siempre y cuando se cubran los valores posibles de la señal de selección.

La sentencia **CASE** se utiliza habitualmente en el diseño de máquinas de estado. Más adelante se verá algún ejemplo de uso de la misma.

### 2.7.2. Sentencias iterativas (**LOOP**, **NEXT**, **EXIT**)

La sentencia **LOOP** permite la ejecución repetida de un bloque de sentencias. Su sintaxis es la siguiente:

```
[etiqueta:] [[WHILE condicion] | [FOR indice IN valor1 TO/DOWNTO valor2]] LOOP
--sentencias secuenciales
END LOOP [etiqueta];
```

Las sentencias **NEXT** y **EXIT** van ligadas a la sentencia **LOOP**. La primera finaliza la iteración actual –omitiendo las sentencias restantes hasta completar dicha iteración– y pasa a la siguiente iteración del bucle. La segunda finaliza el bucle **LOOP** y sale de él. Ambas sentencias permiten una etiqueta de salida opcional y una condición de forma que si ésta se cumple se interrumpe el bucle y si no, no. Estas opciones tienen interés cuando hay bucles anidados (etiquetados) y se quiere especificar a cual de ellos afecta la instrucción. La sintaxis VHDL de ambas instrucciones es la siguiente:

```
NEXT [ [etiqueta] [ WHEN condicion] ]
EXIT [ [etiqueta] [ WHEN condicion] ]
```

A continuación se muestra un ejemplo que suma todos los valores desde 1 hasta un determinado número dado por el parámetro del proceso llamado *entrada*. El resultado se tiene en la variable *suma* de tipo integer. Se realizará con los 3 posibles tipos de bucles:

#### **FOR ... LOOP**

```
PROCESS(entrada)
  VARIABLE suma : INTEGER;
BEGIN
  suma := 0;
  FOR i IN entrada DOWNTO 1 LOOP
    suma := suma + i;
  END LOOP;
END PROCESS;
```

#### **WHILE ... LOOP**

```
PROCESS(entrada)
  VARIABLE i, suma: integer;
BEGIN
  suma := 0;
  i := entrada;
  WHILE (i >= 1) LOOP
    suma := suma + i;
    i := i - 1;
  END LOOP;
END PROCESS;
```

#### **LOOP**

```
PROCESS(entrada)
  VARIABLE i, suma: integer;
BEGIN
  suma := 0;
  i := entrada;
  LOOP
    suma := suma + i;
    i := i - 1;
    IF (i = 0) THEN
      EXIT; -- salir
    END IF;
  END LOOP;
END PROCESS;
```

En el siguiente ejemplo se muestran dos bucles anidados y el uso de las sentencias NEXT y EXIT:

```

PROCESS(x)
  VARIABLE a: INTEGER;
BEGIN
  a:= 0;
  bucle_externo: WHILE(a<15) LOOP
  -- sentencias_secuenciales_1

  bucle_interno: FOR i IN 1 TO 15 LOOP
  --sentencias_secuenciales_2

  NEXT bucle_externo WHEN (i=a);          -- Interrumpe el FOR y sigue en el
                                          -- while justo al comienzo del mismo
                                          -- (nueva iteración del while)

  EXIT bucle_interno WHEN a>x;          -- Interrumpe el FOR y continua la
                                          -- iteración actual del bucle while
  END LOOP;-- por las sentencias_secuenciales_3

  -- sentencias_secuenciales_3;
  END LOOP;
END PROCESS;

```

### 2.7.3. La sentencia WAIT

La sentencia *WAIT* suspende un proceso bien de forma definitiva (fichero de simulación) bien de forma temporal hasta que se cumpla una condición fijada en la propia sentencia. La sintaxis general es la siguiente:

```
WAIT [ [FOR periodo_tiempo] [ UNTIL condicion] [ON senal1 [,señal2, ...]] ];
```

Normalmente esta sentencia se utiliza en procesos utilizados en el fichero de estímulos para la simulación. Un proceso que contenga la sentencia *WAIT* debe tener su lista sensible vacía obligatoriamente. En caso contrario el compilador avisará del error. En la sentencia *PROCESS* se pone un ejemplo de un proceso que genera un reloj de periodo 2ns.

La opción “*WAIT ON ...*” es una alternativa a la lista sensible. La señales de la lista sensible irían en lugar de los puntos suspensivos. Se recomienda utilizar la lista sensible en lugar de esta sentencia.

### 2.7.4. Llamada a subprogramas [No entra]

En VHDL existen procedimientos (*PROCEDURE*) y funciones (*FUNCTION*). Un procedimiento no retorna ningún valor a diferencia de la función. El tipo de sentencias que contienen ambos son del tipo secuencial (como los procesos).

```

nombre_procedimiento[ (parámetros)] IS
nombre_función[ (parámetros)] RETURN tipo IS

```

Un subprograma puede ser llamado desde un entorno secuencial (proceso) o desde un entorno concurrente (arquitectura). En el primer caso se ejecuta el subprograma cada vez que es llamado. En el caso de ser llamado en un entorno recurrente el subprograma actúa como un proceso cuya lista sensible son los parámetros del subprograma de tipo entrada (IN e INOUT).

## 2.8. Subprogramas (PROCEDURE y FUNCTION) [No entra]

Lista de parámetros:

- Los parámetros pueden ser de entrada (IN, INOUT) o de salida (OUT, INOUT). Las funciones solo admiten parámetros de entrada. Si no se especifica nada se entienden que son de tipo IN (valor por defecto).
- Los parámetros pueden ser constantes, variables y señales. Si se omite el tipo se considera variable por defecto. La palabra CONSTANT se puede omitir pero debe ser de tipo entrada (IN).
- La lista de parámetros puede estar vacía. En este caso pueden omitirse los paréntesis.

Retorno de información:

- ❖ Una función solo devuelve un valor de un determinado tipo. Ese tipo se especifica en la propia declaración de la función.
- ❖ Una función requiere obligatoriamente la sentencia RETURN para retornar el dato. Los procedimientos pueden utilizar la sentencia RETURN pero sin opciones con la intención de parar la ejecución del procedimiento en un punto determinado y salir de él.
- ❖ Los procedimientos pueden devolver valores a través de sus parámetros de salida (OUT). Hay que tener mucho cuidado con los posibles efectos colaterales ya que un procedimiento puede modificar el valor de un objeto externo a él que, a su vez, puede ser modificado por la propia arquitectura desde la que se llama al procedimiento.

La estructura genérica de una función y un procedimiento son las siguiente:

```
PROCEDURE nombre [(parámetros)] IS
  declaraciones (solo variables)
BEGIN
  sentencias secuenciales
END nombre;
```

```
FUNCTION nombre [(parámetros)] RETURN tipo IS
  Declaraciones (solo variables)
BEGIN
  sentencias secuenciales -- incluye RETURN
END nombre;
```

Ejemplo de un procedimiento que devuelve el valor entero máximo y mínimo de un vector de enteros definido como *vectInt*.

```
PROCEDURE extremos(CONSTANT dato: IN vectInt; VARIABLE min,max: OUT integer) IS
VARIABLE minimo,maximo: INTEGER; -- variables internas (no ponerles tildes)
BEGIN
  minimo:=dato(dato'LEFT); -- valores iniciales de min y max
  maximo:= dato(dato'RIGHT);
  FOR i IN dato'RANGE LOOP
    IF (minimo > dato (i)) THEN minimo:= dato(i); END IF;
    IF (maximo < dato (i)) THEN maximo:= dato(i); END IF;
  ENDLOOP;
  min:= minimo;
  max:= maximo;
END extremos;
```

Dos posibles formas de llamar al procedimiento anterior pasándole los parámetros: *grupo* (de tipo vectInt), *min\_val* y *max\_val* (de tipo entero):

```
extremos(grupo, min_val, max_val); -- los parámetros en orden
extremos(min=>min_val, dato=> grupo, max=>max_val); -- desordenados
```

Ejemplo de una función que devuelve el valor de una señal de 4 bits como un valor entero sin signo.

```

FUNCTION SL_a_INT(x: IN std_logic_vector(2 DOWNTO 0)) RETURN INTEGER IS

VARIABLE x_sin_signo: UNSIGNED(2 DOWNTO 0);  -- como 'x' pero sin signo
VARIABLE resultado: INTEGER;

BEGIN
    x_sin_signo:= unsigned(x);
    resultado:= conv_integer(x_sin_signo);

    RETURN resultado;
END SL_a_INT;

```

### 3. Fichero de estímulos

Una vez realizado el diseño de un circuito en VHDL el siguiente paso consiste en la simulación del mismo. Se necesitará otro fichero VHDL que contenga los estímulos y que se conoce con el nombre de banco de pruebas (*test bench*). Los estímulos describen como las entradas del diseño cambian en el tiempo. El banco de pruebas permite comprobar si la salida obtenida por el modelo diseñado es la deseada o no; dando el correspondiente aviso de error.

