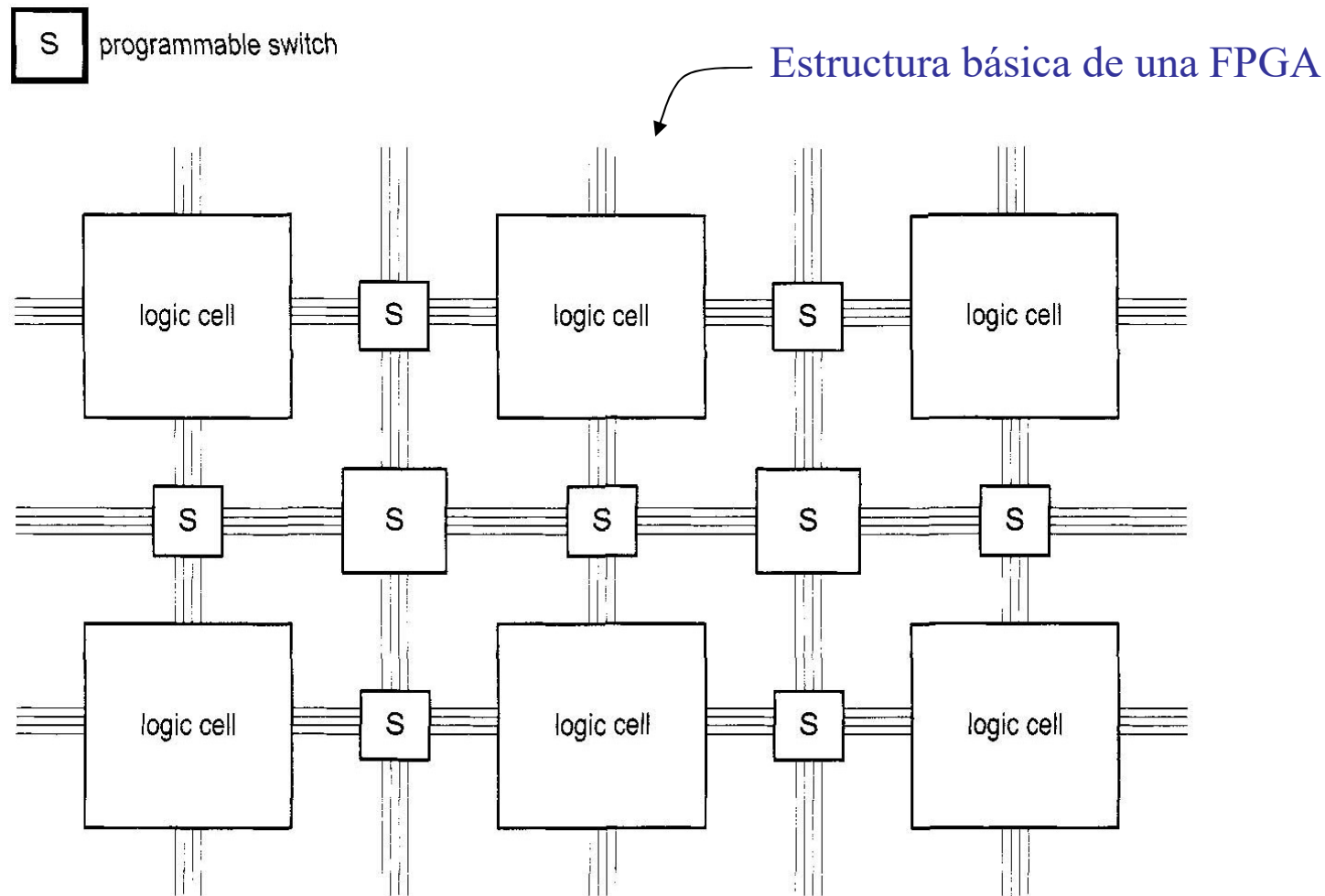
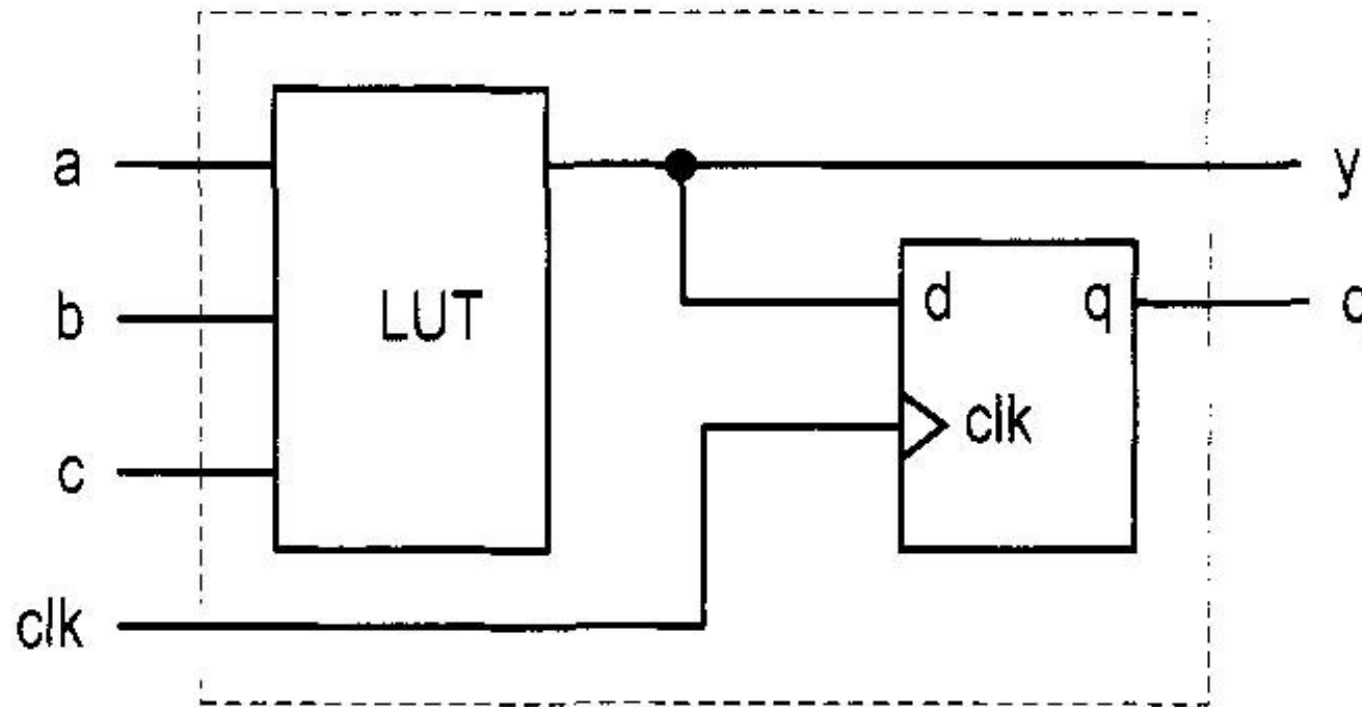


Conceptos básicos sobre FPGAs: (mediados de los años 80 siglo pasado, Xilinx)

- Una FPGA es un dispositivo lógico que contiene una matriz de *celdas lógicas* (*logic cells*) y conmutadores (*switches*) configurables.



- Cada celda lógica (*logic cell*) puede ser configurada para implementar una función lógica sencilla.

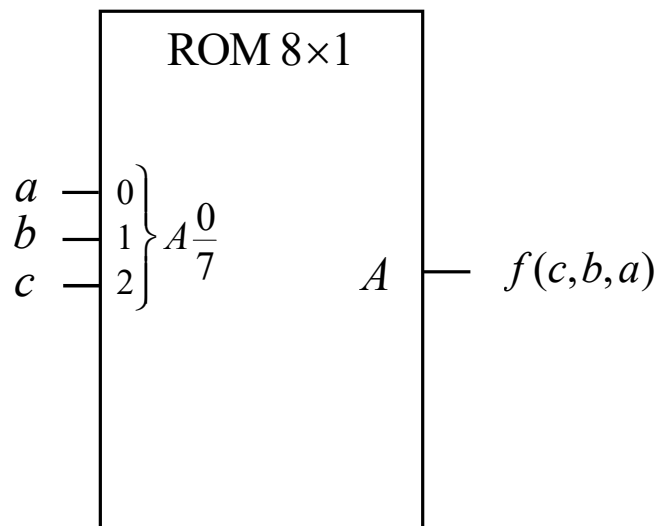


Logic cell

- Una función se puede implementar mediante una *memoria* (Xilinx) o mediante un *multiplexor* (Actel)... (ver Sistemas Digitales).

Ejemplo de implementación de un circuito combinacional con una LUT (*look-up table*)

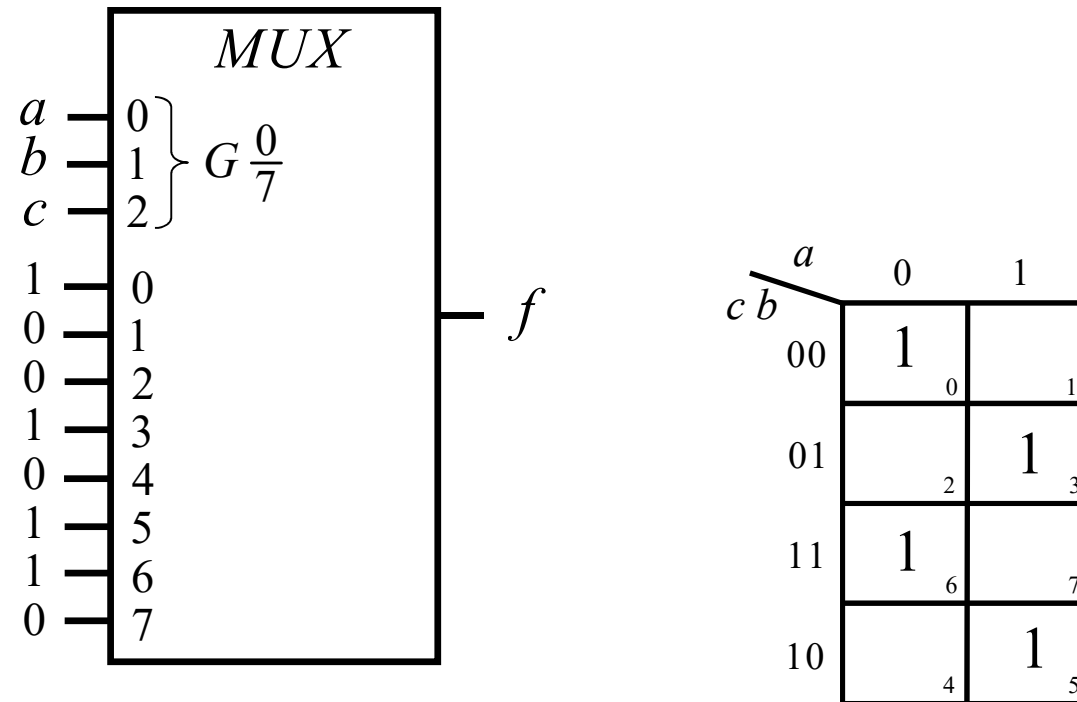
$$f(c,b,a) = \sum_3(0,4,5,6,7)$$



$c$	$b$	$a$	$f$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

## Ejemplo de implementación de una función lógica mediante un multiplexor

$$f(c,b,a) = \sum_3(0,3,5,6)$$



- Un circuito dado se puede implementar especificando las funciones a implementar por una serie de celdas lógicas (las necesarias) y configurando los switches para interconectar adecuadamente los circuitos implementados por las celdas lógicas.
- Los *flip-flops* existentes en las celdas lógicas permiten implementar sistemas secuenciales.

## Macro celdas

La mayoría de las FPGAs contienen macro celdas o macro bloques, cuya funcionalidad complementa a la funcionalidad de las celdas lógicas. Habitualmente una macro celda contiene:

- \_ bloque de memoria

- \_ multiplicador

- \_ circuitos que permiten ‘manejar’ la señal de reloj

- \_ circuitos de interface I/O

- \_ algunas FPGAs contienen uno o más procesadores

## Características específicas de la familia Spartan 3 de Xilinx

- Las celdas lógicas contienen una LUT de 4 entradas y 1 flip-flop D
- Las LUT se pueden configurar como memorias SRAM de 16 bits de capacidad o como registros de desplazamiento de 16 bits.
- Un grupo de 2 celdas forma 1 *slice*
- Un grupo de 4 slices forma 1 *bloque lógico configurable* (CLB)
- Contienen 4 tipos de macro bloques:
  - \_ Multiplicador combinacional: acepta 2 operandos de 18 bits como entradas y calcula su producto
  - \_ Bloque RAM: es una SRAM 18Kx1 que se puede organizar de varias formas
  - \_ *Digital clock manager* (DGM): permite reducir el clock skew y controlar la frecuencia y la fase de la señal de reloj
  - \_ IOB (input output block): controla el flujo de datos entre los pines de la FPGA y el circuito interno.

# Conceptos básicos de VHDL

Very High Speed Integrated Circuit (VHSIC) Hardware Description Language

Nota: los ejemplos indicados en estas notas han sido verificados con la versión 13.4 de ISE.



- VHDL es un lenguaje creado para describir el comportamiento de sistemas digitales.
- La primera versión estándar de vhdL se publicó en 1987 (vhdL-87). En 1993 se publicó una actualización denominada vhdL-93 (ha habido revisiones posteriores\*).
- VHDL fue el primer lenguaje de descripción de hardware establecido como estándar (IEEE 1076). Una norma adicional, la IEEE 1164, fue publicada con posterioridad.
- Los principales campos de aplicación del lenguaje vhdL son el diseño, la simulación (funcional y temporal) y la implementación de circuitos en CPLDs (*complex programmable logic devices*), FPGAs (*field programmable gate arrays*) y ASICs (*application specific integrated circuits*).
- No todo lo que se puede describir con vhdL es sintetizable (implementable)

- Las instrucciones en vhdl se denominan genéricamente como *código* para diferenciarlas de las instrucciones ejecutadas en computadoras, que se denominan genéricamente como *programa*.

- Los pasos a dar en la implementación de un circuito digital en un CPLD, en una FPGA o en un ASIC, utilizando vhdl, se resumen en lo siguiente:

1º paso: escribir en vhdl el código que describe el comportamiento del circuito

2º paso: compilar (sintetizar) el código escrito en vhdl para generar una netlist (lista de conexiones) a nivel de puerta lógica

3º paso: optimizar el circuito (a nivel de puerta) con el fin de reducir el tiempo de propagación de las señales a través del mismo (*speed*) y/o reducir el número de puertas (bloques lógicos configurables) que requiere su implementación (*area*).

4º paso: simular el funcionamiento del circuito.

5º paso: implementar el circuito diseñado en un CPLD, en una FPGA o bien generar la máscara para un ASIC.

## 1: Estructura de un código en vhd1

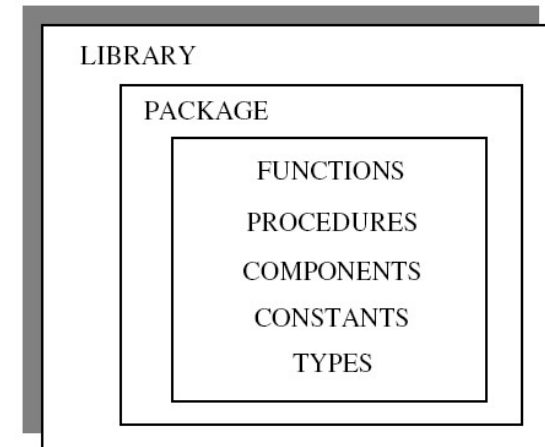
Las principales secciones (partes) que componen un código escrito en vhd1 son:

- Declaración de las bibliotecas (*libraries*) utilizadas: se indica el nombre de cada biblioteca (*library*), así como los *paquetes* utilizados de cada biblioteca.
- Entidad (*entity*): especifica los terminales de entrada y de salida del circuito
- Arquitectura (*architecture*): contiene el código que describe el comportamiento del circuito.

## Bibliotecas (Libraries)

- Una biblioteca no es más que un conjunto de trozos de código que se utilizan habitualmente al describir el comportamiento de circuitos. El situar dichos trozos de código dentro de una biblioteca permite que se puedan utilizar en la descripción de diferentes circuitos.

- En la parte derecha se indica la estructura típica de una biblioteca. El código se suele escribir en forma de *functions*, *procedures* o *components*, los cuales se sitúan dentro de *packages* y se compilan en la biblioteca de destino.



- Para declarar una biblioteca (hacerla visible o utilizable en un diseño) es necesario escribir 2 líneas de código:

\_ en la 1ª línea se indica el nombre de la biblioteca

\_ en la 2ª línea se indica el *package* utilizado de la biblioteca

- La sintaxis de la declaración de una biblioteca es la siguiente:

*library* nombre\_biblioteca; -- biblioteca utilizada      (-- indica comentario)

*use* nombre\_biblioteca.nombre\_paquete.package\_parts; -- package utilizado de la biblioteca

- En la mayoría de los diseños se suelen utilizar las librerías y los paquetes que se indican a continuación:

*library ieee;*

*use ieee.std\_logic\_1164.all;* -- paquete de la biblioteca *ieee*

*use ieee.numeric\_std.all;* -- paquete de la biblioteca *ieee*

*library std;*

*use std.standard.all;* -- paquete de la biblioteca *std*

*library work;*

*use work.all;* -- paquete de la biblioteca *work*

- Las bibliotecas (libraries) *std* y *work* son visibles por defecto (no es necesario declararlas). La biblioteca *ieee* sólo es necesario declararla cuando se emplean los packages *std\_logic* o *std\_ulogic*

El package *std\_logic\_1164* (biblioteca *ieee*, vhdl 2008) contiene, entre otras cosas, la definición de:

- \_ operadores lógicos: *not*, *and*, *or*, *nand*, *nor*, *xor* y *xnor* (y operadores aritméticos para datos de tipo *integer*)
- \_ operadores de comparación (=, /=, <, <=, >, >=)
- \_ algunos operadores de desplazamiento.
- \_ 9 niveles o valores lógicos
- \_ contiene todos los tipos de datos que se emplean en vhdl

El package *std.standard.all* (biblioteca *std*) contiene, entre otras cosas, lo siguiente:

- \_ varias definiciones de tipos de datos: *Bit*, *Integer*, *Boolean*, *Character*, etc.
- \_ operadores lógicos, aritméticos, de comparación, de desplazamiento, de concatenación

La biblioteca *work* es dónde guardamos nuestros diseños (archivos .vhd, los archivos creados por el compilador, por el simulador, etc.)

Nota: se pueden utilizar varios paquetes de una misma biblioteca.

El paquete *ieee.numeric\_std.all* tiene definidas, entre otras cosas, operaciones aritméticas con datos de tipo *signed* y *unsigned*.



## Entity (Entidad)

- En una entidad se declaran los terminales de entrada y de salida de un circuito así como los tipos de datos que pasan por dichos terminales.
- La sintaxis de la declaración de una entidad es:

```
entity nombre_entidad is  
    port (nombre terminal : tipo de terminal tipo de dato;  
          ...  
          nombre terminal : tipo de terminal tipo de dato);  
end nombre_entidad;
```

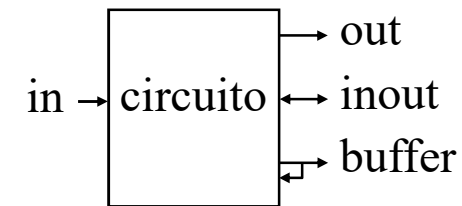
cumpléndose que:

## Nombres

Los nombres (identificadores) que se le pueden utilizar se caracterizan por lo siguiente:

- No tienen longitud máxima.
- Pueden contener caracteres del a 'A' a la 'Z', de la 'a' a la 'z', caracteres numéricos de '0' al '9' y el carácter subrayado '\_'.
- En los nombres no se diferencia entre mayúsculas y minúsculas (CONTADOR, contador y ConTadoR son el mismo nombre  $\equiv$  identificador)
- Un nombre debe comenzar con un carácter alfabético, no puede terminar con un subrayado, ni puede tener dos subrayados seguidos.
- La revisión del lenguaje Vhdl de 1993 (VHDL93) admite el uso de cualquier carácter en un nombre y diferencia mayúsculas de minúsculas si se encuentran entre dos caracteres '\'.
- No puede utilizarse una palabra reservada como nombre.

Los tipos de terminales son:



- *in*: indica un terminal de entrada. El circuito puede leer el valor presente en un terminal de tipo *in*, pero no puede escribir (poner) un valor en él.
- *out*: indica un terminal de salida. El sistema digital (circuito) puede escribir un valor en dicho terminal, pero no puede leer su valor para utilizarlo internamente.
- *buffer*: indica un terminal de salida que es realimentado internamente.
- *inout*: indica un terminal que puede actuar como entrada y como salida (es bidireccional). Se utiliza en salidas con tercer estado y cuando se utiliza el valor de una salida para calcular (internamente) el valor de otra salida (que puede ser la misma salida u otra distinta).

Los **tipos de datos** predefinidos más utilizados en vhdl sintetizable son: *std\_logic*, *std\_logic\_vector*, *signed*, *unsigned* e *integer*.

▪ ***std\_logic***:

\_ Se utiliza para definir datos de 1 bit, que pueden tomar 4 valores sintetizables:

‘1’ : 1 lógico

‘0’ : 0 lógico

‘\_’ : no importa

‘Z’ : tercer estado (alta impedancia  $\nabla$ ) -- la zeta debe estar en mayúsculas

Nota: Los valores U y W no se sintetizan, el valor L se sintetiza como un 0 lógico, el valor H se sintetiza como un 1 lógico y el valor X se sintetiza como un valor ‘\_’.

\_ El tipo *std\_logic* está definido en el paquete *ieee.std\_logic\_1164.all*;

\_ Los datos asociados a los terminales individuales (de entrada y de salida) declarados en una *entidad* se deben definir siempre de tipo *std\_logic* (es un estándar industrial).

▪ *std\_logic\_vector*:

\_ Se utiliza para definir el tipo de dato que se envía por un *bus*. Cada hilo del *bus* puede tomar 4 valores sintetizables distintos:

‘1’ : 1 lógico

‘0’ : 0 lógico

‘\_’ : no importa

‘Z’ : tercer estado (alta impedancia  $\nabla$ ) -- la zeta debe estar en mayúsculas

\_ El tipo *std\_logic\_vector* está definido en el paquete *ieee.std\_logic\_1164.all*;


\_ El tipo de dato asociado a un *bus* (de entrada o de salida) declarado en una *entidad* se debe definir siempre de tipo *std\_logic\_vector* (es un estándar industrial).

*Ejemplo:* signal *aux* : std\_logic\_vector (3 downto 0) := “<sup>3 2 1 0</sup>1100”;

-- aux(3) = 1 y aux(0) = 0

*Ejemplos* de asignación de un valor a los terminales de un *bus*:

```
signal aux : std_logic_vector (7 downto 0) := "01101100"; -- se define una señal de 8 bits
```



```
aux <= "11011001"; -- se le asigna un valor indicado en binario a todos los bits.
```

```
aux <= x"B5"; -- se le asigna un valor indicado en hexadecimal a todos los bits.
```

```
aux(5) <= '1'; -- se le asigna un valor binario a uno de los bits (el bit 5 del bus aux)
```

```
aux(7 downto 4) <= "1101"; -- se le asigna un valor a un grupo de bits del bus
```

```
aux <= (0 => '1', 1 => x or y, others => 'Z');
```

*Nota:*

\_ La asignación de un valor a un 1 bit se indica entre comillas simples.

\_ La asignación de un valor a un conjunto de bits (array, vector) se indica entre comillas dobles.

▪ *signed*:

\_ Se utiliza para representar cantidades **enteras** con signo, con las que se van a realizar operaciones *aritméticas* estándar.

\_ Está definido en el paquete *use ieee.numeric\_std.all*.

\_ Se define de la misma forma que un *std\_logic\_vector* (ver un ejemplo más abajo)

\_ El valor que representa un dato de tipo *signed* está codificado en el código *ca2* y, por lo tanto, el rango de valores que puede tomar un dato de tipo *signed* va desde  $-2^{n-1}$  a  $+2^{n-1} - 1$ , siendo  $n$  el número de bits que se defina para representar las cantidades.

*Ejemplo:* signal x, y, z : *signed* (3 downto 0); --  $n = 4$  bits (*rango*  $-8 \div +7$ )

$x \leq$  "1010"; -- x guarda el valor **-6**

$y \leq$  "0111"; -- y guarda el valor +7

$z \leq y + x$ ; -- suma aritmética (no se puede utilizar *or* en vez de +)

▪ *unsigned*:

\_ Se utiliza para representar cantidades **enteras** sin signo, con las que se van a realizar operaciones *aritméticas* estándar.

\_ Este tipo de dato está definido en el paquete *use ieee.numeric\_std.all*.

\_ Se define de la misma forma que un *std\_logic\_vector*. (ver ejemplo más abajo)


\_ Su valor está codificado en binario natural y, por lo tanto, el rango de valores que puede tomar va desde 0 hasta  $2^n - 1$ , siendo  $n$  el número de bits que se defina para representar las cantidades.

*Ejemplo:* signal  $x, y, z$  : *unsigned* (7 downto 0); --  $n = 8$  bits (rango  $0 \div 255$ )

$x \leq$  "00101101"; --  $x$  guarda el valor 45

$y \leq$  "00000011"; --  $y$  guarda el valor 3

$z \leq y * x$ ; -- producto aritmético ( $\neq$  producto lógico  $\equiv$  *and*) ( $z = 10000111$ )

 no se puede utilizar *and* en vez de  $*$  ya que el tipo de dato es aritmético, no lógico



- *integer*:

\_ Se utiliza para guardar valores enteros, con o sin signo, representables con hasta 32 dígitos binarios (−2,147,483,648 to +2,147,483,647). El rango de valores que puede tomar un dato *integer* se indica como: *range* <lower\_limit> to <upper\_limit>;

\_ Si no se especifica el rango de valores, el compilador asume que es el correspondiente a 32 bits

\_ Se suele utilizar como valor del *índice* en *bucles*, como valor de *constantes* o como un valor *genérico*.

*Ejemplo:* *signal a : integer range 0 to 255; -- no se puede poner de 255 to 0*

*a <= 134;*

*a <= B"00101111"; -- representación en binario de 47*

*a <= x"2F"; -- representación en hexadecimal de 47*

Definición de un tipo de dato *enumerated*: en la descripción de sistemas secuenciales se acostumbra a utilizar un tipo de dato denominado *enumerated*, el cual hay que definir en la parte declarativa de la arquitectura. Su sintaxis es la siguiente:

```
type nombre_tipo_dato is (lita de nombres, separados por comas, que se codifican en
                           en binario con valores consecutivos según el orden en el
                           que se han escrito);
```

*Ejemplo:*

```
type estado is (encendida, apagada, fundida); -- declaración de un tipo de dato
                                                -- enumerated denominado estado.
```

```
signal bombilla : estado; -- declaración de una señal denominada bombilla de tipo estado
```

```
-- Dado que para codificar 3 valores sólo se necesitan 2 bits, el estado:
```

```
-- encendida se codifica con el valor 00
```

```
-- apagada se codifica con el valor 01
```

```
-- fundida se codifica con el valor 10
```

*Ejemplo* de definición de un tipo de dato bidimensional para implementar una *memoria*:

-- declaración del tipo:

```
type memoria is array (1023 downto 0) of std_logic_vector (7 downto 0);
```

-- declaración de una señal denominada *mi\_memoria* de tipo *memoria*:

```
signal mi_memoria : memoria;
```

-- uso de la señal *mi\_memoria*:

```
mi_memoria (0) <= "11100111";
```

## Conversiones de tipos de datos

Para realizar operaciones *aritméticas estándar* (en las *arquitecturas*) se suelen utilizar tipos de datos *signed* y *unsigned*, mientras que los terminales de los puertos (en las *entidades*) se definen siempre de tipo *std\_logic* y/o *std\_logic\_vector*. Esto hace que sea necesario realizar conversiones entre estos tipos de datos.