La sintaxis del fichero de estímulos es idéntica a la de cualquier fichero escrito en lenguaje VHDL. Consta, por tanto, de tres partes diferenciadas:

- **Librerías.**
- **Entidad.** No tiene entradas ni salidas en su caso más simple. Su nombre no puede coincidir con el nombre de la entidad que describe el circuito. Es habitual llamarla *test* o *top*. El nombre de esta entidad es la información que hay que darle al entorno de trabajo para realizar la simulación.
- **Arquitectura.** Contiene un único componente correspondiente a la entidad a simular. También tiene señales de la propia arquitectura que se corresponden con las entradas y salidas del diseño. Normalmente suelen llamarse igual las señales de la arquitectura y las del componente para facilitar la legibilidad del diseño. Para diferenciarlas se han puesto en mayúsculas las señales propias de la arquitectura (son señales internas, por tanto, no son ni tipo IN ni tipo OUT). Estas señales de la arquitectura se pasan como parámetros al componente cuando se instancia (se crea). De esta forma al darle valores a las señales de la arquitectura -accesibles por ésta- conectadas a las entradas del componente, se obtiene la respuesta de éste de forma visible en las señales de la arquitectura conectadas a la salida del componente. Además suele tener dos procesos: (a) uno para definir el reloj -en el caso de que el circuito sea síncrono-, y (b) otro para indicar la evolución de resto de entradas. Los procesos presentes en la arquitectura utilizan la instrucción **WAIT** y, por razones que se estudiarán en cursos superiores, no pueden tener lista sensible.

Para concretar estas ideas se presenta el fichero del diseño y de la simulación de un circuito muy simple: una puerta AND de 2 entradas. En la izquierda se muestra el contenido del fichero *and2.vhd* que describe la lógica del circuito y en la derecha tenemos el contenido del fichero de estímulos que lo hemos llamado *and2\_tb.vhd*. El circuito es puramente combinacional por lo que no se requiere un proceso que genere señal de reloj.

<b>and2.vhd</b>
-----------------

<b>and2_tb.vhd</b>
--------------------



```

LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;

ENTITY and2 IS
PORT (
    a,b:IN std_logic;
    s :OUTstd_logic);
END and2;

ARCHITECTUREand2_arqof and2IS
BEGIN
s <= a AND b;
END and2_arq;

```

```

LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;

ENTITY test_and2 IS --No tiene entradas ni salidas
END test_and2;

ARCHITECTUREtest_and2_arqof test_and2IS
    COMPONENT and2
    PORT (
        a,b:INstd_logic;
        s :OUTstd_logic);
    END COMPONENT;

-- Entradas
SIGNAL a_test: std_logic;
SIGNAL b_test: std_logic;
-- Salida
SIGNALs_test : std_logic;
BEGIN
    -- Se instancia el componente llamado puerta del tipo and2
    -- La descripción del componente está en el fichero and2.vhd
    U1: and2 PORT MAP(a =>a_test,
                     b =>b_test,
                     s =>s_test);

    PROCESS -- Sin lista sensible (por la sentencia WAIT)
    BEGIN
        a_test<= '0';
        b_test<= '0';    WAIT FOR 100ns;
        b_test<= '1';    WAIT FOR 50ns;
        a_test<= '1';    WAIT FOR 200ns;
        -- ASSERT comprueba si la salida es la correcta
        -- para la combinación de entradas actual
        ASSERT (s_test='1') REPORT "salida <>'1'";
        b_test<= '0';    WAIT FOR 50ns;
        a_test<= '0';    WAIT; -- Espera infinita
    END PROCESS;

END test_and2_arq;

```

El resultado de la simulación de la AND2 con los estímulos del fichero *and2\_tb.vhd* semuestra a continuación:

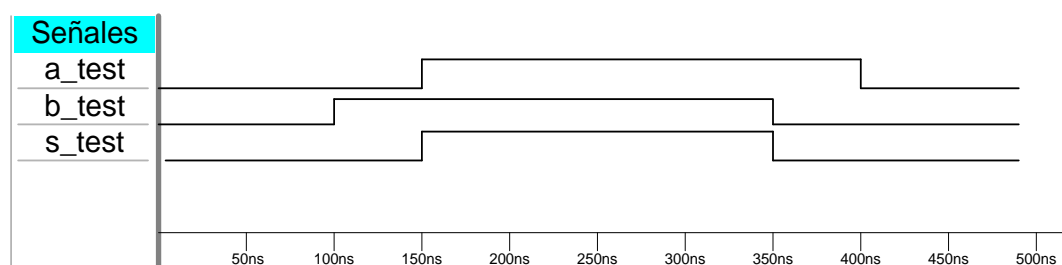


Figura: Resultado de la simulación del diseño de la AND de 2 entradas.

En el siguiente ejemplo se va a diseñar una puerta AND de tamaño variable de entradas. Se van a crear 2 puertas AND con 2 y 4 entradas en el fichero de estímulos, respectivamente. También se muestran las curvas de la simulación del fichero de estímulos.

## andx.vhd

```

LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;

ENTITY andx IS
GENERIC (N: INTEGER:= 2 ); -- Valor por defecto es 2
PORT (
    entradas :IN  std_logic_vector(N-1 DOWNT0 0);
    salida:OUTstd_logic      -- No lleva ';' porque viene un ')'
);
END andx;

ARCHITECTURE andx_arq of andx IS
BEGIN
    PROCESS(entradas)
    BEGIN
        salida<= '1'; -- salida=1, inicialmente. Puede cambiar más adelante
        FOR i IN 0 TO N-1 LOOP-- Alternativa: "FOR i IN entradas'RANGE LOOP"
            IF (entradas(i)='0') THEN
                salida <= '0';-- Se pone a la izquierda de la asignación (tipo OUT)
                EXIT;-- Salir del bucle FOR
            END IF;
        END LOOP;
    END andx_arq;

```

## andx\_tb.vhd

```

LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL; -- sentencia "conv_std_logic_vector(integer,tamaño)"

ENTITY test_andx IS -- No tiene entradas ni salidas
END test_andx;

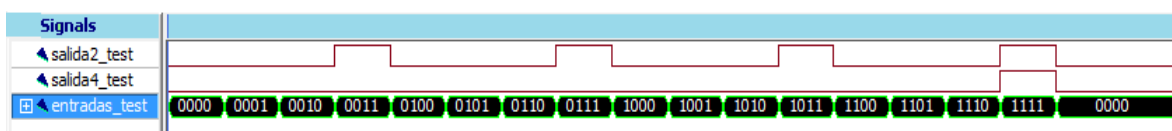
ARCHITECTURE test_andx_arqof test_andx IS
-- Declaración de componentes
COMPONENT andx
GENERIC ( N: INTEGER := 2 ); -- Valor por defecto es 2
PORT (
    entradas:IN  std_logic_vector(N-1 DOWNT0 0);
    salidas :OUTstd_logic );-- No lleva ';' porque viene un ')'
END COMPONENT;
-- Señales internas de la arquitectura que se pasarán como
-- parámetros a los componentes
SIGNAL entradas_test:std_logic_vector(3 DOWNT0 0); -- vector de 4 bits
SIGNAL salida2_test:std_logic;-- salida de la puerta AND2
SIGNAL salida4_test:std_logic; -- salida de la puerta AND4

CONSTANT ciclo: TIME := 100 ns;
BEGIN
-- Se crean 2 puertas AND de 2 y 4 entradas, respectivamente
U2: andxGENERIC MAP (N =>2) PORT MAP( entradas=>entradas_test(1 DOWNT0 0),
salida =>salida2_test); -- AND2
U4: andxGENERIC MAP(N =>4) PORT MAP(entradas =>entradas_test(1 DOWNT0 0),
salida =>salida4_test); -- AND4

PROCESS -- Sin lista sensible ya que contiene la sentencia WAIT.
BEGIN-- En principio se repetiría indefinidamente al no tener lista sensible
    FOR i IN 0 TO 15 LOOP
        entradas_test <= conv_std_logic_vector(i,4);--convierte "int" a vector 4 bits
        WAIT FOR ciclo;
    END LOOP;

    entradas_test<= "0000";-- entradas quedan a nivel bajo
    WAIT;-- Espera infinita -> El proceso NO SE REPITE
END PROCESS;
END test_andx_arq;

```



## 4. Descripción de circuitos combinacionales

Los circuitos combinacionales son aquellos que no almacenan información. No utilizan biestables y, por tanto, no requieren señal de sincronismo (reloj) ni señal de inicio (reset). En estos circuitos se tiene un valor en la salida que depende exclusivamente del valor actual que hay en las señales de entrada. En los siguientes apartados se verán algunos de los circuitos combinacionales más utilizados.

### 4.1. Conexión de varias salidas a una línea bus

Cuando varias salidas se cablean a una misma línea bus existe peligro de cortocircuito -entre dichas líneas de salida- salvo que dicho cableado se realice con cierto control. Existen dos posibilidades:

- salidas a drenador abierto.
- salidas con control de alta impedancia (tres estados: '0', '1' y 'Z')

#### Salidas a drenador abierto

En este caso cualquier salida consta de un transistor nMos que actúa como interruptor. Si el transistor conduce (valor lógico alto en la puerta; interruptor cerrado) entonces dicho transistor fija la tensión de la línea bus a tierra (nivel bajo). En el caso de que el transistor no conduzca (valor lógico bajo en la puerta; transistor cortado; interruptor abierto), la salida del mismo queda en un estado de alta impedancia. El nivel lógico de la línea bus dependerá de la resistencia externa de pull-up (valores típicos son 4K7 o 10k) y de otras salidas conectadas a dicha línea. Si todas las salidas conectadas a la línea bus están en alta impedancia, entonces la línea tomaría el valor lógico alto gracias a la resistencia de pull-up. Basta que una (o varias) salidas estén activas para que el transistor o transistores correspondientes fijen la línea bus a nivel lógico bajo. La función lógica que se obtiene en la línea bus es la AND lógica de las señales de entrada.

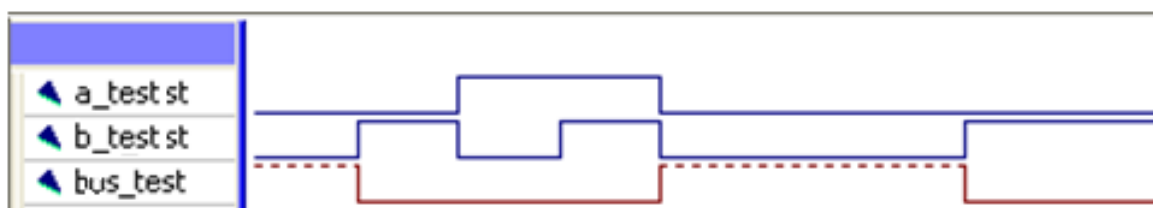
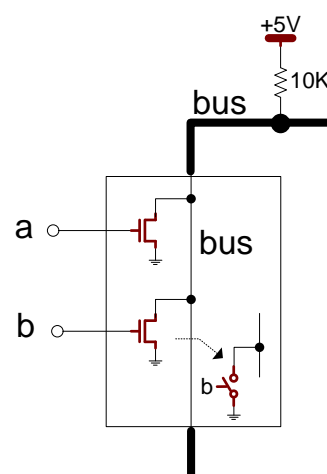
```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY drenadorAbierto IS
  PORT (
    a,b : IN std_logic;
    bus:OUT std_logic);
END drenadorAbierto;

ARCHITECTURE drenadorAbierto_arq OF drenadorAbierto IS
  BEGIN
    bus <= 'H' WHEN (a='0') AND (b='0') ELSE '0' ;
  END ARCHITECTURE drenadorAbierto_arq;

```



### Salidas con control de alta impedancia

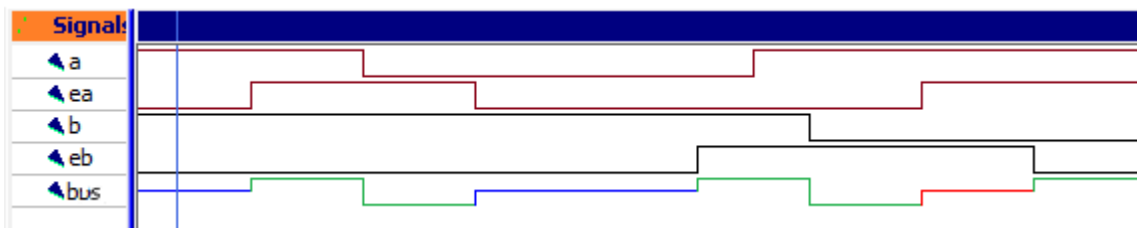
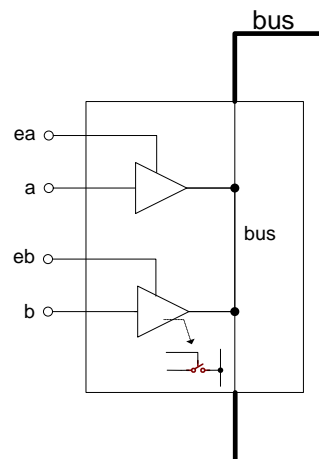
El siguiente diseño es de un circuito que gestiona dos señales de salida que se conectan a una misma línea bus mediante dispositivos que permiten un estado de alta impedancia. La salida toma valor 'Z' (alta impedancia) cuando el habilitador correspondiente toma el valor '0'. El caso de que dos o más habilitadores (si hubiera más dispositivos) estuvieran activossimultáneamente no puede ocurrir ya que generaría el cortocircuito de esas salidas. Para este caso se indica que la salida toma valor indeterminado '-'.

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY triestadoIS
PORT (
    a,b,ea,eb : IN std_logic;
    bus:OUT std_logic
);
END triestado;

ARCHITECTURE triestado_arqOF triestadoIS
BEGIN
    bus <= 'Z'WHEN (ea='0') AND (eb='0') ELSE
    '1'WHEN (ea='1') AND (eb='1') ELSE
    a WHEN (ea='1') ELSE
    b ;
END triestado_arq;
  
```



## 4.2. Multiplexor

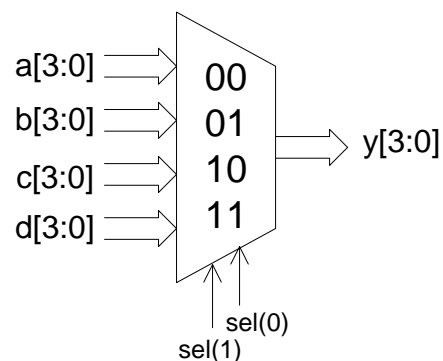
Como ejemplo, se realizará la descripción VHDL de un multiplexor de 4 a 1, cuyas entradas y salidas son elementos de 4 bits. A continuación se muestra el código VHDL del circuito y el esquema del circuito. La sentencia más apropiada es la asignación condicional.

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY muxIS
PORT (
    a,b,c,d : IN std_logic_vector(3 DOWNTO 0);
    sel : IN std_logic_vector(1 DOWNTO 0);
    y : OUT std_logic_vector(3 DOWNTO 0)
);
END mux;

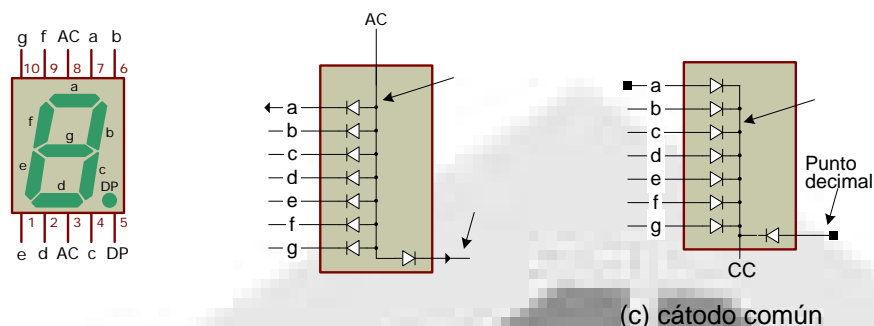
ARCHITECTURE mux_arqOF muxIS
BEGIN
    y <= a WHEN (sel="00") ELSE
    b WHEN (sel="01") ELSE
    c WHEN (sel="10") ELSE
    d ;
END mux_arq;
  
```



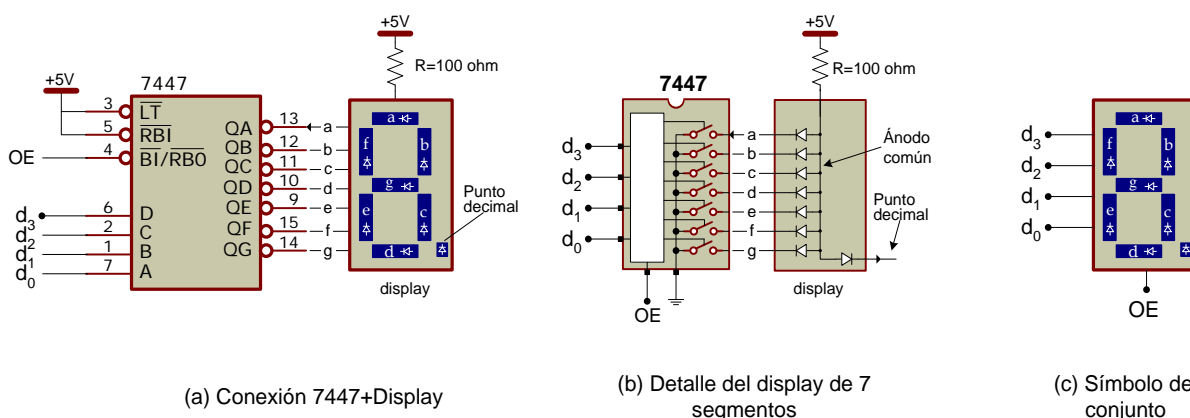
### 4.3. Convertidor BCD a 7 segmentos

El objetivo es diseñar la circuitería correspondiente al convertidor de código comercial 7447 pero con algunas simplificaciones. Previamente se va a presentar el display de 7 segmentos y su conexión al convertidor BCD a 7 segmentos 7447.