Nota: existen bibliotecas/paquetes *no-estándar* que permiten realizar directamente operaciones aritméticas con datos de tipo *std\_logic* y *std\_logic\_vector*. Ahora bien, si se quiere crear código vhdl *portable*, no se deberían utilizar los siguientes paquetes:

```
use ieee.std_logic_signed.all;  
  
use ieee.std_logic_unsigned.all;  
  
use ieee.std_logic_arith.all;
```

Las conversiones que se suelen realizar habitualmente son entre:

\_ un tipo *std\_logic\_vector* y un tipo *unsigned*

\_ un tipo *std\_logic\_vector* y un tipo *signed*

\_ un tipo *std\_logic\_vector* y un tipo *integer*

\_ un tipo *signed* y un tipo *integer*

\_ un tipo *unsigned* y un tipo *integer*

**Nota:** las conversiones deben realizarse entre datos del mismo tamaño

Ejemplos de conversión entre tipos de datos (del mismo tamaño):

```
signal my_slv : std_logic_vector (7 downto 0); -- vector de 8 bits
```

```
signal my_uns : unsigned (7 downto 0); -- use ieee.numeric_std.all
```

```
signal my_sig : signed (7 downto 0); -- use ieee.numeric_std.all
```

```
signal my_int : integer range 0 to 255; -- es un array de 8 elementos
```

```
my_slv <= std_logic_vector (my_uns); -- conversión de unsigned a std_logic_vector
```

```
my_slv <= std_logic_vector (my_sig); -- conversión de signed a std_logic_vector
```

```
my_uns <= unsigned (my_slv); -- conversión de std_logic_vector a unsigned
```

```
my_sig <= signed (my_slv); -- conversión de std_logic_vector a signed
```

```
my_sig <= to_signed (my_int, my_sig'length); -- conversión de integer a signed
```

Nota: *my\_sig'length* proporciona el número de elementos (bits) del vector *my\_sig*

*my\_int* <= *to\_integer* (*my\_sig*); -- conversión de *signed* a *integer*

*my\_uns* <= *to\_unsigned* (*my\_int*, *my\_unsig*'length); -- conversión de *integer* a *unsigned*

*my\_int* <= *to\_integer* (*my\_unsig*); -- conversión de *unsigned* a *integer*

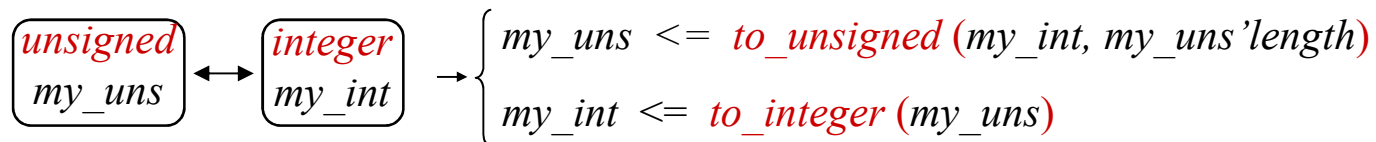
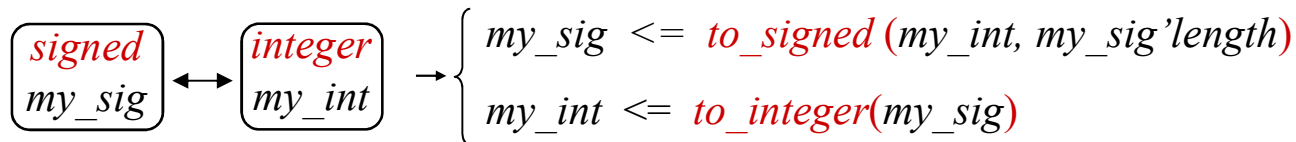
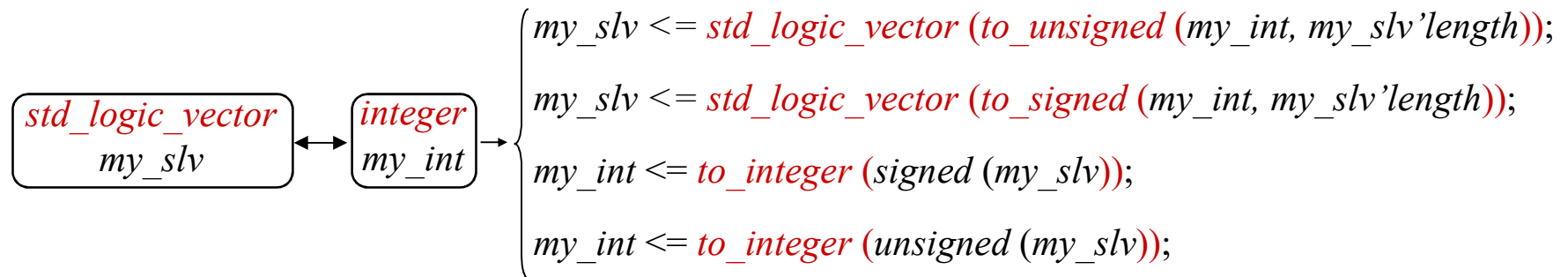
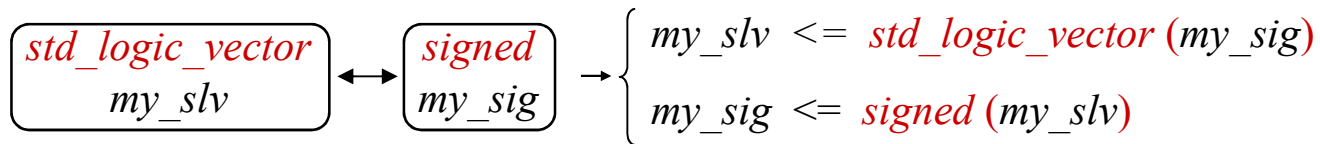
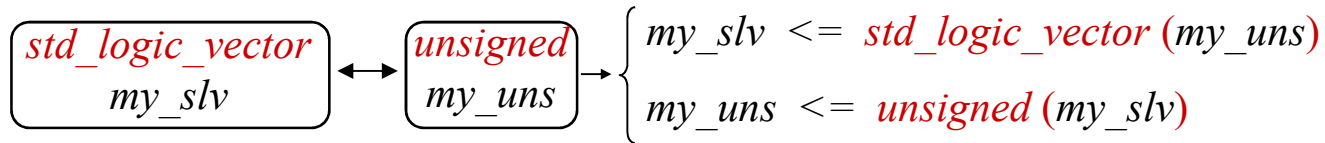
*my\_int* <= *to\_integer* (*to\_signed* (*my\_slv*)); -- para realizar una conversión de datos entre los tipos *integer* y *std\_logic\_vector* hay que realizar una conversión previa a *signed* o bien a *unsigned*. En este caso se realiza la conversión de *std\_logic\_vector* a *signed* y a continuación se realiza la conversión de *signed* a *integer* (La función *to\_integer*() está definida en el paquete *use ieee.numeric\_std.all*

*my\_int* <= *to\_integer* (*to\_unsigned* (*my\_slv*));-- conversión de *std\_logic\_vector* a *unsigned* y a continuación se realiza la conversión de *unsigned* a *integer*.

*my\_slv* <= *std\_logic\_vector* (*to\_unsigned* (*my\_int*, *my\_slv*'length));

*my\_slv* <= *std\_logic\_vector* (*to\_signed* (*my\_int*, *my\_slv*'length));

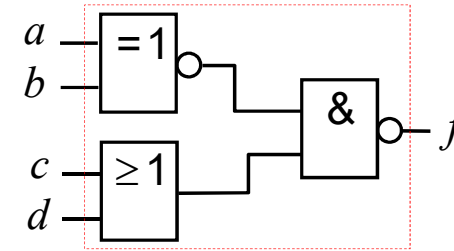
Resumen:





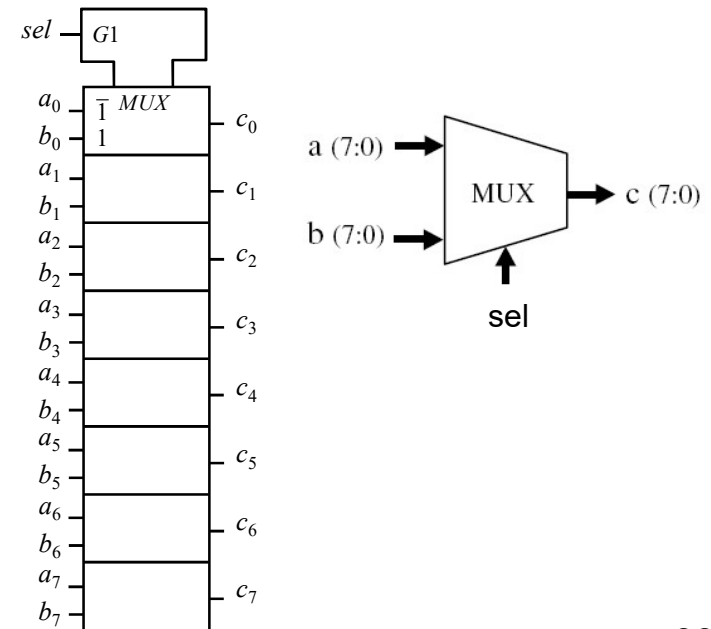
*Ejemplo de declaración de una entidad correspondiente al circuito de la derecha*

```
entity circuito_1 is
  port (d,c,b,a: in std_logic;
        f: out std_logic);
end circuito_1;
```



*Ejemplo de declaración de una entidad correspondiente a un óctuple multiplexor de 2 canales como el indicado en la parte derecha:*

```
entity ocho_mux is
  port(a,b: in std_logic_vector (7 downto 0);
        sel: in std_logic;
        c: out std_logic_vector (7 downto 0));
end ocho_mux;
```



## Architectures (arquitecturas):

- Una arquitectura define el comportamiento del circuito cuyos terminales se indican en la *entidad* a la que pertenece.

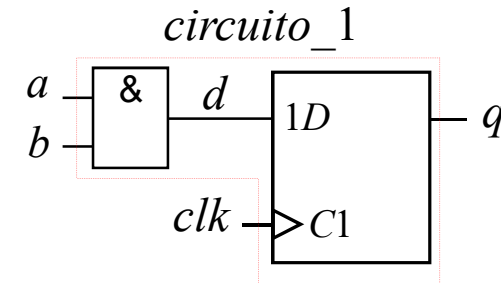
- La sintaxis de una arquitectura es la siguiente:

```
architecture nombre_arquitectura of nombre_entidad is  
[declarations] -- es opcional (parte declarativa)  
begin  
  (code) -- aquí se define el comportamiento del circuito.  
end nombre_arquitectura;
```

- En la parte de las declaraciones (opcional) se declaran (entre otras) señales y constantes.
- Los nombres de las arquitecturas cumplen las mismas reglas que los nombres de las entidades (incluido el mismo nombre de la entidad).
- Si se definen varias arquitecturas de una entidad, hay que tener en cuenta que sólo se puede sintetizar una arquitectura

*Ejemplo:* a continuación se describe el comportamiento del circuito indicado en la parte derecha (la solución propuesta no es única... nunca lo es)

```
library ieee;  
use ieee.std_logic_1164.all;  
  
entity circuito_1 is  
    port (a,b,clk: in std_logic;  
          q: out std_logic);  
end circuito_1;
```



```
architecture circuito_1 of circuito_1 is  
    signal d : std_logic; -- se define una señal local a la arquitectura  
begin  
    d <= a and b; -- se ejecuta concurrentemente (constantemente)  
    process (clk) -- su contenido se ejecuta cada vez que cambia el valor de la señal clk  
    begin  
        if (rising_edge(clk)) then -- si clk describe un flanco de subida, entonces  
            q <= d; -- esta instrucción se podría sustituir  
        end if; -- por q <= a and b; con lo que no  
    end process; -- sería necesario utilizar la señal d  
end circuito_1;
```

## Señales, variables, constantes y valores genéricos (*Signals, variables, constants and generics*)

- Para guardar datos cuyo valor puede cambiar durante el funcionamiento del circuito se pueden utilizar señales (*signals*) y variables (*variables*) ( $\equiv$  valores no estáticos).
- Para guardar datos cuyo valor no puede cambiar durante el funcionamiento del circuito se pueden utilizar *constantes* (*constants*) y valores *genéricos* (*generics*)  $\equiv$  valores estáticos.
- Las *señales* y las *constantes* se pueden utilizar tanto en código concurrente como en código secuencial.
- Las *señales* se pueden declarar en una *entity*, en una *architecture* y en un *package*.
- Las *variables* sólo se pueden declarar y utilizar en un código secuencial ( $\equiv$  sólo se pueden utilizar dentro de *processes*, *functions* y *procedures*) y son *locales* a dicho código.

## Señales (Signals)

- Las *señales* se utilizan para representar (guardar) el valor de las entradas y de las salidas de un circuito. También se pueden utilizar para guardar el valor de conexiones internas (entre bloques dentro de un circuito).
- Los terminales de un puerto (*port*) son *señales* por defecto y son *globales*.
- Las señales declaradas en la parte declarativa de una *arquitectura* (entre las palabras clave *architecture* y *begin*) también son visibles en toda la arquitectura.
- La sintaxis de la declaración de una señal es:

signal *nombre* : tipo [rango] [:= valor inicial]; -- lo indicado entre corchetes es opcional

*Ejemplos:*

signal *control* : std\_logic := '0';

signal *valor* : integer range 0 to 99; -- dato de 7 bits

signal *alfa* : std\_logic\_vector (7 downto 0) := "11010101"; -- dato de 8 bits

- El operador para asignar un valor a una *señal* es:  $\leq$

*Ejemplo:* `aux  $\leq$  '1';` -- se le asigna el valor '1' a la señal *aux*

- Cuando se utiliza una *señal* en código concurrente, la actualización de su valor se produce en el momento en el que se ejecuta la instrucción que le asigna un nuevo valor. Sin embargo, cuando se utiliza una *señal* en una sección de código que se ejecuta secuencialmente (por ejemplo, en un *process*), la actualización de su valor **no** se realiza hasta que finalice la ejecución de dicho código secuencial. En la práctica esto hace que sólo se permita modificar 1 vez el valor de una *señal* dentro de un código secuencial.

**Nota:** en el caso de que una *señal* deba cambiar de valor dos o más veces durante la ejecución de un código secuencial (por ejemplo, en un *process*) lo que debe hacerse es utilizar una *variable* en vez de una *señal*.

- Los valores iniciales de las *señales* (y de las variables) no son sintetizables. Sólo tienen utilidad en simulación.

- Atributos de las señales:

*clk'event* devuelve *true* cuando *clk* cambia de valor

*aux'stable* devuelve *true* si *aux* no ha cambiado de valor

*alfa'length* devuelve el número de bits del vector *alfa*

*Ejemplos:*

if (clk'event and clk = '1') then    -- si *clk* describe un flanco de subida entonces

if (clk'event and clk = '0') then    -- si *clk* describe un flanco de bajada entonces

if (not clk'stable and clk = '1') then    -- si *clk* describe un flanco de subida entonces

## Variables

- Una *variable* representa información local al código en el que se ha declarado
- Las *variables* sólo se pueden utilizar en código secuencial ( $\equiv$  sólo se pueden utilizar dentro de *processes*, *functions* y *procedures*).
- Las *variables* se declaran en la parte declarativa (antes de *begin*) de un *process*, de una *function* o de un *procedure*.
- La sintaxis de la declaración de una *variable* es la siguiente:

variable nombre : tipo [rango] [:= valor inicial]; -- lo que está entre corchetes es opcional

*Ejemplos:*

**variable** aux1 : std\_logic := '0'; -- se le asigna un 0 como valor inicial

**variable** aux2 : integer range 0 to 15;

**variable** aux3 : std\_logic\_vector (7 downto 0) := "11010101";

- El valor inicial de una variable no es sintetizable. Sólo tiene utilidad en simulación.



- El valor de una *variable* se actualiza tan pronto como se ‘ejecuta’ la instrucción en la que se le asigna un nuevo valor. De modo que su nuevo valor se puede utilizar en la siguiente línea de código (secuencial).

Recordatorio: el valor de una *señal* definida en un código secuencial no se actualiza en el momento en el que se ejecuta la instrucción que le asigna un nuevo valor. En la práctica, sólo se puede considerar que se ha actualizado su valor una vez que haya finalizado la ejecución del código secuencial.

- El operador para asignarle un valor a una variable es `:=`

*Ejemplo:* `beta := “00001111”;` -- a la variable beta se le asigna el valor  $15_{10}$

- El valor de una *variable* **no** se puede sacar directamente fuera de un código secuencial. En caso necesario, debe asignarse su valor a una *señal*. La asignación del valor de una *variable* a una *señal* debe realizarse justo antes de la instrucción *end process*; El operador para asignar el valor de una *variable* a una *señal* es `<=`

*Ejemplo:* `alfa <= beta;` -- se asigna el valor de la *variable* beta a la *señal* alfa

- El valor que tenga una *variable* al finalizar la ejecución del proceso en el que esté definida se conservará hasta la siguiente ejecución de dicho proceso.

## Constantes (constants)

- Se utilizan para guardar valores que no cambian durante el funcionamiento del circuito ( $\equiv$  ejecución del código). En la práctica se suelen utilizar para guardar valores de constantes, valores por defecto, etc.
- Una constante se puede declarar en un *paquete*, en una *entidad* o en una *arquitectura*. Si se declara en una *entidad* es global a todas las *arquitecturas* de dicha *entidad*. Si se declara en la parte declarativa de una *arquitectura*, sólo es conocida por el código de dicha *arquitectura*.
- La sintaxis de la declaración de una constante es la siguiente:

*constant* nombre : tipo := valor;

*Ejemplos:*

```
constant alfa : std_logic_vector := "10010110";  
constant beta : integer := 4;  
constant delta : integer range 0 to 15 := 2**beta-1;  
constant theta : memory := (('0','1','1','0'),  
                             ('1','0','1','0'),  
                             ('0','1','1','1'),  
                             ('1','0','0','1'));
```

## Valores genéricos (*generics*)


- Se utilizan para guardar valores que no cambian durante el funcionamiento del circuito ( $\equiv$  ejecución del código). Se diferencian de las *constantes* en que se puede modificar su valor externamente (*en la llamada a un componente o en la especificación de una configuración*). Su propósito es conferir más flexibilidad al código.

- Se pueden declarar en *entidades* y en *componentes*, antes de declarar el correspondiente puerto. El valor guardado por un *generic* declarado en una *entidad* puede ser leído en la propia *entidad* y en la *arquitectura* asociada a dicha *entidad*. En general, un valor genérico se puede tratar en una arquitectura como si fuese una constante.

Nota: la mayoría de los sintetizadores sólo soportan *generics* de tipo *integer*

- Su sintaxis es la siguiente:

*generic* (nombre : tipo de dato := valor por defecto);

Ejemplos: *generic* (*n* : integer := 8);  siempre que en la *entidad* o en la *arquitectura* aparezca *n* se sustituirá por el valor 8

*generic* (*delay* : time := 10 ns);

```
entity sumador_n_bits is
  generic(N : integer := 8);
  port(a, b : in std_logic_vector(N-1 downto 0);
        ci : in std_logic;
        s : out std_logic_vector(N-1 downto 0);
        co : out std_logic);
end sumador_n_bits;
```

ver ejemplo en página 117

## Instrucciones (sentencias)

En vhdl las instrucciones se ejecutan de dos formas: *concurrentemente* y *secuencialmente*. Hay instrucciones que sólo se pueden ejecutar concurrentemente y hay instrucciones que sólo se pueden ejecutar secuencialmente.

- Las instrucciones que se ejecutan *concurrentemente* se caracterizan porque todas se ejecutan al mismo tiempo (completamente en paralelo, a la vez), con independencia del orden en el que estén escritas en el código.
- Las instrucciones que se ejecutan *secuencialmente* se caracterizan porque se ejecutan una después de otra, en el orden en el que están escritas en el código. Estas instrucciones se comportan como las instrucciones de un lenguaje de programación.

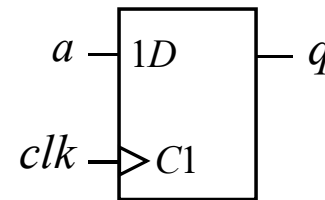
**Importante:** *código secuencial*  $\neq$  *sistema (lógica) secuencial*

Nota: el código concurrente también se denomina *dataflow code*

## Código concurrente

Los componentes que forman un circuito funcionan constantemente. Esto hace que, en general, las instrucciones que establecen su comportamiento deban ejecutarse constantemente, al mismo tiempo (concurrentemente).

Por otra parte, si las instrucciones que describen el comportamiento de un circuito se ejecutan siempre simultáneamente se puede obtener un comportamiento incorrecto. Ya que, por ejemplo, si el comportamiento del circuito indicado en la parte derecha se describe mediante la instrucción:

$$q \leq a;$$


se obtiene un comportamiento incorrecto debido a que la salida  $q$  sólo debe actualizar su valor con el valor que tenga la entrada  $a$  cuando la señal  $clk$  describa un flanco de subida. Este problema se resuelve en vhdl utilizando código que se ejecuta secuencialmente  $\equiv$  código secuencial.



En general, utilizando código concurrente sólo se pueden describir circuitos combinacionales. Para describir circuitos secuenciales debe utilizarse código secuencial (Nota: utilizando código secuencial se pueden describir tanto circuitos secuenciales como circuitos combinacionales).

En vhdl, sólo el código perteneciente a *procesos*, *funciones* y *procedimientos* se ejecuta secuencialmente. El resto del código se ejecuta en paralelo (al mismo tiempo, concurrentemente).

Nota: aunque el contenido de un *process*, de una *function* y de un *procedure* se ejecuta secuencialmente, dichos bloques se ejecutan concurrentemente con el código escrito fuera de ellos.

Para escribir *código concurrente* fuera de *procesos*, de *funciones* y de *procedimientos* se puede utilizar:

*a) Operadores:*

- de *asignación*
- *lógicos*
- *aritméticos*
- de *comparación*
- de *desplazamiento*
- de *concatenación*

*b) When ... else*

*c) With ... select ... when*

*a)* Instrucciones que utilizan *operadores*

En vhdl hay 6 tipos de operadores predefinidos y que se describen a continuación:

Nota: aunque se puede describir cualquier circuito combinacional utilizando únicamente operadores, los circuitos ‘complicados’ se suelen describir más fácilmente utilizando código secuencial.

▪ **Operadores de asignación:** se utilizan para asignar valores a señales (*signals*), a variables (*variables*) y a constantes (*constants*)

**<=** se utiliza para asignar un valor a una señal.


**:=** se utiliza para asignar un valor a una *variable*, a una *constant* o a un *generic*, o para asignar un *valor inicial* (en este caso también a una *signal*)

**=>** se utiliza para asignar valores a elementos individuales de un vector o con *others*

*Ejemplos:*

signal *x* : std\_logic := '0';  se le asigna el valor inicial 0 a la señal *x*

variable *z* : std\_logic\_vector (3 downto 0) := "1101";

 valor inicial de la variable *z*

*x* <= '1'; -- se asigna el valor 1 a la señal *x*

*z* := "0101" -- se asigna el valor 0101 a la variable *z*

*alfa* <= *beta*; -- se asigna el valor de la *variable* *beta* a la *señal* *alfa*

- Operadores lógicos:

- \_ Se utilizan para realizar operaciones lógicas.

- \_ Los operadores lógicos son: *not*, *and*, *or*, *nand*, *nor*, *xor* y *xnor*. (Nota: el operador *not* tiene prioridad sobre los demás operadores lógicos)

- \_ Los datos deben ser de tipo `std_logic` o `std_logic_vector`.

*Ejemplo:*

`x <= not a and b;` -- primero se calcula el negado de *a* y después se realiza la operación *and*

- Operadores aritméticos:

- \_ Se utilizan para realizar operaciones aritméticas.

- \_ Los datos deben ser de tipo *integer*, *signed*, *unsigned*. \*

- \_ Los operadores aritméticos son:

- + suma

- − resta (si actúa sobre 1 operando indica signo negativo, si relaciona a 2 operandos indica resta)

- \* multiplicación

- / división (sólo se permiten las divisiones por potencias enteras de 2)

- \*\* potenciación [*ejemplo*: port (a : in integer range 0 to 2\*\*8;...);] ( $2^{**8} \equiv 2^8 = 256$ )

- mod

- rem

- abs valor absoluto

Notas relativas al uso de los operadores aritméticos:

- No utilices los paquetes:

*ieee.std\_logic.arith.all* (no estándar - Synposys)

*ieee.std\_logic.unsigned.all* (no estándar - Synposys)

*ieee.std\_logic.signed.all* (no estándar - Synposys)

- Siempre que haya que realizar operaciones aritméticas el paquete (estándar) que hay que utilizar es el *ieee.numeric\_std.all*

- El paquete *ieee.numeric\_std.all* no tiene definidas operaciones matemáticas para los tipos *std\_logic* y *std\_logic\_vector*, pero si las tiene definidas para los tipos *signed*, *unsigned* e *integer*. Por lo que las señales y las variables deben definirse de tipo *signed*, *unsigned* o *integer* y una vez obtenidos los resultados realizar su conversión al tipo *std\_logic\_vector*.

- El paquete *ieee.std\_logic\_1164.all* permite realizar operaciones aritméticas con datos de tipo *integer*, pero no con datos de tipo *signed* o *unsigned*.

- Operadores de comparación:

- \_ Se utilizan para comparar cantidades.

- \_ Los datos deben ser de tipo *integer, signed, unsigned*.

- \_ Los operadores son:

- = igual

- /= distinto

- < menor

- > mayor

- <= menor o igual

- >= mayor o igual



- Operadores de desplazamiento:

\_ Se utilizan para desplazar dígitos binarios (no los rota).