Un display de 7 segmentos es un dispositivo que muestra un dígito numérico en formato hexadecimal (0-9 y A-F). Contienen 7 diodos leds –uno para iluminar cada segmento– más un led adicional que permite iluminar el punto decimal –si no se necesita se deja sin conectar–. En el mercado nos encontramos con dos opciones: (a) ánodo común (AC) y (b) cátodo común (CC). Se utiliza preferentemente la opción de ánodo común ya que la corriente que necesitan los segmentos es suministrada por la fuente de alimentación. En este caso los ánodos de los diodos están unidos por lo que tienen un pin en común como se aprecia en la siguiente figura.



El display de 7 segmentos dispone de 7 pines de salida para los 7 segmentos. El punto decimal necesitaría una conexión más si la aplicación lo requiriese. En el caso de que no se dispongan de tantos pines libres en el sistema para gobernar el display se puede utilizar el convertidor de código 7447. Este dispositivo recibe un número BCD en su entrada y genera a la salida los niveles lógicos adecuados para cada uno de los 7 segmentos del display. En la siguiente figura se muestra como se conecta este dispositivo a un display de ánodo común.



El 7447 recibe una entrada de 4 bits –codifica un número del 0 al F– y genera las salidas para que se enciendan los leds correspondientes a dicho número. En realidad lo que hace el 7447 es poner a tierra el cátodo de aquel led –segmento– que ha de iluminarse. Las 7 salidas del 7447 son a colector abierto lo cual permite ponerlas en alta impedancia cuando la señal OE toma valor 0. En este caso los 7 leds del display quedan a circuito abierto y se apagan ya que no circula corriente por ellos.

La resistencia  $R$  de  $100\Omega$  se ha calculado para que la intensidad que circula por un diodo led de un determinado segmento sea de  $5\text{mA}$ . Con este nivel de intensidad se garantiza que un segmento cualquiera tenga la suficiente luminosidad para ser apreciado por el ojo humano. Para el cálculo de la resistencia hacen falta 3 datos más: (a) la caída de un diodo led cuando conduce es de  $1.5\text{V}$ , (b) el caso más desfavorable se produce cuando se encienden los 7 segmentos del display para mostrar el número 8, y (c) la caída de tensión dentro del 7447 para unir el cátodo de un determinado led a tierra es despreciable.

$$R = \frac{5\text{V} - 1.5\text{V}(\text{led})}{5\text{mA} * 7 \text{ segmentos (leds)}} = 100\Omega$$

En la implementación del 7447 en VHDL no se utilizarán las entradas de control  $\overline{LT}$ ,  $\overline{RBI}$  y  $\overline{BI}/\overline{RB0}$ .

```

LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;

ENTITY bcd7seg IS
PORT (
    bcd :IN std_logic_vector(3 downto 0); -- entrada BCD
    segment7 : OUT std_logic_vector(6 downto 0) -- salida 7 bit
);
END bcd7seg;

--'a' corresponde al MSB y g corresponde al LSB del display 7-segmentos

ARCHITECTURE bcd7seg_arq OF bcd7segIS
BEGIN
    PROCESS (bcd)
    BEGIN
        CASEbcd IS
            WHEN "0000"=>segment7<="1000000"; --'0' [gfedcba]="1000000"]
            WHEN "0001"=>segment7<="1111001"; --'1'
            WHEN "0010"=>segment7<="0100100"; --'2'
            WHEN "0011"=>segment7<="0110000"; --'3'
            WHEN "0100"=>segment7<="0011001"; --'4'
            WHEN "0101"=>segment7<="0010010"; --'5'
            WHEN "0110"=>segment7<="0000010"; --'6'
            WHEN "0111"=>segment7<="1111000"; --'7'
            WHEN "1000"=>segment7<="0000000"; --'8'
            WHEN "1001"=>segment7<="0010000"; --'9'

            -- si la entrada 'bcd' supera el número 9 se apagan los segmentos
            WHENOTHERS=>segment7<="1111111";
        END CASE;
    END PROCESS;
END bcd7seg_arq;

```

La instrucción CASE -utilizada para describir el convertidor de código- es secuencial y, por tanto, ha requerido un proceso que la encapsule. Otra opción para describir el mismo circuito sería la instrucción homóloga al CASE pero para entornos concurrentes. La instrucción WITH se escribe directamente en la arquitectura (no va dentro de un proceso). El código VHDL sería el siguiente:

```

ARCHITECTURE bcd7seg_arq OF bcd7seg IS
BEGIN
    WITH (bcd) SELECT
        segment7 <= "1000000" WHEN "0000",      -- '0' [gfedcba]="1000000"]
                    "1111001" WHEN "0001",      -- '1'
                    "0100100" WHEN "0010",      -- '2'
                    "0110000" WHEN "0011",      -- '3'
                    "0011001" WHEN "0100",      -- '4'
                    "0010010" WHEN "0101",      -- '5'
                    "0000010" WHEN "0110",      -- '6'
                    "1111000" WHEN "0111",      -- '7'
        "0000000" WHEN "1000",      -- '8'
                    "0010000" WHEN "1001",      -- '9'
                    "1111111" WHEN OTHERS;      -- Si es cualquier otro núm.
END bcd7seg_arq;

```

#### 4.4. Sumador

Se va a diseñar un sumador de 4 bits -sin signo- con acarreo de entrada y de salida. En el diseño se va a emplear el operador `+` por lo que habrá que incluir la librería `ieee.std_logic_unsigned.ALL` que lo contiene. Si no hubiera acarreos ni problemas de desbordamiento, el diseño sería tan simple como escribir la sentencia `suma <= a + b`.

```

LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY sumador4bits IS
PORT (
    a,b :IN  std_logic_vector(3 downto 0); -- entradas a sumar
    cin :IN  std_logic;                    -- acarreo en la entrada
    cout:OUT std_logic;                    -- acarreo de salida
    suma : OUT std_logic_vector(3 downto 0) -- resultado
);
END sumador4bits;

ARCHITECTURE sumador4bits_arq OF sumador4bits IS
    SIGNAL ax,bx,sumax: std_logic_vector(4 downto 0) --Señales de 5 bits
    CONSTANT cinx:std_logic_vector(4 downto 1) := "0000";
BEGIN
    ax <= '0' & a;      -- ax(4) <= '0';    ax(3 DOWNT0 0) <= a;
    bx <= '0' & b;      -- bx(4) <= '0';    bx(3 DOWNT0 0) <= b;

    sumax <= ax + bx + (cinx & cin); -- Suma tres objetos de 5 bits
    suma <= sumax(3 DOWNT0 0);      -- suma será los 4 LSBs de sumax
    cout <= sumax(4);               -- cout será el bit más signific. de sumax
END sumador4bits_arq;

```

El circuito sumador utilizará 3 señales internas llamadas *ax*, *bx* y *sumax* de tamaño un bit mayor que las originales. El acarreo se almacenará en el bit 4 (nuevo bit más significativo) de *sumax*. Supóngase que  $a=13$ ,  $b=9$  y  $ci='1'$ . El resultado de la suma desborda el tamaño de 4 bits. Se genera acarreo y el valor almacenado en *suma* = 7. En binario se ve mejor:

Operaciones internas (de 5 bits)		Resultado visible (de 4 bits)	
	(cinx & ci) 00001	cin	1
	ax 01101	a	1101
+	bx 01001	+ b	1001
	sumax 10111	suma cout=1, 0111	

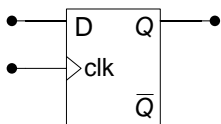
## 5. Descripción de circuitos secuenciales

Los circuitos secuenciales utilizan biestables que permiten almacenar información. Están sincronizados por una señal de reloj que fija el instante para que capturen sus entradas –y cambien su estado-. Este instante se corresponde con el flanco activo de la señal de reloj (normalmente el de subida). Estos circuitos también suelen utilizar una señal de entrada que asigne un valor inicial al estado de los biestables (reset).