\_ Se pueden desplazar datos de tipo *signed* y *unsigned* con las siguientes funciones:

*shift\_right* (*vector*, *n*) -- el valor de *vector* se desplaza *n* posiciones hacia la derecha

*shift\_left* (*vector*, *n*) -- el valor de *vector* se desplaza *n* posiciones hacia la izquierda

*Ejemplo:*

signal *a* : unsigned (3 downto 0); -- *a* es un vector de 4 bits

signal *b* : unsigned (3 downto 0);

*a* <= "1011";

*b* <= shift\_left(*a*,2); -- *b* = "1100" (se ha desplazado *a* 2 posiciones hacia la izquierda)

*b* <= shift\_right(*a*,1); -- *b* = "0101" (se ha desplazado *a* 1 posición hacia la derecha)

- **Operador de concatenación:** & (se utiliza con datos de tipo *signed*, *unsigned*, *std\_logic* y *std\_logic\_vector*)

*Ejemplo:*

```
signal a, b : std_logic_vector (3 downto 0);
```

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

```
a <= "1111";
```

```
b <= "0000";
```

```
c <= a & b;    -- c = "11110000"
```

*Ejemplo* 0 de código *concurrente* utilizando *operadores lógicos*: descripción de un multiplexor de 4 canales (I)

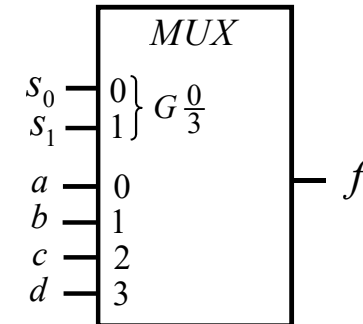
```
library ieee;
use ieee.std_logic_1164.all;

entity mux_4 is
    port(s0,s1,a,b,c,d : in std_logic;
          f : out std_logic);
end mux_4
```

```
architecture mux_4_function of mux_4 is
begin
```

```
    f <= (a and not s1 and not s0) or -- la operación not es prioritaria
          (b and not s1 and s0) or
          (c and s1 and not s0) or
          (d and s1 and s0);
```

```
end mux_4_function;
```



$$f(d,c,b,a,s_1,s_0) = a\bar{s}_1\bar{s}_0 + b\bar{s}_1s_0 + cs_1\bar{s}_0 + ds_1s_0$$

*b) When ... else:* esta estructura permite la asignación condicional de valores a señales. Se utiliza para describir condiciones en las que intervienen varias señales. Su sintaxis es la siguiente:

```
señal <= valor_1 when condición_1 else  
           valor_2 when condición_2 else  
           ...  
           valor_n;
```



a *señal* se le asigna el valor *valor\_1* cuando se cumpla la condición *condición\_1*

Nota: se empieza evaluando la condición *condición\_1*. Si no se cumple, se evalúa la condición *condición\_2* y así sucesivamente.

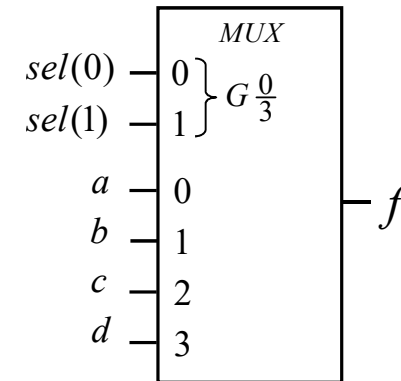
Nota: la estructura *when...else* es parecida a la instrucción secuencial *if ... then ... else*

*Ejemplo 1* de código concurrente utilizando la estructura *when ... else*: descripción de un multiplexor de 4 canales (II)

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

entity mux_4 is
    port(sel : in std_logic_vector (1 downto 0);
          a,b,c,d : in std_logic;
          f : out std_logic);
end mux_4;

architecture mux_4 of mux_4 is
begin
    f <= a when sel = "00" else
           b when sel = "01" else
           c when sel = "10" else
           d;
end mux_4;
```



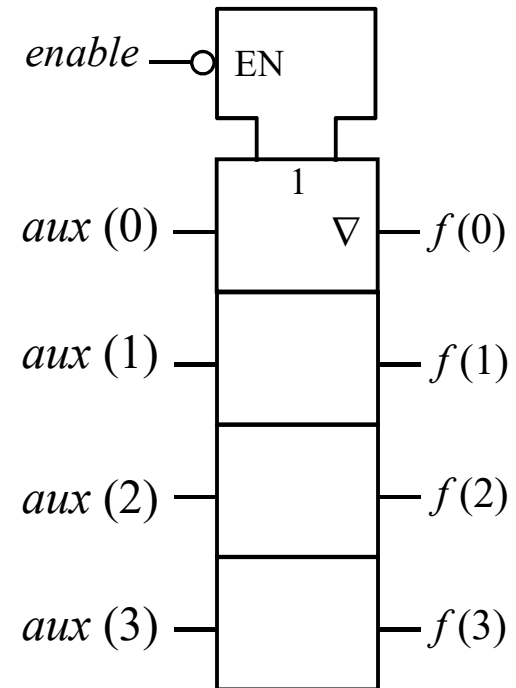
<i>sel(1)</i>	<i>sel(0)</i>	<i>f</i>
0	0	<i>a</i>
0	1	<i>b</i>
1	0	<i>c</i>
1	1	<i>d</i>

*Ejemplo 3* de código concurrente utilizando la instrucción *when ... else*: descripción de un cuádruple buffer

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

entity buffer_4 is
    port(enable : in std_logic;
          aux : in std_logic_vector (3 downto 0);
          f : out std_logic_vector (3 downto 0));
end buffer_4;

architecture buffer_4 of buffer_4 is
begin
    f <= aux when enable = '0' else
        "ZZZZ"; -- hay que poner las zetas en mayúsculas (tercer estado)
end buffer_4;
```

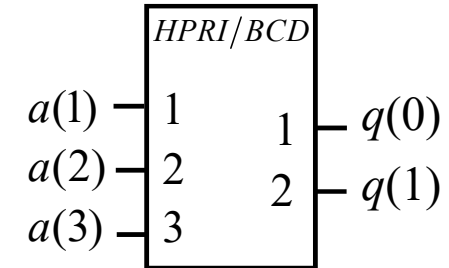


*Ejemplo 5* de código concurrente utilizando la instrucción *when ... else*: descripción de un codificador de 4 a 2 de alta prioridad

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

entity encoder_4_2 is
    port(a : in std_logic_vector (3 downto 1);
          q : out std_logic_vector (1 downto 0));
end encoder_4_2;

architecture encoder_4_2 of encoder_4_2 is
begin
    q <= "11" when a (3) = '1' else
        "10" when a (2) = '1' else
        "01" when a (1) = '1' else
        "00";
end encoder_4_2;
```



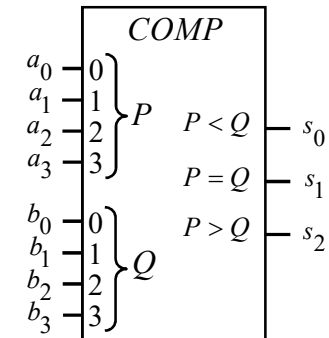
a(3)	a(2)	a(1)	q(1)	q(0)
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	1	0
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

*Ejemplo 6* de código concurrente utilizando la instrucción *when ... else*: descripción de un comparador de magnitud de 4 bits.

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

entity Comparador_4_bits is
    Port ( A, B : in  std_logic_vector (3 downto 0);
          S : out std_logic_vector (2 downto 0));
end Comparador_4_bits;

architecture Behavioral of Comparador_4_bits is
begin
    S <= "001" when A < B else //  $P < Q$ 
        "010" when A = B else //  $P = Q$ 
        "100"; //  $P > Q$ 
end Behavioral;
```





*Ejemplo 7* de código concurrente utilizando la instrucción *when ... else*: otra forma de describir de un *comparador de magnitud* de 4 bits.

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

entity Comparador_4_bits is
  Port (A, B : in  std_logic_vector (3 downto 0);
        S : out std_logic_vector (2 downto 0));
end Comparador_4_bits;
```

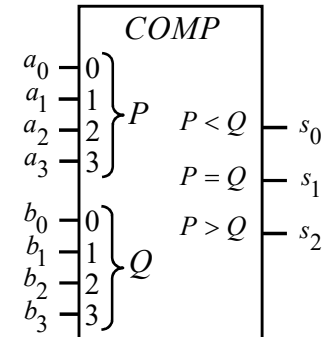
architecture Behavioral of Comparador\_4\_bits is  
begin

```
S(0) <= '1' when A < B else '0';
```

```
S(1) <= '1' when A = B else '0';
```


```
S(2) <= '1' when A > B else '0';
```

```
end Behavioral;
```



c) *With ... select ... when*: esta instrucción se utiliza para seleccionar el valor asignado a una señal dependiendo del valor que tiene otra señal. Su sintaxis tiene los siguientes formatos:

formato 1:


with *aux* select  a **control** se le asigna el valor **alfa** cuando **aux** toma el valor  $\lambda 1$ .

```
control <= alfa when  $\lambda 1$ , -- se usa una coma  
          beta  when  $\lambda 2$ ,  
          ...  
          omega when others;
```

**Nota:** hay que indicar un valor de **control** para todos los valores de **aux**. El último caso / valor se indica poniendo *when others*.

Nota: esta estructura es parecida a la instrucción secuencial *case*

formato 2:

with **aux** select  a **control** se le asigna el valor **alfa** cuando **aux** o bien toma el valor  **$\lambda 1$**  o bien toma el valor  **$\lambda 2$**  o bien toma el valor  **$\lambda 3$**

```
control <= alfa when  $\lambda 1$  |  $\lambda 2$  |  $\lambda 3$ ,  
      beta when  $\beta 1$  |  $\beta 2$  |  $\beta 3$  |  $\beta 4$ ,  
      ...  
      omega when others; -- ISE design suite no admite: omega when  $\lambda n$ 
```

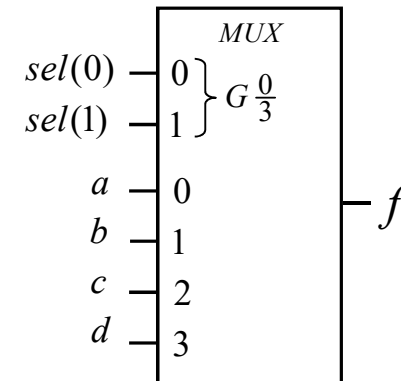
Nota: hay que indicar el valor a tomar por **control** para todos los valores de **aux**.

*Ejemplo* 1 de código concurrente utilizando la instrucción *with ... select ... when*:  
 descripción de un multiplexor de 4 canales (IV)

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

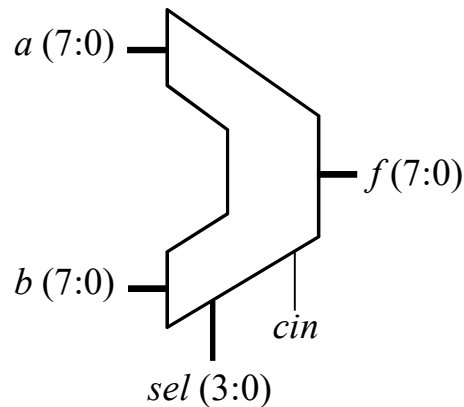
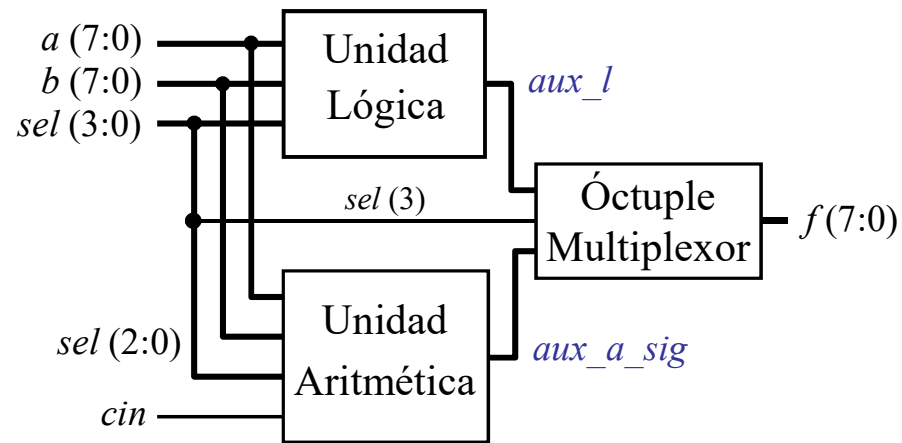
entity mux_4 is
    port(sel : in std_logic_vector (1 downto 0);
          a,b,c,d : in std_logic;
          f: out std_logic);
end mux_4;

architecture mux_4 of mux_4 is
begin
    with sel select
        f <= a when "00", -- se utiliza una coma
              b when "01",
              c when "10",
              d when others;
end mux_4;
```



<i>sel(1)</i>	<i>sel(0)</i>	<i>f</i>
0	0	<i>a</i>
0	1	<i>b</i>
1	0	<i>c</i>
1	1	<i>d</i>

Ejemplo 3 de *código concurrente* utilizando la instrucción *with ... select ... when*:  
 descripción de una ALU definida de la siguiente manera:



<i>sel</i>	Operaciones	
0000	$f \leq a$	lógicas $sel(3) = 0$
0001	$f \leq a + 1$	
0010	$f \leq a - 1$	
0011	$f \leq b$	
0100	$f \leq b + 1$	
0101	$f \leq b - 1$	
0110	$f \leq a + b$	
0111	$f \leq a + b + cin$	
1000	$f \leq not\ a$	aritméticas $sel(3) = 1$
1001	$f \leq not\ a$	
1010	$f \leq a\ and\ b$	
1011	$f \leq a\ or\ b$	
1100	$f \leq a\ nand\ b$	
1101	$f \leq a\ nor\ b$	
1110	$f \leq a\ xor\ b$	
1111	$f \leq a\ xnor\ b$	

*Nota:* los resultados de las operaciones aritméticas están codificados en el código *ca2*

```
library ieee;  
use ieee.std_logic_1164.all;  
use ieee.numeric_std.all; -- para poder utilizar datos de tipo signed (cca2)
```

```
entity alu is
```

```
    port (a, b : in std_logic_vector (7 downto 0);  
          sel : in std_logic_vector (3 downto 0);  
          ci : in std_logic;  
          f : out std_logic_vector (7 downto 0));
```

```
end alu;
```

```
architecture behavioral of alu is
```

```
    signal a_sig, b_sig, aux_a_sig : signed (7 downto 0); -- para las op. aritméticas  
    signal aux_1 : std_logic_vector (7 downto 0);  
    signal ci_in : integer range 0 to 1; -- en las op. aritméticas los datos deben ser de  
begin -- tipo integer, signed o unsigned.  
    a_sig <= signed (a); -- conversión de tipo std_logic_vector a signed (cca2)  
    b_sig <= signed (b);  
    ci_in <= 1 when ci = '1' else 0;
```

*with sel (2 downto 0) select -- cálculo salida unidad aritmética*

```

    aux_a_sig <= a_sig when "000",
    a_sig + 1 when "001",
    a_sig - 1 when "010",
    b_sig when "011",
    b_sig + 1 when "100",
    b_sig - 1 when "101",
    a_sig + b_sig when "110",
    a_sig + b_sig + ci_in when others;

```

*with sel (2 downto 0) select -- cálculo salida unidad lógica*

```

    aux_l <= not a when "000",
    not b when "001",
    a and b when "010",
    a or b when "011",
    a nand b when "100",
    a nor b when "101",
    a xor b when "110",
    a xnor b when others;

```

*with sel(3) select -- selección salida unidad aritmética o lógica*

```

    f <= std_logic_vector(aux_a_sig) when '0', -- conversión de tipo signed a std_logic_vector
    aux_l when others;

```

end behavioral;

<i>sel</i>	Operaciones
0000	$f \leq a$
0001	$f \leq a + 1$
0010	$f \leq a - 1$
0011	$f \leq b$
0100	$f \leq b + 1$
0101	$f \leq b - 1$
0110	$f \leq a + b$
0111	$f \leq a + b + cin$
1000	$f \leq \text{not } a$
1001	$f \leq \text{not } a$
1010	$f \leq a \text{ and } b$
1011	$f \leq a \text{ or } b$
1100	$f \leq a \text{ nand } b$
1101	$f \leq a \text{ nor } b$
1110	$f \leq a \text{ xor } b$
1111	$f \leq a \text{ xnor } b$

Código secuencial (*sequential code*  $\equiv$  *behavioral code*)

- En vhdl, las únicas instrucciones que se ejecutan secuencialmente (que no se ejecutan simultáneamente) corresponden al contenido de los *procesos*, de las *funciones* y de los *procedimientos*.
- Utilizando *código secuencial* se pueden describir tanto *circuitos secuenciales* como *circuitos combinacionales*.
- Las *variables* sólo se pueden utilizar en código secuencial. Es decir, sólo se pueden utilizar dentro de *processes*, *functions* y *procedures*. Esto hace que las *variables* no puedan ser globales y que, por lo tanto, no se pueda sacar directamente su valor fuera de una de estas estructuras.
- Las estructuras cuyo contenido se ejecuta secuencialmente (en el orden en el que se ha escrito) son: *If*, *Wait*, *Case* y *Loop*.



## *Processes* (Procesos)

- Un *proceso* es una estructura en vhdl que describe el comportamiento de un circuito, cuyo contenido se ejecuta *secuencialmente*.
- Un *proceso* puede estar formado por una lista de sensibilidad (*sensitivity list*) y por un conjunto de instrucciones entre las que figura al menos un *if*, un *case* o un *loop*. El contenido de un *process* se ejecuta cada vez que cambia de valor alguna de las señales indicadas en su *lista de sensibilidad*.
- Un *proceso* también puede estar formado por un conjunto de instrucciones entre las que figura al menos un *wait* (en este caso no hay lista de sensibilidad). El contenido del *proceso* se comienza a ejecutar inmediatamente y no se detiene hasta llegar a un *wait*. Hasta que se cumpla la condición relativa a dicho *wait* no se continúa con la ejecución de las instrucciones posteriores a dicho *wait*.
- El contenido de un *proceso* se ejecuta una vez durante la inicialización y posteriormente cada vez que se produce un cambio en el valor de alguna de las señales indicadas en su *lista de sensibilidad* o bien cuando se cumpla la condición relativa a un *wait*.

- La sintaxis de un *process* es la siguiente:

*process* (lista de sensibilidad)

(declaración de *variables*) (opcional) -- parte declarativa

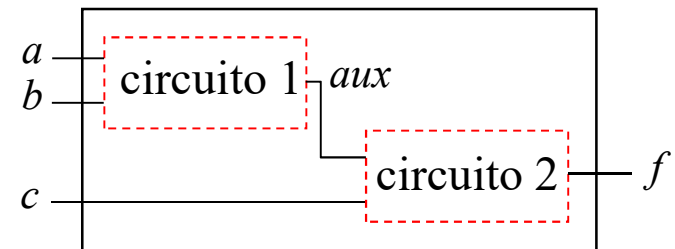
**begin**

(código que se ejecuta secuencialmente)

**end** *process*;

- Una arquitectura puede tener varios *procesos*, los cuales se ejecutan en paralelo (concurrentemente) con el resto del código que haya en la arquitectura.

Esto permite descomponer, por ejemplo, un circuito combinacional en varias partes más sencillas, de modo que cada parte se describe mediante un proceso distinto. Así, por ejemplo, en el circuito de la derecha se puede utilizar un proceso para definir el circuito 1 [process (*a*, *b*)] y otro proceso distinto para definir el circuito 2 [process (*c*, *aux*)].



- En el cuerpo de un *process* puede haber:

- a) *Variables y/o señales*

- b) *if ... then ... else*

- c) *wait*

- d) *case*

- e) *loop*

*a) Variables:*

\_ Sólo pueden ser declaradas y utilizadas en un código secuencial (por ejemplo, en un *proceso*). Se utilizan para guardar valores temporales.

\_ Su valor se actualiza instantáneamente. Es decir, el nuevo valor de una variable se puede utilizar en la siguiente línea de código a la línea en la que se modifica su valor.

Nota: cuando se modifica el valor de una *señal* en un *proceso* hay que tener presente que el valor de la *señal* **no** se actualiza instantáneamente ( $\equiv$  en el momento de ejecutarse la instrucción que modifica su valor). En general, la variable no suele tomar de forma efectiva el nuevo valor hasta que haya finalizado la ejecución del *proceso*. Esto hace que el nuevo valor de la *señal* sólo se pueda utilizar de forma segura una vez que haya finalizado la ejecución del *proceso*.

\_ Una *variable* es local al código secuencial en el que está definida. Lo que implica que su valor no se puede sacar directamente fuera del código secuencial en el que está definida. Para sacar su valor fuera del código secuencial debe asignarse su valor a una *signal*.

\_ Las *variables* deben ser declaradas antes de *begin*. Su valor inicial no es sintetizable (sólo tiene utilidad en simulación). Su sintaxis es la siguiente:

```
[variable nombre type [range] [:= initial value;]]
```

*Ejemplos:*

```
variable temp : integer range 0 to 10;
```

```
variable Q : std_logic_vector (7 downto 0); -- se declara un array-vector de 8 bits
```

```
variable flag : std_logic := '0'; -- se le ha asignado el valor inicial 0
```

\_ Para asignar un valor a una variable se utiliza el operador :=

*Ejemplo:*

```
count := count + 1;
```

b) *if ... then ... else:*

\_ Es una instrucción secuencial parecida a la instrucción concurrente *when ... else*.

\_ Sólo se puede utilizar dentro de un *proceso*, de una *función* o de un *procedimiento*

\_ Su sintaxis es la siguiente:

```
if(condición_1) then  
    instrucciones  
elsif(condición_2) then  
    instrucciones  
    .  
    .  
elsif(condición_n) then  
    instrucciones  
else  
    instrucciones  
end if;
```

Ejemplo de uso **incorrecto** de una señal en un proceso: durante la ejecución del proceso, en principio, se puede llegar a modificar 2 veces el valor de la señal *aux*, lo cual no es posible.

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

entity counter_10 is
    port(clk, reset : in std_logic;
          Q : out std_logic_vector (3 downto 0));
end counter_10;
```

```
architecture contador_10 of counter_10 is
```

```
    signal aux : unsigned (3 downto 0);
```

```
    begin
```

```
        process (clk, reset)
```

```
        begin
```

```
            if(reset = '1') then aux <= "0000";
```

```
            elsif (rising_edge(clk)) then aux <= aux + 1; -- 1ª modificación
```

```
                if (aux >= 10) then aux <= "0000"; -- evaluación y 2ª modificación: ERROR
```

```
                end if;
```

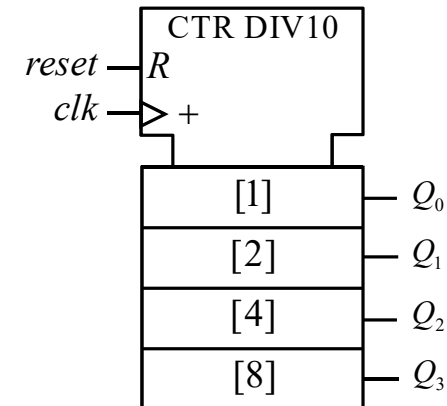
```
            end if;
```

```
            Q <= std_logic_vector (aux);
```

```
        end process;
```

```
    end contador_10;
```

Nota: hay que poner un valor inicial al contenido del contador para simular su funcionamiento



la señal *aux* no toma el nuevo valor establecido por esta instrucción hasta que finalice la ejecución del proceso.

el valor que tiene la señal *aux* al evaluarse esta condición no corresponde al valor que se le ha asignado en la instrucción anterior, sino que corresponde al valor que se le asignado en la ejecución anterior del proceso.

el valor de la señal *aux* se actualiza justo después de ejecutarse esta instrucción

*Ejemplo* de descripción correcta de un *contador* decimal que cuenta los flancos de subida de una señal *clk*.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity counter_10 is
    port(clk, reset : in std_logic;
        Q : out std_logic_vector (3 downto 0));
end counter_10;
```

```
architecture contador_10 of counter_10 is
begin
```

```
    process (clk, reset) -- process se ejecuta cada vez que cambia el valor de clk o de reset
    variable temp : unsigned (3 downto 0); -- para realizar operaciones aritméticas
```

```
begin
```

```
    if(reset = '1') then temp := "0000";
```

```
    elsif (rising_edge(clk)) then temp := temp + 1; -- suma aritmética
```

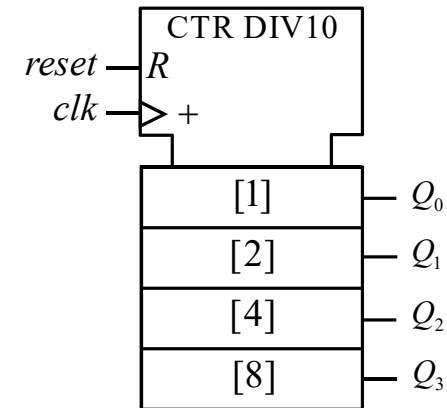
```
        if (temp >= 10) then temp := "0000";
```

```
        end if;
```

```
    end if;
```

```
    Q <= std_logic_vector (temp); -- para sacar el valor de temp fuera del process y
    end process;                                convertirlo a std_logic_vector
```

```
end contador_10;
```





*Nota: comparación signal versus variable*

	<i>Signal</i>	<i>Variable</i>
<i>Tipo de ejecución:</i>	<i>Concurrente y secuencial</i>	<i>Secuencial</i>
<i>Aplicación:</i>	<i>Representa el valor de un terminal</i>	<i>Representa un valor local dentro de un proceso, de una función o de un procedimiento.</i>
<i>Asignación de un valor:</i>	<i>&lt;=</i>	<i>:=</i>
<i>Asignación valor inicial</i>	<i>:=</i>	<i>:=</i>
<i>Visibilidad:</i>	<i>Si la señal está definida en una entidad es global, pero si está definida en un proceso, en una función o en un procedimiento es local a dicha estructura.</i>	<i>Sólo es visible dentro del proceso, función o procedimiento en el que se haya definido</i>
<i>Actualización:</i>	<i>En código concurrente la actualización es inmediata. En código secuencial la actualización de su valor no se produce hasta que finaliza la ejecución del código (secuencial).</i>	<i>Inmediata. Su nuevo valor se puede utilizar en la siguiente instrucción a la que modifica su valor.</i>
<i>Uso:</i>	<i>En paquetes, entidades y arquitecturas</i>	<i>En código secuencial: en procesos, en funciones y en procedimientos</i>

\_ En el momento en el que se inicia la ejecución de un *proceso*, tanto las *señales* como las *variables* utilizadas en el *proceso* tienen los mismos valores que tenían al finalizar la anterior ejecución de dicho *proceso*.

\_ El valor de una *variable* no se puede sacar directamente fuera del código secuencial en el que está definida. En caso necesario, debe asignarse su valor a una *señal*. El operador para asignar el valor de una *variable* a una *señal* es  $\leq$

*Ejemplo* 1 de descripción de un *codificador* de 4 a 2 de alta prioridad utilizando código *secuencial if ... then ... else*.