Todos los circuitos síncronos descritos en VHDL suelen utilizar un proceso con la señal de reloj *clk* en su lista sensible, más una estructura condicional tipo IF que se activa cuando ocurre una transición en la señal de reloj, para lo que se utiliza el atributo *EVENT*.

### 5.1. Biestable tipo D

A continuación se detallan dos versiones equivalentes del diseño VHDL para el biestable tipo *d* activo por flanco de subida del reloj. Una versión con la sentencia IF y la otra con la sentencia IF-ELSE.



```

LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;

ENTITY dff IS
PORT (
    d,clk:IN  std_logic;
    q      :OUTstd_logic;
);
END dff;

ARCHITECTURE dff_arqof dff IS
BEGIN
    PROCESS (clk)
    BEGIN
        IF (clk'EVENT AND clk='1') THEN
            q <= d;
        END IF;
    END PROCESS;
END dff_arq;

```

```

LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;

ENTITY dff IS
PORT (
    d,clk:IN  std_logic;
    q      :OUTstd_logic
);
END dff;

ARCHITECTURE dff_arqof dff IS
    SIGNAL q_aux : std_logic;
BEGIN
    PROCESS (clk)
    BEGIN
        IF (clk'EVENT AND clk='1') THEN
            q_aux <= d;
        ELSE
            q_aux <= q_aux;
        END IF;
    END PROCESS;
    q <= q_aux; -- q a la izq. (salida)
END dff_arq;

```

Los dos diseños generan un biestable tipo D pero es interesante observar que en la primera opción no se utiliza ELSE en la estructura condicional. Esto es posible porque estamos definiendo un elemento de memoria que debe conservar su valor en el caso de que no se verifique la condición del IF (falta el ELSE).

En la segunda opción se ha utilizado el ELSE para fijar de forma explícita qué debe hacer el biestable cuando no ocurre un flanco positivo de reloj (la salida se queda como está). En este caso es necesario utilizar una señal interna (*q\_aux*) ya que la salida (*q*) no puede aparecer a la derecha de una asignación salvo que se declare como BUFFER. Sin embargo el tipo BUFFER no se utilizará porque algunos compiladores no lo admiten. La asignación *q\_aux <= q\_aux* si es válida porque esta señal es interna.

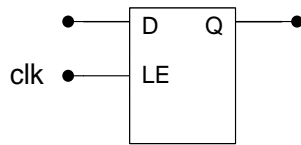
La expresión *clk'EVENT* y el hecho de incluir la señal *clk* en la lista sensible son redundantes. Es decir, ambas sirven para detectar un evento sobre la señal *clk*. La explicación de esta redundancia en el código es que algunas herramientas de síntesis ignoran la lista sensible, por lo que es necesaria esta expresión *clk'EVENT*.

### Biestable tipo D activo por nivel: latch

Si en lugar de describir un biestable activo por nivel se desea describir uno activo por nivel (latch), sería preciso eliminar la condición *clk'EVENT* e incluir la entrada '*d*' en la



listasensible. Recuerdese que en el caso de un latch la salida se modifica siempre que varíe la entrada y que el reloj se encuentre en su nivel activo.



```

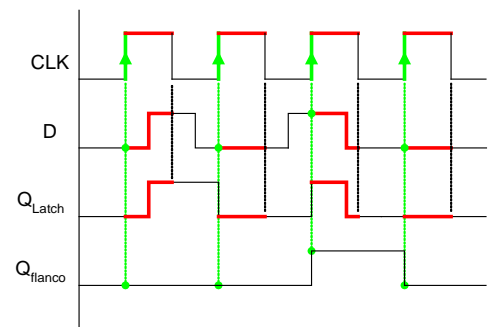
LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;

ENTITY latch IS
PORT (
    d, clk: IN std_logic;
    q: OUT std_logic;
);
END latch;

ARCHITECTURE dff_arq of latch IS
BEGIN
    PROCESS (clk, d)
    BEGIN
        IF (clk='1') THEN
            q <= d;
        END IF;
    END PROCESS;
END dff_arq;

```

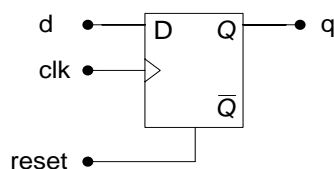
El biestable tipo D por nivel de reloj (Latch) ocupa entre 2 y 3 veces menos área de silicio por lo que es más barato. Pero carece del concepto de sincronismo con la señal de reloj (actualizar su salida en un instante -flanco activo del reloj-). Para comprobar la diferencia de comportamiento entre los dos tipos de biestable tipo D se estimularán con la misma entrada D y la misma señal de reloj. La salida puede ser muy diferente. Supongamos que inicialmente la salida de ambos biestables es '0'.



## Biestable tipo d con reset asíncrono

El siguiente código VHDL ilustra cómo describir un biestable con un reset (o preset) asíncrono, esto es, capaz de poner a cero (o a uno) el biestable, independientemente de la señal de reloj.

En el siguiente código VHDL se ha diseñado un biestable activo por flanco de subida de reloj y con señal de reset activa a nivel alto. Ambas señales de entrada deben figurar en la lista sensible.



```

LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;

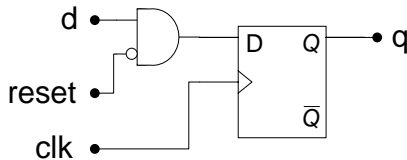
ENTITY dff IS
PORT (
    d, clk, reset: IN std_logic;
    q: OUT std_logic );
END dff;

ARCHITECTURE dff_arq of dff IS
BEGIN
    PROCESS (clk, reset)
    BEGIN
        IF (reset='1') THEN
            q <= '0';
            -- Equivale a (clk'EVENT AND clk='1')
            ELSIF rising_edge(clk) THEN
                q <= d;
            END IF;
        END PROCESS;
END dff_arq;

```

## Biestable tipo d con reset síncrono

En este caso se trata de describir un biestable con reset (o preset) síncrono. Es decir, unbiestable que se pone a '0' o '1' si la señal de reset o preset se activa y ocurre un flanco activo de reloj.



```

LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;

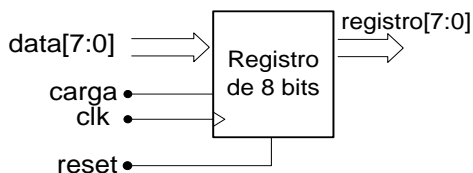
ENTITY dffIS
PORT (
    d,clk,reset:IN std_logic;
    q          :OUTstd_logic);
END dff;

ARCHITECTURE dff_arqof dffIS
BEGIN
    PROCESS (clk)
    BEGIN
        IF (clk'EVENT AND clk='1') THEN
            IF (reset='1') THEN
                q<= '0';
            ELSE
                q<= d;
            END IF;
        END IF;
    END PROCESS;
END dff_arq;

```

## 5.2. Registro de 8 bits con reset asíncrono

La descripción de un registro es análoga a la de un flip-flop, pero definiendo señales como vectores de la forma *std\_logic\_vector*(n-1 DOWNT0 0)). El reset asíncrono solo se ejecuta al principio del programa una sola vez para llevar los biestables a un estado inicial (nomalmente '0').



```

LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;

ENTITY registro8 IS
PORT (
    clk, reset, carga:INstd_logic;
    data :IN std_logic_vector(7 downto 0);
    registro:OUT std_logic_vector(7 downto 0) );
END registro8;

ARCHITECTUREregistro8_arqof registro8 IS
BEGIN
    PROCESS (clk,reset)
    BEGIN
        IF (reset='1') THEN
            registro <= "00000000";
        ELSIF (clk'EVENT AND clk='1') THEN
            IF (carga='1') THEN
                registro <=data;
            END IF;
        END IF;
    END PROCESS;
END registro8_arq;

```

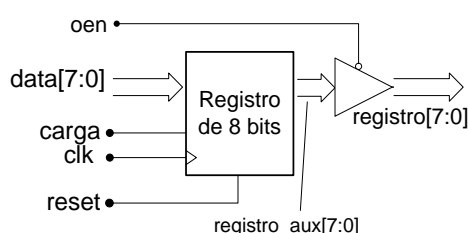
En la sentencia de asignación del reset se podría utilizar la palabra reservada *OTHERS*. En este caso los 8 bits del registro se pueden asignar a '0' escribiendo –de forma alternativa–: `registro<= (OTHERS=>'0');` La palabra reservada *OTHERS* implica que todos los bits del registro se pondrán al valor especificado independientemente del tamaño del registro (útil si el tamaño depende de la sentencia *GENERIC*). El siguiente ejemplo fija los bits más y menos significativos a '1', y el resto a '0':

```
registro<= (7=>'1', 0=>'1', OTHERS=>'0'); --registro <= "10000001";
```

### 5.3. Registro de 8 bits con reset asíncrono, señal de carga y salida en alta impedancia

Sobre el registro definido anteriormente, introduciremos una nueva funcionalidad: salidas en alta impedancia. Para que la herramienta de síntesis traduzca a puertas el código VHDL deberá contar con elementos de librería con salidas en alta impedancia, lo que es habitual en los dispositivos programables de baja capacidad.