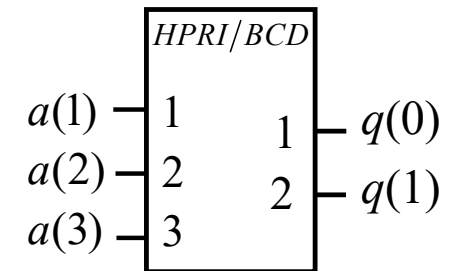
```

library ieee;
use ieee.std_logic_1164.all;

entity encoder_4_2 is
    port (a : in std_logic_vector (3 downto 1);
          q : out std_logic_vector (1 downto 0));
end encoder_4_2;

architecture encoder_4_2 of encoder_4_2 is
begin
    process (a)
    begin
        if (a(3) = '1') then q <= "11";
        elsif (a(3) = '0' and a(2) = '1') then q <= "10";
        elsif (a(3) = '0' and a(2) = '0' and a(1) = '1') then q <= "01";
        else q <= "00";
        end if;
    end process;
end encoder_4_2;

```



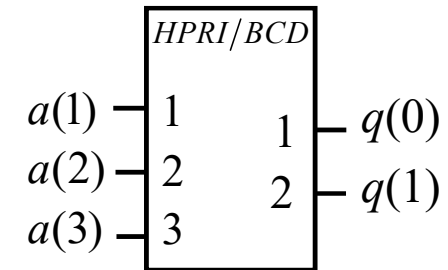
a(3)	a(2)	a(1)	q(1)	q(0)
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	1	0
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

*Ejemplo 1b* (el ejemplo anterior de forma más sencilla)

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

entity encoder_4_2 is
    port (a : in std_logic_vector (3 downto 1);
          q : out std_logic_vector (1 downto 0));
end encoder_4_2;

architecture encoder_4_2 of encoder_4_2 is
begin
    process (a)
    begin
        if (a(3) = '1') then q <= "11";
        elsif (a(2) = '1') then q <= "10";
        elsif (a(1) = '1') then q <= "01";
        else q <= "00";
        end if;
    end process;
end encoder_4_2;
```



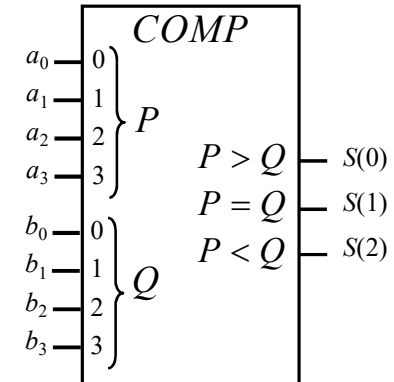
a(3)	a(2)	a(1)	q(1)	q(0)
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	1	0
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

*Ejemplo 2* de descripción de un *comparador de magnitud* de 4 bits, sin entradas de expansión, utilizando código *secuencial if ... then ... else*.

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

entity Comparador_4_bits is
    Port (A, B : in std_logic_vector (3 downto 0);
          S : out std_logic_vector (2 downto 0));
end Comparador_4_bits;
```

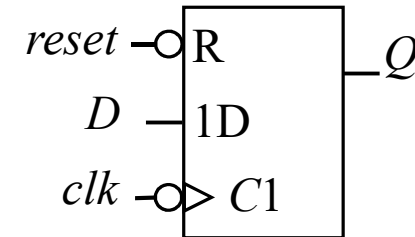
```
architecture Behavioral of Comparador_4_bits is
begin
    process(A, B)
    begin
        if(A > B) then S <= "001";
        elsif(A = B) then S <= "010";
        else S <= "100";
        end if;
    end process;
end Behavioral;
```



*Ejemplo 3* de código secuencial utilizando un *process*: descripción de un *flip-flop D* sincronizado con los flancos de bajada y reset asíncrono.

```
library ieee;
use ieee.std_logic_1164.all;
entity flip_flop_D is
    port (D, clk, reset : in std_logic;
          Q: out std_logic);
end flip_flop_D;

architecture D_bajada of flip_flop_D is
begin
    process (clk, reset) -- su contenido se ejecuta cada vez que cambia el valor de clk o de reset
    begin
        if(reset = '0') then -- si la señal de reset se pone a 0
            Q <= '0';
        elsif (falling_edge(clk)) then -- si clk describe un flanco de bajada
            Q <= D;
        end if;
    end process;
end D_bajada;
```



*Ejemplo 6* de código secuencial utilizando un *process*: descripción de un contador de 4 bits que cuenta los flancos de bajada de una señal *clk*.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all; -- para utilizar funciones aritméticas con signed y unsigned
```

```
entity counter_16 is
    port(clk, reset : in std_logic;
          Q : out std_logic_vector (3 downto 0));
end counter_10;
```

```
architecture contador_10 of counter_10 is
begin
```

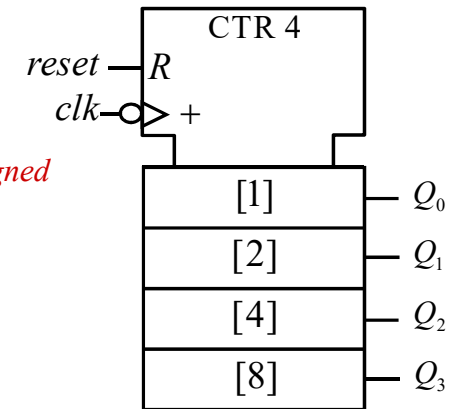
```
    process (clk, reset) -- este proceso se ejecuta cada vez que cambia el valor de clk o de reset
        variable temp : unsigned (3 downto 0); -- para realizar operaciones aritméticas
```

```
    begin
```

```
        if(reset = '1') then temp := "0000";
        elsif (falling_edge(clk)) then temp := temp + 1;
        end if; -- Q no puede tomar el valor "10000"
```

```
        Q <= std_logic_vector (temp); -- para sacar el valor fuera del process y convertir el tipo
    end process;
```

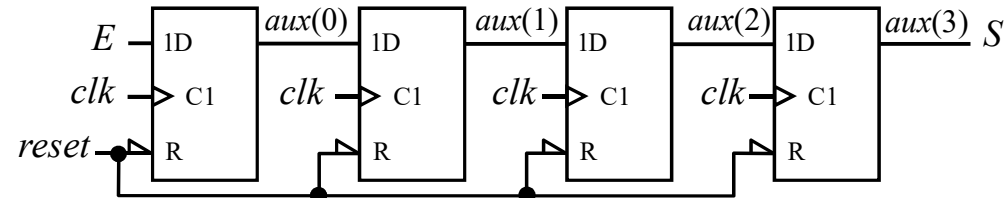
```
end contador_10;
```



*Ejemplo 7* de código secuencial utilizando un *process*: descripción de un registro de desplazamiento de entrada serie, salida serie, de 4 bits. (solución I)

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

entity Registro_serie_serie_4_bits is
    Port (E, clk, reset : in std_logic;
          S : out std_logic);
end Registro_serie_serie_4_bits;
```



```
architecture Behavioral of Registro_serie_serie_4_bits is
begin
```

```
    process (clk, reset)
```

```
        variable aux : std_logic_vector (3 downto 0) := "0000"; -- aux guarda el contenido del reg.
```

```
    begin
```

```
        if (reset = '0') then aux := (others => '0');
```

```
        elsif (rising_edge(clk)) then
```

```
            aux(3) := aux(2); -- las instrucciones de este if se ejecutan en el orden escrito
```

```
            aux(2) := aux(1);
```

```
            aux(1) := aux(0);
```

```
            aux(0) := E;
```

```
        end if;
```

```
        S <= aux(3); -- la variable aux es local al proceso y se actualiza instantáneamente
```

```
    end process;
```

```
end Behavioral;
```



*Ejemplo 8* de código secuencial utilizando un *process*: descripción de un registro de desplazamiento de entrada serie, salida serie, de 4 bits. (solución II: bucle for)

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

entity Registro_serie_serie_4_bits is
    Port (E, clk, reset : in std_logic;
          S : out std_logic);
end Registro_serie_serie_4_bits;
```

```
architecture Behavioral of Registro_serie_serie_4_bits is
begin
```

```
    process (clk, reset)
```

```
        variable aux : std_logic_vector (3 downto 0) := "0000"; -- aux guarda el contenido del reg.
```

```
    begin
```

```
        if (reset = '0') then aux := (others => '0');
```

```
        elsif (rising_edge(clk)) then
```

```
            for i in 0 to 2 loop
```

```
                aux(3 - i) := aux(2 - i);
```

```
            end loop;
```

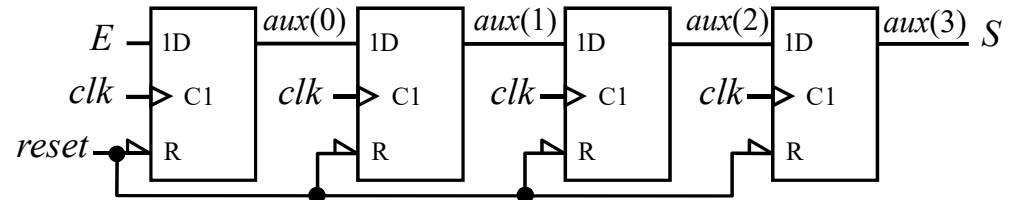
```
            aux(0) := E;
```

```
        end if;
```

```
        S <= aux(3); -- por el reset
```

```
    end process;
```

```
end Behavioral;
```

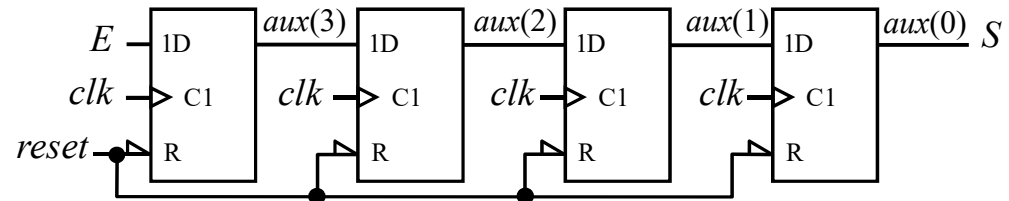


*i no se declara ni como señal ni como variable*

*Ejemplo 9* de código secuencial utilizando un *process*: otra descripción de un registro de desplazamiento de entrada serie, salida serie, de 4 bits. (solución III: **bucle for**)

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

entity Registro_serie_serie_4_bits is
    Port (E, clk, reset : in std_logic;
          S : out std_logic);
end Registro_serie_serie_4_bits;
```



```
architecture Behavioral of Registro_serie_serie_4_bits is
```

```
    signal aux : std_logic_vector (3 downto 0) := "0000"; -- aux guarda el contenido del registro
begin
```

```
    process (clk, reset)
```

```
    begin
```

```
        if (reset = '0') then aux <= (others => '0');
```

```
        elsif (rising_edge(clk)) then
```

```
            for i in 0 to 2 loop
```

```
                aux (i) <= aux (i + 1);
```

```
            end loop;
```

```
            aux(3) <= E;
```

```
        end if;
```

```
    end process;
```

```
    S <= aux(0); -- la señal aux no se actualiza hasta que ha finalizado la ejecución del process
```

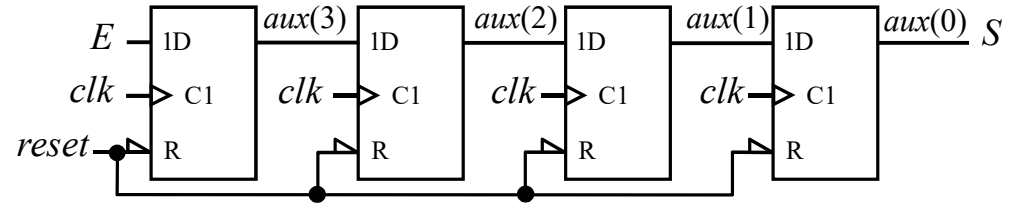
```
end Behavioral;
```

*Nota:* en el código anterior no se puede poner la instrucción  $S \leq aux(0)$  dentro del *proceso*. Ya que al ser *aux* una señal (*signal*), su nuevo valor no está disponible hasta que se hayan ejecutado todas las instrucciones del *proceso* (incluida la instrucción *end process*;). Si se pusiese dentro del *proceso*, el valor de la señal (global) *S* se actualizaría con el valor que adquirió *aux(0)* al ejecutarse la anterior iteración del *proceso*.

*Ejemplo* 10 de código secuencial utilizando un *process*: otra descripción de un registro de desplazamiento de entrada serie, salida serie, de 4 bits. (solución IV: uso de &)

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

entity Registro_serie_serie_4_bits is
    Port (E, clk, reset : in std_logic;
          S : out std_logic);
end Registro_serie_serie_4_bits;
```



```
architecture Behavioral of Registro_serie_serie_4_bits is
begin
```

```
    process (clk, reset)
```

```
        variable aux : std_logic_vector (3 downto 0) := "0000"; -- para la simulación
```

```
    begin
```

```
        if (reset = '0') then aux := "0000";
```

```
        elsif (rising_edge(clk)) then
```

```
            aux := E & aux (3 downto 1); -- concatenación
```

```
        end if;
```

```
        S <= aux(0); -- aux es una variable local al process, y se actualiza instantáneamente
```

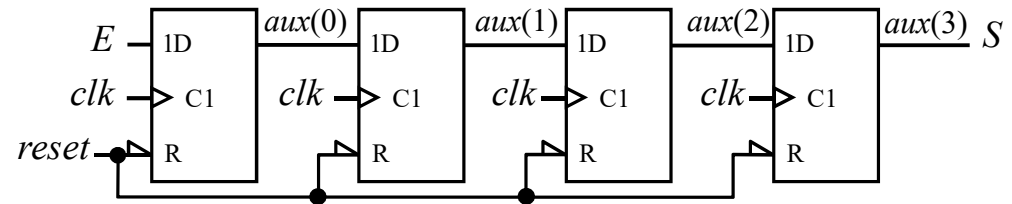
```
    end process;
```

```
end Behavioral;
```

*Ejemplo* 11 de código secuencial utilizando un *process*: otra descripción de un registro de desplazamiento de entrada serie, salida serie, de 4 bits. (solución V)

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

entity Registro_serie_serie_4_bits is
    Port (E, clk, reset : in std_logic;
          S : out std_logic);
end Registro_serie_serie_4_bits;
```



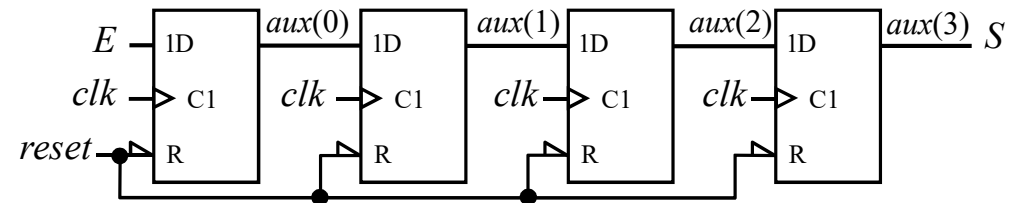
architecture Behavioral of Registro\_serie\_serie\_4\_bits is

```
    signal aux : std_logic_vector (3 downto 0) := "0000"; -- para simulación
begin
    process (clk, reset)
    begin
        if (reset = '0') then aux <= "0000";
        elsif (rising_edge(clk)) then
            aux (3 downto 1) <= aux (2 downto 0);
            aux (0) <= E;
        end if;
    end process;
    S <= aux(3); -- aux es una señal que no se actualiza hasta que se haya ejecutado el process
end Behavioral;
```

*Ejemplo* 12 de código secuencial utilizando un *process*: otra descripción de un registro de desplazamiento de entrada serie, salida serie, de 4 bits. (solución VI)

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

entity Registro_serie_serie_4_bits is
    Port (E, clk, reset : in std_logic;
          S : out std_logic);
end Registro_serie_serie_4_bits;
```



```
architecture Behavioral of Registro_serie_serie_4_bits is
```

```
begin
```

```
    process (clk, reset)
```

```
        variable aux : std_logic_vector (3 downto 0) := "0000"; -- guarda el contenido del reg.
```

```
        begin
```

```
            if (reset = '0') then aux := (others => '0');
```

```
            elsif (rising_edge(clk)) then -- las sig. inst. se ejecutan en el orden en el que están escritas
```

```
                aux (3) := aux (2);
```

```
                aux (2) := aux (1);
```

```
                aux (1) := aux(0);
```

```
                aux(0) := E;
```

```
            end if;
```

```
            S <= aux(3); -- aux es una variable local al process (se actualiza instantáneamente).
```

```
        end process;
```

```
end Behavioral;
```

*Ejemplo* 13 de código secuencial utilizando un *process*: descripción de un *barrel shifter* de 8 bits (realiza un desplazamiento circular hacia la derecha).

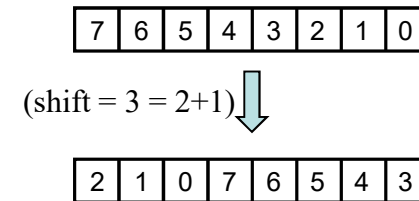
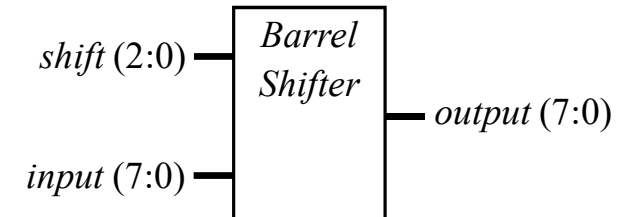
```
library ieee;
use ieee.std_logic_1164.all;

entity barrel_shifter_1 is
    port (input : in  std_logic_vector (7 downto 0);
          shift : in  std_logic_vector (2 downto 0);
          output : out std_logic_vector (7 downto 0));
end barrel_shifter_1;
```

```
architecture behavioral of barrel_shifter_1 is
begin
```

```
    process (input, shift) -- el código dentro del process se ejecuta secuencialmente (por orden)
        variable aux1, aux2 : std_logic_vector (7 downto 0);
    begin
        if (shift (0) = '1') then aux1 := input (0) & input (7 downto 1); -- se desplaza 1 posición
        else aux1 := input; -- con shift (1) se desplaza lo desplazado por shift (0)
        end if;

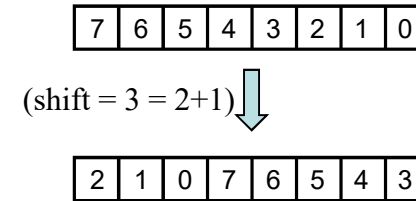
        if (shift (1) = '1') then aux2 := aux1 (1 downto 0) & aux1 (7 downto 2);
        else aux2 := aux1; -- hay que tener en cuenta lo desplazado por shift (0)
        end if;
```



```

    if (shift(2) = '1') then output <= aux2 (3 downto 0) & aux2 (7 downto 4);
    else output <= aux2;
    end if;
end process;
end behavioral;

```



Nota:

- \_ Si shift (0) = '1' se desplaza (rota) 1 posición hacia la derecha
- \_ Si shift (1) = '1' se desplaza (rota) 2 posiciones hacia la derecha lo desplazado por shift (0)
- \_ Si shift (2) = '1' se desplaza (rota) 4 posiciones hacia la derecha lo desplazado por shift (1)



*Ejemplo 14*: descripción de un *contador reversible* de 4 bits.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all; -- para poder usar el tipo unsigned
```

```
entity contador is
    port (reset, mode, enable, input : in std_logic;
          Q : out std_logic_vector (0 to 3));
end contador;
```

```
architecture contador of contador is
begin
```

```
    process (reset, input)
```

```
        variable aux : unsigned (0 to 3) := "0000"; -- para las operaciones aritméticas
```

```
    begin
```

```
        if (reset = '0') then aux := "0000";
```

```
        elsif (rising_edge(input) and enable = '1') then
```

```
            if (mode = '1') then aux := aux + 1; -- operación aritmética
```

```
            else aux := aux - 1; -- operación aritmética
```

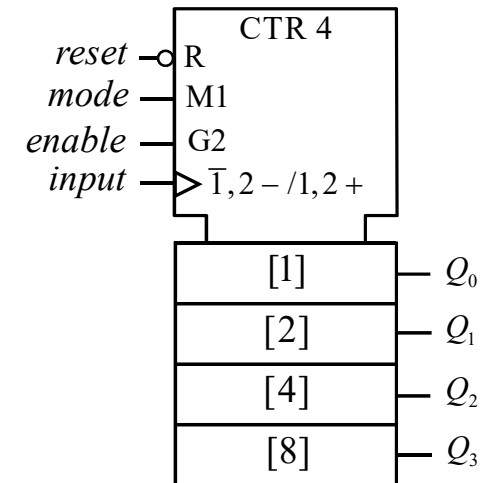
```
            end if;
```

```
        end if;
```

```
        Q <= std_logic_vector (aux);
```

```
    end process;
```

```
end contador;
```



c) *wait*:

En este caso, el *process* no tiene una *lista de sensibilidad*. La instrucción *wait* es secuencial y tiene varios formatos, cuyas sintaxis son:

formato 1: *wait until* condición\_de\_1\_señal;

\_ La instrucción *wait until* sólo acepta 1 señal, lo que la hace más adecuada para código síncrono que asíncrono.

\_ Debe ponerse como primera instrucción en el cuerpo de un *process* (después de *begin*). El contenido del *process* se ejecutará cada vez que se cumpla la condición.

formato 2: *wait on*  $signal_1, signal_2, signal_3, \dots, signal_n$ ;

\_ La instrucción *wait on* acepta varias señales (son como una lista de sensibilidad).

\_ El contenido del *process* se ejecuta cada vez que cambia el valor de alguna de las señales.

formato 3: *wait for* time;

\_ Esta operación sólo se utiliza para generar señales en *test benches*.

\_ *Ejemplos*:

*wait until* rising\_edge(*clk*); -- se espera hasta que *clk* describa un flanco de subida

*wait on* reset, *clk*; -- se espera hasta que cambia de valor de *reset* o *clk* (equivale a  
-- una lista de sensibilidad)

*wait for* 40ns; -- se espera a que transcurran 40 ns antes de continuar con la ejecución  
-- del código (para simulación).

*wait*; -- se utiliza en la finalización del *proceso* que describe cómo varían las señales  
-- de entrada de un circuito en un *test bench*.

```
wait on until rising_edge(ck);  
wait on ck until rising_edge(ck);-- equivale a la instrucción anterior
```

```
process  
  -- parte declarativa  
begin  
  wait on sensitivity list; -- la ejecución se detiene aquí a la espera de que se produzca  
  -- instrucciones          un cambio en el valor de  $a$  o de  $b$  (lista de sensibilidad)  
end process;
```

*Ejemplo:*

```
process  
begin  
  wait on a, b; -- la ejecución se detiene aquí a la espera de que se produzca  
  sum <= a + b;  un cambio en el valor de  $a$  o de  $b$   
end process;
```

-- Este código es equivalente\* al anterior

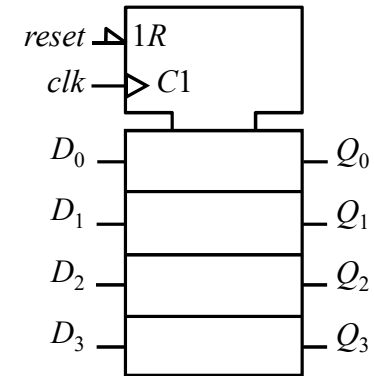
```
process  
begin  
  sum <= a + b;  
  wait on a, b;  
end process;
```

*Ejemplo* de código secuencial utilizando *wait until*: descripción de un registro paralelo de 4 bits con reset síncrono.

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

entity register_parallel_4 is
    port (clk, reset : in std_logic;
          D : in std_logic_vector (3 downto 0);
          Q : out std_logic_vector (3 downto 0);
    end register_parallel_4;

architecture register_parallel_4 of register_parallel_4 is
begin
    process
    begin
        wait until (rising_edge(clk));
        if (reset = '0') then Q <= "0000";
        else Q <= D;
        end if;
    end process;
end register_parallel_4;
```



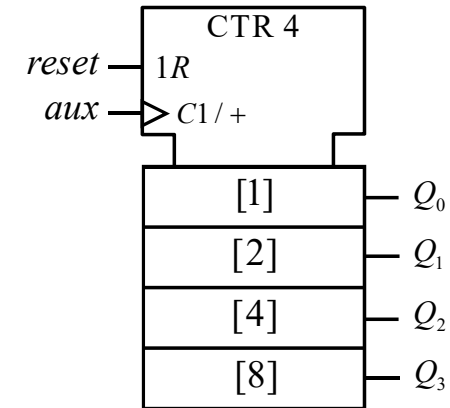
*Ejemplo* de código secuencial utilizando *wait until*: descripción de un contador de 4 bits que cuenta los flancos de subida de una señal *aux*, con reset síncrono.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all; -- para poder usar el tipo unsigned
```

```
entity counter_16 is
    port (aux, reset : in std_logic;
          Q : out std_logic_vector (3 downto 0));
end counter_16;
```

```
architecture contador_16 of counter_16 is
begin
```

```
    process -- no hay lista de sensibilidad porque se utiliza un wait
        variable temp : unsigned (3 downto 0); -- para realizar operaciones aritméticas
    begin
        wait until (rising_edge(aux));
        if(reset = '1') then temp := "0000";
        else temp := temp + 1; -- temp se actualiza instantáneamente
        end if;
        Q <= std_logic_vector(temp); -- para poder sacar el valor de temp fuera del process
    end process;
end contador_16;
```



*Ejemplo* de un *test bench* para el código de la diapositiva anterior.

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

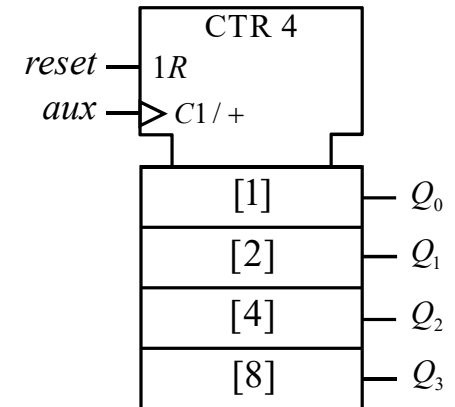
entity tb is
end tb;

architecture behavior of tb is
    component contador_4_bits_reset_sincrono -- component declaration for the UUT
    port(reset, aux : in std_logic;
         Q : out std_logic_vector(3 downto 0));
    end component;

    signal reset : std_logic := '1';
    signal aux : std_logic := '0';
    signal q : std_logic_vector(3 downto 0);

    constant aux_period : time := 20 ns;

begin
    uut: contador_4_bits_reset_sincrono -- instantiate the unit under test (uut)
        port map (reset => reset,
                  aux => aux,
                  Q => q); -- continúa en la siguiente diapositiva
```



```
aux_process : process -- este proceso genera la señal (periódica) aux
begin
    aux <= '0';
    wait for aux_period/2;
    aux <= '1';
    wait for aux_period/2;
end process;
```

```
stim_proc: process -- este proceso genera la señal de reset
begin
    wait for 20 ns;
    reset <= '0';
    wait for aux_period*18;
    reset <= '1';
    wait;
end process;
end;
```

**Nota:** si en vez de *aux* se hubiese puesto *clk* se hubiese podido aprovechar el código que genera automáticamente el entorno ISE para generar una señal de reloj (*clk*).

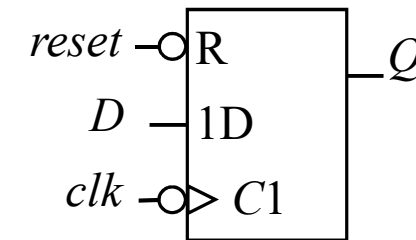


*Ejemplo* de código secuencial utilizando *wait on*: descripción de un *flip-flop D* sincronizado con los flancos de bajada y reset asíncrono.

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

entity flip_flop_D is
    port(reset, D, clk : in std_logic;
          Q: out std_logic);
end flip_flop_D;

architecture D_bajada of flip_flop_D is
begin
    process -- no hay lista de sensibilidad, porque se utiliza un wait
    begin
        wait on reset, clk; -- espera hasta que cambie el valor de reset o de clk
        if (reset = '0') then Q <= '0';
        elsif (falling_edge(clk) then Q <= D;
        end if;
    end process;
end D_bajada;
```



d) *case*:

Es una instrucción secuencial. Su sintaxis es la siguiente:

*case señal is*

*when condición1 | condición2 | ... =>*

*instrucciones secuenciales*

*when condición1 | condición2 | ... =>*

*instrucciones secuenciales*

*...*

*when others => null; -- no se realiza ninguna acción*

*end case;*

*Nota: cuando **señal** cumple la condición **condiciónx** se ejecutan las instrucciones correspondientes*

Notas:

\_ Se deben considerar todos los posibles valores de ‘*señal*’ de forma excluyente.

Para ello resulta útil la palabra clave *others* (ver sintaxis).

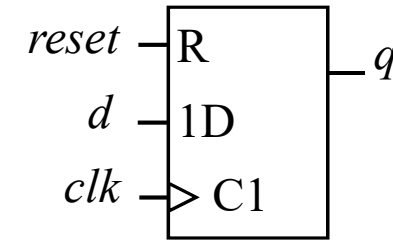
\_ Si en algún caso no se quiere realizar ninguna acción se puede utilizar *null*

*Ejemplo 1* de código secuencial utilizando *case*: descripción de un *flip-flop D* sincronizado con los flancos de subida, con reset asíncrono.

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

entity flip_flop_D is
    port(reset, d, clk : in std_logic;
          q: out std_logic);
end flip_flop_D;

architecture D_subida of flip_flop_D is
begin
    process (reset, clk)
    begin
        case reset is
            when '1' => q <= '0';
            when '0' =>
                if (rising_edge(clk)) then q <= d;
                end if;
            when others => null; -- sin esto no funciona !!!
        end case;
    end process;
end D_subida;
```



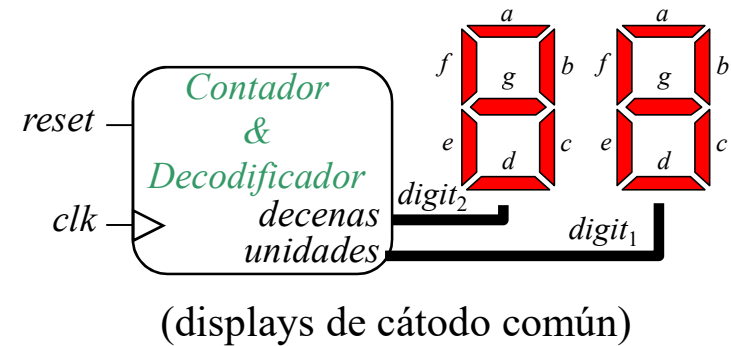
*Ejemplo 2* de código secuencial utilizando *case*: descripción de un circuito que contiene un contador de módulo 60 y dos decodificadores de BCD a 7 segmentos.

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

entity circuito_1 is
port (reset, clk : in std_logic;
      digit1, digit2 : out std_logic_vector (6 downto 0));
end circuito_1;
```

```
architecture circuito_1 of circuito_1 is
begin
```

```
    process (reset, clk)
        variable temp1 : integer range 0 to 10;
        variable temp2 : integer range 0 to 6;
    begin
        if (reset = '1') then
            temp1 := 0;
            temp2 := 0;
        elsif (rising_edge(clk)) then
            temp1 := temp1 + 1;
            if (temp1 = 10) then
                temp1 := 0;
                temp2 := temp2 + 1; -- continua en la página siguiente
```



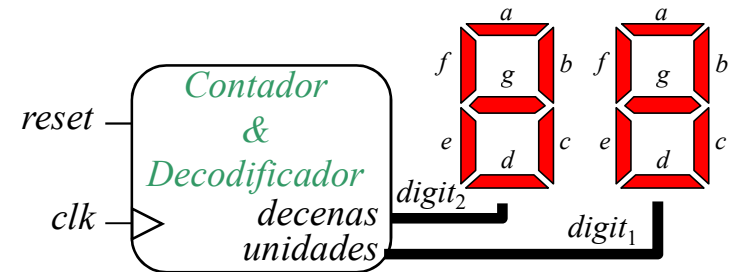
Nota: el paquete *ieee.std\_logic\_1164.all* permite realizar operaciones aritméticas con datos de tipo *integer*, pero **no** con datos de tipo *signed* o *unsigned*.

Nota: en este ejemplo se podrían haber utilizado variables de tipo *unsigned* en vez de tipo *integer*. En tal caso también habría que utilizar el paquete *ieee.numeric\_std.all*

```

    if (temp2 = 6) then
        temp2 := 0;
    end if;
end if;
end if;

```



(displays de cátodo común)

```

case temp1 is
    when 0 => digit1 <= "1111110"; -- a b c d e f g = 7E
    when 1 => digit1 <= "0110000"; -- a b c d e f g = 30
    when 2 => digit1 <= "1101101"; -- a b c d e f g = 6D
    when 3 => digit1 <= "1111001"; -- a b c d e f g = 79
    when 4 => digit1 <= "0110011"; -- a b c d e f g = 33
    when 5 => digit1 <= "1011011"; -- a b c d e f g = 5B
    when 6 => digit1 <= "1011111"; -- a b c d e f g = 5F
    when 7 => digit1 <= "1110000"; -- a b c d e f g = 70
    when 8 => digit1 <= "1111111"; -- a b c d e f g = 7F
    when 9 => digit1 <= "1111011"; -- a b c d e f g = 7B
    when others => null;
end case;

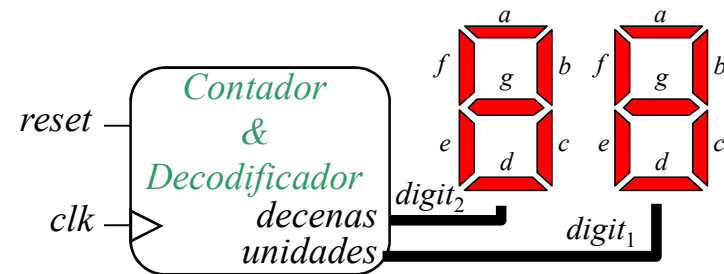
```

-- continua en la siguiente página

```

case temp2 is
  when 0 => digit2 <= "1111110"; -- a b c d e f g = 7E
  when 1 => digit2 <= "0110000"; -- a b c d e f g = 30
  when 2 => digit2 <= "1101101"; -- a b c d e f g = 6D
  when 3 => digit2 <= "1111001"; -- a b c d e f g = 79
  when 4 => digit2 <= "0110011"; -- a b c d e f g = 33
  when 5 => digit2 <= "1011011"; -- a b c d e f g = 5B
  when others => null;
end case;
end process;
end circuito_1;

```



(displays de cátodo común)

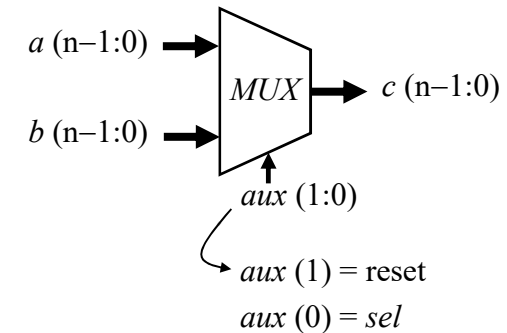
*Ejemplo 3* de código secuencial utilizando *case*: descripción de  $n$  multiplexores de 2 canales, con reset activo a nivel alto.

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

entity multiplexor_n_canales is
    generic (n : integer := 8);
    port (aux : in std_logic_vector (1 downto 0);
          a, b : in std_logic_vector (n - 1 downto 0);
          c : out std_logic_vector (n - 1 downto 0));
end multiplexor_n_canales;
```

architecture behavioral of multiplexor\_n\_canales is  
begin

```
    process (aux, a, b)
    begin
        case aux is
            when "00" => c <= a;
            when "01" => c <= b;
            when "10" => c <= (others => '0');
            when others => c <= (others => '0'); -- es obligatorio poner when others
        end case;
    end process;
end behavioral;
```

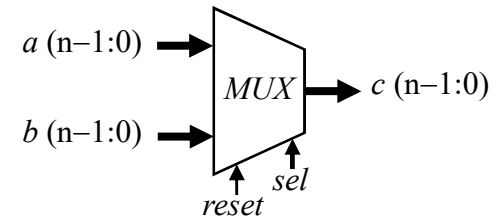


*Otra forma de describir  $n$  multiplexores de 2 canales, con reset activo a nivel alto.*

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

entity multiplexor_n_canales is
    generic (n : integer := 8);
    port (reset, sel : in std_logic;
          a, b : in std_logic_vector (n - 1 downto 0);
          c : out std_logic_vector (n - 1 downto 0));
end multiplexor_n_canales;

architecture behavioral of multiplexor_n_canales is
begin
    process (reset, a, b, sel)
    begin
        if (reset = '1') then c <= (others => '0');
        elsif (sel = '0') then c <= a;
        else c <= b;
        end if;
    end process;
end behavioral;
```






*Nota:*

	<i>When</i>	<i>Case</i>
Tipo de ejecución	<i>Concurrente</i>	<i>Secuencial</i>
Uso	Sólo se puede utilizar fuera de <i>process</i> , <i>functions</i> o <i>procedures</i> .	Sólo se puede utilizar en <i>process</i> , <i>functions</i> o <i>procedures</i> .
Para indicar <i>sin acción</i>	<i>unaffected</i>	<i>null</i>

*Ejemplo con when: (concurrente)*

*with sel select*

*x* <= *a* when “000”,  
    *b* when “001”,  
    *c* when “010”,  
    unaffected when others;

 *x* <= *a* cuando *sel* = “000”

*Ejemplo con case: (secuencial)*

*case sel is*

    when “000” => *x* <= *a*,  
    when “001” => *x* <= *b*,  
    when “010” => *x* <= *c*,  
    when others => null;  
*end case;*

e) *loop*:

Se utiliza cuando se debe repetir varias veces la ejecución de un conjunto dado de instrucciones. Se utiliza exclusivamente con código secuencial, por lo que sólo se puede utilizar en *processes*, *functions* o *procedures*.

Hay varias formas de utilizar un *loop*. Su sintaxis es la siguiente:

- *for* *identificador* *in* *rango* *loop*    -- el bucle se repite un número fijo de veces  
    -- *instrucciones secuenciales*  
*end loop*;
- *while* *condición* *loop*    -- el bucle se repite mientras se cumpla la *condición*  
    -- *instrucciones secuenciales*  
*end loop*;

Se puede definir un *bucle infinito* de la siguiente forma:

*loop*

*-- instrucciones secuenciales*

*end loop;*

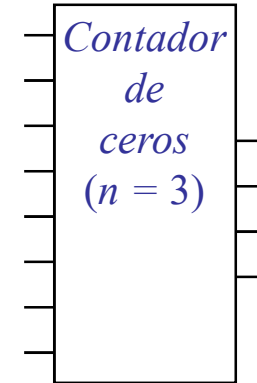
*Nota:* con *exit* se sale de un bucle y con *reset* finaliza la iteración actual y se pasa a la siguiente iteración.

*Ejemplo* 1 de uso de *loop*: circuito que cuenta el número de ceros que hay aplicados en sus  $2^n$  entradas.

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

entity Contador_de_ceros is
    generic (n : integer := 3); -- n establece el número de entradas ( $2^n$ )
    port (a : in std_logic_vector (2**n-1 downto 0);
          b : out std_logic_vector (n downto 0));
end Contador_de_ceros;

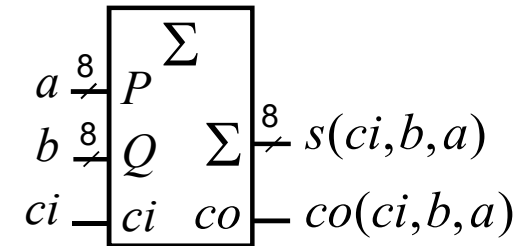
architecture Behavioral of Contador_de_ceros is
begin
    process(a)
        variable aux : unsigned (n downto 0); -- para contar los ceros
    begin
        aux := (others => '0'); -- cada vez que cambia el valor de a la variable aux se pone a cero
        for i in 0 to 2**n-1 loop
            if (a(i) = '0') then aux := aux + 1; -- se realiza una suma aritmética
            end if;
        end loop;
        b <= std_logic_vector(aux); -- conversión de tipo
    end process;
end Behavioral;
```



*Ejemplo 2* de uso de *loop*: descripción de un *sumador total* de 8 bits.

```
library ieee;
use ieee.std_logic_1164.all;
entity sumador_8 is
    port (a, b : in std_logic_vector (7 downto 0);
          ci : in std_logic;
          s : out std_logic_vector (7 downto 0)
          co : out std_logic);
end sumador_8;
```

```
architecture sumador of sumador_8 is
begin
    process (a,b,ci)
        variable carry : std_logic_vector (8 downto 0);
    begin
        carry(0) := ci;
        for i in 0 to 7 loop
            s(i) <= carry(i) xor b(i) xor a(i);
            carry(i+1) := (carry(i) and b(i)) or (carry(i) and a(i)) or (b(i) and a(i));
        end loop;
        co <= carry(8);
    end process;
end sumador;
```



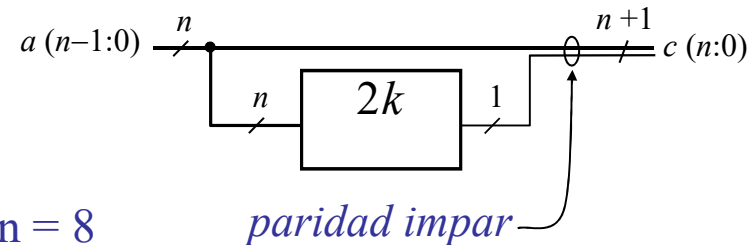
*revisa tema 4 SD*

*Ejemplo 3* de uso de *loop*: descripción de un *generador de paridad impar* de 8 bits.

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

entity generador_paridad_impar_n_bits is
    generic (n : integer := 8); -- valor por defecto n = 8
    port (a : in std_logic_vector (n - 1 downto 0);
          b : out std_logic_vector (n downto 0));
end generador_paridad_impar_n_bits;
```

```
architecture behavioral of generador_paridad_impar_n_bits is
begin
    process (a)
        variable aux : std_logic;
    begin
        aux := '0';
        for i in 0 to n-1 loop
            aux := aux xor a(i);
        end loop;
        b <= not aux & a;
    end process;
end behavioral;
```



*Ejemplo 4* de uso de un *loop*: descripción de un codificador de 8 a 3 de alta prioridad

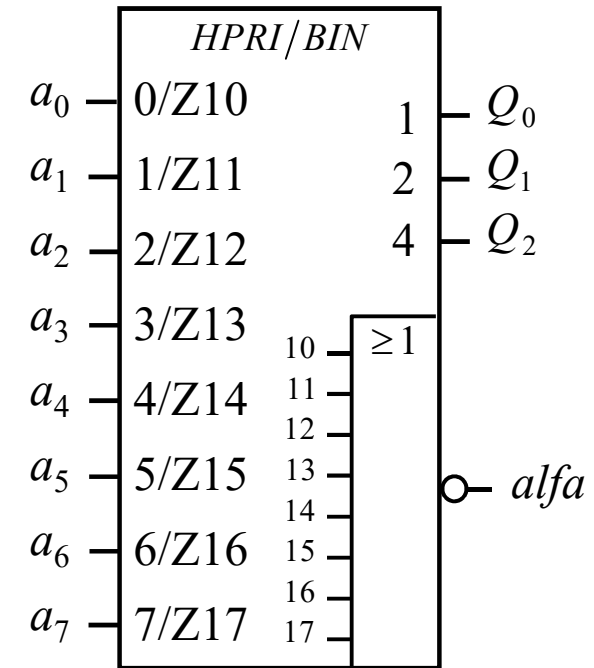
```

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

entity encoder_8_3 is
    port(a : in std_logic_vector (7 downto 0);
          alfa : out std_logic;
          Q : out std_logic_vector (2 downto 0));
end encoder_8_3;

architecture encoder_8_3 of encoder_8_3 is
begin
    process (a)
        variable aux : integer range 0 to 2;
        variable aux2 : std_logic;
    begin
        aux2 := '1';
        for i in 0 to 7 loop
            if (a(i) = '1') then
                aux := i;
                aux2 := '0';
            end if; -- continúa en la siguiente página
        end loop;
    end process;
end architecture;

```



$$\begin{aligned}
 alfa &= \overline{a_0} + a_0 + a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7 = \\
 &= \overline{a_0} \cdot \overline{a_1} \cdot \overline{a_2} \cdot \overline{a_3} \cdot \overline{a_4} \cdot \overline{a_5} \cdot \overline{a_6} \cdot \overline{a_7}
 \end{aligned}$$

```

end loop;
  Q <= std_logic_vector (to_unsigned (aux,3));
  alfa <= aux2;
end process;
end encoder_8_3;

```

Nota: la salida *alfa* se pone a 1 siempre que todas las entradas estén a 0. Es decir:

$$alfa = \overline{a_0 + a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7} = \bar{a}_0 \cdot \bar{a}_1 \cdot \bar{a}_2 \cdot \bar{a}_3 \cdot \bar{a}_4 \cdot \bar{a}_5 \cdot \bar{a}_6 \cdot \bar{a}_7$$



**Notas** sobre describir un circuito *combinacional* utilizando un *proceso*:

- 1: Todas las señales de *entrada* del circuito deben aparecer en su lista de sensibilidad.
- 2: En el código se debe definir el valor de las salidas para todas las combinaciones de valores de las entradas.
- 3: Si se utiliza un *if* entonces debe aparecer un *else*. Y con cada condición evaluada se debe asignar un valor a cada señal.
- 4: Hay que utilizar los tipos de datos *std\_logic* y *std\_logic\_vector* en las declaraciones de los puertos en las entidades.
- 5: Hay que utilizar el paquete *numeric\_std* y los tipos de datos *unsigned* y *signed* con las *señales* y las *variables* que se utilicen en operaciones *aritméticas* en las arquitecturas
- 6: Con *constants* y *generics* hay que utilizar el tipo *integer*

### Creación de *biestables* no deseados en procesos:

La norma VHDL especifica que una señal debe mantener su valor si no se le asigna uno nuevo en un proceso. Esto hace que, en el caso de que no se asigne un nuevo valor a una señal, el sintetizador utilice un *latch* para guardar el valor de la señal.

Para evitar que se generen biestables no deseados hay que hacer lo siguiente:

- \_ incluir todas las entradas en la lista de sensibilidad.
- \_ especificar todas las opciones en una instrucción condicional
- \_ las instrucciones *if... then ... else* deben finalizar con un *else*
- \_ asignar un valor a cada señal en cada condición evaluada
- \_ cuando se utiliza un *case* o bien se especifican todas las alternativas o bien se pone *when others =>*

*Ejemplos de instrucciones concurrentes  
(fuera de procesos):*

```
c <= a when b = '1' else  
    "011" when b = '0' else  
    "110";
```

*with sel select*

```
Q <= a0 when "0001",  
    a1 when "0010",  
    '0' when others;
```

*Ejemplos de instrucciones secuenciales (en procesos):*

```
if (a > b) then  
    mayor <= '1';  
    igual <= '0';  
    menor <= '0';  
elsif (c = b) then  
    mayor <= '0';  
    igual <= '1';  
    menor <= '0';  
else  
    mayor <= '0';  
    igual <= '0';  
    menor <= '1';  
end if;
```

*Ejemplos de instrucciones secuenciales (en procesos):*

*case* aux *is*

*when* “00” => a <= b+c;

*when* “01” => a <= b;

*when* “10” => a <= c;

*when others* => a <= d;

*end case*;

*while* (valor /= tabla(pos) *or* pos < 100) *loop*

pos := pos +1;

*end loop*;

*Ejemplos de instrucciones secuenciales (en procesos):*

*for* i in 0 to 3 *loop*

a(i) <= i+1;

c(i-1) <= c(i);

*end loop*;

*wait for* 10 nsg.; -- sólo en simulación

*wait until* rising\_edge (clk);

*wait on* a, b, clk, reset;

Descripción de  *circuitos secuenciales síncronos*  (máquinas con un número finito de estados)

En la práctica, el comportamiento de un circuito secuencial síncrono se puede describir de varias formas. A continuación se muestran mediante ejemplos diferentes implementaciones de *sistemas secuenciales síncronos* correspondientes a modelos de *Moore*.

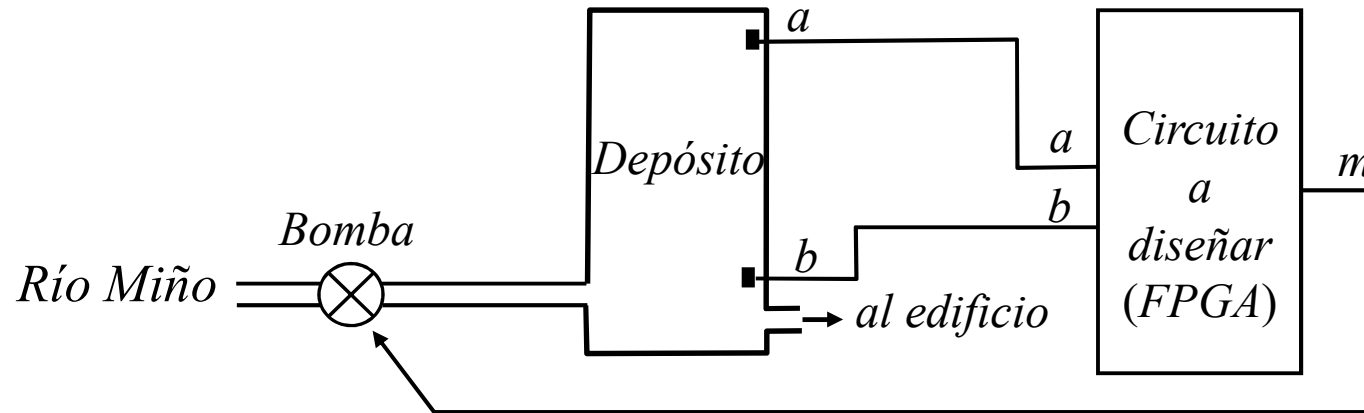
Los diferentes métodos que hay de implementar un sistema secuencial síncrono presentan ventajas e inconvenientes en cuanto a:

- \_ El número de puertas a utilizar en el circuito combinacional de entrada (*cce*), lo cual está directamente relacionado con la máxima frecuencia de la señal de reloj.
- \_ La complejidad del código.
- \_ El número de macroceldas, el número de flip-flops (registros), etc.

Nota: en los circuitos secuenciales síncronos la lista de sensibilidad sólo debe estar formada por la señal de reloj y por la señal de reset (en el caso de que el reset sea asíncrono).

A las entradas dinámicas sólo debe llegar la señal de reloj.

*Ejemplo: control del nivel del agua en un depósito que abastece un edificio (ver figura)*

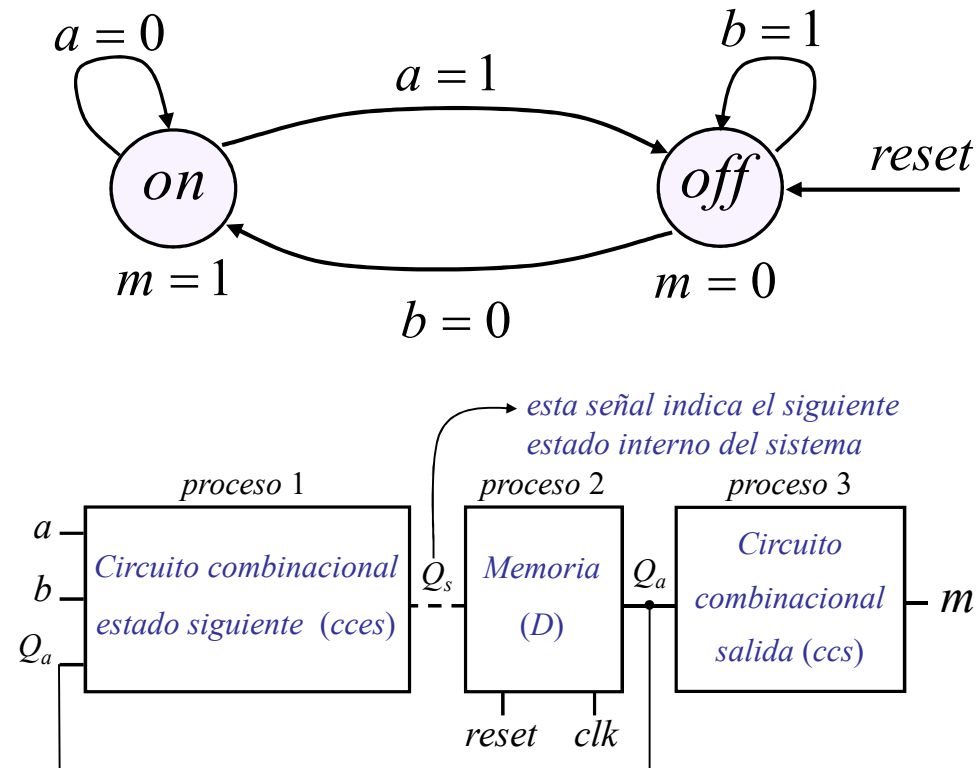


$$a = \begin{cases} 1 & \text{si el nivel del agua supera la posición del sensor (a)} \\ 0 & \text{si el nivel del agua es inferior a la posición del sensor (a)} \end{cases}$$

$$b = \begin{cases} 1 & \text{si el nivel del agua supera la posición del sensor (b)} \\ 0 & \text{si el nivel del agua es inferior a la posición del sensor (b)} \end{cases}$$

$$m = \begin{cases} 1 & \text{el motor bombea agua al depósito} \\ 0 & \text{el motor está parado} \end{cases}$$

El sistema que controla el funcionamiento de la bomba de agua es un sistema secuencial. Su comportamiento se puede describir mediante un diagrama de flujo como el indicado a continuación (modelo de *Moore*).



$Q_a$  : estado actual

$Q_s$  : próximo estado o estado siguiente (al actual)



*Ejemplo* (estilo de diseño con 3 procesos): descripción del sistema secuencial síncrono definido mediante el diagrama de flujo de la diapositiva anterior (modelo de *Moore*).