Para seleccionar entre el estado normal o de alta impedancia de una salida, el registro debe poseer una entrada adicional, que es la entrada 'oen' (output-enable activa a nivel bajo). En el siguiente ejemplo, cuando dicha señal está a '0' habilita la salida normal del registro, pero cuando está a '1' deja la salida en alta impedancia.



```

LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;

ENTITY registro8 IS
PORT (
    clk, reset, carga, oen: IN std_logic;
    data : IN std_logic_vector(7 downto 0);
    registro: OUT std_logic_vector(7 downto 0) );
END registro8;

ARCHITECTURE registro8_arq of registro8 IS
registro_aux: std_logic_vector(7 downto 0);
BEGIN
    -- =====
    -- control de la alta impedancia
    -- =====
    registro <= registro_aux WHEN (oen='0') ELSE "ZZZZZZZZ";

    -- =====
    -- Proceso que gestiona el registro
    -- =====
    PROCESS (clk, reset)
    BEGIN
        IF (reset='1') THEN
            registro_aux <= (OTHERS=>'0'); -- "00000000"
            ELSIF (clk'EVENT AND clk='1') THEN
                IF (carga='1') THEN
                    registro_aux <= data;
                END IF;
            END IF;
        END PROCESS;
    END registro8_arq;

```

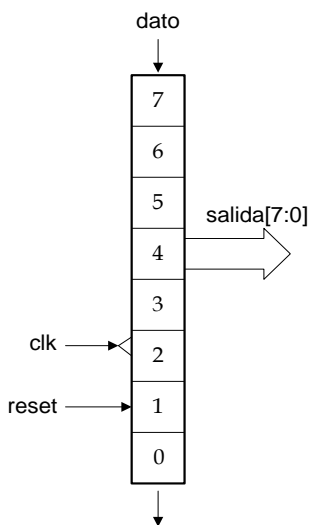
El cuerpo de la arquitectura es un ejemplo de dos procesos que se interpretan de manera concurrente. Este segundo proceso es una asignación concurrente condicional que sirve para definir las salidas bien en su estado normal o bien en alta impedancia. La señal *oen* es el control de la salida en alta impedancia y *registro\_aux* se ha definido como una señal interna de la arquitectura para apoyarnos en ella. Cuando *oen* es '1' utilizamos la asignación `registro <= "ZZZZZZZZ"`, donde el valor Z se entiende como alta impedancia y se encuentra definido en el paquete estándar *std\_logic\_1164*.

Respecto al ejemplo anterior, la señal *registro* continúa en la lista de puertos. La herramienta de síntesis incluirá búferes triestado a la salida.

### 5.4. Registro de desplazamiento de 8 bits con reset asíncrono

El listado presentado a continuación describe un registro de desplazamiento con carga por el bit más significativo y desplazamiento a la derecha. En principio la señal *salida* debería ser de tipo *buffer* ya que debería aparecer a la derecha de una instrucción de asignación. Para evitarlo se declara como tipo OUT y se añade una señal auxiliar

interna de la arquitectura llamada *salida\_aux*. En el apartado del registro de desplazamiento de 8 bits y el del contador, que se detallarán más adelante, se tienen ejemplos sobre como se implementan salidas de tipo *out*.



```

LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;

ENTITY desplaza8 IS
PORT (
    clk, reset, dato:INstd_logic;
    salida:OUT std_logic_vector(7 downto 0) );
END desplaza8;

ARCHITECTURE desplaza8_arqof desplaza8 IS
SIGNAL salida_aux : std_logic_vector(7 downto 0);
BEGIN
    salida<= salida_aux;-- asignación de la señal

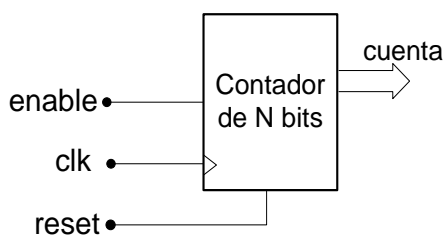
PROCESS (clk,reset)
BEGIN
    IF (reset='1') THEN
        salida_aux <= "00000000";    -- (OTHERS=>'0');
    ELSIF (clk'EVENT AND clk='1') THEN
        salida_aux (6DOWNT0 0) <=salida_aux (7DOWNT0 1);
        salida_aux (7) <=dato;

        -- En lugar de las dos línea anteriores se podría usar
        --salida_aux <= dato &salida_aux(7 DOWNT0 1);
    END IF;
END PROCESS;
END desplaza8_arq;

```

## 5.5. Contador de N bits con reset asíncrono y señal de habilitación

En este diseño se necesita utilizar el operador suma por lo que es necesaria la librería *std\_logic\_unsigned*. Algunos entornos de programación requieren también la librería *std\_logic\_arith*. A continuación se muestra un esquema del circuito y su código en VHDL.



```

LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;-- Necesaria (+)

ENTITY contadorN IS
GENERIC (N: INTEGER := 4);    -- Tamaño por defecto
PORT (
    clk, reset,enable:INstd_logic;
    cuenta:OUT std_logic_vector(N-1 downto 0) );
END contadorN;

ARCHITECTURE contador_arqof contadorN IS
SIGNAL cnt: std_logic_vector(N-1 downto 0); BEGIN
PROCESS (clk,reset)
BEGIN
    IF (reset='1') THEN
        cnt <= (OTHERS =>'0');-- Los N bits a '0'
    ELSIF (clk'EVENT AND clk='1') THEN
        IF (enable='1') THEN
            cnt<= cnt + 1;
        END IF;
    END IF;
    -- =====
    cuenta<= cnt;-- Está a la derecha de una assign.
    -- =====
END PROCESS;
END contador_arq;

```

A continuación se muestra un fichero de estímulos para el diseño anterior en el que se crean dos contadores de 3 y 4 bits, respectivamente. Ambos tienen la misma señal de reloj y reset pero el de 3 bits cuenta de 0 a 7 y el de 4 bits cuenta de 0 a F.

```

LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;-- Necesariapor el signo(+)

ENTITY test_contador IS
END test_contador;

ARCHITECTURE test_contador_arq OF test_contador IS
    --Declaración del componente
    COMPONENT contadorN
        GENERIC (N: INTEGER := 4);-- Tamaño por defecto
        PORT (
            clk, reset,enable:IN std_logic;
            cuenta :OUT std_logic_vector(N-1 downto 0) );
    END COMPONENT;

    SIGNAL SALIDA3_test: std_logic_vector(2downto 0); -- contador de 3 bits
    SIGNAL SALIDA4_test: std_logic_vector(3downto 0); -- contador de 4 bits
    SIGNAL CLK_test, RESET_test, ENABLE_test: std_logic;-- entradas comunes

    CONSTANT ciclo: TIME :=50 ns;

BEGIN
    -- =====
    -- Creación de components U3 y U4
    -- =====
    U3: contadorN GENERIC MAP (N =>3) PORT MAP(clk =>CLK_test,
reset=>RESET_test,enable=>ENABLE_test, cuenta =>SALIDA3_test);
    U4: contadorN GENERIC MAP (N =>4) PORT MAP(clk =>CLK_test,
reset=> RESET_test,enable=>ENABLE_test, cuenta =>SALIDA4_test);

    -- =====
    -- Proceso para la señal de reloj
    -- =====
PROCESS
    BEGIN
        CLK_test<='0'; wait for ciclo/2;
        CLK_test<='1'; wait for ciclo/2;
    END PROCESS;

    -- =====
    -- Proceso para los estímulos
    -- =====
PROCESS
    BEGIN
        RESET_test<= '1';
        ENABLE_test<= '1';
        WAITFOR 5*ciclo/4;-- reset y enable a '1'durante 1.25 ciclos

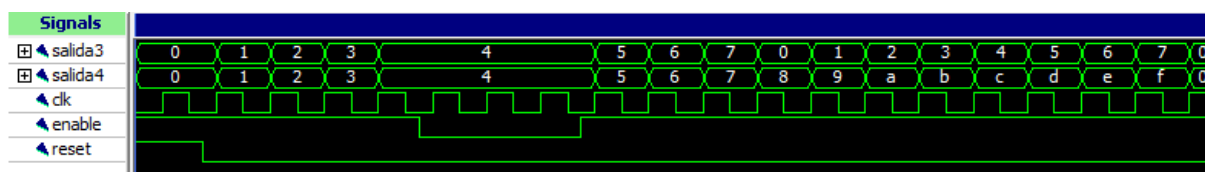
        RESET_test<='0';-- reset=0 y enable=1 -> puede contar
        WAITFOR 4*ciclo;-- 4 ciclos -> 4 flancos -> cuenta hasta 4

        ENABLE_test<='0';-- enable=0 -> desactivado el contador
        WAITFOR 3*ciclo; -- Deja de contar durante 3 ciclos

        ENABLE_test<='1';-- enable=1 -> reactivado el contador
        WAIT; -- Espera indefinida. El reloj sigue contando
    END PROCESS;
END test_contador_arq;

```

El resultado de la simulación anterior se muestra en la siguiente figura:



## 6. Diseño de máquinas de estado

El diseño de máquinas de estado se puede realizar de una forma muy simple en VHDL, y constituye un claro ejemplo de la potencia de los lenguajes de descripción hardware frente a los métodos tradicionales de diseño.