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

entity control is
    port (a, b, clk, reset : in std_logic;
          m : out std_logic);
end control;
```

```
architecture control of control is
    type estado is (off1, on1); -- declaración de un tipo de dato enumerated
    signal Qa, Qs : estado; -- declaración de dos señales de tipo estado
begin
```

```
    process (a, b, Qa) -- determina el siguiente estado
```

```
    begin
```

```
        case Qa is -- Qa : estado actual
```

```
            when off1 =>
```

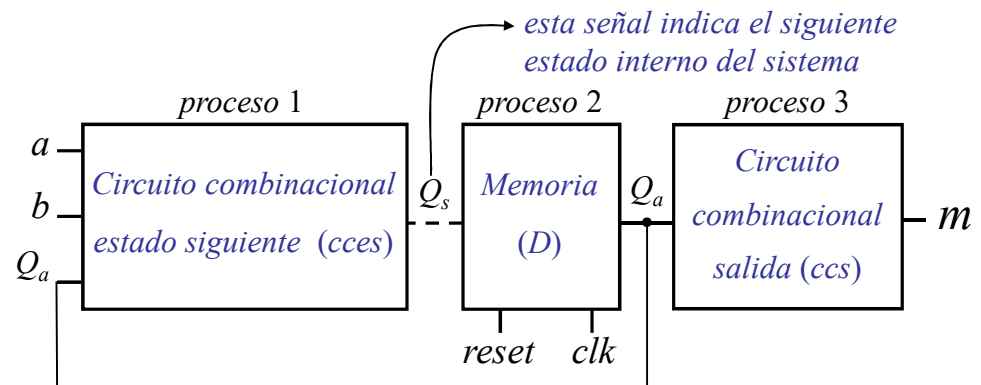
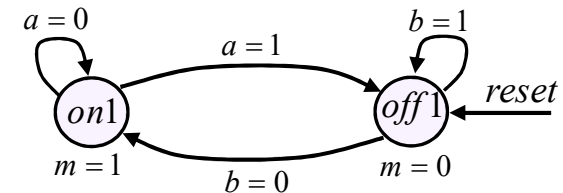
```
                if (b = '0') then
```

```
                    Qs <= on1;
```

```
                else
```

```
                    Qs <= off1;
```

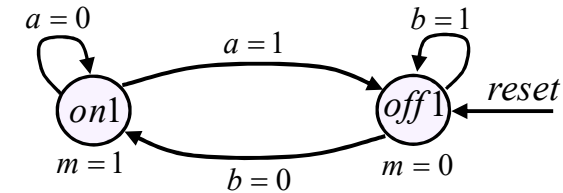
```
                end if;
```



```

when on1 =>
  if (a = '1') then
    Qs <= off1;  -- Qs : estado siguiente
  else
    Qs <= on1;
  end if;
end case;
end process;

```



*process* (reset, clk) -- este proceso implementa el bloque de memoria

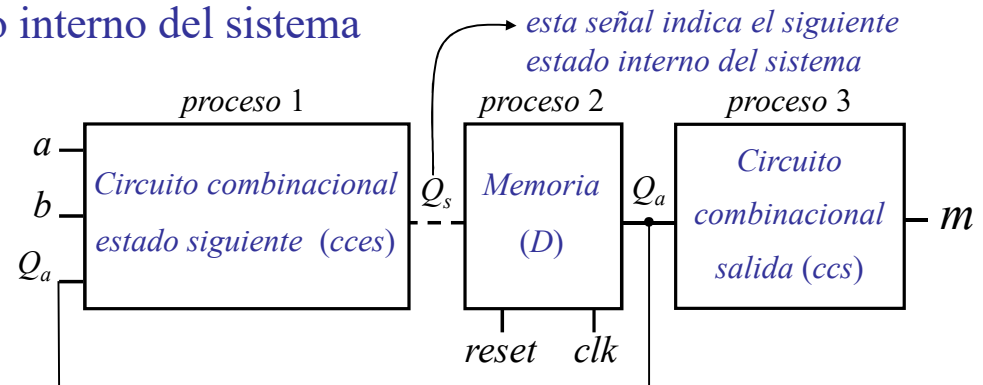
begin -- actualiza el estado interno del sistema

```

if(reset = '1') then
  Qa <= off1;
elsif (rising_edge(clk)) then
  Qa <= Qs;
end if;

```

*end process*;



*process* (Qa) -- este proceso implementa el circuito combinacional de salida (ccs)

begin -- determina la salida (m) del sistema a partir del valor del estado interno (Qa)

```

if (Qa = off1) then
  m <= '0';
else
  m <= '1';
end if;

```

*end process*;

end control;

*Ejemplo* (estilo de diseño con 2 procesos): otra descripción en vhdl del sistema secuencial síncrono anterior (modelo de *Moore*).

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

```
entity control is
    port (a, b, clk, reset : in std_logic;
          m : out std_logic);
end control;
```

```
architecture control of control is
```

```
    type estado is (off1, on1); -- declaración de un tipo de dato enumerated
```

```
    signal Q : estado; -- sólo se actualiza 1 vez en cada ejecución del proceso
```

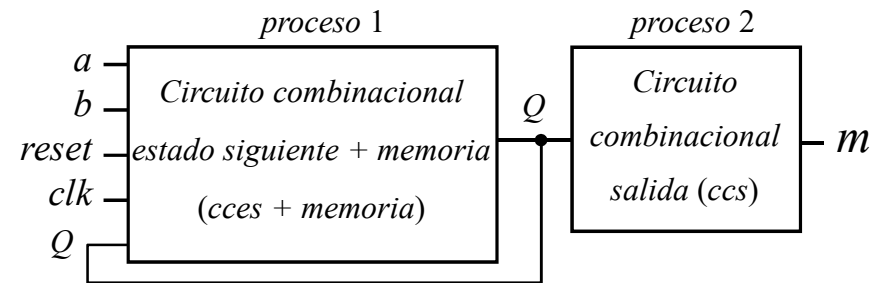
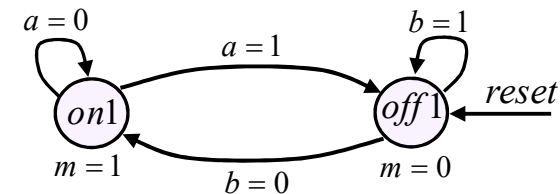
```
begin
```

```
    process (reset, clk) -- este proceso calcula Q (cces + memoria)
```

```
    begin
```

```
        if (reset = '1') then Q <= off1;
```

```
        elsif (rising_edge(clk)) then -- continúa en la siguiente página
```



case  $Q$  is -- se define la evolución del estado interno

when  $off1 \Rightarrow$

if ( $b = '0'$ ) then  $Q \leq on1$ ;

else  $Q \leq off1$ ;

end if;

when  $on1 \Rightarrow$

if ( $a = '1'$ ) then  $Q \leq off1$ ;

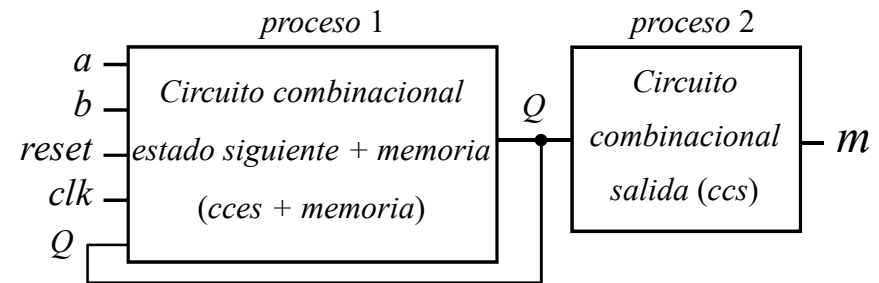
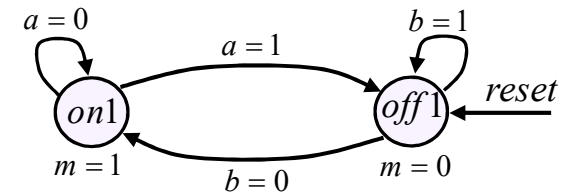
else  $Q \leq on1$ ;

end if;

end case;

end if;

end process;



process ( $Q$ ) -- calcula  $m$  a partir del estado interno  $Q$  (ccs) (no es necesario utilizar un

begin -- proceso para determinar la salida a partir del estado interno)

if ( $Q = off1$ ) then  $m \leq '0'$ ;

else  $m \leq '1'$ ;

end if;

end process;

end control;

*Ejemplo* (estilo de diseño con 2 procesos): otra descripción del sistema secuencial síncrono anterior, pero ahora con la *salida sincronizada* con la señal de *reloj*.

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

```
entity control is
  port (a, b, clk, reset : in std_logic;
        m, ms : out std_logic);
end control;
```

```
architecture control of control is
```

```
  type estado is (off1, on1); -- declaración de un tipo de dato enumerated
```

```
  signal Q : estado; -- declaración de una señal de tipo estado
```

```
begin
```

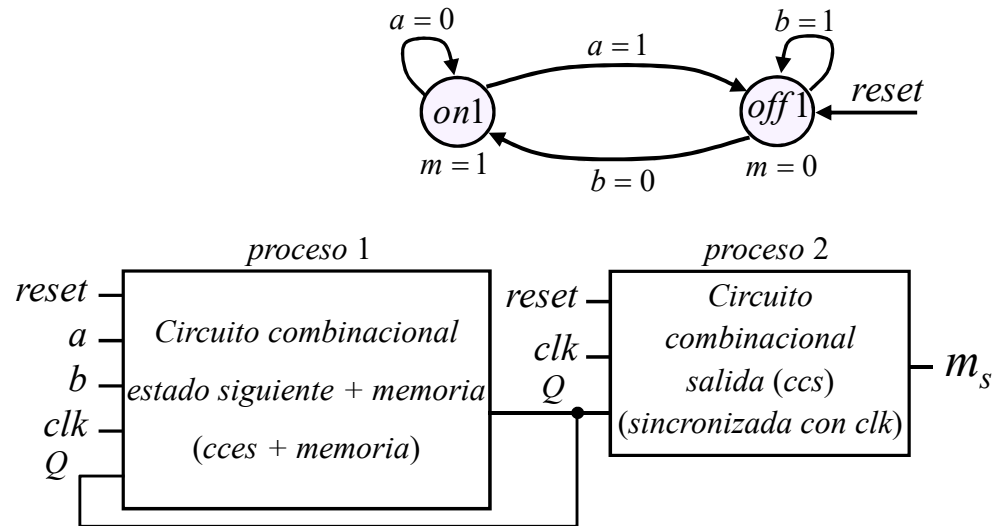
```
  process (reset,clk) -- este proceso calcula el valor del estado interno (Q)
```

```
  begin
```

```
    if (reset = '1') then
```

```
      Q <= off1; -- sólo Q puede tomar los valores off1 y on1
```

```
      m <= '0'; -- para ver el estado interno en la simulación
```



```
elsif (rising_edge(clk)) then
```

```
  case Q is -- se define la evolución del estado interno
```

```
    when off1 =>
```

```
      if (b = '0') then Q <= on1; m <= '1';
```

```
      else Q <= off1; m <= '0';
```

```
      end if;
```

```
    when on1 =>
```

```
      if (a = '1') then Q <= off1; m <= '0';
```

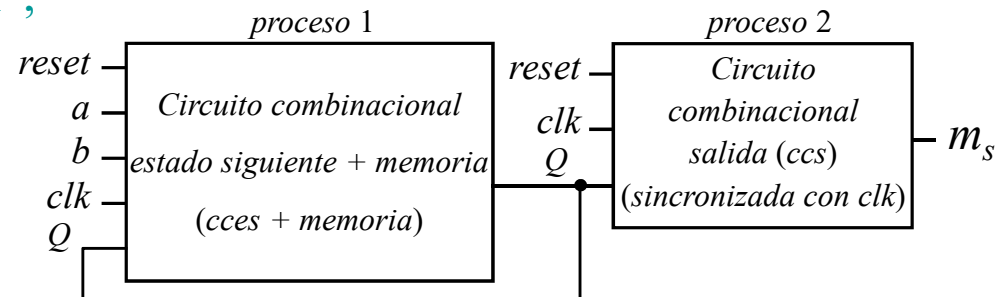
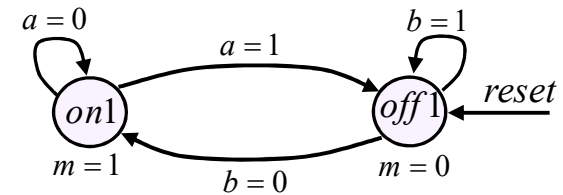
```
      else Q <= on1; m <= '1';
```

```
      end if;
```

```
  end case;
```

```
end if;
```

```
end process;
```



```
process (reset, clk) -- este proceso calcula la salida sincronizada con clk (ccs)
```

```
begin
```

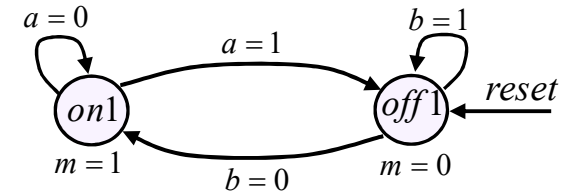
```
  if (reset = '1') then ms <= '0'; -- reset asíncrono
```

```
  elsif (rising_edge(clk)) then -- continúa en la siguiente página
```

```

case  $Q$  is -- se define el valor de la salida
  when off1 => ms <= '0';
  when on1 => ms <= '1';
end case;
end if;
end process;
end control;

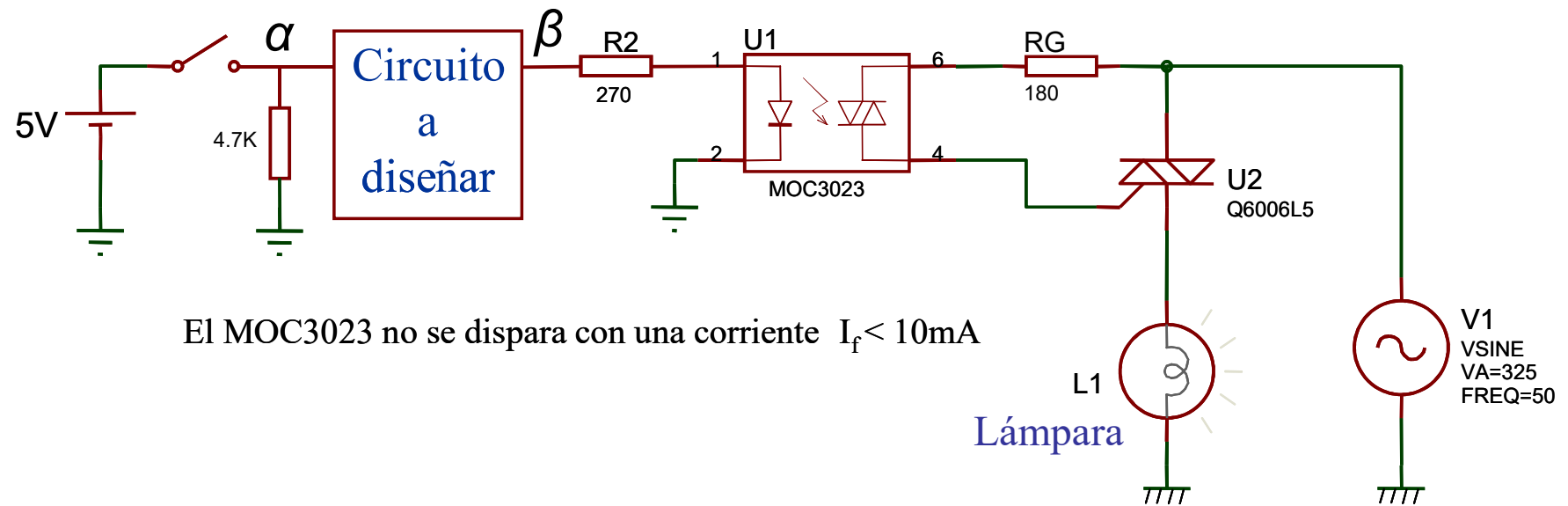
```



**Importante:** con el código anterior, la salida *m* actualiza su valor con un periodo (de *clk*) de retraso con respecto a la actualización del estado interno *Q*. Esto es debido a que el *proceso* que determina el estado interno (*Q*) y el proceso que determina la salida (*m<sub>s</sub>*) se ejecutan al mismo tiempo. Lo que hace que el proceso que calcula el valor de la salida (*m<sub>s</sub>*) utilice el valor de *Q* previo al que se está calculando al mismo tiempo en el otro proceso.

Nota: la salida *m* sólo se utiliza para la simulación (para ver el valor del estado interno)

*Ejemplo:* sistema de control de un pulsador. Cada vez que se presione el pulsador  $\alpha$  cambia de estado (encendido / apagado) de la bombilla.





Se establecen las siguientes definiciones:

Nota: pulsador  $\neq$  interruptor

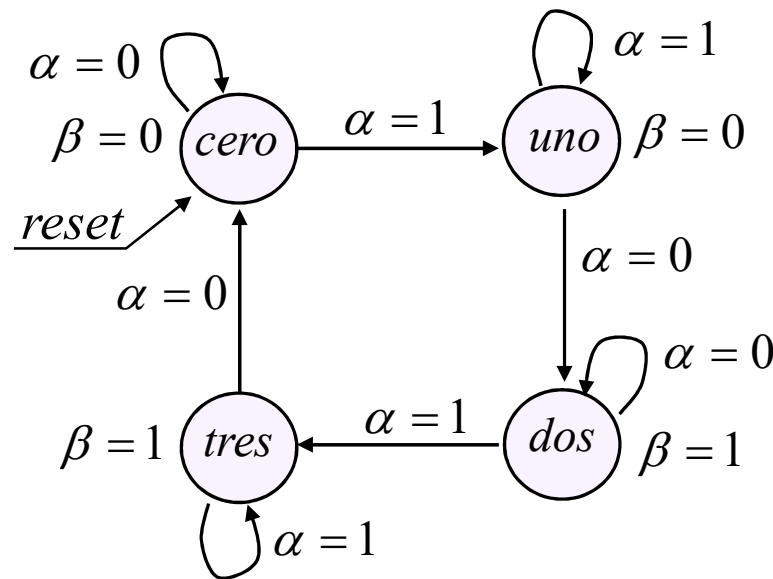
$$\alpha = \begin{cases} 1 \text{ (5v) si el pulsador está presionado} \\ 0 \text{ (0v) si el pulsador no está presionado} \end{cases}$$

$$\beta = \begin{cases} 1 \text{ la bombilla está encendida} \\ 0 \text{ la bombilla está apagada} \end{cases}$$

(modelo de Moore)

$\alpha$  : entrada

$\beta$  : salida



*Ejemplo* (estilo de diseño con 3 procesos): descripción del sistema secuencial síncrono definido mediante el diagrama de flujo de la diapositiva anterior (modelo de *Moore*).

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

entity control is
    port (alfa, clk, reset : in std_logic;
          beta : out std_logic);
end control;
```

```
architecture control of control is
```

```
    type estado is (cero, uno, dos, tres); -- declaración de un tipo de dato enumerated
```

```
    signal Qa, Qs : estado; -- declaración de dos señales de tipo estado
```

```
begin
```

```
    process (alfa, Qa) -- determina el siguiente estado interno del sistema (cces)
```

```
begin
```

```
    case Qa is -- Qa es el estado actual
```

```
        when cero =>
```

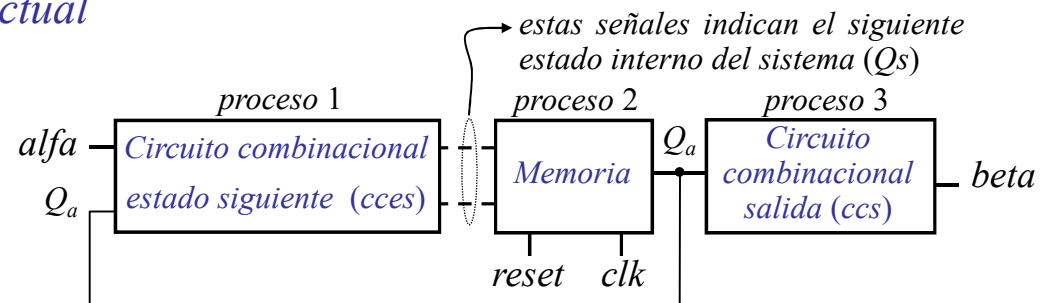
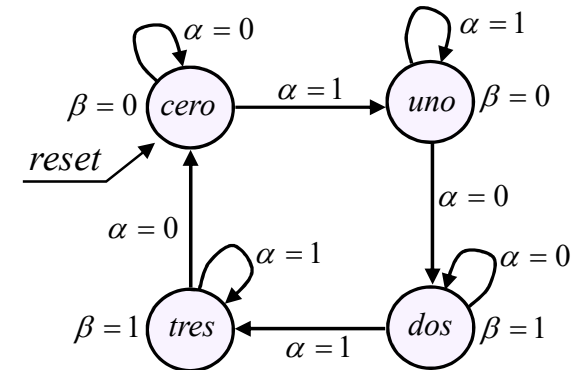
```
            if (alfa = '1') then
```

```
                Qs <= uno;
```

```
            else
```

```
                Qs <= cero;
```

```
            end if;
```



```

when uno =>
    if (alfa = '0') then Qs <= dos;
    else Qs <= uno;
    end if;
when dos =>
    if (alfa = '1') then Qs <= tres;
    else Qs <= dos;
    end if;
when others =>
    if (alfa = '0') then Qs <= cero;
    else Qs <= tres;
    end if;

```

end case;

end process;

*process* (reset, clk) -- este proceso implementa el bloque de *memoria*

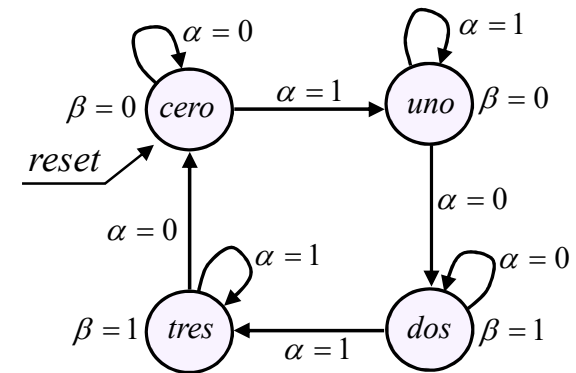
begin -- actualiza el estado interno del sistema

```

if (reset = '1') then Qa <= cero;
elsif (rising_edge(clk)) then Qa <= Qs;
end if;

```

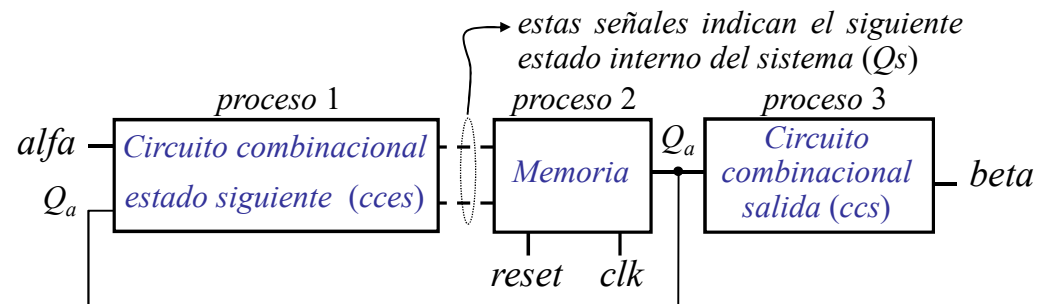
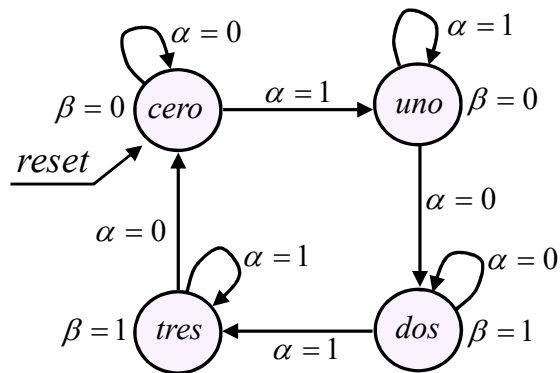
end process;



```

process (Qa) -- este proceso implementa el circuito combinacional de salida (ccs)
begin -- determina la salida (m) del sistema a partir del valor del estado interno (Qa)
  case Qa is
    when cero => beta <= '0';
    when uno  => beta <= '0';
    when dos  => beta <= '1';
    when others => beta <= '1'; -- para evitar la implementación de latches no deseados
  end case;
end process;
end control;

```



*Ejemplo* (estilo de diseño con 2 procesos): otra descripción del sistema secuencial anterior, pero ahora con la *salida sincronizada* con la señal de *reloj*

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

```
entity control is
    port (reset, clk, alfa : in std_logic;
          beta : out std_logic);
end control;
```

```
architecture control of control is
```

```
    type estado is (cero, uno, dos, tres); -- declaración de un tipo de dato enumerated
```

```
    signal Q : estado; -- sólo se actualiza 1 vez en cada ejecución del proceso
```

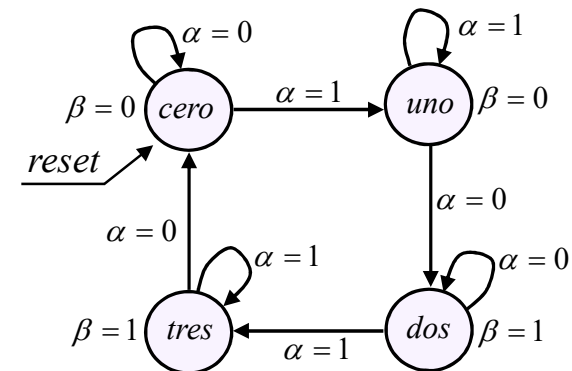
```
begin
```

```
    process (reset, clk) -- este proceso calcula Q (cces + memoria)
```

```
    begin
```

```
        if (reset = '1') then Q <= cero;
```

```
        elsif (rising_edge(clk)) then -- continúa en la siguiente página
```



case  $Q$  is -- se define la evolución del estado interno

when *cero* =>

if (*alfa* = '1') then  $Q \leq uno$ ;

else  $Q \leq cero$ ;

end if;

when *uno* =>

if (*alfa* = '0') then  $Q \leq dos$ ;

else  $Q \leq uno$ ;

end if;

when *dos* =>

if (*alfa* = '1') then  $Q \leq tres$ ;

else  $Q \leq dos$ ;

end if;

when *others* =>

if (*alfa* = '0') then  $Q \leq cero$ ;

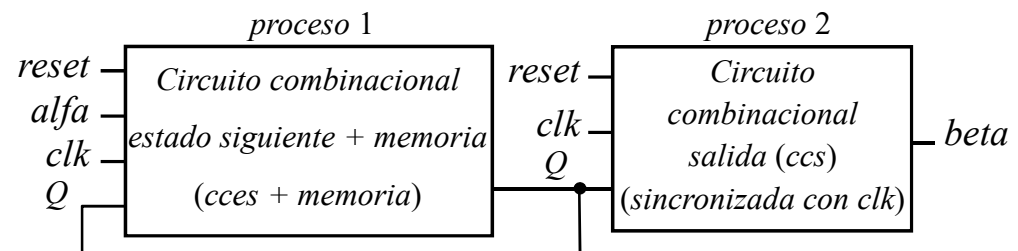
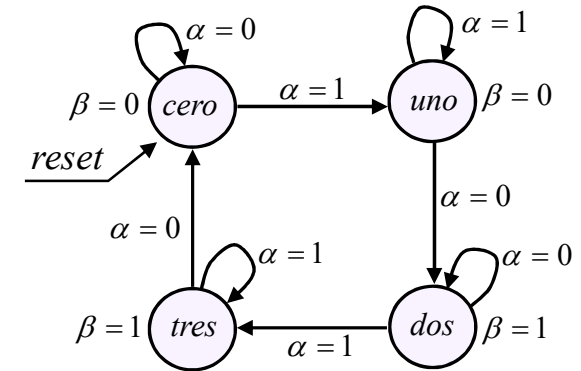
else  $Q \leq tres$ ;

end if;

end case;

end if;

end process;



*process* (*reset*, *clk*) -- este proceso calcula la salida sincronizada con *clk* (*ccs*)

begin

if (*reset* = '1') then *beta* <= '0'; -- reset asíncrono

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

case *Q* is

when *cero* => *beta* <= '0';

when *uno* => *beta* <= '0';

when *dos* => *beta* <= '1';

when others => *beta* <= '1';-- para evitar biestables no deseados

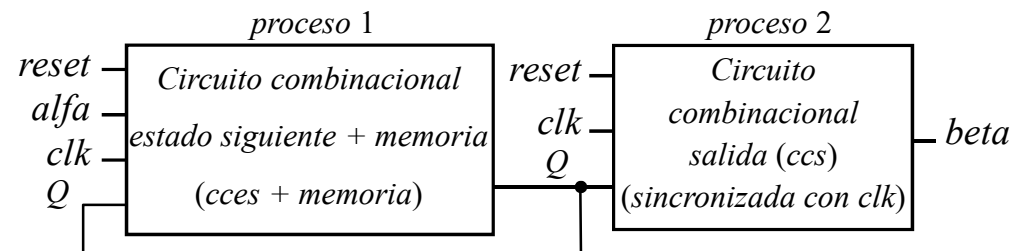
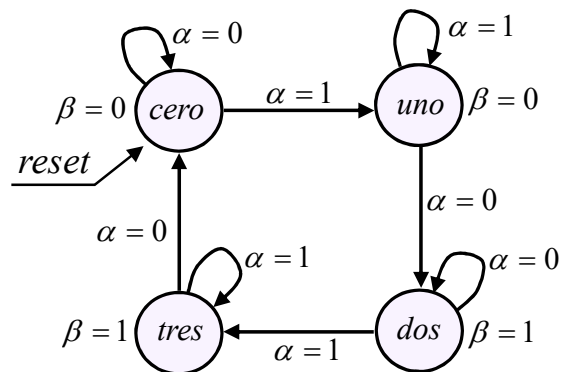
end case;

end if;

end *process*;

end control;

**Importante:** con el código anterior, la salida *m* actualiza su valor con un periodo de reloj de retraso con respecto a la actualización del estado interno *Q*. Esto es debido a que el *proceso* que determina el estado interno (*Q*) y el proceso que determina la salida (*m<sub>s</sub>*) se ejecutan al mismo tiempo. Lo que hace que el proceso que calcula el valor de la salida (*m<sub>s</sub>*) utilice el valor de *Q* previo al que se está calculando al mismo tiempo en el otro proceso



*Ejemplo* (de diseño con 3 procesos): descripción de un sistema secuencial síncrono definido mediante el diagrama de flujo de la derecha (modelo de *Moore*).

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

```
entity control is
    port (a, clk, reset : in std_logic;
          z : out std_logic);
end control;
```

```
architecture control of control is
```

```
    type estado is (cero, uno, dos, tres); -- declaración de un tipo de dato enumerated
```

```
    signal Qa, Qs : estado; -- declaración de dos señales de tipo estado
```

```
begin
```

```
    process (a, Qa) -- determina el siguiente estado interno del sistema (cces)
```

```
begin
```

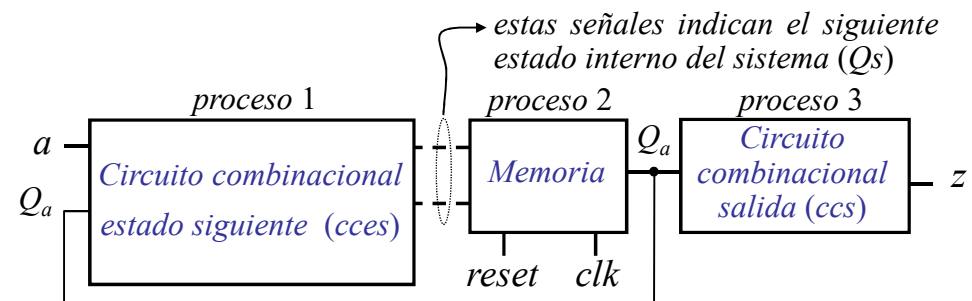
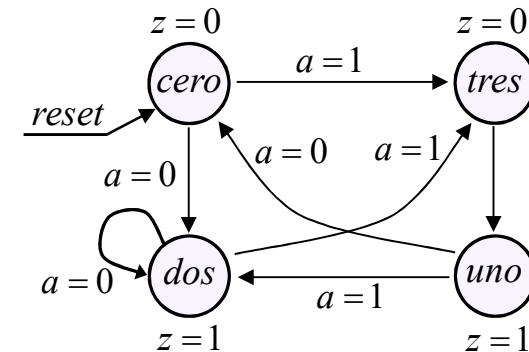
```
    case Qa is -- Qa es el estado actual
```

```
        when cero =>
```

```
            if (a = '1') then Qs <= tres;
```

```
            else Qs <= dos;
```

```
            end if;
```

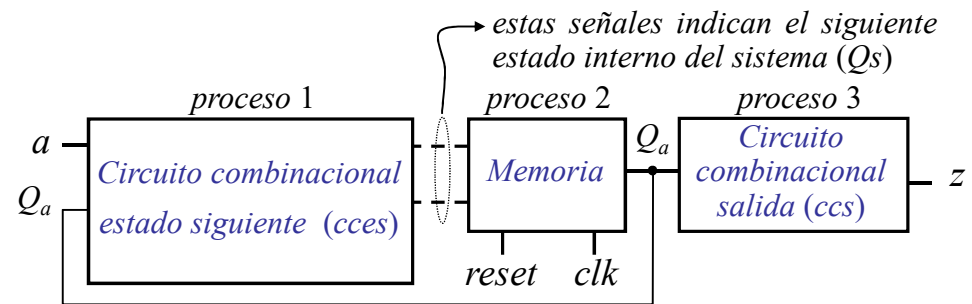
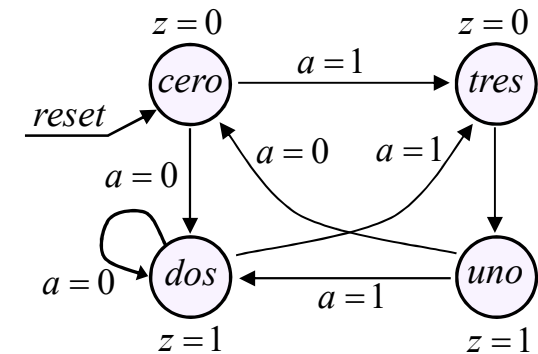




```

when uno =>
    if (a = '0') then Qs <= cero;
    else Qs <= dos;
    end if;
when dos =>
    if (a = '1') then Qs <= tres;
    else Qs <= dos;
    end if;
when others => Qs <= uno;
end case;
end process;

```



```

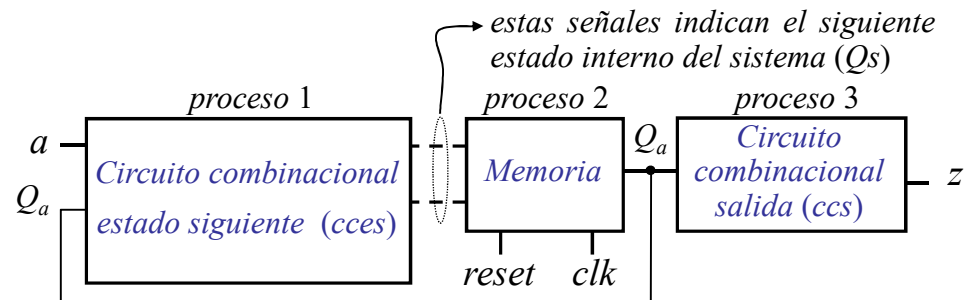
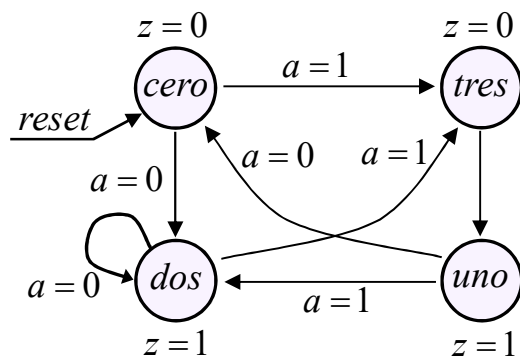
process (reset, clk) -- implementa el bloque de memoria
begin
    -- actualiza el estado interno del sistema
    if (reset = '1') then Qa <= cero;
    elsif (rising_edge(clk)) then Qa <= Qs;
    end if;
end process;

```

```

process ( $Q_a$ ) -- este proceso implementa el circuito combinacional de salida ( $ccs$ )
begin -- determina la salida ( $m$ ) del sistema a partir del valor del estado interno ( $Q_a$ )
  case  $Q_a$  is
    when cero =>  $z <= '0'$ ;
    when uno  =>  $z <= '1'$ ;
    when dos  =>  $z <= '1'$ ;
    when others =>  $z <= '0'$ ; -- para evitar la implementación de biestables no deseados
  end case;
end process;
end control;

```



*Ejemplo* (de diseño con 2 procesos): descripción del sistema secuencial síncrono anterior con la salida sincronizada con la señal de reloj (modelo de *Moore*).

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

```
entity control is
    port (a, clk, reset : in std_logic;
          z : out std_logic);
end control;
```

```
architecture control of control is
```

```
    type estado is (cero, uno, dos, tres); -- declaración de un tipo de dato enumerated
```

```
    signal Q : estado; -- declaración de una señal de tipo estado
```

```
begin
```

```
    process (reset, clk) -- determina el estado interno del sistema (cces)
```

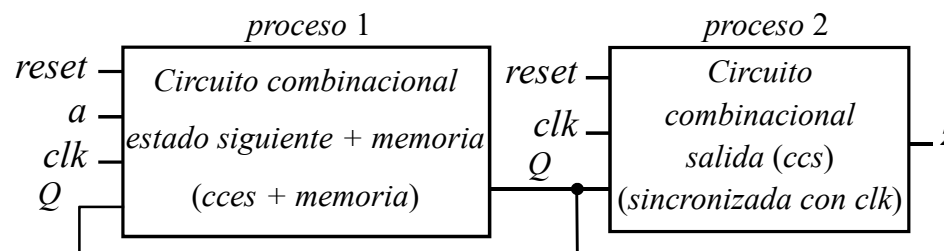
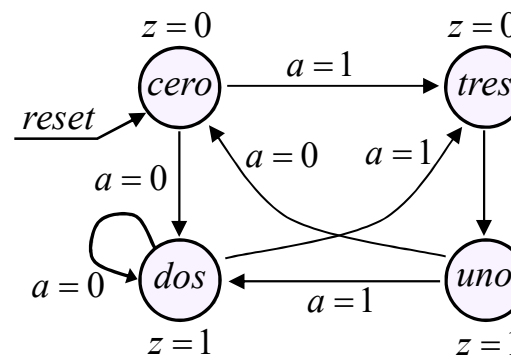
```
    begin
```

```
        if (reset = '1') then
```

```
            Q <= cero;
```

```
        elsif (rising_edge(clk)) then
```

```
        -- continúa en la página siguiente
```



case  $Q$  is -- se define la evolución del estado interno

when *cero* =>

if ( $a = '1'$ ) then  $Q \leq tres$ ;  
 else  $Q \leq dos$ ;  
 end if;

when *uno* =>

if ( $a = '0'$ ) then  $Q \leq cero$ ;  
 else  $Q \leq dos$ ;  
 end if;

when *dos* =>

if ( $a = '1'$ ) then  $Q \leq tres$ ;  
 else  $Q \leq dos$ ;  
 end if;

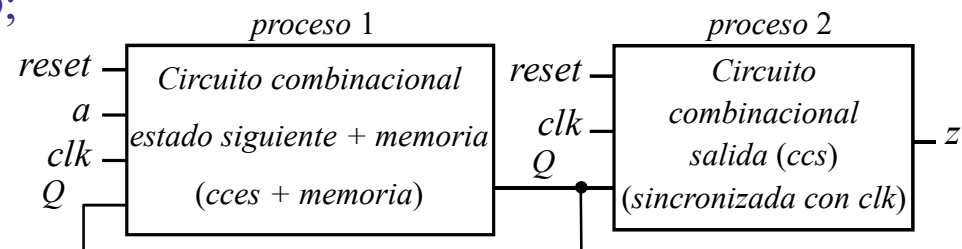
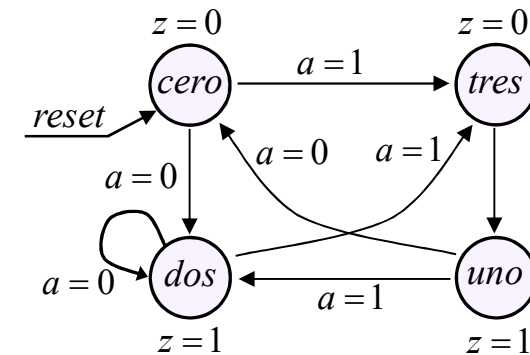
when *others* =>  $Q \leq uno$ ;

end case;

end if;

end process;

-- continúa en la página siguiente



*process* (*reset*, *clk*) -- este proceso calcula la salida sincronizada con *clk* (*ccs*)

begin

if (*reset* = '1') then *z* <= '0'; -- reset asíncrono

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

case *Q* is

when *cero* => *z* <= '0';

when *uno* => *z* <= '1';

when *dos* => *z* <= '1';

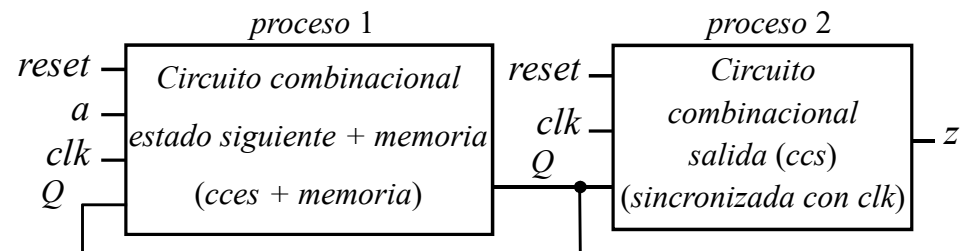
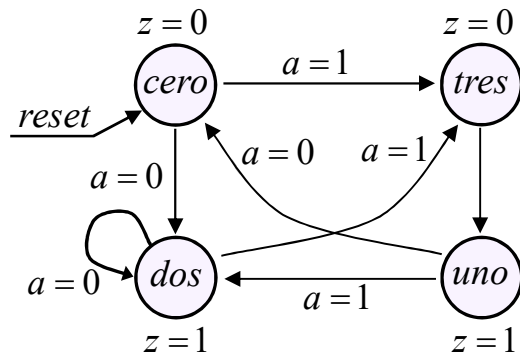
when others => *z* <= '0'; -- para evitar latches no deseados

end case;

end if;

end *process*;

end control;



## *Components* (componentes)

- Un *componente* es un código que describe el comportamiento de un circuito en vhdl y que como tal está formado por la declaración de *bibliotecas* (*libraries*), *paquetes*, una *entidad* y una *arquitectura*.
- Un *componente* está pensado para ser utilizado en la descripción de otros circuitos (equivale a una función en un lenguaje de programación), permitiendo reutilizar código y crear diseños jerárquicos.
- Para utilizar un *componente* en la descripción de un circuito es necesario *declararlo* en la *arquitectura* de dicho circuito (antes de *begin*). En la declaración de un *componente* se indican las características de los terminales (PORT) del circuito que describe así como los parámetros (*generic*) utilizados en su descripción, en caso necesario. Su sintaxis es la siguiente (es muy parecida a la de una entidad):

```
component nombre is    -- declaración de un componente  
port (nombre_entradas : in  tipo_señal;  
      nombre_salidas  : out tipo_señal);  
end component;
```

- La sintaxis de la *llamada* o *uso (instantiation)* de un *componente*, a escribir en el cuerpo de una *arquitectura*, es la siguiente:

*etiqueta* : *nombre\_componente*

*port map* (*relación entre terminales*); -- separados por comas

- \* Es obligatorio que la llamada al *componente* lleve una *etiqueta*
- \* La expresión “*relación entre terminales*” se refiere a la relación entre los nombres de los terminales utilizados al definir el *componente* y los nombres de los terminales que se conectan a dicho componente en el circuito en el que se utiliza. La forma habitual de indicar dicha relación es la siguiente:

*nombre terminal componente* => *nombre terminal del componente en el circuito*

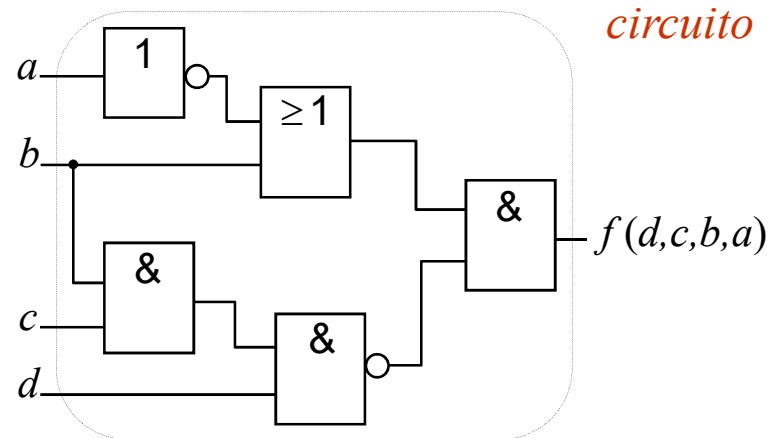
- \* no importa el orden en el que se establezca la relación entre los terminales
- La definición de un *componente* puede estar guardada en una biblioteca (*library*). En tal caso, para poder utilizar el componente es necesario incluir la biblioteca en el archivo vhd1 que lo utiliza, junto con las demás bibliotecas.

- En la descripción de un circuito se pueden utilizar *componentes* que no están guardados en una *biblioteca (library)*, siempre que se incluyan los archivos en los que se definen dichos *componentes* en el proyecto del circuito .

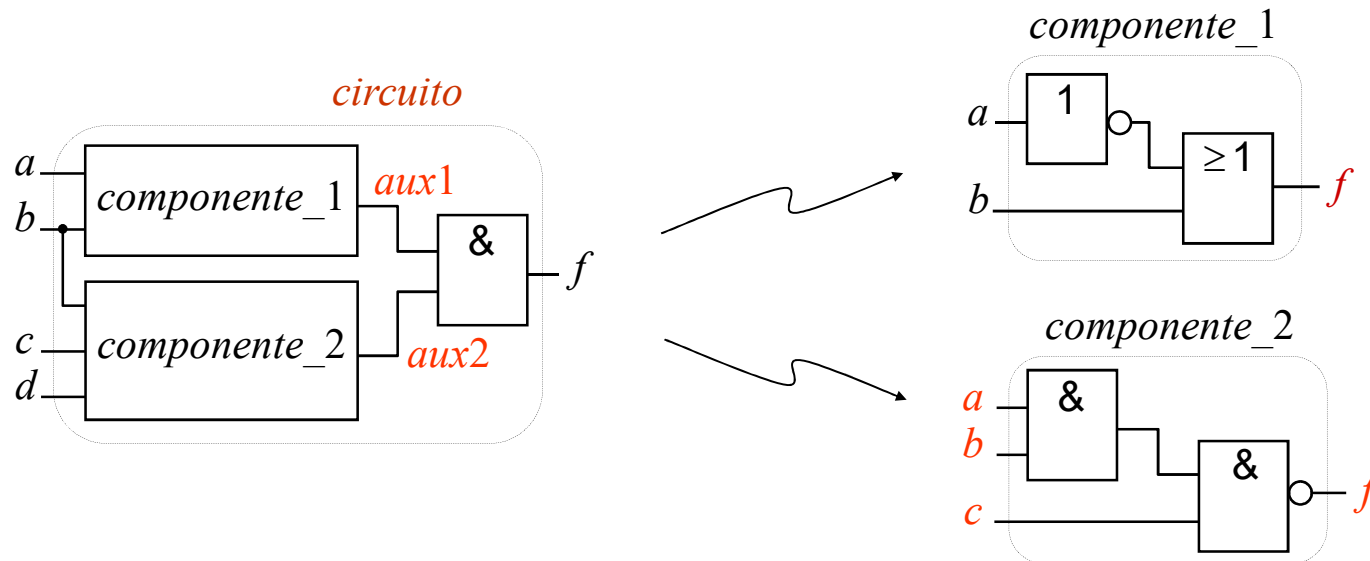
(ver *ejemplo* a continuación)



*Ejemplo* diseño jerárquico 1: descripción del siguiente circuito utilizando componentes



A modo de ejemplo, el circuito anterior se puede considerar formado por las siguientes partes:



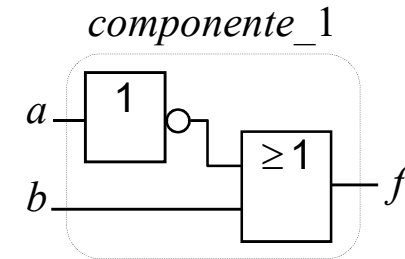
A continuación se definen los componentes 1 y 2 indicados en la diapositiva anterior así como el circuito. A la hora de describir el *circuito* se asume que los componentes 1 y 2 están definidos en archivos incluidos en el proyecto:

Definición del *componente* 1 indicado en la parte derecha y que se utilizará más adelante en la definición del *circuito*.

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

entity componente_1 is -- entidad del componente
    port (a, b : in std_logic;
          f : out std_logic);
end componente_1;

architecture componente_1 of componente_1 is -- arquitectura del componente
begin
    f <= not a or b; -- la operación not es prioritaria
end componente_1;
```

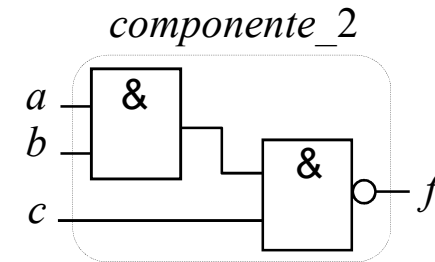


Definición del *componente 2* indicado en la parte derecha y que se utilizará más adelante en la definición del *circuito*.

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

entity componente_2 is -- entidad del componente
    port (a, b, c : in std_logic;
          f : out std_logic);
end componente_1;

architecture componente_2 of componente_2 is -- arquitectura del componente
begin
    f <= (a and b) nand c; -- not(a and b and c)
end componente_2;
```



**Nota:** el que los nombres de los terminales del *componente* sea distintos a los que tiene en el circuito que se va a utilizar es algo que habrá que tener en cuenta al llamar (*instantiate*) al *componente*.

Definición del *circuito* indicado en la parte derecha, teniendo en cuenta que los *componentes* 1 y 2 ya están definidos.

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

entity circuito_1 is -- entidad del circuito 1
    port (a, b, c, d : in std_logic;
          f : out std_logic);
end circuito_1;
```

```
architecture circuito of circuito_1 is -- arquitectura del circuito 1
```

```
    signal aux1, aux2 : std_logic;
```

```
    component Componente_1 -- declaración componente_1 (definido en otro archivo)
```

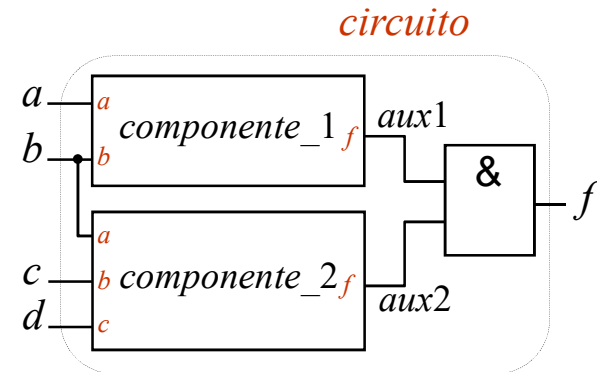
```
    port(a, b : in std_logic;
          f : out std_logic);
```

```
end component;
```

```
    component Componente_2 -- declaración componente_2 (definido en otro archivo)
```

```
    port(a, b, c : in std_logic;
          f : out std_logic);
```

```
end component;
```



begin -- arquitectura del *circuito*

U1 : componente\_1 -- llamada (instantiation) al componente\_1

port map (a => a, b => b, f => aux1); -- terminal componente => terminal circuito

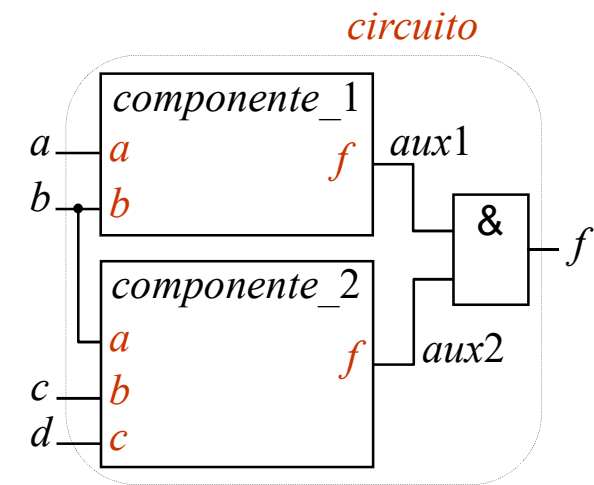
U2 : componente\_2 -- llamada al componente\_2

port map (a => b, b => c, c => d, f => aux2);

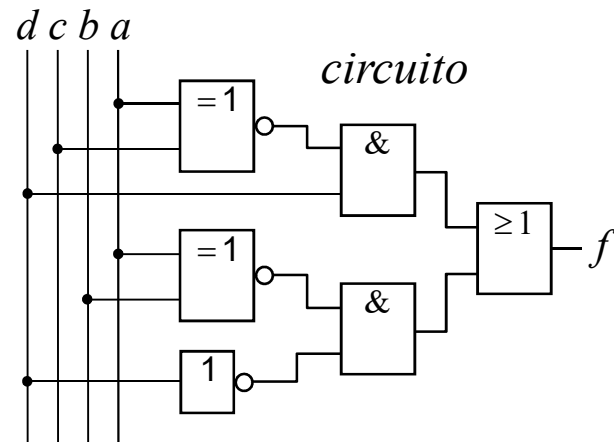
f <= aux1 and aux2;

end circuito\_1;

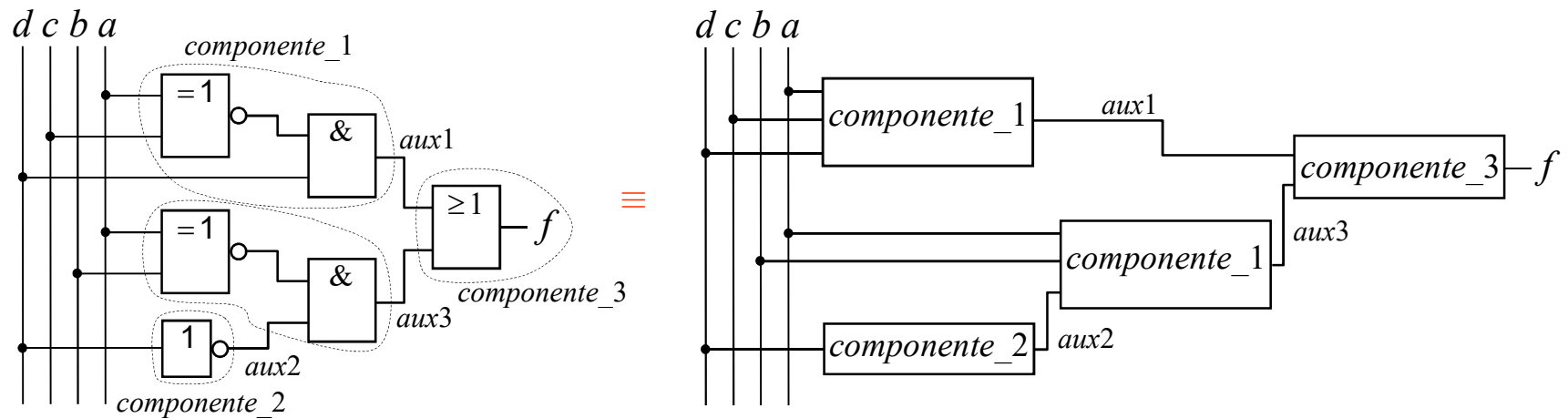
Nota:  $f(d, c, b, a) = \sum_4 (0, 2, 3, 4, 6, 7, 8, 10, 11, 12)$



Ejemplo diseño jerárquico: descripción del siguiente circuito utilizando componentes



el circuito anterior se puede considerar formado por las siguientes partes:

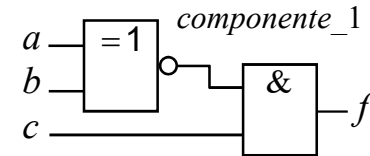


Definición del *componente* 1 indicado en la parte derecha y que se utilizará más adelante en la definición del *circuito*.

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

```
entity componente_1 is -- entidad del componente  
    port (a, b, c : in std_logic;  
          f : out std_logic);  
end componente_1;
```

```
architecture componente_1 of componente_1 is -- arquitectura del componente  
begin  
     $f \leq c \text{ and } (a \text{ xnor } b);$   
end componente_1;
```



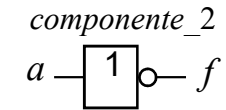


Definición del *componente 2* indicado en la parte derecha y que se utilizará más adelante en la definición del *circuito*.

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

entity componente_2 is -- entidad del componente
    port (a : in std_logic;
          f : out std_logic);
end componente_2;

architecture componente_2 of componente_2 is -- arquitectura del componente
begin
    f <= not a;
end componente_2;
```

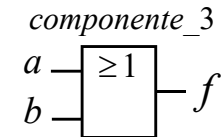


Definición del *componente 3* indicado en la parte derecha y que se utilizará más adelante en la definición del *circuito*.