La metodología tradicional comienza por construir un diagrama de estados o diagrama de bolas, de la que se deriva una tabla de estados. Sobre esta tabla se pueden agrupar estados equivalentes, si se da el caso. A continuación se asignan los estados y se pasa a una tabla de transición de estados de la que se obtienen las ecuaciones lógicas de los estados siguientes y de las salidas según el tipo de biestable elegido para la implementación de los estados.

Se trata de un método algorítmico y, por tanto, susceptible de ser realizado por una herramienta software que libere del mismo al diseñador. Esto es precisamente lo que se consigue en VHDL y otros lenguajes HDL. El código VHDL describe el diagrama de estados. La herramienta de síntesis e implementación es la que realiza todo el trabajo algorítmico. Además, no comete errores como los humanos.

En general, las máquinas de estado se clasifican como máquinas de Moore y máquinas de Mealy. Las primeras se caracterizan porque las salidas dependen únicamente del estado, tanto que en las segundas dependen del estado y las entradas. En lo que sigue nos centraremos en las máquinas de Moore. La descripción de máquinas de Mealy únicamente se diferencia en la forma de expresar las señales de salida.

Las transiciones de estados se pueden expresar con una estructura condicional del tipo IF-THEN-ELSE ó, lo que es más habitual, con una estructura CASE-WHEN. El diagrama de bloques resultante para los diseños que se realicen, basados en máquinas de Moore, se muestra en la figura.

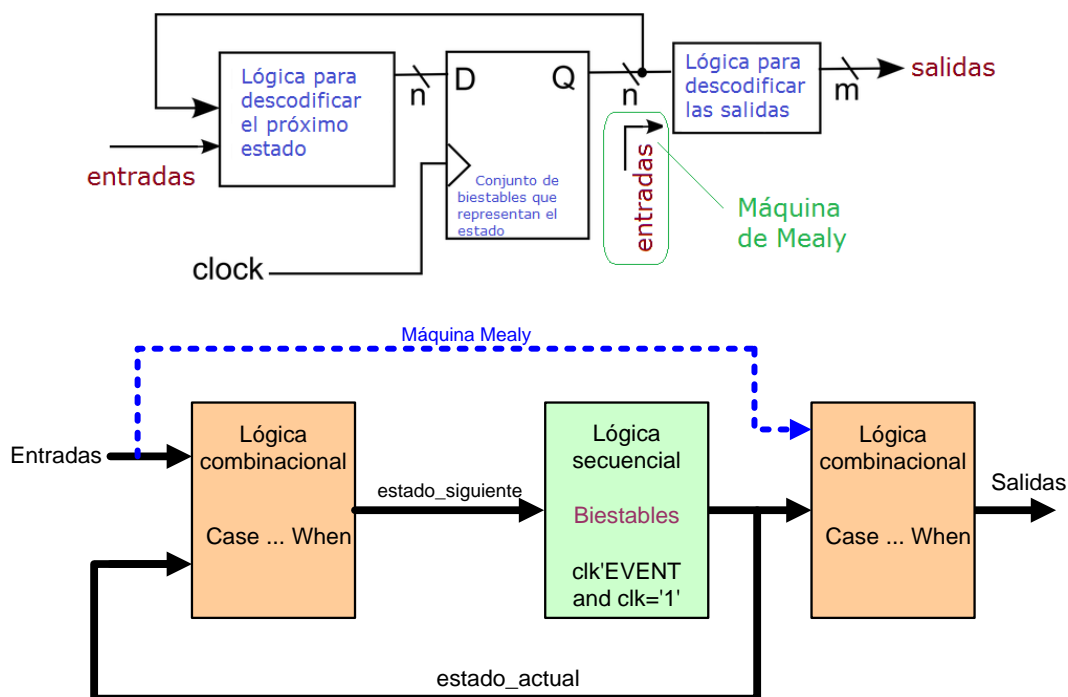


Figura : Estructura genérica de resolución de máquinas de estado

Los estados se definen por una enumeración que agrupa a todos los estados, y se declaran señales según ese tipo: una que corresponde a la salida del bloque

secuencial (estado\_**s**; es el estado actual de los biestables) y otra corresponde a la entrada del bloque secuencial. Esta señal contiene el estado siguiente y procede de un proceso combinacional(estado\_**c**).

```
TYPE ESTADOS IS (reposo,decide,escribe,lee);
SIGNAL estado_s, estado_c: ESTADOS;
```

Se utilizarán 3 bloques para diseñar una máquina de estados. Cada uno de estos bloques suele ser un proceso dentro de la arquitectura:

- Parte combinacional.
  - *Bloque combinacional de las entradas:* Lógica combinacional que va desde las entradas del circuito -declaradas en la entidad- hasta las entradas de los biestables que almacenan el estado.
  - *Bloque combinacional de las salidas:* Lógica combinacional que va desde las salidas de los biestables hasta las salidas del circuito -declaradas en la entidad-.
- Parte secuencial. Describe las transiciones síncronas de estados con el flanco de reloj. Este bloque es el único que recibe las entradas de reset y reloj.

La figura muestra el resultado genérico de la implementación de una máquina de Moore.

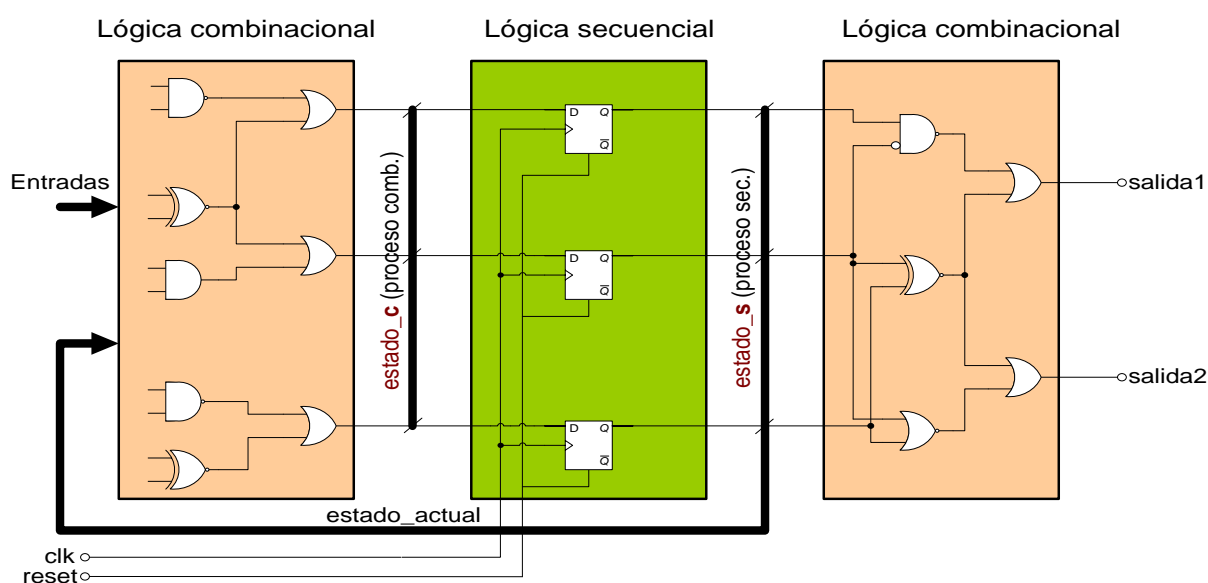


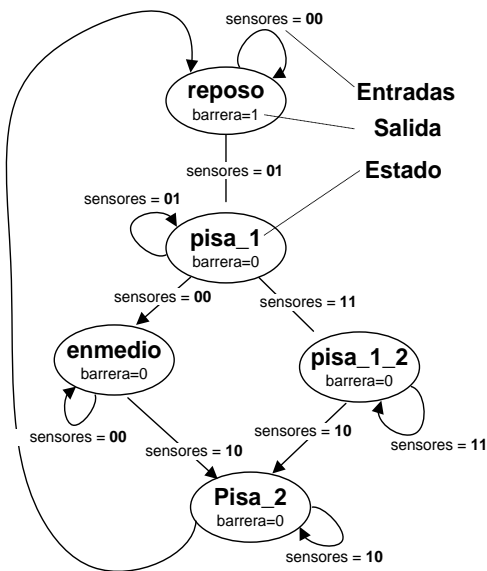
Figura: Esquema genérico de implementación en puertas de una máquina de Moore