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

entity componente_3 is -- entidad del componente
    port (a, b : in std_logic;
          f : out std_logic);
end componente_3;

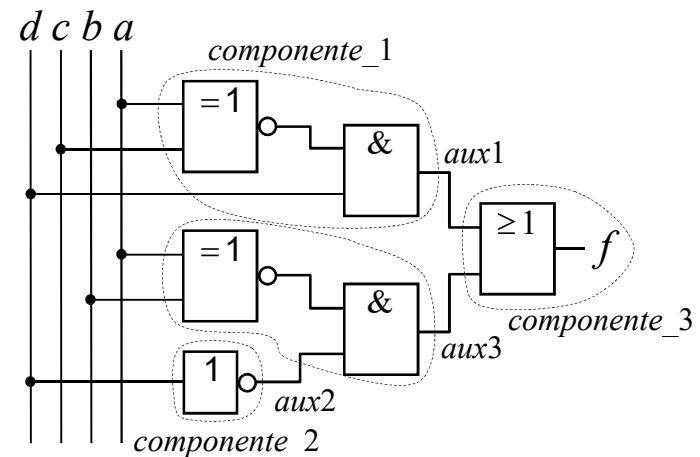
architecture componente_3 of componente_3 is -- arquitectura del componente
begin
    f <= b or a;
end componente_3;
```



Definición del *circuito* indicado en la parte derecha, teniendo en cuenta que los componentes 1 2 y 3 ya están definidos.

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

entity circuito_1 is -- entidad del circuito 2
    port (a, b, c, d : in std_logic;
          f : out std_logic);
end circuito_1;
```



```
architecture circuito of circuito is -- arquitectura del circuito 2
```

```
    signal aux1, aux2, aux3 : std_logic;
```

```
    component Componente_1 -- declaración componente_1 (definido en otro archivo)
```

```
    port(a, b, c : in std_logic;
```

```
          f : out std_logic);
```

```
end component;
```

```
    component Componente_2 -- declaración componente_2 (definido en otro archivo)
```

```
    port(a : in std_logic;
```

```
          f : out std_logic);
```

```
end component;
```

*component* Componente\_3 -- declaración componente\_3 (definido en otro archivo)

port(a, b : in std\_logic;

f : out std\_logic);

end *component*;

begin -- arquitectura del circuito

U1 : componente\_1 -- asignaciones de nombres

port map (a, c, d, aux1);

-- equivale a port map (a => a, b => c, c => d, f => aux1);

U2 : componente\_1 -- llamada (instantiation) al componente\_1

port map (a, b, aux2, aux3);

-- equivale a poner: port map (a => a, b => b, c => aux2, f => aux3);

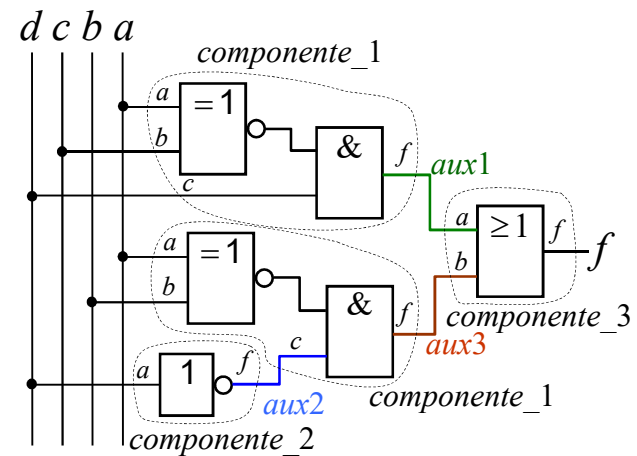
U3 : componente\_2 -- llamada al componente\_2

port map (d, aux2); -- equivale a poner port map (a => d, f => aux2);

U4 : componente\_3 -- llamada al componente\_3

port map (aux1, aux3, f); -- port map (a => aux1, b => aux3, f => f);

end circuito\_2;



Nota:  $f(d, c, b, a) = \sum_4 (0, 3, 4, 7, 8, 10, 13, 15)$

*Test bench (simulación funcional):*

Para simular el comportamiento de un circuito es necesario definir los valores de sus entradas durante un cierto intervalo de tiempo, con el fin de comprobar si los valores de las salidas correspondientes a dichas entradas son los deseados o no. En vhdl, se denomina *test bench* a un código que describe el valor de las entradas de un circuito durante un cierto intervalo de tiempo.

El código correspondiente a un *test bench* debe incluir una *entidad* y una *arquitectura*, las cuales se caracterizan por lo siguiente:

*Entity*: normalmente no se indican ni entradas ni salidas, de modo que su sintaxis suele ser la siguiente:

*entity* nombre *is*  
*end* nombre;

*Nota: haz click con el botón derecho en la zona en blanco de Hierarchy y elige VHDL Module. Modifica el archivo generado, marca la opción Simulation y selecciona el archivo testbench*

*Architecture*: aquí se realiza la llamada (*instantiation*) del circuito que se va a simular, considerado como un *componente* (el cual debe estar definido en otro archivo). Aquí también se incluye la definición de los valores de las entradas (ver siguiente *ejemplo*)

*Ejemplo:* definición de un *testbench* para simular el comportamiento del circuito indicado en la parte derecha (simulación funcional).

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

entity Comparador_2_tb is -- en un test bench no hay PORT
end Comparador_2_tb;
```

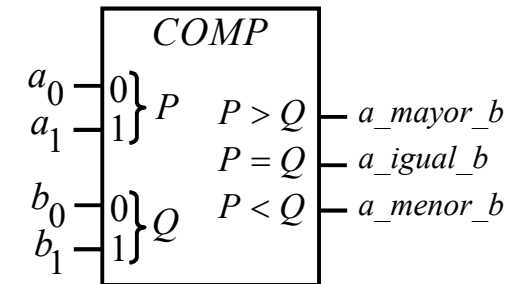
```
architecture Comparador_2 of Comparador_2_tb is
    component Comparador_2 -- declaración del circuito a simular como un componente
    port( a, b : in std_logic_vector (1 downto 0);
          a_mayor_b, a_menor_b, a_igual_b : out std_logic);
    end component;
```

```
    signal a, b : std_logic_vector (1 downto 0); -- declaración señales de interconexión
    signal a_mayor_b, signal a_igual_b, signal a_menor_b : std_logic;
```

```
begin -- architecture
```

```
    uut: Comparador_2 -- llamada al componente denominado Comparador_2
```

```
    port map (a => a, -- Se indica la relación entre los nombres de los terminales del
                    b => b, -- component y los nombres de los terminales a los que se conectan
                    a_mayor_b => a_mayor_b,
                    a_menor_b => a_menor_b,
                    a_igual_b => a_igual_b ); -- continúa en la siguiente página.
```



```
process -- genera las señales de entrada durante un tiempo dado
begin
  a <= "00";
  b <= "00";
  wait for 40ns; -- espera 40ns antes de continuar con la ejecución del código
  a <= "01";
  b <= "10";
  wait for 40ns;
  a <= "10";
  b <= "00";
  wait for 40ns;
  a <= "10";
  b <= "10";
  wait; -- mantiene los últimos valores establecidos hasta que finalice la simulación
end process;
end; -- architecture
```

Nota: el intervalo de tiempo especificado para las señales de entrada debe ser mayor que el tiempo de simulación indicado en el simulador (en *Process properties: simulation run time*). Esto se puede conseguir, por ejemplo, poniendo un *wait* como última instrucción.

*Ejemplo:* definición de un *testbench* para simular el circuito indicado en la parte derecha (simulación funcional). Se utilizan 2 procesos, uno para generar la señal periódica *clk* y otro para generar las señales *reset* y *d*.

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

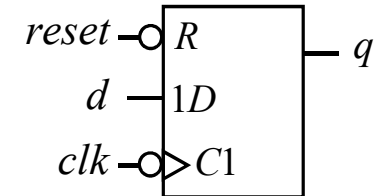
```
entity flip_flop_d_tb is  
end flip_flop_d_tb;
```

```
architecture behavior of flip_flop_d_tb is
```

```
    component flip_flop_d -- declaración del component cuyo funcionamiento se va a simular  
    port (reset, d, clk : in std_logic;  
          q : out std_logic);  
    end component;
```

```
    signal reset : std_logic := '0'; -- declaración de señales de conexión con el componente  
    signal clk : std_logic := '0';   -- a simular  
    signal d : std_logic := '0';  
    signal q : std_logic;
```

```
    constant clk_period : time := 20 ns; -- se define la constante clk_period con el valor 20 ns
```





```
begin -- architecture
```

```
    uut: flip_flop_d -- instantiate the unit under test (llamada al process a simular)
```

```
        port map (reset => reset,
```

```
                  clk => clk,
```

```
                  d => d,
```

```
                  q => q);
```

```
clk_process : process -- con este process se genera una señal periódica denominada clk
```

```
begin -- con un ciclo de trabajo 0.5 y periodo el valor de la constante clk_period
```

```
    clk <= '0'; -- este process se ejecuta cada vez que se cumple la condición de un wait
```

```
    wait for clk_period/2; -- se esperan 10ns antes de ejecutar la siguiente instrucción
```

```
    clk <= '1';
```

```
    wait for clk_period/2; -- se esperan 10ns antes de ejecutar la primera instrucción del process
```

```
end process;
```

```
stim_proc: process -- con este process se generan los valores de las señales de entrada
```

```
begin
```

```
    wait for 30 ns; -- se mantiene activo el reset durante los primeros 30 ns.
```

```
    reset <= '1'; -- se desactiva el reset (activo a nivel bajo)
```

```
    wait for clk_period;
```

```
    d <= '1';
```

```
    wait for clk_period;
```

```
    d <= '0';
```

```
    wait for clk_period;  
    d <= '1';  
    wait for 2*clk_period;  
    d <= '0';  
    wait;  
end process;  
end;
```

Otra versión del *testbench* anterior con un solo process (simulación funcional).

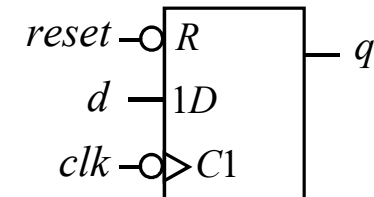
```
library ieee;  
use ieee.std_logic_1164.all;  
  
entity flip_flop_d_tb is  
end flip_flop_d_tb;
```

```
architecture behavior of flip_flop_d_tb is
```

```
    component flip_flop_d -- declaración del component cuyo funcionamiento se va a simular  
    port (reset, d, clk : in std_logic;  
          q : out std_logic);  
    end component;
```

```
    signal reset, clk, d, q : std_logic; -- declaración de señales de conexión con el componente  
                                         -- a simular.
```

```
begin -- architecture  
    uut: flip_flop_d -- llamada al component a simular  
    port map (reset => reset,  
              clk => clk,  
              d => d,  
              q => q);
```



```
stim_proc: process -- con este process se generan los valores de las señales de entrada
begin
    reset <= '0'; -- valores iniciales (se activa el reset)
    clk <= '0';
    d <= '0';
    wait for 30 ns;
    reset <= '1'; -- se desactiva el reset (activo a nivel bajo)
    wait for 10ns; -- 40ns
    clk <= '1';
    wait for 10ns; -- 50ns
    clk <= '0';
    wait for 10ns; -- 60ns
    clk <= '1';
    d <= '1';
    wait for 10ns; -- 70ns
    clk <= '0';
    wait for 10ns; -- 80ns
    clk <= '1';
    d <= '0';
    wait for 10ns; -- 90ns
```

```
    clk <= '0';  
    wait for 10ns; -- 100ns  
    clk <= '1';  
    wait for 10ns; -- 120ns  
    clk <= '0';  
    wait for 10ns; -- 130ns  
    clk <= '1';  
    d <= '1';  
    wait for 10ns; -- 140ns  
    clk <= '0';  
    wait for 10ns; -- 150ns  
    clk <= '1';  
    wait for 10ns; -- 160ns  
    clk <= '0';  
    wait;  
    end process;  
end;
```

*Ejemplo:* definición de un *testbench* para simular el comportamiento del circuito indicado en la parte derecha (simulación funcional).

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

entity tb is
end tb;

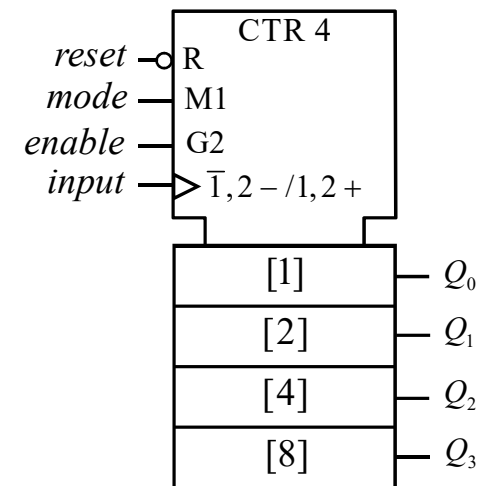
architecture behavior of tb is
component contador_4_reversible
port (reset, mode, enable, input : in std_logic;
      q : inout std_logic_vector(3 downto 0));
end component;

signal reset, mode, enable, clk : std_logic; -- se podrían asignar valores iniciales
signal q : std_logic_vector(3 downto 0);

constant clk_period : time := 20 ns;

begin

uut: contador_4_reversible -- llamada al component que se va a simular
port map (reset => reset,
          mode => mode,
          enable => enable,
          input => clk,
          q => q);
```



```

clk_process : process  -- process que genera una señal clk periódica
begin
    clk <= '0';
    wait for clk_period/2;
    clk <= '1';
    wait for clk_period/2;
end process;

stim_proc : process  -- process que genera las señales de entrada
begin
    reset <= '0'; -- valores iniciales (reset activo a nivel bajo)
    mode <= '0';
    enable <= '0';
    wait for 40 ns;
    reset <= '1'; -- se desactiva el reset
    wait for clk_period;
    enable <= '1';
    wait for clk_period*3;
    mode <= '1';
    wait;
end process;
end;

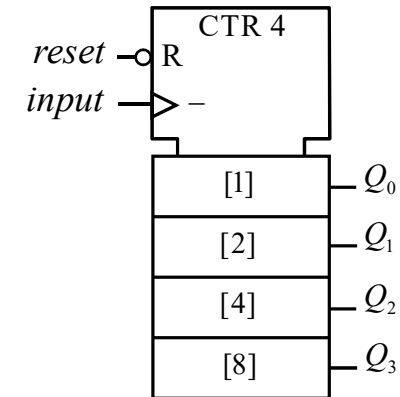
```

Definición del circuito indicado en la parte derecha

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

entity contador_4_bits_reset is
    port (reset, input : in std_logic;
          Q : out std_logic_vector (3 downto 0));
end contador_4_bits_reset;

architecture behavioral of contador_4_bits_reset is
begin
    process (reset, input)
        variable aux : unsigned (3 downto 0) := "1010"; -- en la simulación Q empieza en 10
    begin
        if(reset = '0') then aux := "0000";
        elsif (rising_edge(input)) then aux := aux - 1;
        end if;
        Q <= std_logic_vector(aux);
    end process;
end behavioral;
```





*Ejemplo:* de un *testbench* para simular el comportamiento del circuito indicado en la parte derecha (simulación funcional).

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

entity tb is
end tb;

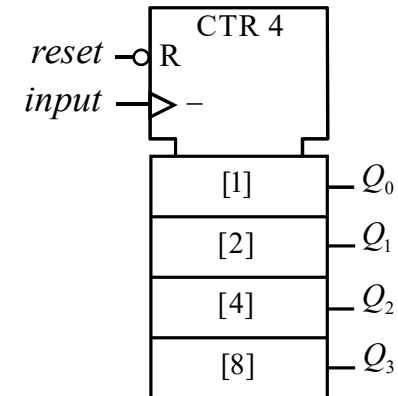
architecture behavior of tb is

component contador_4_bits_reset -- declaración componente
    port(reset, input : in std_logic;
          q : out std_logic_vector(3 downto 0));
end component;

    signal reset : std_logic := '1'; -- reset no activo
    signal input : std_logic := '0';
    signal q : std_logic_vector(3 downto 0);

begin

    uut: contador_4_bits_reset -- instantiate the unit under test (uut)
    port map (reset => reset, input => input, q => q);
```



```
process -- este proceso genera la señal input (es un bucle infinito)
```

```
begin
```

```
    wait for 10 ns;
```

```
    input <= '1';
```

```
    wait for 10 ns;
```

```
    input <= '0';
```

```
end process;
```

```
process -- este proceso genera las señal de reset
```

```
begin
```

```
    wait for 400 ns;
```

```
    reset <= '0';
```

```
end process;
```

```
end;
```

*Ejemplo de un testbench para el circuito de la figura (simulación funcional)*

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

```
entity tb is  
end tb;
```

```
architecture behavior of tb is
```

```
    component comparador_2bits  -- declaración del componente a simular
```

```
    port( a, b : in std_logic_vector(1 downto 0);
```

```
          c : out std_logic);
```

```
end component;
```

```
signal a, b : std_logic_vector(1 downto 0);
```

```
signal c : std_logic;
```

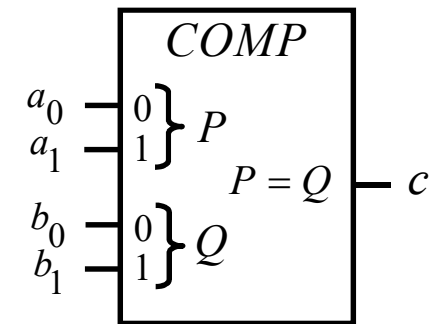
```
begin
```

```
    uut: comparador_2bits -- llamada al circuito a simular
```

```
    port map (a => a, b => b, c => c);
```

```
    stim_proc: process  -- process que genera las señales de entrada
```

```
    -- continúa en la página siguiente
```



```
begin
wait for 10 ns;
a <= "01";
b <= "10";
wait for 10 ns;
a <= "00";
b <= "01";
wait for 10 ns;
a <= "11";
b <= "01";
wait for 10 ns;
a <= "11";
b <= "11";
wait;
end process;
end;
```

## Notas para ISE:

- A la hora de crear un *test bench*, en View hay que tener seleccionada la opción *Simulation* y a continuación hacer 1 click con el ratón en el nombre de la FPGA

- Para realizar una *simulación funcional*, hay que hacer lo siguiente:

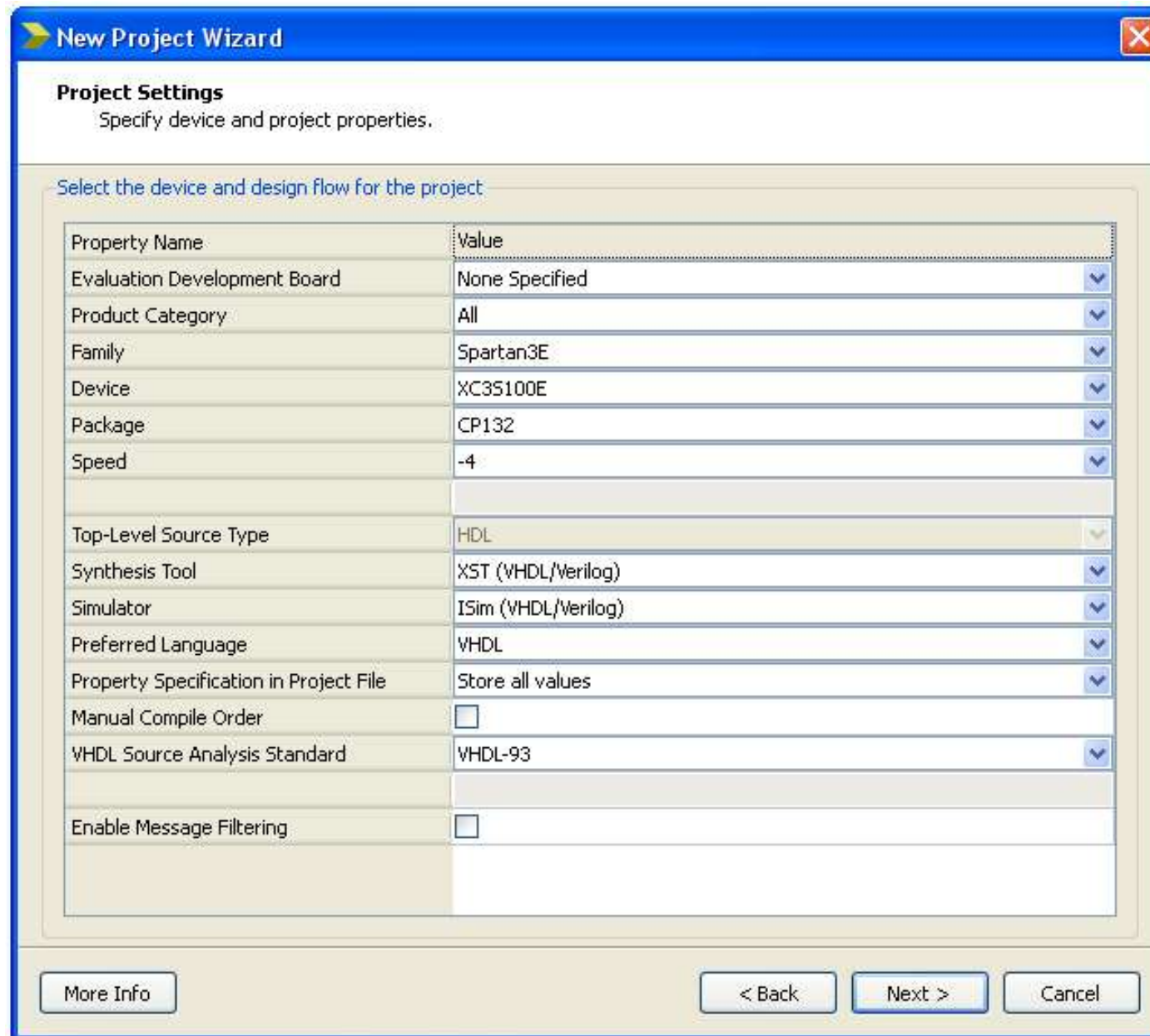
- \_ en *View* hay que elegir *Simulation*

- \_ en *Hierarchy* hay que seleccionar el *test bench*

- \_ en *Processes* hay que hacer doble click en *Simulate Behavioral Model* o bien hacer click con el botón derecho en *Simulate Behavioral Model* y elegir *Rerun All*. (conviene que antes se haga doble click en *Behavioral check syntax*)

- Para establecer el tiempo que se va a simular el funcionamiento del circuito, hay que hacer click con el botón derecho en *Simulate Behavioral Model* y seleccionar *Process Properties*. En la ventana que se abre se indica el tiempo en *Simulation Run Time*.

## Para la Basys 2



**New Project Wizard**

**Project Settings**  
Specify device and project properties.

Select the device and design flow for the project

Property Name	Value
Evaluation Development Board	None Specified
Product Category	All
Family	Spartan3E
Device	XC3S100E
Package	CP132
Speed	-4
Top-Level Source Type	HDL
Synthesis Tool	XST (VHDL/Verilog)
Simulator	ISim (VHDL/Verilog)
Preferred Language	VHDL
Property Specification in Project File	Store all values
Manual Compile Order	<input type="checkbox"/>
VHDL Source Analysis Standard	VHDL-93
Enable Message Filtering	<input type="checkbox"/>

## # ICF (*Implementation Constraints File*)... ejemplo para Basys 2

### # Interruptores

```
net "a<7>" LOC = "N3"; # SW7
net "a<6>" LOC = "E2"; # SW6
net "a<5>" LOC = "F3"; # SW5
net "a<4>" LOC = "G3"; # SW4
net "a<3>" LOC = "B4"; # SW3
net "a<2>" LOC = "k3"; # SW2
net "a<1>" LOC = "L3"; # SW1
net "a<0>" LOC = "P11"; # SW0
```

### # Leds

```
net "b<0>" LOC = "M5"; # Led 0
net "b<1>" LOC = "M11"; # Led 1
net "b<2>" LOC = "P7"; # Led 2
net "b<3>" LOC = "P6"; # Led 3
```

### # Botones

```
net "c" LOC = "G12"; # BTN0
net "d" LOC = "C11"; # BTN 1
net "e" LOC = "M4"; # BTN 2
net "f" LOC = "A7"; # BTN 3
```

### # Reloj

```
net "clk" LOC = "B8";
```

## VHDL or VERILOG

To answer the question of VHDL vs Verilog, you should really keep in mind your goals and where you'll be using the language. In the United States, the commercial industries tend to use more Verilog, while the aerospace and defense industries more heavily favor VHDL; the language you learn should reflect which industry you're more interested in.

Also consider the depth and style you're comfortable working at. VHDL lends itself to describing hardware at more abstracted levels (like case statements, if/then, etc.), while Verilog is good at describing hardware down to the gate level (nand, xor, etc.). If you're not familiar with schematic layout or your background is heavy in computer programming, you might be better off using VHDL until you really get familiar with the physical FPGA constructs.