## 6.1. Ejercicio del control de la barrera del tren

Como ejemplo, realizaremos una máquina de Moore consistente en el control de la barrera de un paso a nivel -una sola salida-. Se supondrá que el tren va en un único sentido y que nunca da marcha atrás. Las entradas son dos sensores: el sensor que está antes de la barrera (*sensores(1)*) y el sensor ubicado después de la barrera (*sensores(2)*). La salida del sistema estará a '0' en el estado de reposo (barrera abierta) y a '1' en el resto de estados (barrera cerrada). El diagrama de bolas de la máquina de Moore, correspondiente a este sistema, presenta una bifurcación debido a que puede darse el caso de que pase un tren largo que pise ambos sensores simultáneamente y el caso de que pase un tren corto cuya longitud sea menor a la distancia entre ambos sensores. La entrada de reset será activa a nivel bajo.

La descripción de cada uno de los estados es la siguiente:

- **reposo.** No ha llegado ningún tren todavía. Barrera abierta (salida).
- **pisa\_1.** El tren está pisando el primer sensor. Barrera cerrada.
- **pisa\_1\_2.** El tren es largo y está pisando los dos sensores. Barrera cerrada.
- **enmedio.** El tren es corto. Ha dejado de pisar el primer sensor pero no ha llegado al segundo. No está activo ninguno de los dos sensores. Barrera cerrada.
- **pisa\_2.** El tren está marchándose. Pisa el segundo sensor. Barrera cerrada.



El reset asíncrono debe utilizarse cuando el dispositivo sobre el que se va a implementar el circuito dispone de biestables con reset asíncrono. Por ejemplo la PAL22V10.

Los autómatas se implementarán con tres procesos: (a) Proceso secuencial. Se asignan la salida de los biestables (estado\_s), (b) Proceso combinacional de las salidas. Se asigna la salida (barrera) y (c) Proceso combinacional de los estados. Se asigna las entradas de los biestables (estado\_c). En nuestro caso se utilizará una máquina de Moore lo que implica que en el diagrama de estados la salida se representa dentro del estado (bola). En la máquina de Mealy las salidas irían en los arcos entre bolas (junto con las entradas). Si fuera una máquina de Mealy, en el proceso combinacional de los estados, habría que incluir las entradas en la lista sensible.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
```

```
ENTITY tren IS
```

```
PORT
```

```
clk, reset : IN std_logic;
sensores : IN std_logic_vector(2 DOWNTO 1);
barrera : OUT std_logic -- '1': abierta, '0': cerrada
);
```

```
END tren;
```

```
ARCHITECTURE tren_arq OF tren IS
```

```
TYPE ESTADOS IS (reposo, pisa_1, pisa_1_2, enmedio, pisa_2);
```

```
SIGNAL estado_s, estado_c : ESTADOS;
```

```
BEGIN
```

```
--Se asigna "estado_s". Proceso síncrono. Solo depende del reloj.
```

```
PROCESS (clk, reset)
```

```
BEGIN
```

```
IF (reset='0') THEN --Reset activo a nivel bajo
```

```
estado_s <= reposo;
```

```
ELSIF (clk'EVENT AND clk='1') THEN
```

```
estado_s <= estado_c;
```

```
END IF;
```

```
END PROCESS;
```

```
--Se asigna la salida "barrera". Proceso combinacional.
```

```
PROCESS (estado_s)
```

```
BEGIN
```

```
IF (estado_s = reposo) THEN
```

```
barrera <= '1'; --Barrera abierta (salida)
```

```
ELSE
```

```
barrera <= '0'; --Barrera cerrada (salida)
```

```
END IF;
```

```
END PROCESS;
```

```
--Se asigna "estado_c". Proceso combinacional
```

```
PROCESS (estado_s, sensores)
```

```
BEGIN
```

```
-- Valor por defecto
```

```
-- Valor por defecto
```

```
-- Valor por defecto
```

```
estado_c <= estado_s;
```

```
-- Valor por defecto
```

```
-- Valor por defecto
```

```
-- Valor por defecto
```

```
CASE estado_s IS
```

```
WHEN reposo =>
```

```
IF (sensores="01") THEN
```

```
estado_c <= pisa_1;
```

```
--ELSE-- innecesario
```

```
--estado_c <= reposo; -- innecesario
```

```
END IF;
```

```
WHEN pisa_1 =>
```

```
IF (sensores="00") THEN
```

```
estado_c <= enmedio;
```

```
ELSIF (sensores="11") THEN
```

```
estado_c <= pisa_1_2;
```

```
--ELSE-- innecesario
```

```
--estado_c <= pisa_1; -- innecesario
```

```
END IF;
```

```
WHEN pisa_1_2 =>
```

```
IF (sensores="10") THEN
```

```
estado_c <= pisa_2; --tren saliendo
```

```
--ELSE-- innecesario
```

```
--estado_c <= pisa_1_2; -- innecesario
```

```
END IF;
```

```
WHEN enmedio =>
```

```
IF (sensores="10") THEN
```

```
estado_c <= pisa_2; --tren saliendo
```

```
END IF;
```

```
WHEN pisa_2 =>
```

```
IF (sensores="00") THEN
```

```
estado_c <= reposo;
```

```
END IF;
```

```
WHEN OTHERS =>
```

```
estado_c <= reposo;
```

```
END CASE;
```

```
END IF;
```

```
END PROCESS;
```

```
END tren_arq;
```



```

LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;

ENTITY test_tren_corto IS
END test_tren_corto;

ARCHITECTURE test_tren_corto_arq OF test_tren_corto IS
--Declaración del componente
COMPONENT tren
PORT (sensores: IN std_logic std_logic_vector(2 downto 1);
clk, reset: IN std_logic;
barrera: OUT std_logic); -- No lleva ';' porque viene un ')'
ENDCOMPONENT;

SIGNAL SENSORES_test: std_logic_vector(2 downto 1);
SIGNAL CLK_test, RESET_test: std_logic;
SIGNAL BARRERA_test: std_logic;

SIGNAL FIN_test: std_logic := '0'; -- Se pondrá a '1' al finalizar la simulación
CONSTANT ciclo: TIME := 50 ns;
BEGIN
-- =====
-- Creación del componente U1 de tipo 'tren'
-- =====
U1: tren PORT MAP(sensores=>SENSORES_test, reset => RESET_test,
Clk=>CLK_test, barrera=>BARRERA_test);

-- =====
-- Proceso para la señal de reloj
-- =====
PROCESS
BEGIN
CLK_test<='0'; wait for ciclo/2;
CLK_test<='1'; wait for ciclo/2;
END PROCESS;

-- =====
-- Proceso para los estímulos
-- =====
PROCESS
BEGIN
SENSORES_test<= "00";
RESET_test<= '1';
WAITFOR 9*ciclo/4; -- reset activo a '1' durante 2.25 ciclos

RESET_test<='0'; -- Termina el reset.
WAITFOR 2*ciclo; -- 2 ciclos

-- Se pisa el sensor 1. Se pasa al estado 'pisa_1'. Barrera cerrada
SENSORES_test<= "01";
WAITFOR 3*ciclo;

-- No se pisa sensores. Se pasa al estado 'enmedio'. Tren corto
SENSORES_test<= "00";
WAITFOR 3*ciclo;

-- Se pisa el sensor 2. Se pasa al estado 'pisa_2'. Barrera cerrada
SENSORES_test<= "10";
WAITFOR 3*ciclo;

-- No se pisa sensores. El tren ya pasó. Barrera abierta
SENSORES_test<= "00";
WAITFOR 2*ciclo;

FIN_test<= '1'; -- Esta señal es artificial. Se utiliza para parar el reloj.
WAIT; -- Espera indefinida. El proceso no se repite
END PROCESS;
END test_tren_corto_arq;

```

El resultado de la simulación de los estímulos para el tren corto sería la siguiente:

